SAVCBS 2004
Specification and Verification of Component-Based Systems

SIGSOFT 2004/FSE-12
12th ACM SIGSOFT Symposium on the Foundations of Software Engineering
Newport Beach, California, USA
October 31-November 5, 2004

Technical Report #04-09, Department of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, IA 50011-1041, USA

SAVCBS 2004 PROCEEDINGS

Specification and Verification of Component-Based Systems

http://www.cs.iastate.edu/ SAVCBS/

October 31-November 1, 2004
Newport Beach, California, USA

Workshop at SIGSOFT 2004/FSE-12
12th ACM SIGSOFT Symposium on the
Foundations of Software Engineering

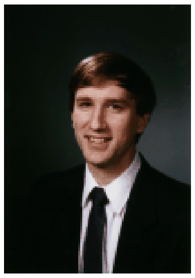
SAVCBS 2004
TABLE OF CONTENTS

SAVCBS 2004 ORGANIZING COMMITTEE

Mike Barnett (Microsoft Research, USA)
Mike Barnett is a Research Software Design Engineer in the Foundations of Software Engineering group at Microsoft Research. His research interests include software specification and verification, especially the interplay of static and dynamic verification. He received his Ph.D. in computer science from the University of Texas at Austin in 1992.

Stephen H. Edwards (Dept. of Computer Science, Virginia Tech, USA)
Stephen Edwards is an associate professor in the Department of Computer Science at Virginia Tech. His research interests are in component-based software engineering, automated testing, software reuse, and computer science education. He received his Ph.D. in computer and information science from the Ohio State University in 1995.

Dimitra Giannakopoulou (RIACS/NASA Ames Research Center, USA)
Dimitra Giannakopoulou is a RIACS research scientist at the NASA Ames Research Center. Her research focuses on scalable specification and verification techniques for NASA systems. In particular, she is interested in incremental and compositional model checking based on software components and architectures. She received her Ph.D. in 1999 from the Imperial College, University of London.



Gary T. Leavens (Dept. of Computer Science, Iowa State University, USA)
Gary T. Leavens is a professor of Computer Science at Iowa State University. His research interests include programming and specification language design and semantics, program verification, and formal methods, with an emphasis on the object-oriented and aspect-oriented paradigms. He received his Ph.D. from MIT in 1989.

Natasha Sharygina (Carnegie Mellon University, SEI, USA)
Natasha Sharygina is a senior researcher at the Carnegie Mellon Software Engineering Institute and an adjunct assistant professor in the School of Computer Science at Carnegie Mellon University. Her research interests are in program verification, formal methods in system design and analysis, systems engineering, semantics of programming languages and logics, and automated tools for reasoning about computer systems. She received her Ph.D. from The University of Texas at Austin in 2002.

Program Committee:
Jonathan Aldrich (Carnegie Mellon University)
Mike Barnett (Microsoft Research)
Manfred Broy (Universität München)
Betty H. C. Cheng (Michigan State University)
Edmund M. Clarke (Carnegie Mellon University)
Matthew Dwyer (University of Nebraska)
Stephen H. Edwards (Virginia Tech)
Dimitra Giannakopoulou (RIACS /NASA Ames Research Center)
Gary T. Leavens (Iowa State University)
K. Rustan M. Leino (Microsoft Research)
Jeff Magee (Imperial College, London)
Rupak Majumdar (UCLA)
Peter Müller (ETH Zürich)
Wolfram Schulte (Microsoft Research)
Natalia Sharygina (Carnegie Mellon University, SEI)
Murali Sitaraman (Clemson University)
Clemens Szyperski (Microsoft Research)

Sponsors:

Microsoft® Research

SAVCBS 2004 WORKSHOP INTRODUCTION

This workshop is concerned with how formal (i.e., mathematical) techniques can be or should be used to establish a suitable foundation for the specification and verification of component-based systems. Component-based systems are a growing concern for the software engineering community. Specification and reasoning techniques are urgently needed to permit composition of systems from components. Component-based specification and verification is also vital for scaling advanced verification techniques such as extended static analysis and model checking to the size of real systems. The workshop will consider formalization of both functional and non-functional behavior, such as performance or reliability.

This workshop brings together researchers and practitioners in the areas of component-based software and formal methods to address the open problems in modular specification and verification of systems composed from components. We are interested in bridging the gap between principles and practice. The intent of bringing participants together at the workshop is to help form a community-oriented understanding of the relevant research problems and help steer formal methods research in a direction that will address the problems of component-based systems. For example, researchers in formal methods have only recently begun to study principles of object-oriented software specification and verification, but do not yet have a good handle on how inheritance can be exploited in specification and verification. Other issues are also important in the practice of component-based systems, such as concurrency, mechanization and scalability, performance (time and space), reusability, and understandability. The aim is to brainstorm about these and related topics to understand both the problems involved and how formal techniques may be useful in solving them.

The goals of the workshop are to produce:

1. An outline of collaborative research topics,
2. A list of areas for further exploration,
3. An initial taxonomy of the different dimensions along which research in the area can be categorized. For instance, static/dynamic verification, modular/whole program analysis, partial/complete specification, soundness/completeness of the analysis, are all continuums along which particular techniques can be placed, and
4. A web site that will be maintained after the workshop to act as a central clearinghouse for research in this area.

SAVCBS 2004 PAPERS

SPECIFICATION & VERIFICATION OF COMPONENT-BASED SYSTEMS

Verification of Multithreaded Object-Oriented Programs with Invariants

Bart Jacobs[*]
Dept. of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
bart.jacobs@cs.kuleuven.ac.be

K. Rustan M. Leino
Microsoft Research
One Microsoft Way
Redmond, WA, USA
leino@microsoft.com

Wolfram Schulte
Microsoft Research
One Microsoft Way
Redmond, WA, USA
schulte@microsoft.com

ABSTRACT

Developing safe multithreaded software systems is difficult due to the potential unwanted interference among concurrent threads. This paper presents a sound, modular, and simple verification technique for multithreaded object-oriented programs with object invariants. Based on a recent methodology for object invariants in single-threaded programs, this new verification technique enables leak-proof ownership domains. These domains guarantee that only one thread at a time can access a confined object.

0. INTRODUCTION

A primary aim of a reliable software system is ensuring that all objects in the system maintain *consistent* states: states in which all fields, and all fields of other objects on which they depend, contain legal meaningful values. In this paper, we formalize consistency constraints as *object invariants*, which are predicates over fields.

An object is consistent if it is in a state where its invariant must hold. We also allow an object to be in a mutable state, where its invariant may temporarily be violated.

It is hard to maintain object invariants in sequential programs, and it is even harder in concurrent programs. For example, consider the following method:

```
void $Transfer(DualAccounts\ o, \textbf{int}\ amount)$ {
    $o.a := o.a - amount$ ;
    $o.b := o.b + amount$ ;
}
```

Suppose this method is to maintain the invariant that for all dual accounts d: $d.a + d.b = 0$. In a concurrent setting, this invariant can be violated in several ways. Even if the programming system ensures that each read or write of a field is atomic, the interleavings might cause the invariant to be violated. For example, if one thread executes method $Transfer$ and reads $o.a$, but before the thread performs the write to $o.a$, another thread causes some update of $o.a$, then the invariant will not be maintained.

In a concurrent setting, consistency of an object can be ensured by exclusion at a level coarser than individual reads and writes. For example, while one thread updates an object, another is not allowed to perform any operation on the object. In contemporary object-oriented languages, exclusion is implemented via locking.

Guaranteed exclusion simplifies the automatic verification of multithreaded code a lot. It means that we can simply split the proof of the concurrent program into a proof for exclusion and a proof for a sequential program [17].

In this paper, we present a new programming methodology for sound modular verification of multithreaded object-oriented programs with object invariants. The methodology not only guarantees that every object protects itself from consistency violations, but it also allows aggregates of objects to define *leak-proof ownership domains*. These domains guarantee that only one thread at a time can access an object of the aggregate.

The methodology achieves modular static verification by requiring methods to be annotated with simple ownership requirements. The methodology is a extension of the Boogie methodology for sequential code, as described in our previous work [1].

The paper proceeds as follows. The next three sections gradually introduce our methodology: Section 1 introduces object invariants, Section 2 introduces confinement within objects, and Section 3 presents our extension to confinement within threads. In Section 4, we sketch a proof of the soundness of our verification method. We discuss additional issues of static verification in Section 5, of implementation in Java and C# in Section 6, and of run-time checking in Section 7. Sections 8 and 9 mention related work and conclude.

1. OBJECT INVARIANTS

We consider an object-oriented programming language with classes, for example like the class in Figure 0. Each class can declare an invariant, which is a predicate on the fields of an object of the class.

To allow a program temporarily to violate an object's invariant, the Boogie methodology [1] introduces into each object an auxiliary boolean field called inv.[0] We say that an object o is *consistent* if $o.inv = true$, otherwise we say the object is *mutable*. Only in the mutable state is the object's invariant allowed to be violated. The inv field can be mentioned in method contracts (*i.e.*, pre- and postconditions). It cannot be mentioned in invariants or in program

[*]Bart Jacobs co-authored this paper during an internship at Microsoft Research. Bart Jacobs is a Research Assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.-Vlaanderen).

[0]The Boogie methodology also deals with subclasses, but for brevity we here consider only classes without inheritance. Extending what we say to subclasses is straightforward.

```
class $IntList$ {
   rep int[] $elems$ := new int[10] ;
   int $count$ := 0 ;
   invariant $0 \leq count \land count \leq elems.Length$ ;

   void $Add$(int $elem$)
      requires $inv$ ;
   {
      unpack (this) ;
      if ($count = elems.Length$)
         { $elems := elems.Copy(count * 2)$ ; }
      $elems[count] := elem$ ;
      $count := count + 1$ ;
      pack (this) ;
   }
}
```

Figure 0: An example class, representing a extensible list of integers. The invariant links the $count$ field with the array length of the $elems$ field. The Add method maintains the invariant.

code. The inv field can be changed only by two special statements, **unpack** and **pack**. These statements delineate the scope in which an object is allowed to enter a state where its invariant does not hold.

The rules for maintaining object invariants are as follows:

- A new object is initially mutable.
- Packing an object takes it from a mutable state to a consistent state, provided its invariant holds.
- Unpacking an object takes it from a consistent state to a mutable state.
- A field assignment is allowed only if the target object is mutable.

We formalize these rules as follows, where $Inv_T(o)$ stands for the invariant of class T applied to instance o.

```
pack$_T$ $o$  $\equiv$
   assert $o \neq null \land \neg o.inv \land Inv_T(o)$ ;
   $o.inv \leftarrow true$

unpack$_T$ $o$  $\equiv$
   assert $o \neq null \land o.inv$ ;
   $o.inv \leftarrow false$

$o.f := E$  $\equiv$
   assert $o \neq null \land \neg o.inv$ ;
   $o.f \leftarrow E$
```

In this formalization, an **assert** statement checks the given condition and aborts program execution if the condition does not hold.

Our methodology guarantees the following program invariant for all reachable states, for each class T:

PROGRAM INVARIANT 0.

$$(\forall o\colon T \bullet\ o.inv \implies Inv_T(o)\ )$$

Here and throughout, quantifications are over non-null allocated objects.

```
class $Account$ {
   rep $IntList$ $hist$ := new $IntList$() ;
   int $bal$ := 0 ;
   invariant $bal =$
         $(\Sigma\, i \mid 0 \leq i < hist.count \bullet\ hist.elems[i]\ )$ ;

   void $Deposit$(int $amount$)
      requires $inv$ ;
      ensures $bal = \mathbf{old}(bal) + amount$ ;
   {
      unpack (this) ;
      $hist.Add(amount)$ ;
      $bal := bal + amount$ ;
      pack (this) ;
   }
}
```

Figure 1: An example class illustrating aggregate objects.

2. CONFINEMENT WITHIN OBJECTS

The accessibility modifiers (like **private** and **public**) in contemporary object-oriented languages cannot guarantee consistency. Consider for example the class $Account$ in Figure 1, which uses an $IntList$ object to represent the history of all deposits ever made to a bank account. A bank account also holds the current balance, which is the same as the sum of the history, as is captured by the invariant.

We say an $Account$ object is an *aggregate*: its *part* is the object referenced through the field $hist$. Part objects are also known as *representation objects*. We qualify fields holding representation objects with a **rep** modifier (*cf.* [16]).

A part is said to be *leaked* if it is accessible outside the aggregate. In a sequential setting, leaking is not considered harmful, as long as the parts are leaked only for reading [15, 1].

An aggregate *owns* its parts. Object ownership, here technically defined via **rep** fields, establishes a hierarchy among objects. Invariants and ownership are related as follows: the invariant of an object o can depend only on the fields of o and on the fields of objects reachable from o by dereferencing only **rep** fields. (We don't allow an invariant to mention any quantification over objects.)

To formulate ownership properly, we introduce for each object an $owner$ field. Like inv, the $owner$ field cannot be mentioned in program code. We say an object o is *free* if $o.owner = null$. An object is *sealed* if it has a non-null owner object and that owner is consistent. The *ownership domain* of an object o is the set collecting o and all objects that o transitively owns. The rules for **pack** and **unpack** enforce that ownership domains are packed and unpacked only according to their order in the ownership hierarchy. Furthermore, **pack** and **unpack** change the ownership of representation objects as described by the following rules, which extend the ones given earlier.[1] We use the function $RepFields_T$ to denote the fields marked **rep** within class T.

```
pack$_T$ $o$  $\equiv$
   assert $o \neq null \land \neg o.inv \land Inv_T(o)$ ;
   foreach ($f \in RepFields_T$ where $o.f \neq null$)
      { assert $o.f.inv \land o.f.owner = null$ ; }
   foreach ($f \in RepFields_T$ where $o.f \neq null$)
      { $o.f.owner \leftarrow o$ ; }
   $o.inv \leftarrow true$
```

[1]This is a slightly different use of the $owner$ field than in [13].

```
$\textbf{unpack}_T\ o\ \ \equiv$
  $\textbf{assert}\ o \neq null \wedge o.inv$ ;
  $o.inv \leftarrow false$ ;
  $\textbf{foreach}\ (f \in RepFields_T\ \ \textbf{where}\ o.f \neq null)$
    $\{\ o.f.owner \leftarrow null\ ;\ \}$
```

For illustration purposes, let us inspect a trace of the invocation $acct.Deposit(100)$ for a non-null $Account$ object $acct$ that satisfies the precondition of $Deposit$, where we focus only on the involved inv and $owner$ fields of the involved objects. First, $Deposit$ unpacks $acct$: $acct$ is made mutable, $hist$ is made free. Next, Add is called, which first unpacks $hist$ and makes it mutable. Next, the updates happen. On return from the Add method, $hist$ is packed again: the invariant of $hist$ is checked and $hist$ is made consistent. Finally, the $Deposit$ method packs $acct$: the invariant of $acct$ is checked, $acct$ is made consistent, and $hist$ is sealed. And that's exactly our pre-state restricted to inv and $owner$ fields of the objects in the ownership domain.

Generalizing from this example, we observe that the methodology ensures the following program invariant, for each class T:

PROGRAM INVARIANT 1.

$$
\begin{array}{l}
(\forall\, o\colon T \bullet\ o.inv \implies Inv_T(o)\ ) \wedge \\
(\forall\, f \in RepFields_T, o\colon T \bullet \\
\qquad o.inv \implies o.f = null \vee o.f.owner = o\ ) \wedge \\
(\forall\, o\colon T \bullet\ o.owner \neq null \implies o.inv\ )
\end{array}
$$

3. CONFINEMENT WITHIN THREADS

In the object ownership scheme above, objects are either part of an aggregate object or they are free, which means they do not have any owner. For modular verification of multithreaded code, we now refine this scheme again. We say that an object can either be *free*, it can be *owned by an aggregate object*, or it can be *owned by a thread*. Correspondingly, the owner field is $null$, an object, or a thread.[2]

To support sequential reasoning about field accesses, we require a thread to have exclusive access to the fields during the execution of the program fragment to which the sequential reasoning applies. We require a thread to transitively own an object whenever it reads one of its fields, and to directly own an object whenever it writes one of its fields. Since no two threads can (transitively) own the same object concurrently, this guarantees exclusion.

The rules for thread ownership are as follows:

- A thread owns any object that it creates, and the new object is initially mutable.
- A thread can additionally attempt to **acquire** any object. This operation will block until the object is free. At that point, we know that the object is consistent and the thread gains ownership of the object.
- A thread can relinquish ownership of a consistent object using the **release** statement.
- A thread that owns a consistent aggregate object can gain ownership of its sealed representation objects by unpacking the aggregate object using the **unpack** statement. This transfers ownership of the representation objects from the aggregate object to the thread.

[2]In this text, threads are not objects. In some languages, like Java and C#, a thread has a representation as an object; we can avoid ambiguity in these languages by requiring that thread objects have no **rep** fields, which allows us to stipulate that when a thread object appears as an owner, it denotes the thread, not the object.

- A thread can, via a **pack** statement, transfer ownership of a consistent object that it owns to an aggregate object.
- A thread can perform a field assignment only if it owns the target object and the target object is mutable.
- A thread can read a field only if it transitively owns the target object. We actually enforce this rule by a slightly stricter rule: a thread can evaluate an access expression $o.f_1.\cdots.f_n.g$ only if it owns o and each object in the sequence $o.f_1.\cdots.f_n$ owns the next one.

These rules are an extension of the rules presented in the previous section. They give rise to the object lifecycle shown in Figure 2. Fully spelled out, they are formalized as follows, where we denote the currently executing thread by **tid** .

```
$\textbf{pack}_T\ o\ \ \equiv$
  $\textbf{assert}\ o \neq null \wedge o.owner = \textbf{tid} \wedge \neg o.inv$ ;
  $\textbf{foreach}\ (f \in RepFields_T\ \ \textbf{where}\ o.f \neq null)$
    $\{\ \textbf{assert}\ o.f.owner = \textbf{tid} \wedge o.f.inv\ ;\ \}$
  $\textbf{foreach}\ (f \in RepFields_T\ \ \textbf{where}\ o.f \neq null)$
    $\{\ o.f.owner \leftarrow o\ ;\ \}$
  $\textbf{assert}\ Legal[Inv_T(o)] \wedge Inv_T(o)$ ;
  $o.inv \leftarrow true$

$\textbf{unpack}_T\ o\ \ \equiv$
  $\textbf{assert}\ o \neq null \wedge o.owner = \textbf{tid} \wedge o.inv$ ;
  $o.inv \leftarrow false$ ;
  $\textbf{foreach}\ (f \in RepFields_T\ \ \textbf{where}\ o.f \neq null)$
    $\{\ o.f.owner \leftarrow \textbf{tid}\ ;\ \}$

$\textbf{acquire}\ o\ \ \equiv$
  $\textbf{assert}\ o \neq null \wedge o.owner \neq \textbf{tid}$ ;
  $\textbf{await}\ (o.owner = null)\ \{\ o.owner \leftarrow \textbf{tid}\ ;\ \}$

$\textbf{release}\ o\ \ \equiv$
  $\textbf{assert}\ o \neq null \wedge o.owner = \textbf{tid} \wedge o.inv$ ;
  $o.owner \leftarrow null$

$o.f := v\ \ \equiv$
  $\textbf{assert}\ o \neq null \wedge o.owner = \textbf{tid} \wedge \neg o.inv$ ;
  $o.f \leftarrow v$

$x := E\ \ \equiv$
  $\textbf{assert}\ Legal[E]$ ;
  $x \leftarrow E$
```

In the above, we write $Legal[E]$ to denote the predicate that says that every access expression in E is transitively owned by the current thread, as stipulated by the last bullet above. In particular,

$$
\begin{array}{l}
Legal[x]\ \equiv\ true \\
Legal[E_0\ \textbf{op}\ E_1]\ \equiv\ Legal[E_0] \wedge Legal[E_1] \\
Legal[o.f_1.\cdots.f_n.g]\ \equiv \\
\quad o.owner = \textbf{tid} \wedge \\
\quad o.f_1.owner = o \wedge \\
\quad \cdots \wedge \\
\quad o.f_1.\cdots.f_n.owner = o.f_1.\cdots.f_{n-1}
\end{array}
$$

When a thread attempts to execute a statement **await** $(P)\ \{\ S\ \}$, it blocks until the condition P is $true$, at which point the statement S is executed; the evaluation of P that finds P to be $true$ and the execution of S are performed as one indivisible action.

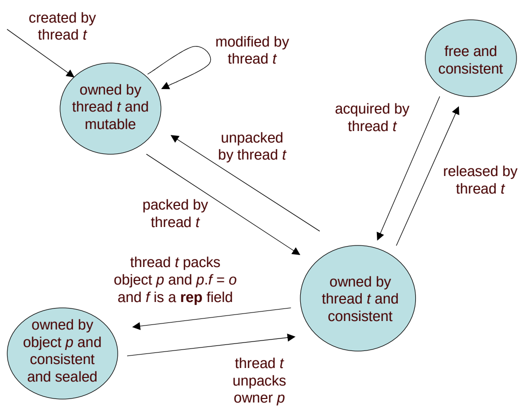

created by thread *t*
modified by thread *t*
free and consistent
owned by thread *t* and mutable
acquired by thread *t*
unpacked by thread *t*
released by thread *t*
packed by thread *t*
thread *t* packs object *p* and *p.f* = *o* and *f* is a **rep** field
owned by thread *t* and consistent
owned by object *p* and consistent and sealed
thread *t* unpacks owner *p*

Figure 2: Object lifecycle for an arbitrary object o.

```
class Account {
  void Deposit(int amount)
    requires owner = tid ∧ inv ;
    ...
}

class IntList {
  void Add(int value)
    requires owner = tid ∧ inv ;
    ...
}
```

Figure 3: The example classes *Account* and *IntList*, revised for their use in a multithreaded environment.

Now let us extend our running example, so that we can verify it in a multithreaded environment. First, we have to make sure that *Account* and *IntList* objects are accessed only when they are owned by the current thread. We choose to relegate the responsibility of exclusion to the client, a pattern which is often called *client-side locking*. We indicate this by including the requirement $owner = \mathbf{tid}$ in the preconditions for methods *Deposit* and *Add*, see Figure 3. A program is allowed to mention $o.owner$ only in the form $o.owner = \mathbf{tid}$ and only in method contracts.

We extend the example with a *Bank* class, which allows transfers between different accounts, see Figure 4. The method *Transfer* requires that the *from* and *to* accounts are owned by the current thread. If the precondition holds, *Transfer* performs the intended account operations without blocking. The method *Transaction* does the same thing, but has no requirement on thread ownership. Therefore, *Transaction* acquires the *from* and *to* objects, each of which might block.

Note that method *Transfer* declares a postcondition, whereas method *Transaction* does not. In fact, *Transaction* cannot ensure the same postcondition as *Transfer*, since other threads might intervene as soon as the account objects are released. For method *Transfer*, on the other hand, the postcondition is stable, since the calling thread owns the account objects, which affords it exclusive access.

Our methodology ensures the following program invariant, for

```
class Bank {
  static void Transfer(Account from,
                       Account to,
                       int amount)
    requires from ≠ null ∧ to ≠ null ∧ from ≠ to ∧
             from.owner = tid ∧ to.owner = tid ;
    ensures from.bal = old(from.bal) − amount ∧
            to.bal = old(to.bal) + amount ;
  {
    from.Deposit(−amount) ;
    to.Deposit(amount) ;
  }
  static void Transaction(Account from,
                          Account to,
                          int amount)
    requires from ≠ null ∧ to ≠ null ∧ from ≠ to ;
  {
    acquire from ;
    acquire to ;
    Transfer(from, to, amount) ;
    release to ;
    release from ;
  }
}
```

Figure 4: A safe multithreaded bank example.

each class T:

PROGRAM INVARIANT 2.

$$(\forall\, o\colon T \bullet\ o.inv \implies Inv_T(o)\ ) \tag{0}$$

$$(\forall\, f \in RepFields_T, o\colon T \bullet\ o.inv \implies o.f = null \lor o.f.owner = o\ ) \tag{1}$$

$$(\forall\, o\colon T \bullet\ o.owner \notin \mathfrak{thread} \implies o.inv\ ) \tag{2}$$

4. SOUNDNESS

In this section, we prove two results for our methodology. First, there are no data races. Second, if an object is consistent, its invariant holds.

A *data race* occurs when a field is accessed concurrently by two threads and at least one of the threads is performing a write to the field. If a data race occurs, the values read or written by a thread may be unpredictable, which severely complicates reasoning about the program.

As we have formalized our methodology in the previous section, there actually are data races, in particular on the *owner* field. Fortunately, we can eliminate these data races by introducing redundant thread-local data into our program state, as follows:

- With each thread t, we associate a thread-local table *owns*, which maps object references to booleans.
- We extend the semantics of all statements that perform updates on *owner* fields so that they also update the local thread's *owns* variable. These updates will maintain the following invariant, for any object o and thread t:

  $$t.owns[o] \implies o.owner = t$$

- We modify the semantics of all statements whose preconditions require $o.owner = \mathbf{tid}$ for some o, so that these preconditions instead require $\mathbf{tid}.owns[o]$.

- We assume any write to the $owner$ field of an object to be an indivisible action.

With these modifications, we can now prove the following lemma and theorem:

LEMMA 0. *The methodology guarantees that (1) holds in all reachable states.*

THEOREM 1 (RACE FREEDOM). *Consider any object o in an execution of a program. If t is a thread that transitively owns o, then t is the only thread that can read or write a field of o or change the transitive ownership of o. Furthermore, if the transitive owner of o is $null$, then the only field of o that a thread reads or writes is $o.owner$, and the thread reads and writes $o.owner$ only at a moment when $o.owner = null$.*

We prove Lemma 0 and Theorem 1 together:

PROOF. Consider an arbitrary execution of the program. We prove by induction that the required properties hold in every prefix of the execution.

We look at our formalization of each program construct, as given in the previous section. Except for the **unpack** and **acquire** statement, these rules guarantee that each read or write of a field $o.f_1.\cdots.f_n.g$ is protected by an expression equivalent to the expansion of $Legal[\![o.f_1.\cdots.f_n.g]\!]$ (we assume the evaluation of $\wedge$ to be conditional-and). By the induction hypothesis, these conditions are stable (with respect to the execution of other threads).

This property is also guaranteed for the **unpack** statement, except for its update of $o.f.owner$. Here's where we need the lemma. By the inductive hypothesis of the lemma, we have the disjunction $o.f = null \vee o.f.owner = o$ immediately after checking $o.inv$. By the inductive hypothesis of the theorem, this disjunction is stable. Therefore, $o.f.owner = o$ holds inside the **foreach** loop (unless a previous iteration of the **foreach** loop has already assigned **tid** to $o.f.owner$, which is also okay; this situation arises if o has two **rep** fields referencing the same part).

For the **acquire** statement, the reading and writing of $o.owner$ happens at a time when $o.owner = null$, as required by the theorem.

For the lemma, (1) holds in the empty prefix of the execution, since no objects are allocated then, which means the quantifications are vacuously true. We now turn to nonempty prefixes of the execution.

Condition (1) can be violated if the quantifier's range is enlarged to a newly allocated object. But new objects are initially mutable, so (1) is maintained.

Condition (1) can be violated if an inv field is set to $true$, which happens only in the **pack** statement. There, the update of $o.inv$ is preceded by assignments to $o.f.owner$ for representation fields $o.f$. By the theorem, the effect of these assignments is stable, and thus **pack** maintains (1).

Condition (1) can also be violated if a representation field $o.f$ is changed to a non-null value when $o.inv$ holds. But only the field update statement writes to fields, and its update is protected by $\neg o.inv$, which by the theorem is stable.

Finally, condition (1) can be violated if $p.owner$ is changed to a value q, when there is an object r and representation field g such that

$$r \neq q \wedge r.inv \wedge r.g = p$$

for then, after the assignment, we would have

$$r.inv \wedge r.g \neq null \wedge r.g.owner = q$$

The assignment to $o.f.owner$ in the **pack** statement is okay, because we argue that there are no r and g such that $r.g = o.f \wedge r.inv$: For a contradiction, suppose there are such an r and g. Then, by the induction hypothesis of (1), $r.g = null \vee r.g.owner = r$. It can't be $r.g = null$, because $o.f \neq null$. And it can't be $r.g.owner = r$, because the **pack** statement checks $o.f.owner$ to be a thread, not the object r.

The **unpack** statement changes $o.f.owner$, so we again argue that there are no r and g such that $r.g = o.f \wedge r.inv$. At the time the **unpack** statement checks $o.inv$, the induction hypothesis of (1) tells us that $o.f = null \vee o.f.owner = o$ for all representation fields f. The update of $o.f.owner$ happens only if $o.f \neq null$, so if $o.f.owner$ is updated, then $o.f.owner$ starts off as o. So the only r in danger is o itself. But at the time of the update of $o.f.owner$, $o.inv$ is $false$.

The **acquire** statement changes $o.owner$, but does so from a state where $o.owner$ is $null$.

The **release** statement changes $o.owner$, but does so from a state where $o.owner$ is a thread, not an object. ☐

Because of Theorem 1, we no longer have to argue about race conditions. That is, in the proof of the Soundness Theorem below, we can assume values to be stable.

THEOREM 2 (SOUNDNESS). *The methodology guarantees that Program Invariant 2 holds in all reachable states.*

PROOF. Lemma 0 already proves (1), so it remains to prove (0) and (2).

Consider an arbitrary execution of the program. We prove by induction that Program Invariant 2 holds in every prefix of the execution.

Program Invariant 2 holds in the empty prefix of the execution, since no objects are allocated then, which means the quantifications are vacuously true.

Consider any prefix of the execution leading to a state in which Program Invariant 2 holds. Let t be the thread that is about to execute the next atomic action. We prove by case analysis that this action maintains Program Invariant 2. In all cases, we make use of the fact that the $owner$ field is not mentioned in invariants.

- **Case** creation of a new object o. This operation affects only quantifications over objects, since the operation enlarges the range of such quantifications. Since $o.owner = t$ and $\neg o.inv$, and since for all p, $Inv_T(p)$ does not mention quantifications over objects, all conditions are trivially satisfied.

- **Case** $\mathbf{pack}_T\ o$. (0) and (2) follow directly from the semantics.

- **Case** $\mathbf{unpack}_T\ o$. (0) and (2) follow directly from the semantics.

- **Case** **acquire** o. (0) is vacuously maintained. (2) follows directly from the semantics.

- **Case** **release** o. (0) is vacuously maintained. (2) follows directly from the semantics.

- **Case** $o.f := v$. (2) is vacuously maintained. We prove the maintenance of (0) for an arbitrary object p of a type T. Suppose for a contradiction that $p.inv$ holds and that $Inv_T(p)$ depends on $o.f$. Then o must be reachable from p via non-null **rep** fields. Through repeated application of (1) and (2), we obtain that $o.inv$ holds. This contradicts the action's precondition, which incorporates $\neg o.inv$.

This concludes the proof. □

Having proved the Soundness Theorem, we can simplify the definition of $Legal$. In particular, we only need to check that the current thread owns the root object of an access expression and that all fields in the intermediate dereferences in the access expression are **rep** fields:

$$\begin{array}{l} Legal[o.f_1.\cdots.f_n.g] \;\equiv \\ \quad o.owner = \mathbf{tid} \\ \quad \text{and } f_1, \ldots, f_n \text{ are all } \mathbf{rep} \text{ fields} \end{array}$$

Program invariant (1) takes care of the rest.

The soundness proof assumes an interleaving semantics. This implies that memory accesses are sequentially consistent. Sequential consistency means that there is a total order on all memory accesses, such that each read action yields the value written by the last write action.

Unfortunately, most execution platforms do not actually guarantee sequential consistency. However, many do guarantee the following property, see for instance Manson and Pugh's proposed memory model for Java [14]:

> If in all sequentially consistent executions of a program P, all conflicting accesses are ordered by the happens-before relation, then all executions of P are sequentially consistent.

Since Theorem 1 proves the absence of data races, our Soundness Theorem is relevant even in these systems, provided a happens-before edge exists between writing the $owner$ field in the **release** statement and reading the $owner$ field in the **acquire** statement.

5. STATIC VERIFICATION

Our Soundness Theorem proves three properties that hold in every reachable state. These properties can therefore be assumed by a static program verifier at any point in the program.

By Theorem 1, we know that the values read by a thread are stable with respect to other threads. That is, as long as an object remains in the thread's ownership domain, the fields of the object are controlled exactly in the same way that fields of objects are controlled in a sequential program. Therefore, static verification proceeds as for a sequential program.

For objects outside the thread's ownership domain, all bets are off (as we alluded to in the discussion of the $Transaction$ method in Figure 4). But since a thread cannot read fields of such objects (Theorem 1), static verification is unaffected by the values of those fields.

When an object o enters a thread's ownership domain, we know that the invariants of all objects in o's ownership domain hold. In particular, due to our non-reentrant **acquire** statement and program invariant (2) of the Soundness Theorem, we have $o.inv$. To model the intervention of other threads between exclusive regions, a static verifier plays havoc on the fields of all objects in o's ownership domain after each **acquire** o operation. The static verifier can then assume $o.inv$. By repeated applications of program invariants (1) and (2), the verifier infers $p.inv$ for all other objects p in the ownership domain of o. Thus, by program invariant (0), the verifier infers that the invariants of all of these objects hold.

To check our methodology at run time, we only need to check the assertions prescribed in Section 3. However, to reason modularly about a program, as in static modular verification, one needs method contracts. We have already seen examples of pre- and postconditions, but method contracts also need to include *modifies clauses*, which frame the possible effects a method can have within the thread's ownership domain, see [1].

```
public class AcqRel {
  private boolean free ;
  public final synchronized void acquire()
    { while (!free) { wait() ; } free = false ; }
  public final synchronized void release()
    { free = true ; notify() ; }
}
```

Figure 5: Example implementation of acquire and release in Java.

6. SAFE CONCURRENCY IN JAVA AND C#

Our methodology uses **acquire** and **release** as synchronization primitives. But how, if at all, does this apply to the $synchronized$ of Java (or, equivalently, C#'s **lock** statement)? One might think that it would suffice to map Java's **synchronized** statement to **acquire** and **release** statements as follows:

```
[[ synchronized (o) { S } ]] =
    acquire o ;
    try { S } finally { release o ; }
```

Unfortunately, this approach is incorrect. Specifically, entering a **synchronized** statement is not semantically equivalent to the **acquire** statement because Java considers an object to be initially not owned, whereas our methodology considers an object to be initially owned by the thread that creates it. This manifests itself in the following specific behavior: in Java, the first thread that attempts to enter a synchronized statement always succeeds immediately; in our methodology, a **release** operation must occur on an object before any thread can successfully acquire it, even the first time.

Additionally, in this approach there is no syntax for an object's initial release operation; as a result, an object could never become free. One might suggest having an implicit release operation when an object is created, and requiring even the creating thread to synchronize on the object, even in the object's constructor. But this is problematic, since it would not give the creating thread a chance to establish the object's invariant before it is released.

But there are at least two ways to achieve a correct mapping between our methodology and Java and C#. The first consists of implementing $acquire$ and $release$ methods on top of the language's built-in primitives. An example implementation in Java is shown in Figure 5. With this implementation, acquiring an object o would correspond to calling the $acquire$ method of the $AcqRel$ object associated with object o. The latter association could be achieved using *e.g.* a hash table, or, depending on platform constraints, more efficient methods, such as merging the $AcqRel$ class into class $Object$.

The second way to apply our methodology to Java and C#, is by modifying the methodology. Specifically, a modified methodology exists such that executing an **acquire** or **release** statement on an object corresponds exactly with entering or exiting a **synchronized** statement that synchronizes on the object. The modification involves the introduction of an additional boolean field, called *shared*, in each object. The field is initially *false*, it can be mentioned only in method contracts, and it can be updated only through a special **share** statement.

In the modified methodology, the semantics of the statements

share, **acquire**, and **release** are as follows:

```
**acquire** $o$ $\equiv$
    **assert** $o \neq null \wedge o.shared \wedge o.owner \neq \mathbf{tid}$ ;
    **await** $(o.owner = null)$ { $o.owner \leftarrow \mathbf{tid}$ ; }

**release** $o$ $\equiv$
    **assert** $o \neq null \wedge o.owner = \mathbf{tid} \wedge o.shared \wedge o.inv$ ;
    $o.owner \leftarrow null$

**share** $o$ $\equiv$
    **assert** $o \neq null \wedge o.owner = \mathbf{tid} \wedge \neg o.shared \wedge o.inv$ ;
    $o.owner \leftarrow null$ ;
    $o.shared \leftarrow true$
```

In the modified methodology, exclusive access to an object by its creating thread during initialization is ensured not through run-time synchronization, but through constraints on the the newly introduced *shared* field imposed by the methodology.

7. RUN-TIME CHECKING

Our methodology supports both static verification and run-time checking. The advantage of static verification is that it decides the correctness of the program for all possible executions, whereas run-time checking decides whether the running execution complies with the methodology. The disadvantage of static verification is that it requires method contracts, including preconditions, postconditions, and modifies clauses, whereas run-time checking does not.

If a program has been found to be correct through static verification, no run-time checks would ever fail and they can be omitted. When running a program without run-time checks, the only run-time cost imposed by our methodology is the implementation of the **acquire** and **release** statements (as in Figure 5, for example); none of the fields or other data structures introduced by our methodology need to be present, and none of the **assert** statements need to be executed. In particular, the **pack** and **unpack** statements become no-ops.

For run-time checking, two fields, the *inv* field and the *owner* field, need to be inserted into each object. To prove race freedom, we eliminated the races on the *owner* fields by introducing an *owns* table for each thread; however, on most platforms, including Java and C#, these races are in fact benign and the *owns* tables can be omitted.

8. RELATED WORK

The Extended Static Checkers for Modula-3 [6] and for Java [8] attempt to statically find errors in object-oriented programs. These tools include support for the prevention of data races and deadlocks. For each field, a programmer can designate which lock protects it. However, these two tools trade soundness for ease of use; for example, they do not take into consideration the effects of other threads between regions of exclusion. Moreover, various engineering trade-offs in the tools notwithstanding, the methodology used by the tools was never formalized enough to allow a soundness proof.

Method specifications in our methodology pertain only to the pre-state and post-state of method calls. Some systems [18, 9] additionally support specification and verification of the atomic transactions performed during a method call, even though this information does not translate into knowledge about the post-state (because of intervening transactions by other threads).

A number of type systems have been proposed that prevent data races in object-oriented programs. For example, Boyapati *et al.* [4] parameterize classes by the protection mechanism that will protect their objects against data races. The type system supports thread-local objects, objects protected by a lock (*i.e.*, another object), read-only objects, and unique pointers. However, the ownership relationship that relates objects to their protection mechanism is fixed. Also, their type system does not support object invariants.

Quite similar to ours is the methodology used by Vault (*cf.* [5]), which can be applied in a concurrent setting. In Vault, linear types guarantee that objects are owned by a single thread only. The **pack** and **unpack** operations are implicit in Vault. The **acquire** operation is not supported, because the object to be acquired may have been deleted; however, it would be possible to add the **release acquire** operation pair to a version of Vault for a garbage-collected language. Vault's methodology is enforced by a static type system, which has advantages but limits its supported invariants. For example, Vault supports neither general predicates on the fields of an object nor relations on the fields of more than one object in an aggregate.

Atomizer [7] dynamically checks the atomicity of unannotated methods. It ensures that all statements in the method can be reasoned about sequentially. However, Atomizer does not easily support atomicity at different abstraction levels, which our methodology does.

Ábrahám-Mumm *et al.* [0] propose an assertional proof system for Java's reentrant monitors. It supports object invariants, but these can depend only on the fields of **this**. No claim of modular verification is made.

The rules in our methodology that an object must be consistent when it is released, and that it can be assumed to be consistent when it is acquired, are taken from Hoare's work on monitors and monitor invariants [10].

There are also tools that try dynamically to detect violations of safe concurrency. A noteable example is Eraser [19]. It finds data races by looking for locking-discipline violations. The tool has been effective in practice, but does not come with guarantees about the completeness nor the soundness of the method.

The basic object-invariant methodology that we have built on [1] has also been extended in other ways for sequential programs [13, 3, 12].

9. CONCLUSIONS

Our new sound, modular, and simple locking methodology helps in defining leak-proof ownership domains. Several aspects of this new approach are noteworthy. First, sequentially verifiable programs are race free. Due to the necessary preconditions for reading and writing, only one thread at a time can access the objects of an ownership domain. Second, the owner of an object can change over time. In particular, an object may move between ownership domains. Third, our methodology can be efficient; it acquires only one lock per ownership domain, where the domain consists of many objects. Further, at run time, we only need to keep track of a bit per object that says whether or not there exists a thread that transitively owns the object.

We are in the process of adding support for this methodology to Spec#, an extension of C# with contracts [2]. Spec# performs both run-time checking and static verification (via the program verifier Boogie).

But there is obviously much more left to be done. One important area of work is the assessment and optimization of the efficiency of both static verification and run-time checking on realistic examples. Also, we are currently extending the approach to deal with other design patterns, like traversals, wait and notification, condition variables, multiple reader writers, *etc.* In fact, our ambition is to cover many of the design patterns described by Doug Lea [11]. Another area of future work is the treatment of liveness properties, such as deadlock freedom.

Since our methodology is an extension of an object-invariant methodology for sequential programs, it would be interesting to automatically infer for given sequential programs the additional contracts necessary for concurrency.

Acknowledgments. We thank Manuel Fähndrich, Tony Hoare, the members of the Boogie team, and the referees for insightful remarks and suggestions.

10. REFERENCES

[0] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verification for Java's reentrant multithreading concept. In *Foundations of Software Science and Computation Structures, 5th International Conference, FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 5–20. Springer, April 2002.

[1] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Start devices (CASSIS)*, Lecture Notes in Computer Science. Springer, 2004. To appear.

[3] Mike Barnett and David Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 54–84. Springer, July 2004.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, volume 37, number 11 in *SIGPLAN Notices*, pages 211–230. ACM, November 2002.

[5] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 36, number 5 in *SIGPLAN Notices*, pages 59–69. ACM, May 2001.

[6] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[7] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 2004ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, volume 39, number 1 in *SIGPLAN Notices*, pages 256–267. ACM, January 2004.

[8] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.

[9] Stephen N. Freund and Shaz Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, June 2004.

[10] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[11] Doug Lea. *Concurrent Programming in Java*. Addison Wesley, 2000.

[12] K. Rustan M. Leino and Peter Müller. Modular verification of global module invariants in object-oriented programs. Technical Report 459, ETH Zürich, 2004.

[13] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[14] Jeremy Manson and William Pugh. Requirements for a programming language memory model. Workshop on Concurrency and Synchronization in Java Programs, in association with PODC, July 2004.

[15] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. PhD thesis, FernUniversität Hagen.

[16] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98—Object-oriented Programming: 12th European Conference*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, July 1998.

[17] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[18] Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing procedures in concurrent programs. In *Proceedings of the 2004 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, volume 39, number 1 in *SIGPLAN Notices*, pages 245–255. ACM, January 2004.

[19] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997. Also appears in *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 27–37, Operating System Review 31(5), 1997.

Encapsulating Concurrency as an Approach to Unification

Santosh Kumar[*], Bruce W. Weide[*], Paolo A.G. Sivilotti[*], Nigamanth Sridhar[†], Jason O. Hallstrom[‡], Scott M. Pike[#]

∗ The Ohio State University, Computer Science & Engineering, {kumars,weide,paolo}@cse.ohio-state.edu
† Cleveland State University, Electrical & Computer Engineering, n.sridhar1@csuohio.edu
‡ Clemson University, Computer Science, jasonoh@cs.clemson.edu
# Texas A&M University, Computer Science, pike@cs.tamu.edu

ABSTRACT

We extend traditional techniques for sequential specification and verification to systems involving intrinsically concurrent activities. Our approach uses careful design of component specifications to encapsulate inherent concurrency, and hence isolate clients from associated verification concerns. The approach has three parts: (i) relational specifications to capture the interleaved effects of concurrent threads of execution, (ii) intermediate components to support a client's view of being the only active thread of computation, and (iii) a new specification clause to express requirements on a client's future behavior. We illustrate these ideas, and discuss their merits, in the context of a case study specified using RESOLVE.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification—*Methodologies*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Design By Contract*

General Terms

Verification

Keywords

Unification problem, sequential verification techniques, concurrent systems, relational specification, mutual exclusion, expects clause.

1. INTRODUCTION

Design-by-contract [10] has long been recognized as a foundation for component-based specification and verification. For sequential systems, contract specification mediates interactions between a component and its *client*. By contrast, for concurrent systems the contract specification is between a component and its *environment*. The difference is significant since in a sequential system, the client and component operate in the same thread of execution, whereas in a concurrent system, the environment may include other threads of execution. Taking a sequential system view leads to traditional input-output specifications and Hoare-style verification techniques. On the other hand, taking a concurrent system view leads to explicit progress requirements and guarantees, where circularities in reasoning must be carefully avoided. As a result, research in these nominally related areas has focused on separate and largely orthogonal issues. There has been remarkably little cross-fertilization.

Specification and verification techniques for sequential systems (particularly component-based systems) generally involve showing that, given acceptable input values, a computation produces specified output values. The operational model is normally a standard von Neumann machine. Some issues that arise include (1) the contract style (*e.g.*, algebraic or model-based), (2) the impact of aliased references on sound reasoning, and (3) the difficulties introduced by modern programming language constructs (*e.g.*, inheritance, user-defined types, etc).

Specification and verification techniques for concurrent systems generally involve showing that cooperating processes adhere to specified temporal properties. Models of computation may vary, with message passing or shared memory, various degrees of synchrony, etc. Programming language features and variable types tend to be simpler in order to abstract away from complex details concerning "what is being computed" in the more traditional sense. Still, there are new communication constructs that are not present in the sequential case. Some issues that arise include (1) the expressiveness of various temporal logics, (2) the non-determinism introduced by interleaved access to shared resources, (3) progress and fairness properties for scheduling individual processes, and (4) non-interference requirements to guarantee sound proof systems, etc.

1.1 The Unification Problem

The *unification problem* in the title refers to the following impediment to fully successful specification and verification of component-based software: There is a need for a unified theory that reconciles work in the now largely disjoint research areas of specification and verification of sequential systems, and of concurrent systems. This problem is important because modern software systems involve both aspects. A typical modern program is charged not only with com-

Specification and Verification of Component-Based Systems (SAVCBS'04) Workshop at the 12^{th} ACM SIGSOFT Symposium on Foundations of Software Engineering. Newport Beach CA, USA. Oct 31 – Nov 1, 2004.

puting specified data values, but with doing so reliably in the context of concurrent threads of activity that may be accessing resources that it shares with them.

This paper partially develops an approach to addressing the unification problem. Based on an early case study, we believe it will permit extending the realm of traditional sequential specification and verification techniques to systems that involve intrinsically concurrent activities. The idea is to use certain component design and contract specification techniques to create what appear to be ordinary sequential components for the purposes of client interaction, and to "bury" or encapsulate concurrency inside the implementations of those components. This shields otherwise sequential clients of shared resources from the complications normally associated with concurrency. Among other things, for example, specification and verification of client correctness need not be extended from the sequential situation by introducing temporal logic whenever the client also has to deal with concurrent activities.

To give the flavor of the approach—and to emphasize that we are claiming novelty from the standpoint of specification and verification issues and *not* from the standpoint of, say, a new software design pattern—consider a nominally sequential program that needs to use a printer. The printer is a shared physical device that other programs like it might also be using. Without some method to encapsulate a protocol for ensuring mutually exclusive access to the printer, each program that shares it must understand how to negotiate access with the printer's other clients. This necessarily involves modeling the other clients, specifying liveness and safety properties associated with mutual exclusion, detailing some particular protocol for mutual exclusion, etc. But if there is a carefully-designed printer-monitor component that allows each client to create its own logical printer object and simply print to it as though it were alone in the world, all the complications associated with sharing—indeed, all the problems associated with specifying and reasoning about the concurrency involved in accessing the physical printer—are moved down one level in the system. The clients of the printer-monitor component see an ordinary sequential component. The problem of concurrency is not moved over the horizon, just down one level. But the impact is that there are more clients for which sequential contract specifications and verification techniques can be used.

This is, of course, just one possible approach to the unification problem. It is premature to discuss whether it might be the best one or whether it could ultimately lead to a complete solution to the problem. Instead, the goal of this paper is to report on progress and to describe some open issues in the hope that others will step forward to address the unification problem, too.

Paper Organization. The rest of the paper is organized as follows. Section 2 outlines the main technical features of our approach. Section 3 presents the case study. Section 4 sketches some related work. Section 5 concludes the paper and Section 6 points to open issues.

2. TECHNICAL FEATURES

There are three main technical features to the approach we present. Individually, each is not new; the primary contribution is combining them in a special new way. Each feature is highlighted during the case study in Section 3.

2.1 Using Relational Specifications

The first idea is to use relational specifications to model potential interference from concurrent threads of execution. In this way, each client of a component can maintain an especially simple view of the system: it is as though there are no other threads of execution. The hoped-for result is that specification and reasoning can be done using ordinary techniques for sequential systems — so long as they admit relational behavior (*i.e.*, are not restricted to purely functional input-output specifications).

There is an interesting connection between relational specifications and the philosophical theory of *epistemic solipsism*. Roughly stated, this doctrine draws a methodological distinction between ontology (that which exists) and epistemology (that which we can know exists). Epistemic solipsism witnesses a gap between these concepts, such that it is possible for certain entities to exist, despite our inability to know them. We make no judgment here about the coherence of solipsism as a philosophical doctrine, but only about the similarity in principle between solipsism and the use of relational specifications to mask concurrency.

By encapsulating concurrency inside an observably sequential component, we seek to create a cover story that makes it *appear to the client programmer* as though all state changes in all objects are the result of method calls made by the client. In particular, no object's value ever changes spontaneously; indeed, nothing changes unless the client initiates such a change. Concurrency may *exist* in the implementations of such components, but it cannot be *known* to the client insofar as it is unobservable through any component interface. Consequently, there is never a need for the client programmer to postulate the existence of concurrent threads of activity to explain or verify the correctness of the observed behavior with respect to the given sequential specifications.

2.2 Separating Proxy and Core Components

Direct application of the above idea in principle might lead to a sequential-component contract specification for the client. But it is not necessarily a pretty one. One can make visible, in the client's nominally sequential view of behavior, all the state needed to capture the underlying concurrency in the implementation of the component. The specification merely says that this state changes in bizarre though technically explainable ways. But probably most of this state is not needed to explain the same behavior, *i.e.*, the specification is not fully abstract. So, the second idea is to simplify the contract specification for the client by introducing an intermediate component that acts as a proxy in client dealings with the underlying component that encapsulates concurrency (which we call the core component).

Figure 1 illustrates how this works in the case study. Round-cornered boxes stand for contract specifications, rectangles for implementations, and arrows for the relationships on their labels. For example, the top box represents the client specification, which involves only sequential constructs. The rectangle below it stands for the implementation of this specification—what we have been calling the client program. This program claims to implement the specification above it, which is something that needs to be verified. It uses some implementation of the Mutex_Proxy specification, which in turn hides the concurrency inherent in the Mutex_Core specification below it. The client program could, in principle, directly use something like the specification Mutex_Core (although the specification details would be quite different than those developed in the case study). However, that specification would be unnecessarily complex compared to the much simpler Mutex_Proxy specification.

2.3 Additional Client Obligations

In reasoning about the correctness of implementations of the core and proxy components, one is faced with the usual verification situation for traditional sequential components: clients of a contract specification are responsible for establishing preconditions on calls, implementations are responsible for establishing postconditions. However, because of the need to reason about properties that otherwise would involve temporal logic specifications, we must introduce an additional reciprocal obligation on every client of the proxy. This is needed to show that *other* clients will make progress. The objective of proving total correctness of the client cannot be achieved if other clients—whose very existence is not knowable to any one of them—can thwart termination of any of the calls to proxy methods.

The third idea, then, is to introduce an additional specification construct in the sequential specification language. We call this new construct the expects clause. While a method's requires clause defines an obligation the client must meet *before* calling the method, its expects clause describes an obligation the client must meet at some point *after* calling the method. This obligation is given as a set of method calls that must be made in the future.

For example, consider the following path expression specification for a read-only file:

```
(Open; Read*; Close)
```

This expression stipulates that once a file has been opened, the Read() operation can be invoked several times (or never at all), and finally the file must be closed. That is, as a consequence of invoking Open(), the client is required to invoke Close() in the future. This obligation could be reflected in the contract as follows:

```
operation Open()
    requires true
    ensures self.Is_Open
    expects self.Close()
```

A brief operational sketch of the semantics of the expects clause is as follows. In addition to the program context and variable values in each state of a program, we maintain a "promises set" PS of all the method calls that the program has promised to other components via expects clauses, and has not yet discharged. Whenever an operation is invoked, it is removed from PS, if it is there. If the call is to an operation that has its own expects clause, the calls in that clause are added to PS. For other statements, PS is not modified. At termination, if the program has honored all the promises it made, then PS is empty. Therefore, the client, in addition to dispatching proof obligations for all pre-conditions, also has to dispatch an additional proof obligation that PS is an empty set at the point of termination. (Please see Section 6 for issues raised by this over-simplified view.)

3. CASE STUDY: MUTUAL EXCLUSION

In this section, we present our specification of a mutual exclusion component we call Mutex, using the RESOLVE specification language [7]. This single-object component is divided into two parts — the first, called Mutex_Proxy, presents a sequential view to the client, and the second, called Mutex_Core, encapsulates a conflict resolution protocol for mutual exclusion. All aspects of concurrency in the conflict resolution protocol are encapsulated inside Mutex_Core, and do not bleed through to the client. In effect, there are several instances of Mutex_Proxy in a distributed system (one for each client), all of which interact with a single instance of Mutex_Core. The design of this system structure is shown in Figure 1.

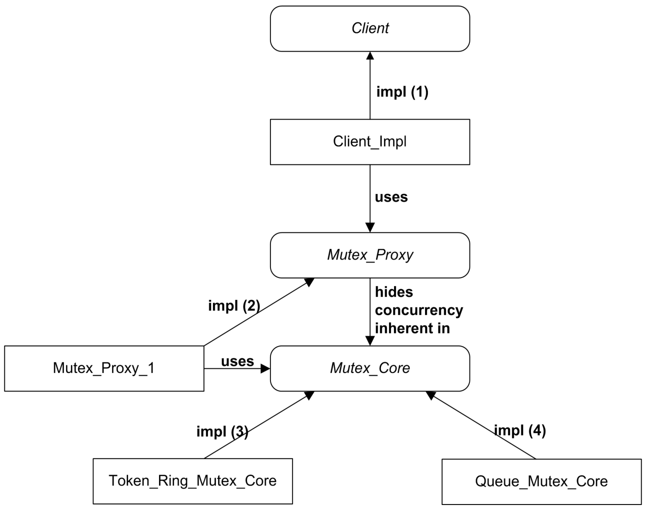


Figure 1: Component coupling diagram (CCD) for a system using the Mutex component

3.1 Mutex_Proxy

Listing 1 shows the specification of our Mutex_Proxy abstract component. The mathematical model of the Mutex_Proxy type is defined by MUTEX_PROXY_MODEL (Lines 8—14). This is a 3-tuple of two booleans, requested and available, and a NATURAL_NUMBER, wait_index. If the client that uses this proxy has requested access to the critical section, requested becomes **true**, and when it is safe for the client to access the critical section, available becomes **true**. The purpose of wait_index is to allow a client to prove its progress, *i.e.*, if it requests access to the resource, it will eventually have it available. For a given proxy, available can only become **true** if requested is also **true** (the client has requested access to the critical section). Upon initialization, every instance of Mutex_Proxy has both requested and available equal to **false** and wait_index equal to 0 (Lines 16—18).

The Mutex_Proxy component exports the following operations:

Request() **(Lines 20—26):** A client invokes the Request() method when it wants access to the critical section. The various preconditions make sure that this method can only be invoked once per "cycle" — once a client requests access, it cannot make another request until it has either used the critical section and then released it, or simply canceled its request by calling Release(). The post-condition of the Request() operation says that requested becomes **true** and wait_index assumes some natural number value (which, it turns out, the client has no way to observe, except to know whether it is zero). In addition, as a consequence of this invocation, the client is committed to invoking Release() in every possible future computation.

Is_Requested() **(Lines 28—30):** This operation can be invoked to check whether the client has requested access to the resource. (It is provided for functional completeness but plays no other substantive role.)

Check_If_Available() **(Lines 32—40):** After a client has requested access to the resource, a call to this procedure returns with b equal to **true** when the client can safely access the resource. Also, with every invocation of this method, the value of wait_index decreases if it is positive. Finally, if and only if wait_index is already 0, then it stays the same and available becomes **true** (as does b). Notice that Check_If_Available(), by virtue of its relational specification, has the property that (sometimes) it makes the client believe that available changes *as a result* of the Check_If_Available() call. That is, there is no reason for the client to explain the change in the value of available by postulating the existence of some other process having concurrent access; the client's call to Check_If_Available() is what has *caused* available to change.

Release() **(Lines 42—46):** The Release() operation is invoked to signal the end of the critical section or to cancel an outstanding request. This operation results in requested and available both becoming **false**.

3.2 Mutex_Core

Listing 2 shows the mathematical model of Mutex_Core.

PROXY_SET **(Lines 3—4):** This denotes a mathematical set of proxy identities, each of which is a natural number.

WAITING_PROXY **(Lines 6—8):** This denotes a pair of natural numbers, id to represent the identity of the proxy, and ticket to represent some metric that determines when the proxy will get access to the resource.

WAITING_PROXY_SET **(Lines 10—16):** This denotes a set of waiting proxies, *i.e.*, those that have requested access to the resource.

MUTEX_CORE_MODEL **(Lines 18—35):** The mathematical model of the Mutex_Core type is a tuple that consists of the set, all_proxies, of all the proxies that have "registered" with this Mutex_Core instance; the set, waiting_proxies, of just those proxies in all_proxies that have requested access to the resource but have not yet released it or canceled the request; and the integer, current_id, which is the (non-negative integer) identity of the proxy that currently has access to the resource because it has a minimum ticket value among all waiting proxies (or a negative number if there are no waiting proxies). Notice that a proxy can request access to a resource only after registering with the Mutex_Core instance that is responsible for that resource. Upon initialization, a Mutex_Core instance has no proxies registered (and hence no waiting proxies) and current_id is -1 (no proxy is accessing the resource).

Listings 3 and 4 present the specifications of the operations that the Mutex_Core component exports.

Listing 1: Specification of abstract Mutex_Proxy component

```
 1 abstract component Mutex_Proxy
 2
 3 math subtype NATURAL_NUMBER is integer
 4   exemplar n
 5   constraint
 6     n >= 0
 7
 8 math subtype MUTEX_PROXY_MODEL is
 9   (requested: boolean,
10    available: boolean,
11    wait_index: NATURAL_NUMBER)
12   exemplar mpm
13   constraint
14     if mpm.available then mpm.requested
15
16 Mutex_Proxy is modeled by MUTEX_PROXY_MODEL
17   initialization ensures
18     self = (false, false, 0)
19
20 procedure Request ()
21   requires
22     not self.requested
23   ensures
24     there exists i: NATURAL_NUMBER such that
25       (self = (true, false, i))
26   expects self.Release()
27
28 function Is_Requested (): Boolean
29   ensures
30     Is_Requested = self.requested
31
32 procedure Check_If_Available (replaces b: Boolean)
33   requires
34     self.requested
35   ensures
36     self.requested and
37     (if #self.wait_index /= 0
38      then self.wait_index < #self.wait_index
39      else #self.wait_index = self.wait_index) and
40     b = self.available = (self.wait_index = 0)
41
42 procedure Release ()
43   requires
44     self.requested
45   ensures
46     self = (false, false, 0)
```

Listing 2: Specification of abstract Mutex_Core component

```
abstract component Mutex_Core

math subtype PROXY_SET is
    finite set of NATURAL_NUMBER

math subtype WAITING_PROXY is
  (id: NATURAL_NUMBER,
   ticket: NATURAL_NUMBER)

math subtype WAITING_PROXY_SET is
    finite set of WAITING_PROXY
  exemplar wps
  constraint
    for all p,q: WAITING_PROXY
        where ({p,q} is subset of wps)
      (if p.id = q.id then p = q)

math subtype MUTEX_CORE_MODEL is
  (all_proxies: PROXY_SET,
   waiting_proxies: WAITING_PROXY_SET,
   current_id: integer)
  exemplar mcm
  constraint
    for all p: WAITING_PROXY
        where (p is in mcm.waiting_proxies)
      (p.id is in mcm.all_proxies) and
    if mcm.waiting_proxies = {}
      then mcm.current_id < 0
      else
        there exists p: WAITING_PROXY such that
          (p is in mcm.waiting_proxies and
           mcm.current_id = p.id and
           (for all q: WAITING_PROXY
                where (q is in mcm.waiting_proxies)
              (q.ticket >= p.ticket)))

Mutex_Core is modeled by MUTEX_CORE_MODEL
  initialization ensures
    self = ({}, {}, -1)
```

Listing 3: Operations of abstract Mutex_Core component

```
math definition IS_REQUESTED
    (id: integer, wps: WAITING_PROXY_SET): boolean
  explicit definition
    there exists wp: WAITING_PROXY such that
      ((wp is in wps) and (wp.id = id))

math definition MIN_TICKET
    (wps: WAITING_PROXY_SET): NATURAL_NUMBER
  explicit definition
    min ({0} union {wp: WAITING_PROXY
         where (wp is in wps) (wp.ticket)})

procedure Add_Proxy (replaces id: Integer)
  ensures
    self.waiting_proxies = #self.waiting_proxies and
    self.current_id = #self.current_id and
    id is not in #self.all_proxies and
    self.all_proxies = #self.all_proxies union {id}
```

Listing 4: Operations of abstract Mutex_Core component (contd.)

```
procedure Remove_Proxy (evaluates id: Integer)
  requires
    id is in self.all_proxies and
    not IS_REQUESTED(id, self.waiting_proxies)
  ensures
    self.waiting_proxies = #self.waiting_proxies and
    self.current_id = #self.current_id and
    self.all_proxies = #self.all_proxies - {id}

procedure Request (evaluates id: Integer)
  requires
    id is in self.all_proxies and
    not IS_REQUESTED(id, self.waiting_proxies)
  ensures
    self.all_proxies = #self.all_proxies and
    there exists ticket: NATURAL_NUMBER such that
      ((if #self.waiting_proxies = {}
        then (ticket = 0 and self.current_id = id)
        else (ticket >
                MIN_TICKET(self.waiting_proxies) and
              self.current_id =
                    #self.current_id)) and
       self.waiting_proxies =
         #self.waiting_proxies union
           {(id, ticket)})
  expects self.Release(id)

function Is_Requested (id: Integer): Boolean
  requires
    id is in self.all_proxies
  ensures
    Is_Requested =
      IS_REQUESTED(id, self.waiting_proxies)

procedure Check_If_Available
    (evaluates id: Integer, replaces b: Boolean)
  requires
    id is in self.all_proxies and
    IS_REQUESTED(id, self.waiting_proxies)
  ensures
    self.all_proxies = #self.all_proxies and
    self.current_id = #self.current_id and
    b = there exists wp: WAITING_PROXY such that
        (wp is in #self.waiting_proxies and
           id = wp.id = self.current_id)

procedure Release (preserves id : integer)
  requires
    id is in self.all_proxies and
    IS_REQUESTED(id, self.waiting_proxies)
  ensures
    self.all_proxies = #self.all_proxies and
    there exists wp: WAITING_PROXY such that
      (wp is in #self.waiting_proxies and
       wp.id = id and
       self.waiting_proxies =
         #self.waiting_proxies - {wp}) and
    if self.waiting_proxies = {}
    then self.current_id = -1
    else
      (self.current_id,
       MIN_TICKET (self.waiting_proxies)) is in
         self.waiting_proxies
```

Useful Mathematical Definitions

IS_REQUESTED **(Lines 1—5):** Is the proxy with the given id in the set of waiting proxies?

MIN_TICKET **(Lines 7—11):** The minimum value of ticket in the set of waiting proxies.

Operations

Add_Proxy() **(Lines 13—18):** This operation adds a new proxy to the set of proxies that want to use the resource. The operation ensures the uniqueness of id's in the set of all proxies. It returns the id of the newly created proxy.

Remove_Proxy() **(Lines 1—8):** Given an id, this operation removes the proxy with this id. The proxy can no longer request access to the resource once it is removed.

Request() **(Lines 10—26):** When this operation is invoked by a valid proxy, the proxy is given a ticket, whose value can not be directly observed by any proxy. The proxy's id and the associated ticket are added to the set of waiting proxies. If, at the time of invocation, no other proxy had requested access to the resource, the proxy invoking Request() gets access to the resource, *i.e.*, the current_id becomes this proxy's id. This means that the proxy will be able to access the resource immediately after its first call to Check_If_Available(). If there are pending requests from other proxies at the time of invocation, the ticket assigned is some larger value than MIN_TICKET(waiting_proxies).

Is_Requested() **(Lines 28—33):** This operation returns **true** if the given id belongs to a waiting proxy, and **false** otherwise.

Check_If_Available() **(Lines 35—45):** After requesting access to the resource, a proxy invokes this operation to see if the resource is available to it. The procedure returns with b equal to **true** if the given id is equal to current_id and the waiting proxy with this id has a ticket value equal to MIN_TICKET(waiting_proxies).

Release() **(Lines 47—63):** Once a proxy is done using the resource, it invokes Release(). This removes it from the set of waiting proxies, and results in current_id becoming either -1 (in case there are no other waiting proxies), or the id of some still-waiting proxy whose ticket is equal to MIN_TICKET(waiting_proxies). The next time the proxy whose id equals current_id invokes Check_If_Available(), it will be granted access to the resource.

3.3 Proofs of Implementations

As shown in Figure 1, there are four *implements* relations. Each of these has a set of associated proof obligations to show that the implementations meet the specifications. We focus on the new aspects of these (the *impl(1)* and *impl(2)* relations in Figure 1) in this paper and defer the other two, which will be similar to each other, for future work.

3.3.1 Proof of Client Progress

We notice that the proof of the client implementation will be similar to any sequential component, although it is using a concurrent program. As mentioned in Section 2.1 and Section 2.2, the verification for client implementation is reasonably simple because it is using the sequential specification of the proxy as opposed to that of the core. For example, the client will be able to prove loop termination in the following code snippet:

```
proxy1.Request();
while(not b) {
    proxy1.Check_If_Available(b); }
proxy1.Release();
```

When it calls Request() on proxy1, which is an instance of Mutex_Proxy_1, wait_index assumes the value of some natural number. With every call to Check_If_Available() this number goes down. Eventually, it will hit zero and the client will be able to access the critical section. We also notice that the client is able to prove, based on the termination of the loop, that it satisfies the expects obligation it acquired by calling Request().

3.3.2 Proof of Proxy Implementation

The key part to prove in this proof obligation is that the proxy can meet the ensures clause of Check_If_Available() by using the specification of Mutex_Core.

A sketch of the implementation of the Mutex_Proxy is as follows: In the constructor, it calls Add_Proxy() and gets an *id*. In Request(), it calls Request(id) on Mutex_Core and sets its requested to true. In the Check_If_Available(), it sets b and available to true, if it gets a true answer from the Mutex_Core. In Release(), it calls the corresponding operation of the Mutex_Core and sets both requested and available to false.

In the following lemmas and theorems, we assume that mc is the common instance of a Mutex_Core implementation that all proxies are using. Further, we use MP to denote the Mutex_Proxy type. We also use min_ticket to denote MIN_TICKET(mc.waiting_proxies). Lastly, we use the terms "getting access to the resource" for a proxy synonymously with the event that the associated *id* becomes equal to mc.current_id.

LEMMA 3.1. $\forall$ wp$\in$ mc.waiting_proxies, (wp.ticket-min_ticket,k)*, where* k *is the number of proxies with ticket equal to* min_ticket*, decreases with every call to* mc.Release() *by proxies who get access to the critical section, until* mc.current_id = wp.id.

PROOF. The value of wp.ticket is unchanged once it is assigned by mc.Request(). If |mc.waiting_proxies| = 0 at the time wp calls mc.Request(), it follows from the post-condition of Request() that mc.current_id = wp.id. On the other hand, if at the time of wp's call to mc.Request(), |mc.waiting_proxies| $>$ 0, then (from the constraint in lines 29-35 in Listing 2), as long as |mc.waiting_proxies-wp|$>$ 0, there is some proxy ap such that mc.current_id = ap.id. From the expects clause of mc.Request(), ap eventually calls mc.Release(). With this call to mc.Release(), either min_ticket increases (if there were no other waiting proxies with the same value of ticket as ap.ticket) or k decreases. In either case, the tuple (wp.ticket-min_ticket,k) decreases[1]. The same situation now

[1]We use the standard ordering relation on tuples, where (a, b) $<$ (c, d) iff (a $<$ c) or a = c and b $<$ d.

recurs with some other proxy np taking the place of ap until mc.current_id = wp.id. This completes the proof. □

THEOREM 3.1. ∀ wp ∈ mc.waiting_proxies *eventually* mc.current_id = p.id, *unless* wp *cancels its request by calling* mc.Release() *prematurely.*

PROOF. From Lemma 3.1, (wp.ticket - min_ticket, k) decreases with every call to mc.Release() by proxies who get access to the resource before wp. When the value of (wp.ticket - min_token, k) becomes (0, 0), wp is guaranteed to have mc.current_id = p.id, although it may get access to the resource earlier when min_ticket = 0 and k > 0. □

Theorem 3.1 ensures that every proxy can meet the ensures clause of its Check_If_Available() method. A more formal proof of this claim can be done by using an abstraction relation [12] to relate wait_index to its associated ticket in mc.

3.4 Proof of Safety and Starvation Freedom

The entire mutual exclusion program is making some guarantees to the system designer. These are the safety and progress guarantees [2]. The safety specification says that "at most one client gets access to the critical section at a time." The progress specification says that "every requesting client eventually gets access to the critical section." The system uses the Mutex_Proxy and Mutex_Core components to meet these specifications.

The progress property follows immediately from Theorem 3.1. We prove the safety property below. In the following, we assume that the *id* assigned by the Mutex_Core to a proxy is one of the fields of proxy. We use the definitions and assumptions from Section *3.3.2*.

LEMMA 3.2. *The following invariant holds in the system:* ∀ p ∈ MP *:* p.available⇒ (p.id = mc.current_id).

PROOF. The invariant vacuously holds at initialization since available is false for all proxies. Now, when p.available changes from false to true (in a call to mc.Check_If_Available()) for some proxy p, it is only when p.id = mc.current_id. Therefore, the invariant is preserved by this transition. When the assertion p.id = mc.current_id changes from true to false (in a call to mc.Release()), p.available is set to false, thus preserving the invariant in this transition too. Therefore, the invariant holds. □

THEOREM 3.2. ∀ p ∈ MP : p.available ⇒ (∀ op ∈ MP *:* op ≠ p ⇒ ¬ op.available*).*

PROOF. The theorem holds vacuously if ∀ p ∈ MP : ¬ p.available. So, let us assume ∃ p ∈ MP: p.available. From Lemma 3.2, p.id = mc.current_id. Further, from the constraint on Line 16, ids are unique. Now, the assignment p.id = mc.current_id is done either in mc.Request() or in mc.Release(). If the assignment was done in mc.Request(), then there were no other waiting proxies and p is the unique proxy with available set to true. If the assignment was done in mc.Release(), then the proxy with available set to true previously, had set its available to false at the time that mc.current_id was assigned to p.id. Therefore, in this case also p is the unique proxy with its available set to true. From the uniqueness of ids, no other proxy can get an affirmative answer in a call to mc.Check_If_Available(), as long as p does not call mc.Release(). Therefore, p remains the unique proxy with its available set to true as long as it does not call mc.Release(). When p calls mc.Release(), either there are no more waiting proxies, in which case no proxy has its available set to true, or some waiting proxy q takes the place of p. This completes the proof. □

4. RELATED WORK

Concurrent systems often exhibit reactive behavior, so specification techniques for such systems usually include explicit treatment of both safety and progress properties. Many different compositional approaches have been investigated, including rely-guarantee [13, 1, 8], hypothesis-conclusion [4], and assumption-commitment [5]. All of these techniques address the problem of circularities in proofs in some way, for example by restricting the properties on which a component relies to be safety properties only. In contrast, the expects clause introduced here allows a component to explicitly require a progress property of its client. Circularities are avoided by preventing a client from requiring any progress properties from the components it uses, apart from termination of each called method.

A common way to model concurrency is with nondeterministic interleaving, as in Unity [4] and TLA [9]. Our use of relational specifications to capture the potential effects of concurrently executing threads of execution is certainly not new. The introduction of an intermediary component (the proxy), however, facilitates a solipsistic view on the part of a client, and in this way promotes a novel way to compositionally reason about program behavior.

The Seuss methodology [11] is similar to our proposed approach in that it draws on both concurrent and sequential techniques. Whereas we begin with a sequential framework (RESOLVE) and add elements to address concurrency, Seuss begins with a concurrent framework (similar to Unity) and incorporates program structures such as processes and methods with sequential invocation semantics.

5. CONCLUSION

We proposed an approach to unify the specification and verification of sequential systems with those of concurrent ones. We proposed a specification approach that characterizes concurrent programs as a pair of components — a Proxy and a Core component. The Proxy component uses relational specifications to present a sequential veneer over the Core component, allowing clients of a concurrent program to be verified without concern for concurrency. We also introduced a new expects clause in the contractual specification of operations to formalize the well-behavedness requirements from the clients. We illustrated our approach in the context of the traditional mutual exclusion problem. In our case study, we also demonstrated how the proof obligation associated with the expects clause can be carried out by the clients.

6. OPEN ISSUES

As the work presented in this paper is preliminary, there are a number of avenues for future investigation. We plan to investigate the utility of the expects construct in specifying other RESOLVE components. We also plan to investigate other possible versions of the mathematical structure involved in the operational semantics of the expects clause, i.e. multi-set, string or some other model in place of a set. Further, we plan to investigate what proof obligations a nonterminating program should have with respect to the expects clauses in the components it uses. We would like to point out here that it is tempting to make the structure and semantics of the expects clause very rich, but it is not yet clear whether that will be useful. For example, in our case study of the Mutex component, nested calls to methods with expects clause do not arise, although this may have been the case had we designed our component in a different way.

Another area of investigation is the evaluation of the applicability limits of our approach. A successful application of our approach to the case study of mutual exclusion program in this paper is evidence that our approach can work for conflict resolution programs, also called *competitive systems*, such as dining philosophers [6] and drinking philosophers [3]. Currently, we are in the process of applying this approach to some systems in the other domain of concurrent programming, *cooperative systems.* In particular, we are working on designing components for barrier synchronization and network protocol stack, both of which are representative of cooperative concurrent systems.

In the more distant future, we plan to focus on a proof system for verifying the correctness of concurrent component implementations.

7. ACKNOWLEDGMENTS

We thank the anonymous referees for their helpful insights, and especially for more suggestions for future work than we could elaborate here. We also thank the Reusable Software Research Group (RSRG) at the Ohio State University for their comments and criticisms on this work.

8. REFERENCES

[1] Abadi, M., and Lamport, L. Composing specifications. *TOPLAS 15*, 1 (Jan 1993), 73–132.

[2] Alpern, B., and Schneider, F. B. Defining liveness. *Information Processing Letters 21*, 4 (Oct 1985), 181–185.

[3] Chandy, K. M., and Misra, J. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst. 6*, 4 (1984), 632–646.

[4] Chandy, K. M., and Misra, J. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, USA, 1988.

[5] Collette, P. Composition of assumption-commitment specifications in a UNITY style. *SCP 23* (Dec 1994), 107–125.

[6] Dijkstra, E. W. Hierarchical ordering of sequential processes. *Acta Informatica 1*, 2 (Oct 1971), 115–138.

[7] Edwards, S. H., Heym, W. D., Long, T. J., Sitaraman, M., and Weide, B. W. Specifying Components in RESOLVE. *Software Engineering Notes 19*, 4 (1994), 29–39.

[8] Jones, C. B. Tentative steps toward a development method for interfering programs. *TOPLAS 5*, 4 (1983), 596–619.

[9] Lamport, L. The temporal logic of actions. *TOPLAS 16*, 3 (May 1994), 872–923.

[10] Meyer, B. *Design by contract.* Prentice Hall, 1992, ch. 1.

[11] Misra, J. *A Discipline of Multiprogramming.* Monographs in Computer Science. Springer-Verlag, 2001.

[12] Sitaraman, M., Weide, B. W., and Ogden, W. F. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Transactions on Software Engineering 23*, 3 (1997), 157–170.

[13] Stark, E. W. A proof technique for rely guarantee properties. In *Foundations of software technology and theoretical computer science*, no. 306 in LNCS. Pergamon-Elsevier Science Ltd., 1985, pp. 369–391.

Basic Laws of Object Modeling

Rohit Gheyi [*]
rg@cin.ufpe.br

Tiago Massoni [†]
tlm@cin.ufpe.br

Paulo Borba [‡]
phmb@cin.ufpe.br

Informatics Center
Federal University of Pernambuco
Recife, Brazil

ABSTRACT

Semantics-preserving model transformations are usually proposed in an *ad hoc* way because it is difficult to prove that they are sound with respect to a formal semantics. So, simple mistakes lead to incorrect transformations that might, for example, introduce inconsistencies to a model. In order to avoid that, we propose a set of simple modeling laws (which can be seen as bi-directional transformations) that can be used to safely derive more complex semantics-preserving transformations, such as refactorings which are useful, for example, to introduce design patterns into a model. Our laws are specific for Alloy, a formal object-oriented modeling language, but they can be leveraged to other object modeling notations. We illustrate their applicability by formally refactoring Alloy models with subtypes in order to improve the analysis performed by the Alloy Analyzer tool.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal Methods, Model Checking; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification

General Terms

Design, Verification

Keywords

Model Transformations, Refactorings, Formal Methods, Verification

1. INTRODUCTION

Laws of programming [14] are important not only to define the axiomatic semantics of programming languages but also to assist in the software development process. In fact, these laws can be used as the foundation for informal development practices, such as refactorings [7], widely adopted due to modern methodologies, in particular Extreme Programming [1].

Modeling laws might bring similar benefits, such as refactoring models, but with a greater impact on cost and productivity, since they are used in earlier stages of the software development process. However, semantics-preserving model transformations are usually proposed in an *ad hoc* way because it is difficult to prove that they are sound with respect to a formal semantics. So, simple mistakes lead to incorrect transformations that might, for example, introduce inconsistencies to a model.

In this paper, we propose a set of modeling laws [9] for Alloy [15], a formal object-oriented modeling language. Each law proposed here defines two small-grained model transformations that preserve semantics. We proved their soundness based on a denotational semantics for Alloy [9]. We regard them as basic because they are simple and able to derive more complex transformations. This set can be considered comprehensive, if compared to what have been proposed so far [6, 12, 18]. In addition, we propose an equivalence notion for Alloy models.

These laws can be useful to refactor Alloy models with subtypes in order to improve the analysis performed by the Alloy Analyzer tool. Moreover, these laws can be applied to derive refactorings [9], which are useful, for instance to introduce design patterns [8] into a model. Additionally, they can be applied to reason whether one component, annotated with Alloy, can be reused or substituted by another. Our laws can also be used for educational purposes in object modeling, since they clarify the meaning of a number of important constructs.

Related work [6, 18, 12] has proposed transformations for Unified Modeling Language (UML) [2] class diagrams. However, they do not offer a comprehensive set of modeling laws that can derive more complex transformations. In addition, some of them do not completely preserve semantics. We proposed laws for Alloy, rather than UML and the Object Constraint Language (OCL) [17], due to Alloy's simpler semantics, although expressive enough to model a broad variety of applications. Nevertheless, our laws can also be useful for reasoning about UML class diagrams, by provid-

[*]Supported by CNPq.
[†]Supported by CAPES.
[‡]Partially supported by CNPq (521994/96-9).

ing a semantics for UML class diagrams based on Alloy [20]. Similarly, the results can also be leveraged to other object modeling notations.

The remainder of this paper is organized as follows. Section 2 overviews the Alloy language. Section 3 presents some basic laws for Alloy. In Section 4, we show applications for the laws. The following section discusses some related work. Finally, Section 6 concludes the paper.

2. ALLOY

Alloy is formal object-oriented specification language that is strongly typed and assumes a universe of elements partitioned into subsets, each of which associated with a basic type. Alloy can be used for specifying, verifying and validating properties about object and component-based systems.

An Alloy model or specification is a sequence of *paragraphs* of two kinds: signatures that are used for defining new types, and formula paragraphs, namely facts and functions, used to record constraints. Each signature contains a set of objects (elements). These objects can be related by the relations declared in the signatures. A signature paragraph introduces a basic type and a collection of relations, called fields, along with the types of the fields and other constraints on the values they relate. Besides subtyping with signature extension, Alloy includes other important structures and operators such as modules, predicates and commands for analyzing the specification, which are discussed elsewhere [15].

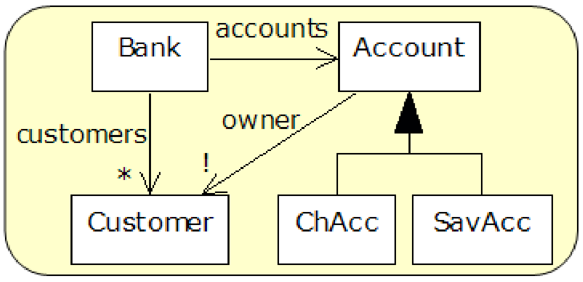


Figure 1: Bank System Object Model

Suppose that we want to model part of a banking system in Alloy, on which each bank contains a set of accounts and a set of customers. Each account is owned by a customer. Also, accounts may be checking or savings. Figure 1 describes the object model [19] of the system. Each box in an object model represents a set of objects. The arrows are relations and indicate how objects of a set are related to objects in other sets. For instance, the arrow labeled `owner` shows that each object from `Account` has a field whose value object is a `Customer` object.

An arrow with a closed head form, such as from `ChAcc` to `Account`, denotes a subset relationship. In this case, `ChAcc` is a subset of `Account`. If two subsets share an arrow, they are disjoint. For instance, `ChAcc` and `SavAcc` are disjoint. If the arrowhead is filled, the subsets exhaust the superset, so there are no members of the superset that are not members of one of the subsets. In the banking system, the subsets form a *partition*: every member of the superset belongs to exactly one subset.

The multiplicity symbols are as follows: `!` (exactly one), `?` (zero or one), `*` (zero or more) and `+` (one or more). Multiplicity symbols can appear on both ends of the arrow. If a multiplicity symbol is omitted, `*` is assumed. The following fragment introduces three signatures and three relations modeling part of the banking system.

```
sig Bank {
  accounts: set Account,
  customers: set Customer
}
sig Customer {}
sig Account {
  owner: set Customer
}
```

In the field declaration of `Bank`, the `set` relation qualifier specifies that `accounts` maps each element in `Bank` to a set of elements in `Account`. When we omit the keyword, we specify a total function.

In Alloy, one signature can extend another, establishing that the extended signature is a subset of the parent signature. For instance, the values given to `ChAcc` is a subset of the values given to `Account`.

```
sig ChAcc, SavAcc extends Account {}
```

Signature extension introduces a subtype in Alloy version 3, establishing that each subsignature is disjoint. In this case, `ChAcc` and `SavAcc` are disjoint. In Alloy, we can declare several signatures at once if they do not declare any relation, as showed in the previous fragment.

Facts are formula paragraphs. They are used to package formulae that always hold, such as invariants about the elements. The following example introduces a fact named `BankConstraints`, establishing general properties about the previously introduced signatures.

```
fact BankConstraints {
  all acc: Account | one acc.owner
  Account = ChAcc + SavAcc
}
```

The first formula states that each account is owned by exactly one customer (the . operator can be seen as relational dereference), while the last one states that each account is a checking or savings account. The keyword `all` represents the universal quantifier. The `one` keyword, when applied to an expression, denotes that the expression has exactly one element. The operator `+` corresponds to the union set operator. In Alloy, the fact formulae are implicitly declared as a conjunction.

3. BASIC LAWS

In this section, we present a set of basic laws proposed for Alloy. These laws state properties about signatures, relations, facts and formulae. With a comprehensive set of simple basic laws [9], we aim to provide powerful guidance on the derivation of complex transformations, such as refactorings and optimizations. The models on the left and the right side of each law have the same meaning, since each law preserves semantics, as described elsewhere [9].

For instance, the laws can be useful to transform the model shown in Figure 2, which shows a transformation introducing a collection between `Bank` and `Account`. We establish that these models have the same semantics considering a set of signature and relation names that we call alphabet (Σ). The alphabet includes the element names which are considered to be relevant in a model. For instance, suppose that Σ contains the `Bank`, `Account` and `accounts` names. If these models have the same set of values to all names in the alphabet, they are equivalent under this equivalence notion. The other names (`col`, `Vector` and `elems`) are auxiliary, thus not taken into consideration.

However, some models may not have all names considered in the alphabet. For instance, in Figure 2, `accounts` does not belong to the right-hand side model. In this case, relevant names must be represented by others names. Hence, we define a view (v), consisting of a set of items such as $n \rightarrow exp$, where n is a name and exp is an expression not containing n, and they have the same type. In the example of Figure 2, we can choose a v containing the following item: $accounts \rightarrow col.elems$. Now we can compare both models. Notice that `accounts` is defined in terms of two names that belong to the right-hand side model; hence we can compute the `accounts`'s value. So, a view allows a strategy for representing relevant names using an equivalent combination of other elements. There are some constraints when choosing a view, such as the items cannot be recursive. This is a general idea of the equivalence notion considered for our basic laws and it is described in more detail elsewhere [11].

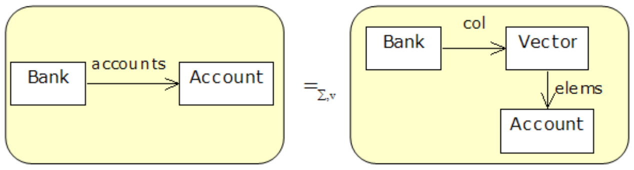


Figure 2: Equivalence Notion

In the proposed laws, we used *ps* to denote a set of signature and fact paragraphs. We do not consider other Alloy paragraphs since some of them are syntactic and others are used to perform analysis in the Alloy Analyzer.

3.1 Laws for Signatures

The first law states that we can introduce a generalization between two signatures when the name of the new parent signature is not previously used in the specification. We can also remove a generalization between them if the parent signature, the relations of its family and the subsignatures are not being used elsewhere. This proviso guarantees that there is no formula containing S, T and their relations, except the two formulae stating the partition. Since S and T have different types after the generalization removal, this proviso assures that the generalization removal does not introduce type errors.

Law 1. ⟨introduce generalization⟩

```
ps
sig S {
  rs
}
sig T {
  rs'
}
fact F {
  forms
}
```

$=_{\Sigma,v}$

```
ps
sig U {}
sig S extends U {
  rs
}
sig T extends U {
  rs'
}
fact F {
  forms
  U = S + T
}
```

provided
($\leftrightarrow$) if U belongs to Σ then v contains the $U \rightarrow S + T$ item;
($\rightarrow$) *ps* does not declare any paragraph named U;
($\leftarrow$) U and the relations declared by its family, S and T do not appear in *ps*, *rs*, *rs'* and *forms*.

We write ($\rightarrow$), before the proviso, to indicate that this proviso is required when applying this law from left to right. Similarly, we use ($\leftarrow$) to indicate what is required when applying the law in the opposite direction, and we use ($\leftrightarrow$) to indicate that the proviso is necessary in both directions. It is important to notice that each basic law, when applied in any direction, defines one transformation that preserves semantics.

Notice that both models of the law have the same names and constraints, except U, its definition and the implicit constraints stating that S and T are subset of U. Since the view has an item for U that is equivalent to its definition and it contains the union of the values given to S and T, the left side model yields the same value for U of right side model; hence the law preserves semantics.

The operator `&` denotes the intersection set operator. The keyword `no` when applied to an expression denotes that the expression has no elements. We write *forms* and *rs* to denote a set of formulae and a set of relation declarations, respectively. We also propose trivial laws allowing us to introduce empty signatures [9].

3.2 Laws for Relations

Besides laws for dealing with signatures, we also define laws for manipulating relations. The next law states that we can introduce a new relation along with its definition, which is a formula of the form $r = exp$, establishing a value for the relation. We can also remove a relation that is not being used.

Law 2. ⟨introduce relation and its definition⟩

```
ps
sig S {
  rs
}
fact F {
  forms
}
```

$=_{\Sigma,v}$

```
ps
sig S {
  rs,
  r : set T
}
fact F {
  forms
  r=exp
}
```

provided
(↔) if r belongs to Σ, r does not appear in *exp* and v contains the $r \rightarrow exp$ item;
(→) The family of S in *ps* does not declare any relation named r; T is a signature name declared that does not extend other signatures;
(←) The r relation does not appear in *ps* and *forms*.

The *exp* expression can be either r or an expression having the same type of r and not containing r. It is important to stress that the previous law can be used to simply introduce a relation, without any definition. We have just to take *exp* as being r itself, introducing a tautology. Moreover, constraints involving Σ and v must be carefully introduced. When introducing or removing a relation in Σ, we must guarantee that the $r \rightarrow exp$ item belongs to v and r does not appear in *exp* in order to avoid a recursive definition in v. The family of a signature is the set of all signatures that extend or are extended by it direct or indirectly. Alloy does not allow two relations with the same name in the same family.

A relation qualified as a `set` of T, declared in the S signature, indicates that every element in S relates to any number of T elements. Since it does not impose any constraint on the relation, we ensure that the previous law preserves the constraints, not introducing inconsistency. In contrast, due to its constraint, we cannot always introduce a relation declared with the `one` (stating a total function) qualifier since this constraint can contradict previous specification constraints. For instance, the introduction of the r relation with `one` in the previous law can introduce an inconsistency if there are constraints stating that T is empty and S has at least one element. After applying this transformation, r must relate every element of S to one element of T. However, S is not empty and T is empty, introducing an inconsistency.

Notice that both models of the law have the same names and constraints, except r and its definition. Since the view has an item for r that is equivalent to its definition, the left side model yields the same value for r of right side model; hence the law preserves semantics.

Next, we establish a law for pulling up relations. We can pull up a relation from a signature to its parent by adding a formula stating that this relation only maps elements of the subsignature. Similarly, we can push down a relation if the specification has a formula stating that the relation only relates elements of the subsignature.

Law 3. ⟨pull up relation⟩

```
ps
sig T {
  rs
}
sig S extends T {
  rs',
  r : set U
}
fact F {
  forms
}
```

$=_{\Sigma,v}$

```
ps
sig T {
  rs,
  r : set U
}
sig S extends T {
  rs'
}
fact F {
  forms
  no (T − S).r
}
```

The operator `-` corresponds to the difference set operator. Notice that the values given to r, which is pulled up or down, are the only values that are subject to change. On the left side of the law, r relates elements from S to U. On the right side of the law, r relates elements from T to U. However, there is an explicit constraint indicating that r relates elements from S to U. Therefore, both modes have the same meaning. We have proposed other laws for relations, such as splitting a relation [9].

3.3 Laws for Facts and Formulae

Besides proposing some trivial laws for facts and formulae, we proposed a law establishing that we can add or remove a formula from a fact, as long as it can be deduced from other formulae in the specification.

Law 4. ⟨introduce formula⟩

```
ps
fact F {
  forms
}
```

$=_{\Sigma,v}$

```
ps
fact F {
  forms
  f
}
```

provided

(↔) The formula f can be deduced from the formulae in *ps* and *forms*.

Since f is derived from other formulae, we guarantee that both specifications have the same meaning. The constraints imposed by this formula are already imposed by the others. From predicate calculus, we infer `'P and Q'` from the `'P => Q'` and `'P'` formulae, where `P` and `Q` are arbitrary predicates. Therefore, this law is trivially valid. The laws presented here focus on Alloy structures, although relational [24] and predicate [22] calculi can also be applied to Alloy formulae.

Besides these laws, we proposed laws for syntactic sugar constructs [9]. We prove these laws using a denotational semantics for Alloy [9]. We aim at proposing simple small-grained transformations because it is easier to prove that they are semantics-preserving. Although they are simple, we can derive a number of complex large-grained transformations by composing them, which consequently preserve semantics. Examples of the use of the laws can be found in Section 4.

4. APPLICATIONS

The basic laws provide an axiomatic semantics for Alloy, clarifying the meaning of its constructs. In this section, we describe how we can use them to transform Alloy models with subtypes in order to improve the analysis performance by the Alloy Analyzer. As previously illustrated [5], analysis performance of Alloy models with subtypes can be increased by *atomization*. When performing analysis, the Alloy Analyzer internally transforms (atomizes) a model by pushing relations down to the lowest subtype level in order to improve its performance. Atomization applies a number of model transformations to remove a relation in a parent signature and introduce one relation to each subsignature. However, some transformations are not proved to be semantics-preserving [5], such as introducing and removing relations. Other transformations, such as deducing formulae, are semantics-preserving considering they are derived from relational calculus laws. Next, we show how the proposed laws, which are sound with respect to an Alloy denotational semantics, can formalize the atomization scheme.

Consider an addition to the signature `Account`, described in Section 2, which is partitioned by `ChAcc` and `SavAcc`. Each account relates to a customer by the `owner` relation. In order to improve the performance analysis, the atomization scheme removes `owner` from `Account` while introducing relations in `ChAcc` and `SavAcc`, called `chOwner` and `savOwner`, respectively.

Suppose that we consider an alphabet Σ containing all names declared in the initial specification, and a view v having the $owner \rightarrow chOwner + savOwner$ item. First of all, we can introduce the new relations into `Account` and their definitions by applying Law 2 from left to right. Since both relations do not belong to Σ, no item is needed to v. Next, we show the resulting specification. We consider that `ps` contains the `Bank` and `Customer` signatures.

```
ps
sig Account {
  owner: set Customer,
  chOwner: set Customer,
  savOwner: set Customer
}
sig ChAcc extends Account {}
sig SavAcc extends Account {}

fact BankConstraints {
  all acc: Account | one acc.owner
  Account = ChAcc + SavAcc
  chOwner = owner & (ChAcc->Customer)
  savOwner = owner & (SavAcc->Customer)
}
```

The notation `->` represents the product operator that combines every element in the left operand with every element in the right operand. When applied to sets, this operator represents the standard Cartesian product.

Our aim is to pull down `chOwner` and `savOwner` to `ChAccount` and `SavAccount`, respectively. In order to do that, we first derive the `no (Account - ChAcc).chOwner` formula, by applying some relational calculus properties, from the `chOwner` definition. Similarly, we can derive a formula for `savOwner`, and introduce both formulae, by applying Law 4 from left to right, which results in the following specification.

```
ps
sig Account {
  owner: set Customer,
  chOwner: set Customer,
  savOwner: set Customer
}
sig ChAcc extends Account {}
sig SavAcc extends Account {}

fact BankConstraints {
  all acc: Account | one acc.owner
  Account = ChAcc + SavAcc
  chOwner = owner & (ChAcc->Customer)
  savOwner = owner & (SavAcc->Customer)
  no (Account - ChAcc).chOwner
  no (Account - SavAcc).savOwner
}
```

Next we apply Law 3 from right to left and pull down both relations, as shown next.

```
ps
sig Account {
  owner: set Customer
}
sig ChAcc extends Account {
  chOwner: set Customer
}
sig SavAcc extends Account {
  savOwner: set Customer
}

fact BankConstraints {
  all acc: Account | one acc.owner
  Account = ChAcc + SavAcc
  chOwner = owner & (ChAcc->Customer)
  savOwner = owner & (SavAcc->Customer)
}
```

Our aim is to derive the `owner` definition in order to replace it by its definition and eventually remove it from the specification. Applying some predicate and relational calculus properties, which are within brackets to justify every step in the derivation, we deduce that:

```
(chOwner + savOwner =
  owner & (ChAcc->Customer) +
  owner & (SavAcc->Customer))
  [(P&Q + P&R) = (P&(Q+R))] =
(chOwner + savOwner =
  owner & ((ChAcc->Customer) + (SavAcc->Customer)))
  [(P->R) + (Q->R) = (P+Q)->R)] =
(chOwner + savOwner =
  owner & ((ChAcc+SavAcc)->Customer))
  [Account = ChAcc + SavAcc] =
(chOwner + savOwner = owner & (Account->Customer)) =
(owner = chOwner + savOwner)
```

Since this formula is deduced from formulae in the specification, using Law 4 from left to right, we can introduce this formula in the specification. After that, we can replace `owner` by its definition. It is important to notice that from every formula containing `owner`, except its definition, we can derive a new formula replacing `owner` by its definition, which is inserted to the specification applying Law 4 from left to right. Consequently, these new formulae can also derive the formulae with `owner`. Next, we can remove all formulae that contain `owner`, except its definition, from the specification by applying Law 4 from right to left.

Finally, since `owner` does not appear in the model, except in its definition, we can remove this relation and its definition using Law 2 from right to left. Since `owner` belongs to Σ, we have to check whether v has the $owner \rightarrow chOwner + savOwner$ item. Moreover, the third and fourth formula of `BankConstraints` can be deduced from the model. Therefore, we can remove them from the specification applying Law 4 from right to left. The final specification is described next.

```
ps
sig Account {}
sig ChAcc extends Account {
  chOwner: set Customer
}
sig SavAcc extends Account {
  savOwner: set Customer
}

fact BankConstraints {
  all acc: Account | one acc.(chOwner + savOwner)
  Account = ChAcc + SavAcc
}
```

Notice that our laws deal with equivalent models; hence the atomization process can be reversed, similarly. This process can be generalized and we can state the atomization semantics-preserving transformation similarly to the laws. Figure 3 summarizes the order of the application of the laws. Each box has the direction and number of the law to be applied. The filled arrow denotes the next law to be applied. Some boxes have filled arrows on top of it indicating that this law can be applied repeatedly.

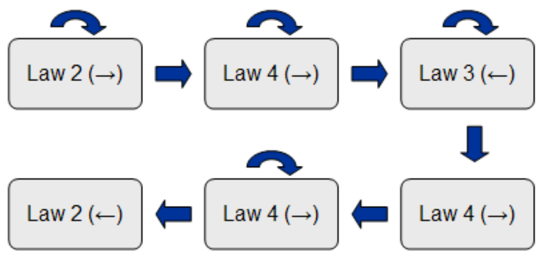


Figure 3: Atomization

In the banking system, every account is a checking or savings account. The atomization process also considers parent signatures that are not partitioned by the subsignatures. In this case, we have to create a new subsignature, extending the parent signature. We do not have a law that allows us to introduce a subsignature in a parent signature already declared. We regard this law as future work.

The basic laws proposed here can also be useful to refactor models [10, 9]. For instance, we refactored a simple but non-trivial Java types specification. Applying the laws, a model describing Java types in terms of subtyping relations can be transformed into another in terms of supertypes, having the same semantics.

We can also use our laws to derive complex large-grained transformations (refactorings) by composing them [9]. Since these refactorings are derived using semantics-preserving laws, they also preserve semantics. We have derived large-grained transformations such as the extract interface refactoring, introducing a collection, and move and reverse a relation [9]. These refactorings can be useful, for example, to introduce design patterns [8] into a model. However, we will not show these here due to the lack of space. Furthermore, by using the laws, we can verify whether two models have the same meaning.

5. RELATED WORK

Zaremski and Wing [25] determines whether two software components are related by a specification matching process. This can be useful, among other things, to reuse components and substitute one component by another without affecting the observable behaviour. To verify whether the specification of one component matches the other, the authors use a theorem prover. We believe that our set of laws can be useful in this case. Suppose that each Java component is annotated with Alloy, similarly as described elsewhere [16]. Since we already proved that our laws are semantics-preserving [9], applying the laws, we just have to check syntactically whether one specification component is equivalent to another, instead of proving it. However, since we do not prove that our set of laws is complete, we may not use them always.

Related work [23, 6, 12, 18] has been carried out on transformation of UML class diagrams. They do not state in which conditions a transformation can be applied. Therefore, some transformations do not preserve semantics in some situations. For instance, creating a generalization between classes not always preserve semantics (Figure 4). Given the constraints in a specification, it can become inconsistent by introducing a generalization. For instance, we cannot declare the `S` class to extend the `T` class when a explicit constraint in the specification states that `S` has more elements than `T`. The introduction of a generalization in this case makes the specification inconsistent, since the generalization constrains `T` to include `S`. Therefore, we can deduce that `T` has the same number or more elements than `S`, which contradicts the explicit constraint in the specification. We cannot apply Law 1 to introduce a generalization in Figure 4, since `T` is already declared.

These transformations do not preserve semantics because some of them use a semi-formal UML semantics. Others partly define a semantics for UML but do not verify soundness of the transformations, or do not consider OCL constraints. We conclude that it is important to prove the soundness of the transformations, in order to guarantee that a transformation preserves semantics. It is easy to make a small change in a model and make it inconsistent.

A similar work proposes basic laws for Refinement Object-

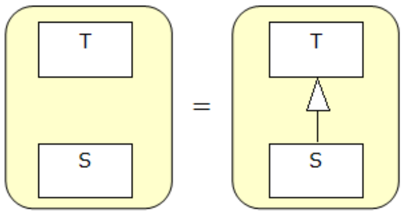


Figure 4: Introduce Generalization

Oriented Language (ROOL) [3]. ROOL is less powerful for specifying structural properties among types compared to Alloy. Whereas ROOL supports only attribute declarations, as in Java [13], Alloy supports the declaration of bidirectional relations with arbitrary arities and multiplicities, as in UML. Another difference is that we cannot define global constraints in ROOL. This related work is similar to ours in the sense that they propose basic laws that are used not only for giving the axiomatic semantics of the language, but also for deriving refactorings.

Laws for top level design elements of UML-RT (Real Time) [4] have also been proposed [21]. Our laws do not deal with refinements, as theirs. Moreover, their work does not intend to propose basic laws, as ours. They propose laws not only for structural constructs, as our laws, but also laws for behavioural constructs, such as laws for capsules. They assume that relationships are directed and predicates involve only relationships as attributes, as in ROOL. Additionally, the authors consider implementation-oriented models. Moreover, their proposed laws rely on the absence of global constraints on the model, such as those involving cardinality (number of instances) of classes in the entire system. Our laws also work for models containing global constraints.

6. CONCLUSIONS

In this paper, we propose basic laws for Alloy and show how they can be used for deriving complex transformations, such as refactorings and optimizations. In contrast to model transformations usually defined in an *ad hoc* way, these laws describe semantics-preserving transformations. Additionally, we propose an equivalence notion for Alloy models.

The laws presented here have been proven sound with respect to a formal semantics for Alloy [9]. Consequently, they should act as a tool for carrying out model transformations. One immediate application of the basic laws is to define an interface from which one can derive more complex transformations, as illustrated in Section 4, and to refactor specifications [10]. Although our laws are specific to Alloy, they can be leveraged to object modeling in general. For instance, we can leverage them to UML class diagrams giving a precise semantics for it in Alloy [20].

All basic laws are very simple to apply since their preconditions are simple syntactic conditions. Nevertheless, these laws can be used as powerful guidance for deriving complex transformations. The law for introducing a formula that is deduced from the model also has syntactic conditions, if we consider relational and predicate calculi. We extended the Alloy Analyzer tool to include the implementation of a number of the basic laws, in such a way that the user does not need to verify the preconditions and apply the laws [9]. The user is only required to inform the parameter values for the transformations. Furthermore, our laws can be used for educational purposes in object modeling, since they clarify the meaning of a number of important constructs. Additionally, they could be useful to verify, with syntactic conditions, whether the specification of one component is equivalent to another. In case they are equivalent, we can substitute a component by another.

Although we have a comprehensive set, relative to what have been proposed so far, of basic laws for Alloy, we still need to prove a reduction theorem stating that our set of laws is complete, in the sense of allowing reduction of arbitrary Alloy specifications to a normal form. We need more laws such as for introducing an empty subsignature. This normal form is expressed in a small subset of the language operators, following approaches adopted for ROOL [3] and imperative languages [14], among others. We also intend to study and formalize the relationship between modeling and programming laws. In particular, we need to investigate whether model refactorings have corresponding program refactorings. This might be useful for implementing tools that apply model and code refactorings in a synchronized way.

7. ACKNOWLEDGMENTS

This work benefited from the discussions during the Software Productivity Group's meetings. In addition, we would like to thank all the anonymous referees, whose appropriate comments helped improving the paper. This work was partially funded by CAPES and CNPq, grant 521994/96-9.

8. REFERENCES

[1] K. Beck. *Extreme Programming Explained.* Addison-Wesley, 2000.

[2] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[3] P. Borba, A. Sampaio, and M. Cornélio. A refinement algebra for object-oriented programming. In *17th European Conference on Object-Oriented Programming, ECOOP'03*, pages 457–482, Darmstadt, Germany, 2003.

[4] B. Douglass. *Real Time UML - Developing Eficient Objects for Embedded Systems.* Addison-Wesley, 1998.

[5] J. Edwards, D. Jackson, E. Torlak, and V. Yeung. Faster constraint solving with subtypes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 232–242. ACM Press, 2004.

[6] A. Evans. Reasoning with UML class diagrams. In *Second IEEE Workshop on Industrial Strength Formal Specification Techniques, WIFT'98, Boca Raton/FL, USA*, pages 102–113. IEEE CS Press, 1998.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[9] R. Gheyi. Basic laws of object modeling. Master's thesis, Federal University of Pernambuco, 2004.

[10] R. Gheyi and P. Borba. Refactoring alloy specifications. In A. Cavalcanti and P. Machado, editors, *Electronic Notes in Theoretical Computer Science, Proceedings of the Brazilian Workshop on Formal Methods*, volume 95, pages 227–243. Elsevier, 2004.

[11] R. Gheyi, T. Massoni, and P. Borba. An equivalence notion of object models. Technical Report, 2004.

[12] M. Gogolla and M. Richters. Equivalence rules for UML class diagrams. In *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France*, pages 87–96, 1998.

[13] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[14] C. Hoare, J. Spivey, I. Hayes, J. He, C. Morgan, A. Roscoe, J. Sanders, I. Sorenson, and B. Sufrin. Laws of programming. *Communications of the Association for Computing Machinery*, 30(8):672–686, 1987.

[15] D. Jackson. Alloy 3.0 reference manual. At http://alloy.mit.edu/beta/reference-manual.pdf, 2004.

[16] S. Khurshid, D. Marinov, and D. Jackson. An Analyzable Annotation Language. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 231–245. ACM Press, 2002.

[17] A. Kleppe and J. Warmer. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1999.

[18] K. Lano and J. Bicarregui. Semantics and transformations for UML models. In *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 97–106, 1998.

[19] B. Liskov and J. Guttag. *Program Development in Java.* Addison-Wesley, 2001.

[20] T. Massoni, R. Gheyi, and P. Borba. A UML class diagram analyzer. In *Third International Workshop on Critical Systems Development with UML (CSDUML), affiliated with UML Conference*, Lisbon, Portugal, 2004.

[21] A. Sampaio, A. Mota, and R. Ramos. Class and capsule refinement in UML for real time. In A. Cavalcanti and P. Machado, editors, *Electronic Notes in Theoretical Computer Science, Proceedings of the Brazilian Workshop on Formal Methods*, volume 95, pages 23–51. Elsevier, 2004.

[22] J. Spivey. *The Z Notation: A Reference Manual.* C. A. R. Hoare Series Editor. Prentice Hall, 1989.

[23] G. Sunyé, D. Pollet, Y. Traon, and J.-M. Jézéquel. Refactoring UML models. In *The Unified Modeling Language, UML'01 - Modeling Languages, Concepts, and Tools. Fourth International Conference, Toronto, Canada*, volume 2185 of *LNCS*, pages 134–148. Springer-Verlag, 2001.

[24] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(9):73–89, 1941.

[25] A. Zaremski and J. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.

Selective Open Recursion: Modular Reasoning about Components and Inheritance

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
jonathan.aldrich@cs.cmu.edu

Kevin Donnelly
Computer Science Department
Boston University
111 Cummington Street
Boston, MA 02215, USA
kevind@bu.edu

ABSTRACT

Current component-based systems with inheritance do not fully protect the implementation details of a class from its subclasses, making it difficult to evolve that implementation without breaking subclass code. Previous solutions to the so-called fragile base class problem *specify* those implementation dependencies, but do not *hide* implementation details in a way that allows effective software evolution.

In this paper, we show that one instance of the fragile base class problem arises because current object-oriented languages dispatch methods using *open recursion* semantics even when these semantics are not needed or wanted. Our solution, called *Selective Open Recursion*, makes explicit the methods to which open recursion should apply. As a result, classes can be more loosely coupled from their subclasses, and therefore can be evolved more easily without breaking subclass code.

We have implemented Selective Open Recursion as an extension to Java, along with an analysis that automatically infers the necessary program annotations. We have collected data for the Java standard library suggesting that the additional programmer effort required by our proposal is low, and that Selective Open Recursion aids in automated reasoning such as compiler optimizations.

1. Inheritance and Information Hiding

In his seminal paper, Parnas laid out the classic theory of information hiding: developers should break a system into modules or components in order to hide information that is likely to change [10]. Thus if change is anticipated with reasonable accuracy, the system can be evolved with local rather than global system modifications, easing many software maintenance tasks. Furthermore, the correctness of each component can be verified in isolation from other components, allowing developers to work independently on different sub-problems.

Unfortunately, developers do not always respect the information hiding boundaries of components—it is often tempting to reach across the boundary for temporary convenience, while causing more serious long-term evolution problems. Thus, encapsulation mechanisms such as Java's packages and public/private data members were developed to give programmers compiler support for enforcing information hiding boundaries.

While the encapsulation mechanisms provided by Java and other languages can help to enforce information hiding boundaries between an object and its clients, enforcing information hiding between a class and its subclasses is more challenging. The `private` modifier can be used to hide some method and fields from subclasses. However, inheritance creates a tight coupling between a class and its subclasses, making it difficult to hide information about the implementation of `public` and `protected` methods in the superclass. In component-based systems with inheritance, it is easy for a subclass to become unintentionally dependent on the implementation details of its superclass, and therefore to break when the superclass changes in seemingly innocuous ways.

```
public class CountingSet extends Set {
  private int count;

  public void add(Object o) {
    super.add(o);
    count++;
  }
  public void addAll(Collection c) {
    super.addAll(c);
    count += c.size();
  }
  public int size() {
    return count;
  }
}
```

Figure 1: The correctness of the `CountingSet` class depends on the independence of `add` and `addAll` in the implementation of `Set`. If the implementation is changed so that `addAll` uses `add`, then `count` will be incremented twice for each element added.

1.1 The Fragile Base Class Problem

This issue is known as the *Fragile Base Class Problem*, one of the most significant challenges faced by designers of object-oriented component libraries. Figure 1 shows an example of the fragile base class problem, taken from the literature [12, 9, 2].

In the example, the `Set` class has been extended with an optimization that keeps track of the current size of the set in an additional variable. Whenever a new element or collection of elements is added to the set, the variable is updated appropriately.

Unfortunately, the implementation of CountingSet makes assumptions about the implementation details of Set—in particular, it assumes that Set does not implement addAll in terms of add. This coupling means that the implementation of Set cannot be changed without breaking its subclasses. For example, if Set was changed so that the addAll method calls add for each member of the collection in the argument, the count variable will be updated not only during the call to addAll, but also for each individual add operation—and thus it will end up being incremented twice for each element in the collection.

The root of the problem described above (which is just one instance of the larger fragile base class problem) is that the subclass depends on the calling patterns between methods in its superclass. Object-oriented languages provide *open recursion*, in which self-calls are dynamically dispatched, allowing subclasses to intercept self-calls from the superclass and thus depend on when it makes these calls. Open recursion is useful for many object-oriented programming idioms—for example, the template method design pattern [7] uses open recursion to invoke customized code provided by a subclass. However, sometimes making a self-call to the current object is just an implementation convenience, not a semantic requirement. The whole point of encapsulation is to ensure that subclasses do not depend on such implementation details, so that a class and its subclasses can be evolved independently. Thus inheritance breaks encapsulation when implementation-specific self-calls are made.

Examples like these have led some to call inheritance anti-modular. Most practitioners, recognizing the value of inheritance for achieving software reuse in component-based systems, would not go so far, but this example illustrates that reasoning about correctness is challenging in the presence of inheritance.

A number of previous papers have addressed the fragile base class problem in various ways [8, 12, 13, 2, 11]. These solutions, however, either give up the power of open recursion entirely, or expose details of a class's implementation that ought to be private (such as the fact that the addAll method calls or does not call add in the example above).

1.2 Contributions

The contribution of this paper is Selective Open Recursion, a new approach that provides the benefits of inheritance and open recursion where they are needed, but allows programmers to effectively hide many details of the way a class is implemented. In our system, described in the next section, method calls on the current object this are dispatched statically by default, meaning that subclasses cannot intercept internal calls and thus cannot become dependent on those implementation details. External calls to the methods of an object—i.e., any method call not explicitly invoked on this—are dynamically dispatched as usual.

If an engineer needs open recursion, she can declare a method "open," in which case self-calls to that method are dispatched dynamically. By declaring a method "open," the author of a class is promising that any changes to the class will preserve the ways in which that method is called.

In section 3, we describe our implementation of Selective Open Recursion as an extension to Java. We have implemented a static, whole-program analysis that annotates an existing Java program with the minimal set of "open" declarations that are necessary so that the program has the same semantics in our system. Results of applying our analysis to the JDK 1.4 standard library show that open annotations are rarely needed and that Selective Open Recursion increases the potential for program optimizations such as inlining. Section 4 discusses related work and section 5 concludes.

2. Selective Open Recursion

We argue that the issue underlying the instance of the fragile base class problem described above is that current languages do not allow programmers to adequately express the intent of various methods in a class. There is an important distinction between methods used for communication between a class and its clients, vs. methods used for communication between a class and its subclasses.

Some methods are specifically intended as callbacks or extension points for subclasses. These methods are invoked recursively by a class so that its subclasses can provide customized behavior. Examples of callback methods include methods denoting events in a user interface, as well as abstract "hook" methods in the template method design pattern [7]. Because callback methods are intended to be invoked whenever some semantic event occurs, any changes to the base class must maintain the invariant that the method is always invoked in a consistent way.

In contrast, many accessor and mutator functions are primarily intended for use by clients. If the implementation of a class also uses these functions, it is typically as a convenience, *not* because the class expects subclasses to override the function with customized behavior. The fragile base class problem described above occurs exactly when a "client-oriented" method is called recursively by a superclass, but is also overridden by a subclass.[1] Because the recursive call to the method was never intended to be part of the subclassing interface, the maintainer of the base class should be able to evolve the class to use (or not use) such methods without affecting subclasses.

The key insight underlying Selective Open Recursion is that subclasses do not need to intercept recursive calls to methods that were not intended as callbacks or extension points—they can always provide their behavior by overriding the external interface of a class. At most, intercepting recursive calls to "client-oriented" methods is only a minor convenience, and one that creates an undesirable coupling between subclass and superclasses.

We thus propose to add a new modifier, open, which allows developers to more fully declare their underlying design intent. An open method has open recursion semantics; it is treated as a callback for subclasses that will always be recursively invoked by the superclass whenever some conceptual event occurs. Ordinary methods—those without the open keyword—are not part of the subclassing interface. While external calls to ordinary methods are dynamically dispatched as usual, recursive calls where the receiver is explicitly stated to be the current object this are dispatched statically.[2] Because open recursion does not apply to methods that are not marked open, subclasses cannot depend on

[1]There are other instances of the fragile base class problem—for example, name collisions between methods in a class and its subclasses—that we do not consider here.

[2]If the receiver is not syntactically this but is an alias, we treat the call as external. This ensures that the dispatch mechanism used is easily predicted by browsing the source code.

```
public class Set {
  List elements;

  public void add(Object o) {
    if (!elements.contains(o))
      elements.add(o);
  }
  public void addAll(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext())
      this.add(i.next());
  }
}
```

Figure 2: In the first solution to the problem described in Figure 1, the developer decides not to mark either `add` or `addAll` as `open`. Thus, when `addAll` invokes `add`, the call is dispatched statically, so that `Set`'s implementation of `add` executes even if a subclass overrides the `add` method (Client calls to `add` are dispatched dynamically as usual). Thus, subclasses cannot tell if `addAll` was implemented in terms of `add` or not, allowing the maintainer of `Set` to change this decision.

```
public class Set {
  List elements;

  /* called once for every added element */
  public void open add(Object o) {
    if (!elements.contains(o))
      elements.add(o);
  }
  public void addAll(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext())
      this.add(i.next());
  }
}
```

Figure 3: In the second solution to the problem described in Figure 1, the developer decides that the `add` method denotes a semantic event of interest to subclasses, and therefore marks `add` as `open`. By doing this, the developer is promising that any correct implementation of `Set` will call `add` once for each element added to the set. Therefore, a subclass interested in "add element" events can override the `add` method without overriding `addAll`.

when they are invoked by the superclass, and the fragile base class problem cannot occur.

In our proposal, there are two choices a designer can make to solve the problem described in Figure 1. In the first solution, shown in Figure 2, the designer of the `Set` class has decided that neither `add` and `addAll` are intended to act as subclass callbacks, and so neither method was annotated `open`. In this case, subclasses cannot tell whether `addAll` is implemented in terms of `add` or not, and so the fragile base class problem cannot arise. Even if `addAll` calls `add` on the current object `this`, this call will be dispatched statically and so subclasses cannot intercept it. Note that calls to `add` from clients are dispatched dynamically as usual, so that an implementation of `CountingSet` can accurately track the element count simply by overriding both `add` and `addAll`.

In the second solution, shown in Figure 3, the designer of the `Set` class has decided that `add` represents a semantic event (adding an element to the set) that subclasses may be interested in reacting to. The designer therefore annotates `add` as `open`, documenting the promise that even if the implementation of `Set` changes, the `add` method will always be called once for each element added to the set. The implementor of `CountingSet` can keep track of the element count by overriding just the `add` function. Any changes to the `Set` class will not break the `CountingSet` code, because the implementor of `Set` has promised that any changes to `Set` will preserve the semantics of calls to `add`.

2.1 Using Selective Open Recursion

With any new language construct, it is important not only to describe the construct's meaning but also how to use it effectively. We offer tentative guidelines for the use of `open`, which can be refined as experience is gained with the construct.

We expect that `public` methods will generally not be `open`. The rationale for this guideline is that `public` methods are intended for use by clients, not by subclasses. In general, any internal use of these public methods is probably coincidental, and subclasses should not rely on these calls. There are exceptions—for example, the `add` method could be both `public` and `open`, depending on the designer's intent—but these idioms can also be expressed by having the `public` method invoke a `protected open` method. For example, instead of making the `add` method `open`, the developer could implement both `add` and `addAll` in terms of a `protected`, `open internalAdd` method that serves as the subclass extension point. Using this `protected` method solution is potentially cleaner because it separates the client interface from the subclassing interface.

On the other hand, we expect that `protected` methods will either be `final` or `open`. Protected methods are usually called on the current object `this`, so overriding them is useful only in the presence of open recursion. Protected methods that are not intended to represent callbacks or extension points for subclasses should be marked as `final`.

Private methods in languages like Java are unaffected by our proposal; since they cannot be overridden, open recursion is not relevant.

2.2 An Alternative Proposal

The discussion above suggests an alternative proposal: instead of adding a new keyword to the programming language, simply use open recursion dispatch semantics for all (non-`final`) `protected` methods and treat all `public` methods as if they were non-open. This alternative has the advantage of simplicity; it takes advantage of common patterns of usage, does not add a new keyword to the language, and encourages programmers to cleanly separate the public client interface from the protected subclass interface.

However, there are two disadvantages to the alternative. If, in addition to performing a service for a client, a public method also represents an event that subclasses may want to extend, the programmer will be forced to create an additional protected method for the subclass interface, creating a minor amount of code bloat. Furthermore, the proposal that makes `open` explicit is a more natural evolutionary path;

existing Java code need only be annotated with open (perhaps with the analysis described in Section 4), whereas in the alternative proposal public methods that are conceptually open would have to be re-written as a pair of public and protected methods.

2.3 Applications to Current Languages

Our proposal extends languages like Java and C# in order to capture more information about how a class can be extended by subclasses. However, the idea of "open" methods can also be applied within existing languages, providing engineering guidelines for avoiding problematic uses of inheritance.

The discussion above suggests that developers should avoid calling public methods on the current object this. If a public method contains code that can be reused elsewhere in the class, the code should be encapsulated in a protected or private method, and the public method should call that internal method. This guideline was previously suggested by Ruby and Leavens [11], and appears to be common practice within the Java standard library in any case. For example, the java.util.Vector class in the JDK 1.4.2 internally calls a protected method, ensureCapacityHelper, to verify that the underlying array is large enough—even though the public method ensureCapacity could be used instead.

Protected methods should be final if they don't represent an explicit extension point for subclasses. The author of a library should carefully document under which circumstances non-final protected methods are called, so that subclasses can rely on the semantics.

Methodological solutions like this one have the advantage that they do not change the semantics of the language. However, for a methodology to be effective, it must be followed. The advantage of Selective Open Recursion is that the keyword open encourages developers to make an explicit choice about the nature of the methods they define, then enforces that choice naturally through the dispatch semantics o the language. Thus, both the syntax and semantics of our proposal work together to reinforce good use of inheritance, while a methodological solution relies primarily on programmer discipline, possibly augmented with lint-like style checkers.

2.4 A Rejected Alternative Design

Based on the insight that the fragile base class problem arises when open recursion is used unintentionally, there is a natural alternative design to be considered. In the discussion above, we chose to annotate methods as being open or not; an alternative is to annotate call sites as using dynamic or static dispatch. We rejected this alternative for two reasons. First, it is a poor match for the design intent, which associates a method—not a call site—with a callback or extension point. Second, because the design intent is typically associated with methods, it would be very surprising if different recursive calls to the same method were treated differently. By annotating the method rather than the call site, our proposal helps developers be consistent.

2.5 Family Polymorphism

The fragile base class problem can be generalized to sets of classes that are closely related. For example, if a Graph module defines classes for nodes and edges, it is likely that the node and edge class are closely related and will often be inherited together. Just as self-calls in an object-oriented setting can be mistakenly "captured" by subclasses, calls between node and edge superclasses might be mistakenly captured by node and edge subclasses.

This paper is primarily focused on the version of the problem that is restricted to a single subclass and superclass, in part because the right solution is more clear-cut in this setting. However, some languages provide first-class support for extending related classes together through mechanisms like Family Polymorphism [6]. In this setting, our proposal could potentially be generalized to distinguish between inter-object calls that should be dispatched dynamically and those that should be dispatched statically. Further work is needed to understand how to apply our proposal effectively in this setting.

2.6 Pure Methods

A central aspect of our approach is that a class must document the circumstances under which all of its open methods are called internally. As suggested by Ruby and Leavens [11], it is possible to relax this requirement for *pure* methods which have no (visible) side-effects and do not change their result with inheritance. Since these methods have no effects and always return the same result, a class can change the way in which they are called without affecting subclasses. An auxiliary analysis or type system could be used to verify that pure methods have no effects, including state changes (other than caches), I/O operations, or non-termination.

2.7 Specification and Verification Benefits

We believe that Selective Open Recursion has potential benefits to formal specification and verification techniques. Intuitively, reasoning about non-open methods is easier than reasoning about open methods, since calls to open methods on this must be formally treated as callbacks to methods of a subclass. Reasoning about code with callbacks to an unknown function defined in a subclass is inherently more challenging than analyzing a locally-defined set of functions, because the analysis results will be dependent or parameterized by the behavior of that function. By making open recursion selective, our technique can reduce the number of callbacks that formal verification techniques must confront. We are currently working to make these intuitions more precise by proving a representation independence theorem in a formal model of selective open recursion.

3. Implementation and Analysis

We have implemented Selective Open Recursion as an extension to the Barat Java compiler [3]. Our implementation strategy leaves open methods and private methods unchanged. For each non-open public/protected method in the source program, we generate another protected final method containing the implementation, and rewrite the original method to call the new method. We leave all calls to open methods unchanged, as well as all calls to methods with a receiver other than this. For every call to a non-open method that has this as the receiver, including implicit uses of this but not other variables aliased to this, we call the corresponding implementation method, thus simulating static dispatch.

Our implementation of Selective Open Recursion is available at http://www.archjava.org/ as part of the open source ArchJava compiler.

3.1 Inference of Open Recursion

In order to ease a potential transition from standard Java or C# to a system with Selective Open Recursion, we have implemented an analysis that can automatically infer which methods must be annotated with `open` in order to preserve the original program's semantics. Of course, our system is identical to Java-like languages if every method is `open`, so the goal of the analysis is to introduce as few `open` annotations as possible. Extra `open` annotations are problematic because they create the possibility of using open recursion when it was not intended, thus triggering fragile base class problems like the example above. In general, no analysis can do this perfectly, because the decision to make a method `open` is a design decision that may not be expressed explicitly in the source code. However, an analysis can provide a reasonable (and safe) default that can be refined manually later.

In order to gain precision, our analysis design assumes that whole-program information is available. A local version of the analysis could be defined, but it would have to assume that every method called on `this` is `open`, because otherwise some unknown subclass could rely on the open recursion semantics of Java-like languages. This assumption would be extremely conservative, so much so that it would be likely to obscure any potential benefits of Selective Open Recursion.

Our analysis design examines each `public` and `protected` method m of every class C. The program potentially relies on open recursion for calls to m whenever there is some method m' in a class $C' \leq C$ that calls m on `this`, and some subclass C'' of C' overrides m, and that subclass either doesn't override m' or makes a super call to m'. The analysis conservatively checks this property, and determines that the method should be annotated `open` whenever the property holds.

3.2 Experiments

We have applied our analysis to a large portion of the Java library, namely all of the packages starting with `java` except for `java.nio` (which was more difficult to compile due to the code generation that is used in that package), and `java.sql` (which triggered a bug in our implementation). We used the JDK 1.4.2 as our codebase.

A threat to validity of this experiment is that a true determination of which methods might be open would have to make a closed world assumption, implicitly considering all possible clients of the library. The library developer might have created "hook" methods for use by future clients that are self-called and ought to be open, but which are not overridden within the library. Our analysis will not catch these methods as being open. However, we believe that because there is substantial use of inheritance and overriding within the library, our analysis should find most of the relevant open methods.

Open Annotations. There are 9897 method declarations in the portion of the standard library that we analyzed. Of these, we determined that only 246 would require `open` annotations in our system to preserve the current semantics of the standard library. This is a small fraction (less than 3%) or the methods in the library, suggesting that open annotations would be infrequently needed in practice.

In principle, it is possible that open annotations would not be needed on a codebase because that codebase was not making use of inheritance in any case. In fact, however, we found 1394 of the methods in the standard library are actually overridden, indicating substantial though not ubiquitous use of inheritance. The 246 `open` methods still make up less than 18% of the methods that were overridden.

The evidence that few `open` annotations are needed in practice supports the utility of Selective Open Recursion. Calling patterns within a class can be easily changed if few of that class's methods are open. In order to support correct usage of inheritance it is important that the ways in which open methods are called are documented [8, 12, 11], and so having fewer open methods lessens the documentation burden on implementors.

Optimization Potential. Since having few "open" annotations makes it easier for developers to reason about correctness of changes to a class, it is natural to expect that it might aid in automated reasoning—such as for compiler optimizations—as well. We tested this hypothesis on the same part of the Java library by looking at the potential for method inlining. We found that the library contains 22339 method calls, of which 6852 were self-calls. Only 716 of these self-calls were to open methods, meaning that they need to be dynamically dispatched. The remaining 6136 calls could potentially be inlined in a system with Selective Open Recursion. In standard Java, however, it would be unsafe in general to inline these calls, as Java treats all methods as implicitly open.

This data indicates that Selective Open Recursion allows 27% of all method calls in the libraries analyzed to be inlined. It is possible that some of these calls could already have been inlined because the target method is `private` or `final`, however. We are currently working to gather data that will tell us the true increase in optimization potential.

A whole program analysis could catch many of the optimization opportunities that Selective Open Recursion does, simply by observing that a particular program does not use all of the open recursion that Java supports. Whole program optimization of Java programs is complicated, however, because many programs load code dynamically, and this code could invalidate optimizations such as inlining. Because our Selective Open Recursion proposal changes the semantics of dispatch so that subclasses cannot intercept non-open self calls, it enables optimizations like inlining without the need for whole-program information.

4. Related Work

A significant body of related research focuses on documenting the dependencies between methods in a *specialization interface*. Kiczales and Lamping proposed that a method should document which methods it depends on, so that subclasses can make accurate assumptions about the superclass implementation [8]. Steyaert et al. propose a similar approach in a more formal setting [12]. Ruby and Leavens suggest documenting method call dependencies as part of a broader focus on modular reasoning in the presence of inheritance [11]. They also document a number of design guidelines that are applicable to the setting of Selective Open Recursion.

A common weakness of the "dependency documentation" approaches described above is that they solve the fragile base class problem not by hiding implementation details, but rather by exposing them. Since the calling patterns of a class are part of the subclassing interface—and since subclasses may depend on them—making significant changes to the implementation of the class become impossible. Steyaert et al. acknowledge this and suggest documenting only the "important method calls," but the fragile base class problem can still occur unless unimportant method calls are hidden from subclasses using a technique like ours. Our work requires that calling patterns be maintained for calls to `open` methods, but does not impose this requirement for non-open methods, allowing a much wider range of implementation changes.

Bloch, Szyperski, and others suggest using forwarding in place of inheritance as a way of avoiding the fragile base class problem [2, 13]. However, as Szyperski notes, not all uses of inheritance can be replaced by forwarding because open recursion is sometimes needed [13]. Selective Open Recursion provides a middle ground between inheritance and forwarding, providing open recursion when it is needed but the more modular forwarding semantics where it is not.

Mikhajlov and Sekerinski consider a number of different ways in which an incorrect use of inheritance can break a refinement relationship between a class and its subclasses [9]. They prove a flexibility theorem showing that under certain conditions, when a superclass C is replaced with a new implementation D, then C's subclasses still implement refinements of the original implementation C. Their results, however, do not appear to guarantee that the semantics of C's subclasses are unaffected by the new implementation D, which is the contribution of our work.

Our use of static dispatch for calls on `this` is related to the *freeze* operator provided by module systems such as Jigsaw [4]. The freeze operation statically binds internal uses of a module declaration, while allowing module extensions to override external uses of that declaration. The freeze operator, however, has not been previously proposed as a solution to the fragile base class problem, nor (to our knowledge) has it previously been integrated into an object-oriented language implementation.

Some languages, including C++, provide a way to statically call a particular implementation of a method [5]. While this technique can be used as an implementation strategy for our proposal, we believe it is cleaner to associate "open-ness" with the method that is called rather than the call site, as discussed earlier.

Our solution to the fragile base class problem was inspired by our earlier work on a related modularity problem in aspect-oriented programming [1]. Just as a `CountingSet` subclass of `Set` can observe whether `addAll` is implemented in terms of `add`, a `Counting` aspect can be defined that uses advice to make the same observation. Our solution there was to prohibit aspects from advising internal calls within a class or module—just as we solve the fragile base class problem by using static dispatch to prevent subclasses from intercepting implementation-dependent calls in their superclass. In the aspect-oriented setting, we allow modules to export pointcuts that act as disciplined extension points, similar to `open` methods.

Relative to previous work, ours is the first to address the fragile base class problem by distinguishing methods for which open recursion is needed from methods for which it is not.

5. Conclusion

This paper argued that the fragile base class problem occurs because current object-oriented languages do not distinguish internal method calls that are invoked for mere convenience from those that are invoked as explicit extension points for subclasses. We proposed to make this distinction explicit by labeling as `open` those methods to which open recursion should apply. Our results mean that object-oriented component library designers can freely change more aspects of a library's implementation without the danger of breaking subclass code.

6. Acknowledgments

We thank Craig Chambers, Donna Malayeri, Todd Millstein, Frank Pfenning, and the anonymous reviewers for their feedback on earlier drafts of this material.

7. References

[1] J. Aldrich. Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming. In *AOSD Workshop on Foundations of Aspect Languages*, March 2004.

[2] J. Bloch. *Effective Java*. Addison-Wesley, Reading, Massachusetts, 2001.

[3] B. Bokowski and A. Spiegel. Barat–A Front-End for Java. Freie Universitt Berlin Technical Report B-98-09, 1998.

[4] G. Bracha. The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. Ph.D. Thesis, Dept. of Computer Science, University of Utah, 1992.

[5] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, May 1990.

[6] E. Ernst. Family Polymorphism. In *European Conference on Object-Oriented Programming*, June 2001.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[8] G. Kiczales and J. Lamping. Issues in the Design and Documentation of Class Libraries. In *Object-Oriented Programming Systems, Languages, and Applications*, 1992.

[9] L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *European Conference on Object-Oriented Programming*, 1998.

[10] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[11] C. Ruby and G. T. Leavens. Safely Creating Correct Subclasses without Seeing Superclass Code. In *Object-Oriented Programming Systems, Languages, and Applications*, October 2000.

[12] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Object-Oriented Programming Systems, Languages, and Applications*, October 1996.

[13] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

CTL Model-checking for Systems with Unspecified Components*

[Extended Abstract]

Gaoyan Xie and Zhe Dang
School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA 99164, USA
{gxie,zdang}@eecs.wsu.edu

ABSTRACT

In this paper, we study a CTL model-checking problem for systems with unspecified components, which is crucial to the quality assurance of component-based systems. We introduce a new approach (called *model-checking driven black-box testing*) that combines model-checking with traditional black-box software testing to tackle the problem in an automatic way. The idea is, with respect to some requirement (expressed in a CTL formula) about the system, to use model-checking techniques to derive a condition (expressed in terms of *witness graphs*) for an unspecified component such that the system satisfies the requirement iff the condition is satisfied by the component. The condition's satisfiability can be established by testing the component. Test sequences are generated on-the-fly by traversing the witness graphs with a bounded depth. With a properly chosen bound, a complete and sound algorithm is immediate.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model-checking*; D.2.5 [**Software Engineering**]: Testing/Debugging—*Black-box testing*; F.4.1 [**Mathematic Logic and Formal Languages**]: Mathematical Logic—*Temporal Logic*

General Terms

Verification, Component-based systems

Keywords

Component-based systems, Model-checking, Black-box testing

1. INTRODUCTION

Although component-based software development [22, 6] enjoys the great benefits of reusing valuable software assets, reducing development costs, improving productivity, etc., it also poses serious challenges to the quality assurance problem [3, 27] of component-based systems. This is because prefabricated components could be a new source of system failures. In this paper, we are interested in one problem that system developers often face:

> (*) *how to ensure that a component whose design detail and source code are unavailable will function correctly in a host system.*

This is a rather challenging problem and yet to be handled in a satisfying way by available techniques. For instance, in practice, testing is almost the most natural resort to solve the problem. When integrating a component into a system, system developers may either choose to thoroughly test the component separately or to hook the component with the system and conduct integration testing. However, software components are generally built with multiple sets of functionality [17], and indiscriminately testing all the functionality of a software component separately is not only expensive but also infeasible. Also, integration testing is often not applicable in applications where software components are used for dynamic upgrading or extending a running system [32] that is too costly or not supposed to shut down for testing at all. Even without the above limitations, purely testing techniques are still considered to be insufficient to solve the problem for mission-critical or safety-critical systems where formal methods like model-checking are highly desirable. But, it is very common that design details and source code of an externally obtained component are not available to the developers of its host system. This makes existing formal verification techniques (like model-checking) not directly applicable to these cases.

In this paper, we study how to extend CTL model-checking techniques to solve the problem in (*). Specifically, we consider systems with only one such unspecified component. Denote such a system as $Sys = \langle M, X \rangle$, where M is the host system and X is an unspecified component. Both M and X are finite-state transition systems (the actual specification of X is unknown), which communicate with each other by synchronizing a finite set of given input and output symbols. Then our problem can be further formulated as to check whether $\langle M, X \rangle \models f$ holds, where f is a CTL formula specifying some requirement for Sys.

Our approach to solve the above model-checking problem is a combination of both model-checking and traditional black-box testing techniques (called *model-checking driven black-box testing*). First, a model-checking procedure is used to derive from M and f a condition P over the unspecified components X. The condition P guarantees that the system Sys satisfies the requirement f iff P is satisfied by X. The satisfiability of the condition P over the unspecified component X is then checked through adequate black-

*The research was supported in part by NSF Grant CCF-0430531.

box testing on X with test-cases generated automatically from P. Our study shows that the obtained condition P is in the form of a hierarchy of communication graphs (called a witness graph), each of which is a subgraph of M. Test-cases can be generated by traversing witness graphs when the unspecified component X is a finite or infinite state system. In particular, when X is a finite state system (m is an upper bound of its state number), our study shows that traversing the witness graphs up to a depth bounded by $O(k \cdot n \cdot m^2)$ is sufficient to answer the model-checking query, where k is the number of CTL operators in the formula f and n is the state number in the host system M. Thus, in this case, with a properly chosen search depth, a complete and sound solution is immediate.

The advantages of our approach are obvious: a stronger confidence about the reliability of the system can be established through both model-checking and adequate functional testing; system developers can customize the testing of a component with respect to some specific system properties; intermediate model-checking results (the witness graphs) for a component can be reused to avoid (repetitive) integration testing when the component is updated, if only the new component's interface remains the same; the whole process can be carried out in an automatic way.

The rest of this paper is organized as follows. Section 2 defines the system model and introduces some background on black-box testing. Section 3 presents algorithms for deriving the condition as well as testing the condition over the unspecified component. Section 4 compares our research with some related work. Section 5 concludes the paper with discussions on issues to be solved in the future.

Due to space limit, details of most algorithms are omitted in this extended abstract. Readers can find the full version of this paper at `http://www.eecs.wsu.edu/∼gxie/`.

2. PRELIMINARIES

2.1 The System Model

In this paper, we consider systems with only one unspecified component, which is denoted by

$$Sys = \langle M, X \rangle,$$

where M is the host system and X is the unspecified component. Both M and X are finite-state transition systems communicating synchronously with each other via a finite set of input and output symbols.

Formally, the unspecified component X is viewed as a deterministic Mealy machine whose internal structure is unknown (but an implementation of X is available for testing). We write X as a triple $\langle \Sigma, \nabla, m \rangle$, where Σ is the set of X's input symbols, ∇ is the set of X's output symbols, and m is an upper bound for the number of states in X (the m is given). Upon receiving an input symbol, X may perform some internal actions, move to a new state, and then send back an output symbol immediately. Assume that X has an initial state s_{init} and X is always in this initial state when the system starts to run. A *run* of X is a sequence of alternating symbols in Σ and ∇: $\alpha_0\beta_0\alpha_1\beta_1...$, such that, starting from the initial state s_{init}, X outputs exactly the sequence $\beta_0\beta_1...$ when it is given the sequence $\alpha_0\alpha_1...$ as input. In this sense, we say that the input sequence is accepted by X.

The host system M is defined as a 5-tuple

$$\langle S, \Gamma, R_{env}, R_{comm}, I \rangle$$

where

- S is a finite set of states;
- Γ is a finite set of events;
- $R_{env} \subseteq S \times \Gamma \times S$ defines a set of *environment transitions*, where $(s, a, s') \in R_{env}$ means that M moves from state s to state s' upon receiving an event (symbol) $a \in \Gamma$ from the outside environment;
- $R_{comm} \subseteq S \times \Sigma \times \nabla \times S$ defines a set of *communication transitions* where $(s, \alpha, \beta, s') \in R_{comm}$ means that M moves from state s to state s' when X outputs a symbol $\beta \in \nabla$ after M sends X an input symbol $\alpha \in \Sigma$; and,
- $I \subseteq S$ defines M's initial states.

Without loss of generality, we further assume that, there is only one transition between any two states in M (but in general, M could still be nondeterministic).

An *execution path* of the system $Sys = \langle M, X \rangle$ is a (potentially infinite) sequence τ of states and symbols, $s_0c_0s_1c_1...$, where each $s_i \in S$, each c_i is either a symbol in Γ or a pair $\alpha_i\beta_i$ (called a *communication*) with $\alpha_i \in \Sigma$ and $\beta_i \in \nabla$. Additionally, τ satisfies the following requirements:

- s_0 is an initial state of M, i.e., $s_0 \in I$;
- for each $c_i \in \Gamma$, (s_i, c_i, s_{i+1}) is an environment transition of M;
- for each $c_i = \alpha_i\beta_i$, $(s_i, \alpha_i, \beta_i, s_{i+1})$ is a communication transition of M.

The *communication trace* of τ, denoted by τ_X, is the sequence obtained from τ by retaining only symbols in Σ and ∇ (i.e., the result of projecting τ onto Σ and ∇). For any given state $s \in S$, we say that the system Sys can *reach* s iff Sys has an execution path τ on which s appears and τ_X (if not empty) is also a run of X. In the case when X is fully specified, the system can be modeled as an I/O automaton [26] (which is not input-enabled) or as two interface-automata [10]. In the latter case, the state number m can also be considered as the number of states in an interface automaton of X (instead of the state number in X itself).

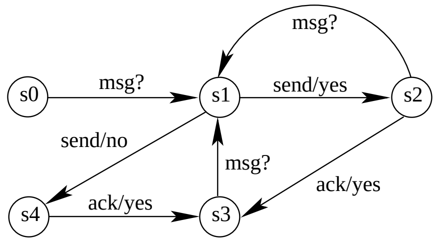


Figure 1: An example system

As an illustrating example, consider a system $Sys = \langle M, X \rangle$ where the host system M keeps receiving messages from the outside environment and then sends the message through the unspecified component X. The only event symbol in M is msg, while X has two input symbols $send$ and ack that make X send a message and ask X for an acknowledge respectively. X also has two output symbols yes and no that indicate whether the internal actions related with a previous input symbol succeeded or not (i.e., whether a message is sent and whether an ack is available). The transition graph of M is depicted in Figure 1 where we use a suffix ? to denote events from the outside environment (e.g., msg?), and use an infix / to denote communications of M with X (e.g., $send/yes$).

2.2 Black-box Testing

Black-box testing is a technique to test a system without knowing its internal structure. A system is regarded as a "black-box" in the sense that its behaviors can only be determined by observing its implementation's the input/output sequences, and a test over a black-box is simply to run its implementation with a given input sequence.

Studies [33] have shown that if only an upper bound for the number of states in the system and the system's input/output symbols are known, then its (equivalent) internal structure can be recovered through black-box testing. Clearly, a naive solution to the CTL model-checking problem over the system Sys is to first recover the full structure of the component X through black-box testing, and then solve the classic model-checking problem over the fully specified system composed from M and the recovered X. Notice that, in the naive solution, when black-box testing is performed over X, the selected test sequences have nothing to do with the host system M. Therefore, it is desirable to find more sophisticated solutions such as the algorithms introduced in below, which only select "useful" test sequences w.r.t. the M as well as its temporal requirement.

The unspecified component X in this paper can be treated as a black-box. And as a common practice in black-box testing, X is assumed to always have a special input symbol $reset$ which always makes it return to the initial state s_{init} regardless of its current state. Throughout this paper, we use $Experiment$ to denote a test over the unspecified component X. Specifically, we use $Experiment(X, reset\pi)$ to denote the output sequence obtained by testing X with the input sequence $reset\pi$ (i.e., run X from its initial state with input sequence π). Suppose after testing X with the input sequence $reset\pi$, we continue to run X by feeding it with an input symbol α. Corresponding to this α, we may obtain an output symbol β from X, and we use $Experiment(X, \alpha)$ to denote this β. Notice that this $Experiment(X, \alpha)$ is actually a shorthand for "the last output symbol in $Experiment(X, reset\pi\alpha)$".

3. CTL MODEL-CHECKING DRIVEN BLACK-BOX TESTING

In this section, we introduce algorithms for CTL model-checking driven black-box testing for the system $Sys = \langle M, X \rangle$.

3.1 Ideas

Recall that the CTL model-checking problem is, for a Kripke structure $K = (S, R, L)$, a state $s_0 \in S$, and a CTL formula f, to check whether $K, s_0 \models f$ holds. The standard algorithm [9] to solve this problem operates by exhaustively searching the structure and, during the search, labeling each state s with the set of subformulas of f that are true at s. Initially, labels of s are just $L(s)$. Then, the algorithm goes through a series of stages—during the i-th stage, subformulas with the $(i-1)$-nested CTL operators are processed. When a subformula is processed, it is added to the labels of each state where the subformula is true. When all the stages are completed, the algorithm returns $true$ when s_0 is labeled with f, or $false$ otherwise.

However, for a system like Sys that contains an unspecified component, the standard algorithm does not work, since transitions of the host system M may depend on communications with the unspecified component X, which cannot be statically resolved. For instance, for the system depicted in Figure 1, a simple check like $\langle M, X \rangle, s_1 \models EX\ s_2$ (i.e., whether s_2 is reachable from s_1) cannot be done by the standard algorithm. In this section, we adapt the standard CTL model-checking algorithm [9] to handle systems like Sys; i.e., to check whether

$$\langle M, X \rangle, s_0 \models f \tag{1}$$

holds, where s_0 is an initial state in M and f is a CTL formula.

Our new algorithm follows a similar structure to the standard one. It also goes through a series of stages to search M's state space and label each state during the search. The labeling of a state, however, is far more complicated when processing a subformula during each stage. The central idea of our algorithm can be summarized as follows. When the truth of a subformula h at a state s cannot be statically decided (due to communications), we construct some communication graph (called a *witness graph*, written as $[\![h]\!]$) by picking up all the communications that shall witness the truth of h at state s and then label s with the witness graph. The witness graph serves as a sufficient and necessary condition for h to be true at s, and this condition shall be later evaluated by testing the unspecified component X.

Actually, we do not have to construct one witness graph for every subformula at every state. Instead, we construct one witness graph only for a subformula h that contains a CTL operator, and this witness graph encodes all the witnesses (communications) to the truth of the formula at every state in M. Thus, totally we shall construct k witness graphs where k is the number of CTL operators in f, and we associate each witness graph with a unique ID number that ranges from 2 to $k+1$. Let $\mathcal{I}$ be the mapping from the witness graphs to their IDs; i.e., $\mathcal{I}([\![h]\!])$ denotes the ID number of h's witness graph, and $\mathcal{I}^{-1}(i)$ denotes the witness graph with i as its ID number, $2 \leq i \leq k+1$. Notice that the witness graph to different CTL operators shall be evaluated differently, so we call $[\![h]\!]$ as an *EX graph*, an *EU graph*, or an *EG graph* when h takes the form of $EX\ g$, $E[g_1\ U\ g_2]$, or $EG\ g$, respectively.

Specifically, we label a state s with 1 (resp. nothing) if h is true (resp. false) at s regardless of the communications between M and X. Otherwise, we shall label s with $i = \mathcal{I}([\![h]\!])$ when h takes the form of $EX\ g$, $E[g_1\ U\ g_2]$, or $EG\ g$, which means that h could be true at s and the truth would be witnessed by some (communication) paths starting from s in $\mathcal{I}([\![h]\!])$. When h takes the form of a Boolean combination of subformulas using $\neg$ and $\vee$, the truth of h at state s shall also be a logic combination of the truths of its component subformulas at the same state. To this end, we shall label s with an *ID expression* ψ defined as follows:

- $ID := 1 \mid 2 \mid \ldots \mid k+1;$
- $\psi := ID \mid \neg\psi \mid \psi \vee \psi.$

Let Ψ denote the set of all ID expressions. For each subformula h, in addition to the possible witness graph of h, we also construct a labeling (partial) function $L_h : S \to \Psi$ to record the ID expression labeled to each state during the processing of the subformula h. The labeling function is returned when the subformula is processed.

In summary, our new algorithm to solve the model-checking problem $\langle M, X \rangle, s_0 \models f$ can be sketched as follows:

```
**Procedure** $CheckCTL(M, X, s_0, f)$
    $L_f := ProcessCTL(M, f)$
    **If** $s_0$ is labeled by $L_f$ **Then**
        **If** $L_f(s_0) = 1$ **Then**
            **Return** $true$;
        **Else**
            **Return** $TestWG(X, reset, s_0, L_f(s_0))$;
    **Else**
        **Return** $false$.
```

In the above algorithm, a procedure $ProcessCTL$ (will be introduced in Section 3.2) is called to process all subformulas of f, and it returns a labeling function L_f for the outer-most subformula (i.e., f itself). The algorithm returns $true$ when s_0 is labeled with 1 by L_f or $false$ when s_0 is not labeled at all. In other cases, a procedure $TestWG$ (will be introduced in Section 3.3) is called to test whether the ID expression $L_f(s_0)$ could be evaluated true at s_0.

3.2 Process a CTL Formula

Processing a CTL formula h is implemented through a recursive procedure $ProcessCTL$. Recall that any CTL formula can be expressed in terms of $\vee$, $\neg$, EX, EU, and EG. Thus, at each intermediate step of the procedure, depending on whether the formula h is atomic or takes one of the following forms: $g_1 \vee g_2$, $\neg g$, $EX\ g$,$E[g_1\ U\ g_2]$, or $EG\ g$, the procedure has six cases to consider. When it finishes, a labeling function L_h is returned for the formula h.

3.2.1 Process atom

When h is an atomic formula, $ProcessCTL$ simply returns a function that labels each state where h is true with 1.

3.2.2 Process negation

When $h = \neg g$, we first process g by calling $ProcessCTL$, then construct a labeling function L_h for h by "negating" g's labeling function L_g as follows:

- For every state s that is not in the domain of L_g, let L_h label s with 1;
- For each state s that is in the domain of L_g but not labeled with 1 by L_g, let L_h label s with ID expression $\neg L_g(s)$.

3.2.3 Process union

When $h = g_1 \vee g_2$, we first process g_1 and g_2 respectively by calling $ProcessCTL$, then construct a labeling function L_h for h by "merging" g_1 and g_2's labeling functions L_{g_1} and L_{g_2} as follows:

- For each state s that is in both L_{g_1}'s domain and L_{g_2}'s domain, let L_h label s with 1 if either L_{g_1} or L_{g_2} labels s with 1 and label s with ID expression $L_{g_1}(s) \vee L_{g_2}(s)$ otherwise;
- For each state s that is in L_{g_1}'s domain (resp. L_{g_2}'s domain) but not in L_{g_2}'s domain (resp. L_{g_1}'s domain), let L label s with $L_{g_1}(s)$ (resp. $L_{g_2}(s)$).

3.2.4 Process an EX subformula

When $h = EX\ g$, subformula g is processed first by recursively calling $ProcessCTL$. Then, the procedure $ProcessEX$ is called with g's labeling function L_g to create a witness graph for h and to construct a labeling function L_h.

In $ProcessEx$, the witness graph for $h = EX\ g$, called an EX graph, is created as a triple: $[\![h]\!] = \langle N, E, L_g \rangle$, where N is a set of nodes and E is a set of annotated edges. It is created as follows:

- Add one node to N for each state that is in the domain of L_g.
- Add one node to N for each state that has a successor in the domain of L_g.
- Add one edge between two nodes in N to E when M has a transition between two states corresponding to the two nodes respectively; if the transition involves a communication with X then annotate the edge with the communication symbols.

The labeling function L_h is constructed as follows. For each state s that has a successor s' in the domain of L_g, if s can reach s' through an environment transition and s' is labeled with 1 by L_g then let L_h also label s with 1, otherwise let L_h label s with $\mathcal{I}([\![h]\!])$.

3.2.5 Process an EU subformula

The case when $h = E[g_1\ U\ g_2]$ is more complicated. We first process g_1 and g_2 respectively by calling $ProcessCTL$, then call the procedure $ProcessEU$ with g_1 and g_2's labeling functions L_{g_1} and L_{g_2} to create a witness graph for h and to construct a labeling function L_h.

In $ProcessEU$, the witness graph for $h = E[g_1\ U\ g_2]$, called an EU graph, is created as a 4-tuple: $[\![h]\!] := \langle N, E, L_{g_1}, L_{g_2} \rangle$, where N is a set of nodes and E is a set of edges. N is constructed by adding one node for each state that is in the domain of L_h, while E is constructed in the same way as that of $ProcessEX$.

We construct the labeling function L_h recursively. First, let L_h label each state s in the domain of L_{g_2} with $L_{g_2}(s)$. Then, for state s that has a successor s' in the domain of L_h, if s (resp. s') is labeled with 1 by L_{g_1} (resp. L_h) and s can reach s' through an environment transition, then let L_h also label s with 1, otherwise let L_h label s with $\mathcal{I}([\![h]\!])$. Notice that, in the latter step, if a state s can be labeled with both 1 and $\mathcal{I}([\![h]\!])$, let L_h label s with 1. Thus, we can guarantee that the constructed L_h is indeed a function.

3.2.6 Process an EG subformula

To handle formula $h = EG\ g$, we first process g by calling $ProcessCTL$, then call the procedure $ProcessEG$ with g's labeling function L_g to create a witness graph for h and to construct a labeling function L_h.

In $ProcessEG$, the witness graph for h, called an EG graph, is created as a triple: $[\![h]\!] := \langle N, E, L_g \rangle$, where N is a set of nodes and E is a set of annotated edges. The graph is constructed in the same way as that of $ProcessEU$.

The labeling function L_h is constructed as follows. For each state s that can reach a loop C through a path p such that every state (including s) on p and C is in the domain of L_g, if every state (including s) on p and C is labeled with 1 by L_g and no communications are involved on the path and the loop, then let L_h also label s with 1, otherwise let L_h label s with $\mathcal{I}([\![h]\!])$.

3.3 Evaluate an ID Expression

As seen from the previous subsection, the $ProcessCTL$ procedure labels states with ID expressions for each subformula h, which are essentially conditions under which the subformula h is true at a state. Also, as seen in Section 3.1, the $CheckCTL$ procedure either gives a definite $true$ or $false$ answer to the CTL model-checking problem, i.e., $\langle M, X \rangle, s_0 \models f$, or it reduces the problem to checking whether the ID expression $\psi = L_f(s_0)$ can be evaluated true at state s_0. The evaluation is carried out by a recursive procedure $TestWG$, which is essentially a testing process.

According to the definition of an ID expression, $TestWG$ only needs to consider six cases. When the ID expression ψ is the value 1, $TestWG$ returns $true$; when $\psi = \neg\psi_1$, $TestWG$ returns $false$ (resp. $true$) if ψ_1 is evaluated true (resp. false) at s_0; when $\psi = \psi_1 \vee \psi_2$, $TestWG$ returns $true$ if either ψ_1 or ψ_2 can be evaluated true at s_0, and returns $false$ if neither can be evaluated true at s_0. The remaining three cases are when ψ represents an EX graph, an EU graph, or an EG graph. We discuss the evaluation for these three cases as follows.

3.3.1 Evaluate an EX graph

To check whether an EX graph $G = \langle N, E, L_g \rangle$ can be evaluated true at a state s_0 is simple. We just test whether the system M can reach from s_0 to another state $s' \in \mathbf{dom}(L_g)$ along one edge in G such that the ID expression $L_g(s')$ can be evaluated true at s'.

3.3.2 Evaluate an EU graph

To check whether an EU graph $G = \langle N, E, L_{g_1}, L_{g_2} \rangle$ can be evaluated true at a state s_0, we need to traverse all paths p in G with length less than mn,[1] and test the unspecified component X to see whether the system can reach some state $s' \in \mathbf{dom}(L_{g_2})$ through one of those paths. In here, m is the given upper bound for the number of states in the unspecified component X and n is the number of nodes in G. In the meantime, we should also check whether $L_{g_2}(s')$ can be evaluated true at s' and whether $L_{g_1}(s_i)$ can be evaluated true at s_i for each s_i on p (excluding s') by calling $TestWG$.

3.3.3 Evaluate an EG graph

To check whether an EG graph $G = \langle N, E, L_g \rangle$ can be evaluated true at a state s_0, we need to find an infinite path in G, along which the system can run forever. The following procedure $TestEG$ first decomposes G into a set of SCCs. Then, for each state s_f in the SCCs, it calls another procedure $SubTestEG$ to test whether the system can reach s_f from s_0 along a path not longer than mn,[1] as well as whether the system can further reach s_f from s_f for $m - 1$ times[1]. Here, m is the same as before while n is the number of nodes in G.

```
Procedure $TestEG(X, \pi, s_0, G = \langle N, E, L_g \rangle)$
    $SCC := \{C | C$ is a nontrivial SCC of $G\}$;
    $T := \bigcup_{C \in SCC} \{s | s \in C\}$;
    For each $s \in T$ Do
        $Experiment(X, reset\pi)$;
        If $SubTestEG(X, \pi, s_0, s, G, level = 0, count = 0)$;
            Return $true$;
    Return $false$.
```

The maximal length of the paths that the above evaluation process shall traverse depends on how many witness graphs are involved in an ID expression, the sizes of the witness graphs, and the number of states of the unspecified component. One can show that the maximal length is bounded by $O(k \cdot n \cdot m^2)$, where k is the number of CTL operators in the formula f, m is the upper bound for the number of states in the unspecified component X, and n is the number of states in the host system M.

3.4 Example

To better understand how our algorithms work, consider such a model-checking problem for the system depicted in Figure 1: starting from the initial state s_0, whenever the systems reaches state s_2, it would eventually reach s_3; i.e., the problem is to check whether $(M, X), s_0 \models AG(s_2 \rightarrow AFs_3)$ holds. Taking the negation of the original problem, we describe how the problem $(M, X), s_0 \models f$, where $f = E[true\ U(s_2 \wedge EG\neg s_3)]$ is solved by our algorithms.

Step 1. Atomic subformula s_2 (in f) is processed by *ProcessCTL*, which returns a labeling function $L_1 = \{(s_2, 1)\}$.

Step 2. Atomic subformula s_3 is processed by *ProcessCTL*, which returns a labeling function $L_2 = \{(s_3, 1)\}$.

Step 3. Subformula $\neg s_3$ is processed by $Negation$ (see Section 3.2.2), which returns a labeling function $L_3 = \{(s_0, 1), (s_1, 1), (s_2, 1), (s_4, 1)\}$.

[1]Since the unspecified component X is treated as a finite state transition system, these bounds can be easily obtained from a Cartesian product of M and X.

Step 4. Subformula $EG\neg s_s$ is processed by $ProcessEG$ (see Section 3.2.6), which constructs an EG graph $G_1 = \langle N, E, L_3 \rangle$ with an ID 2 (see Figure 2) and returns a labeling function $L_4 = \{(s_0, 2), (s_1, 2), (s_2, 2)\}$.

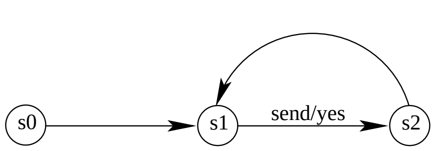


Figure 2: The witness graph for $EG\neg s_s$

Step 5. Subformula $s_2 \wedge EG\neg s_3$ is processed by $Negation$ and $Union$ (see Section 3.2.3), which return a labeling function $L_5 = \{(s_2, 2)\}$.

Step 6. Finally, the formula $E[true\ U(s_2 \wedge EG\neg s_3)]$ is processed by procedure $ProcessEU$ (see Section 3.2.5), which constructs an EU graph $G_2 = \langle N, E, L_{true}, L_5 \rangle$[2] with an ID 3 (see Figure 3) and returns a labeling function $L_f = \{(s_0, 3), (s_1, 3), (s_2, 3), (s_3, 3), (s_4, 3)\}$.

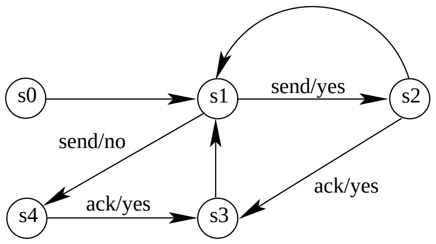


Figure 3: The witness graph for $E[true\ U(s_2 \wedge EG\neg s_3)]$

When $ProcessCTL$ finishes, s_0 is labeled by L_f with an ID expression 3 instead of 1 (i.e., $true$). This indicates that the original model-checking problem can not be statically decided and its truth depends on a condition that the ID expression 3 be evaluated true at s_0. Hence, procedure $TestWG$ must be called to test the condition as follows.

Step 7. The ID expression 3 is evaluated by $EvaluateEU$ since the witness graph with ID 3, G_2 constructed in **Step 6**, is an EU graph.

Step 8. $EvaluateEU$ traverses every path p of G_2 that is between s_0 and some state in the domain of L_5 (recall that L_5 is the fourth component of G_2 in **Step 6** and s_2 is the only state in its domain) to see:

- Whether the annotations (communication symbols) on p constitute a run of the unspecified component X. For instance, if $p = s_0s_1s_2s_1s_3s_1s_2$, then we need to test the "black-box" X with an input sequence "$send\ ack\ send$" to see whether the corresponding output sequence is "$yes\ yes\ yes$".

- Whether the ID expression $L_{true}(s)$ (recall that L_{true} is the third component of G_2 in **Step 6**) can be evaluated true at each state s along p. Obviously, that is true from the definition of L_{true}.

- Whether the ID expression $L_5(s_2) = 2$ can be evaluated true at s_2 by calling $EvaluateEG$ (since the witness graph with ID 2, G_1 constructed in **Step 4**, is an EG graph).

[2]L_{true} labels every state with 1 (i.e., $true$).

- $EvaluateEG$ tries to find in G_1 a loop C as well as a path p_1 from s_2 to C such that the annotations (communication symbols) on the concatenated path pp_1C constitute a run of the unspecified component X. As we can see from Figure 2, the only loop in G_1 is $s_1s_2s_1\cdots$. So, if $p = s_0s_1s_2s_1s_3s_1s_2$, then we need to test the "black-box" X with an input sequence "$send\ ack\ send\ send\ send\cdots$" to see whether the output sequence is "$yes\ yes\ yes\ yes\ yes\cdots$".

Step 9. If none of such paths satisfies the conditions in **Step 8**, then $false$ is returned to indicate that the original model-checking problem is true. Otherwise, $true$ is returned. In this case, the maximal length of test sequences generated is bounded by $5m + 3(m-1)$ according to the evaluation algorithms for EU graphs and EG graphs.

It is easy to see that, in this example, $TestWG$ essentially would be testing whether some communication trace (of bounded length) of sys with two consecutive symbol pairs $(send\ yes)$ is a run of the unspecified component X.

Note. Notice that the condition L_f, a sufficient and necessary condition on the unspecified component X to ensure the truth of the model-checking problem $(M, X), s_0 \models f$, does not depend on the state number m of X. Therefore, even when X is an *infinite-state* system, the condition can also be useful in generating test cases for X and a testing procedure similar to $TestWG$ could be formulated to answer the model-checking query conservatively.

4. RELATED WORK

The quality assurance problem for component-based software has attracted lots of attention in the software engineering community. However, most work are based on the traditional testing techniques and they consider the problem from component developers' point of view; i.e., how to ensure the quality of components before they are released.

Voas [34, 35] proposed a component certification strategy with the establishment of independent certification laboratories performing extensive testing of components and then publishing the results. Technically, this approach would not provide much improvement, since independent certification laboratories can not ensure the sufficiency of their testing either. Some researchers [28] suggested an approach to augment a component with additional information to increase the customer's understanding and analyzing capability of the component behavior. A related approach [36] is to automatically extract a finite-state machine model from the interface of a software component, which is delivered along with the component. This approach can provide some convenience for customers to test the component, but again, how much a customer should test is still a big problem.

Bertolino et. al. [4] recognized the importance of testing a software component in its deployment environment. They developed a framework that supports functional testing of a software component with respect to customer's specification, which also provides a simple way to enclose with a component the developer's test suites which can be re-executed by the customer. Yet their approach requires the customer to have a complete specification about the component to be incorporated into a system, which is not always possible.

In the formal verification area, there has been a long history of research on verification of systems with modular structure. A key idea [24, 23, 20] in modular verification is the *assume-guarantee* paradigm: A module should guarantee to have the desired behavior once the environment with which the module is interacting has the assumed behavior. There have been a variety of implementations for this idea (see, e.g., [19, 1, 29, 11, 8, 37]). The assume-guarantee ideas can be applied to our problem setup if we consider the unspecified component as the host system's environment (though this is counter-intuitive). But the key issue with the assume-guarantee style reasoning is how to obtain assumptions about the environment. Giannakopoulou et. al. [16, 15] introduced a novel approach to generate assumptions that characterize exactly the environment in which a component satisfies its property. Their idea is the closest to ours, still there are non-trivial differences: (1) theirs is a purely formal verification technique (model-checking) while we combine both model-checking and black-box testing to handle systems with unspecified components; and (2) theirs uses a labeled transition system to specify the reachability property of a system while we use CTL formulas, which are more expressive and harder to manipulate. Although not within the assume-guarantee paradigm, Fisler et. al. [13, 25] introduced a similar idea of deducing a model-checking condition for extension features from the base feature for model-checking feature-oriented software designs. Unfortunately, their algorithms are not sound (have false negatives). Furthermore, their approach is not applicable to component-based systems where unspecified components exist. This paper is also different from our previous work [38] where an automata-theoretic approach is used to solve a similar LTL model-checking problem.

In the past decade, there has also been lots of research on combining model-checking and testing techniques for system verification, which can be grouped into a broader class of techniques called specification-based testing. But many of the work only utilizes model-checkers' ability of generating counter-examples from a system's specification to produce test cases against an implementation [7, 21, 12, 14, 2, 5], and they do not generalize the problem setup in this paper. Peled et. al. [31, 18, 30] studied the issue of checking a black-box against a temporal property (called black-box checking). But their focus is on how to efficiently establish an abstract model of the black-box through black-box testing , and their approach requires a clearly-defined property (LTL formula) about the black-box, which is not always possible in component-based systems.

5. CONCLUSIONS

In this paper, we studied the CTL model-checking problem

$$(M, X), s_0 \models f$$

where X is an unspecified component. Our approach is a combination of both model-checking and traditional black-box testing techniques. For such a problem, our algorithm $CheckCTL$ in Section 3.1 either gives a definite $true/false$ answer or gives a sufficient and necessary condition in the form of ID expressions and witness graphs. The condition is evaluated through black-box testing over the unspecified component X. Test sequences are generated by traversing the witness graphs with bounded depth as we evaluate the condition. The evaluation process terminates with a $true/false$ answer. One can show that our algorithm is both complete and sound with a properly chosen search depth (as the ones given in this paper). Basically only theoretic results on the approach are presented in this paper, and in the future we plan to continue investigating the following issues that are important to the implementation of our approach.

- Symbolic Algorithms. The algorithms presented in this paper are essentially explicit state-space searches, which may not scale well to large systems. So it would be interesting our approach can be implemented with symbolic algorithms.

- Scalability. Another issue concerning the scalability of our approach is the choice of the search depth for the generations test sequences. In practice we could sacrifice the completeness of the algorithm by choosing a smaller search depth.
- More Complex Models. The system model considered in this paper is rather restricted. At the present, we are working to extend our approach to more complex system models that allow multiple unspecified components, asynchronous communications between unspecified components and the host system as well as among unspecified components, and unspecified components with an infinite state space.

6. REFERENCES

[1] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *CAV'98*, volume 1427 of *LNCS*, pages 521–525. Springer, 1998.

[2] P. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *ICFEM'98*, pages 46–. IEEE Computer Society, 1998.

[3] B. Balzer. Living with cots. In *ICSE'02*, pages 5–5. ACM Press, 2002.

[4] A. Bertolino and A. Polini. A framework for component deployment testing. In *ICSE'03*, pages 221–231. IEEE Computer Society, 2003.

[5] P. E. Black, V. Okun, and Y. Yesha. Mutation operators for specifications. In *ASE'00*, pages 81–. IEEE Computer Society, 2000.

[6] A. Brown and K. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, Sep/Oct 1998.

[7] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model checking. In *SPIN'96*, 1996.

[8] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *ICSE'03*, pages 385–395. IEEE Computer Society Press, 2003.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[10] L. de Alfaro and T. A. Henzinger. Interface automata. In *ASE'01*. ACM Press, 2001.

[11] J. Dingel. Computer-assisted assume/guarantee reasoning with verisoft. In *ICSE'03*, pages 138–148. IEEE Computer Society Press, 2003.

[12] A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *TACAS'97*, volume 1217 of *LNCS*, pages 384–398. Springer, 1997.

[13] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *FSE'01*, pages 152–163. ACM Press, 2001.

[14] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/FSE'99*, volume 1687 of *LNCS*, pages 146–163. Springer, 1999.

[15] D. Giannakopoulou, C. S. Pasareanu, and J. M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *ICSE'04*, pages 211–220. IEEE Press, 2004.

[16] D. Giannakopoulou, C. S. Psreanu, and H. Barringer. Assumption generation for software component verification. In *ASE'02*, pages 3–13. IEEE Computer Society, 2002.

[17] I. Gorton and A. Liu. Software component quality assessment in practice: successes and practical impediments. In *ICSE'02*, pages 555–558. ACM Press, 2002.

[18] A. Groce, D. Peled, and M. Yannakakis. Amc: An adaptive model checker. In *CAV'02*, volume 2404 of *LNCS*, pages 521–525. Springer, 2002.

[19] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16:843–872, 1994.

[20] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 1998.

[21] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special Issue: Formal Methods in Software Practice.

[22] W. Kozaczynski and G. Booch. Component-based software engineering. *IEEE Software*, 15(5):34–36, Sep/Oct 1998.

[23] O. Kupferman and M. Vardi. Module checking revisited. In *CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 1997.

[24] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[25] H. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. *ACM SIGSOFT Software Engineering Notes*, 27(6):89–98, 2002.

[26] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *6th ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[27] B. Meyer. The grand challenge of trusted components. In *ICSE'03*, pages 660–667. IEEE Computer Society Press, 2003.

[28] A. Orso, M. J. Harrold, and D. Rosenblum. Component metadata for software engineering tasks. volume 1999 of *LNCS*, pages 129–144, 2001.

[29] C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *SPIN*, pages 168–183, 1999.

[30] D. Peled. Model checking and testing combined. In *ICALP'03*, volume 2719 of *LNCS*, pages 47–63. Springer, 2003.

[31] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *FORTE/PSTV'99*, pages 225–240. Kluwer, 1999.

[32] C. Szyperski. Component technology: what, where, and how? In *ICSE'03*, pages 684–693. IEEE Computer Society, 2003.

[33] B. A. Trakhtenbrot and Y. M. Barzdin. *Finite automata; behavior and synthesis*. North-Holland Pub. Co., 1973.

[34] J. Voas. Certifying off-the-shelf software components. *IEEE Computer*, 31(6):53–59, June 1998.

[35] J. Voas. Developing a usage-based software certification process. *IEEE Computer*, 33(8):32–37, August 2000.

[36] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA'02*, pages 218–228. ACM Press, 2002.

[37] F. Xie and J. C. Browne. Verified systems by composition from verified components. In *FSE'03*, pages 277–286. ACM Press, 2003.

[38] G. Xie and Z. Dang. An automata-theoretic approach for model-checking systems with unspecified components. In *FATES'04*, LNCS. Springer, to appear.

Automatic Extraction of Sliced Object State Machines for Component Interfaces

Tao Xie David Notkin

Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195, USA
{taoxie,notkin}@cs.washington.edu

ABSTRACT

Component-based software development has increasingly gained popularity in industry. Although correct component-interface usage is critical for successful understanding, testing, and reuse of components, interface usage is rarely specified formally in practice. To tackle this problem, we automatically extract sliced object state machines (OSM) for component interfaces from the execution of generated tests. Given a component such as a Java class, we generate a set of tests to exercise the component and collect the concrete object states exercised by the tests. Because the number of exercised concrete object states and transitions among these states could be too large to be useful for inspection, we slice concrete object states by each member field of the component and use sliced states to construct a set of sliced OSM's. These sliced OSM's provide useful state-transition information for helping understand behavior of component interfaces and also have potential for being used in component verification and testing.

1. INTRODUCTION

Component-based software development has become an emerging discipline that manages the growing complexity of software systems [18]. In component-based software development, software components are the building blocks of a software system. When component users try to reuse an existing component in their applications, they need to understand behavior of the component's interface, such as usage rules that they are required to obey or expected results of some component usage scenarios. When component developers or users test their components before being released or reused, they need to know whether their components behave correctly against some usage rules or expectations. However, in practice, component-interface-usage rules or behavioral specifications are usually not equipped for many components. Even if usage rules or behavioral specifications are provided, they are often informally written in interface documentation such as Java API documentation [17], being prone to errors or difficult to be understood.

In this work, among a variety of specifications, we propose to use the form of *object state machines* (OSM) to characterize behavior of component interfaces and dynamically extract OSMs from automatically generated tests for component interfaces. We have proposed OSM in our previous work [27]. A state in an OSM represents the state that a component object is in at runtime. A transition in an OSM represents method calls invoked through the component interface transiting the component object from one state to another. States in an OSM can be concrete or abstract. A concrete state of a component object is characterized by the values of all transitively reachable fields of the component object. A concrete OSM is an OSM with concrete states. Given a component, we generate a set of tests for the component and then collect all exercised concrete states of component objects and transitions (method calls through component interfaces) among states. These collected states and transitions are used to construct a concrete OSM; however, the concrete OSM is often too complicated to be useful for understanding. To address this problem, our previous work has proposed the *observer abstraction* approach [27]; the approach uses the return values of observers (interface methods with non-void returns) invoked on a component object as an abstract state in an OSM. This paper proposes a new supplementary approach of slicing a concrete state by each member field of the component[1]. Different from our previous observer abstraction approach [27], our new approach is not affected by the availability or complexity of observers in component interfaces. Our state slicing technique is inspired by Whaley et al's model slicing by member fields in dynamically extracting component interfaces [22]; however, our new approach is more accurate in characterizing component behavior and does not require a good set of existing system tests for exercising component interfaces. In this work, we focus on components in the form of Java classes and component interfaces in the form of public methods in classes; however, we expect the approach could be easily extended to components in other forms.

The rest of this paper is organized as follows. Section 2 describes a nontrivial illustrative example. Section 3 introduces the formal definition of an OSM. Section 4 illustrates the automatic approach of extracting sliced OSM's. Section 5 discusses main issues of the approach and proposes future work. Section 6 presents related work and Section 7 concludes.

[1]We define state slicing or OSM slicing following the definition of model slicing by Whaley et al. [22]. The use of *slicing* in these definitions differs from the one in a more common definition: program slicing [21], which is closely related to some notion of dependence.

```
public class LinkedList extends AbstractSequentialList
     implements List, Cloneable, java.io.Serializable {
  private transient Entry header
                       = new Entry(null, null, null);
  private transient int size = 0;
  private static final long serialVersionUID
                  = 876323262645176354L;

  public LinkedList() {...}
  public void add(int index, MyInput element) {...}
  public boolean add(MyInput o) {...}
  public boolean addAll(int index, Collection c) {...}
  public void addFirst(MyInput o) {...}
  public void addLast(MyInput o) {...}
  public void clear() {...}
  public Object remove(int index) {...}
  public boolean remove(MyInput o) {...}
  public Object removeFirst() {...}
  public Object removeLast() {...}
  public Object set(int index, MyInput element) {...}
  public Object get(int index) {...}
  public ListIterator listIterator(intindex) {...}
  public Object getFirst() {...}
   ...
}
```

Figure 1: A LinkedList implementation

2. ILLUSTRATIVE EXAMPLE

As an illustrative example, we use a nontrivial data structure: a LinkedList class, which is the implementation of linked lists in the Java Collections Framework, being a part of the standard Java libraries [17]. Figure 1 shows declarations of LinkedList's fields and some public methods that we shall refer to in the rest of this paper (these public methods either modify object states or throw uncaught exceptions).[2] This implementation uses doubly-linked, circular lists that have a `size` field and a `header` field, which acts as a sentinel node. In addition, it also has a static `serialVersionUID` field, which is used during serialization. It inherits a `modCount` field from a super class `AbstractList`; this field records the number of times the list has been structurally modified. LinkedList has 25 public methods, 321 noncomment, non-blank lines of code, and 708 lines of code including comments and blank lines.

3. OBJECT STATE MACHINE

We have defined an object state machine for a component in our previous work [27]:

DEFINITION 1. *An* object state machine *(OSM) M of a component c is a sextuple M = $(I, O, S, \delta, \lambda, INIT)$ where I, O, and S are nonempty sets of method calls in c's interface, returns of these method calls, and states of c's objects, respectively. $INIT \in S$ is the initial state that the machine is in before calling any constructor method of c. $\delta : S \times I \rightarrow P(S)$ is the state transition function and $\lambda : S \times I \rightarrow P(O)$ is the output function where $P(S)$ and $P(O)$ are the power sets of S and O, respectively. When the machine is in a current state s and receives a method call i from I, it moves to one of the next states specified by $\delta(s, i)$ and produces one of the method returns given by $\lambda(s, i)$.*

When a method call in a component interface is executed, an uncaught exception might be thrown. To represent the state where an object is in after an exception-throwing method call, we introduce a special type of states in an OSM: *exception states*. After a method call on an object throws an uncaught exception, the object is in an exception state represented by the type name of the exception. The exception-throwing method call transits the object from the object state before the method call to the exception state.

[2]We change those `Object` argument types to `MyInput` so that we can guide ParaSoft Jtest 5.1 [15] (being used in our test generation described in Section 4.1) to generate better arguments; `MyInput` is a class that contains an integer field `v`.

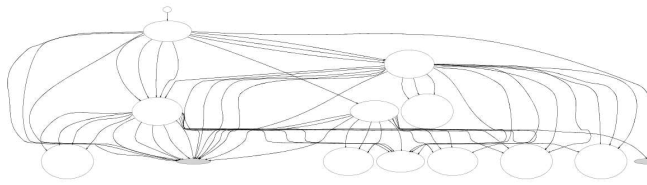


Figure 2: An overview of LinkedList concrete OSM (containing only state-modifying transitions) exercised by generated tests

An OSM can be deterministic or indeterministic. To help characterize indeterministic transitions, we have defined two statistics in a dynamically extracted OSM: transition counts and emission counts [27]. Assume a transition t transits state s to s', the *transition count* associated with t is the number of concrete states enclosed in s that are transited to s' by t. Assume m is the method call associated with t, the *emission count* associated with s and m is the number of concrete states enclosed in s and being at entries of m (but not necessarily being transited to s'). If the transition count of a transition is equal to the associated emission count, the transition is deterministic and indeterministic otherwise.

The object states in an OSM can be concrete or abstract. A concrete OSM is an OSM where all states are concrete object states. We have proposed several techniques to represent object states in our previous work [24]; we use the WholeState technique to represent concrete object states in this work. Given an object, the WholeState technique collects the values of all fields reachable from the object and uses these field values to represent the concrete state of the object. When we encounter a reference-type field with a non-null value during field-value collection, we use a linearization algorithm [24] to collect the field value as the field name of the earliest collected aliased field; if we cannot find any earlier collected aliased field for the field, we collect its value as "not_null". Two concrete object states are nonequivalent if their representations are different. A set of nonequivalent concrete object states contain concrete object states any two of which are nonequivalent.

For example, there are 11 nonequivalent concrete object states of LinkedList exercised by tests generated in our test generation step (Section 4.1). There are 161 transitions among these states (including both state-modifying and state-preserving transitions). There are two exception states: `IndexOutOfBoundsException` and `NoSuchElementException`. Figure 2 shows a concrete OSM (containing only state-modifying transitions) exercised by generated tests. [3] We have observed that the concrete OSM is too complex to be useful for inspection.

To reduce the complexity of an OSM, we shall extract an abstract OSM containing abstract states instead of concrete states. An *abstract state* of an object is defined by an *abstraction function* [14]; the abstraction function maps each concrete state to an abstract state. In this work, for each member field of a component, we define an abstraction function that maps each concrete state to an abstract state characterized by the values of those fields reachable from the member field. The next section describes the details of the state slicing approach.

[3]We display OSM's by using the Grappa package, which is part of graphviz [9].

4. SLICED-OSM EXTRACTION

Given a Java class, we automatically generate a set of tests for extensively exercising object states within a (small) scope (Section 4.1). During the execution of the generated tests, we slice each exercised concrete object state by member fields and construct abstract OSM's (Section 4.2). For a member field with a reference type, we additionally conduct structural abstraction on the sliced state to further abstract primitive field values reachable from the member field (Section 4.3).

4.1 Test Generation

Given a Java class, we first use Parasoft Jtest 5.1 [15] (a commercial Java testing tool) to generate method arguments for each public method of the class. Jtest generates a small set of method arguments and invoke public methods with these arguments after invoking class constructors. For example, Jtest 5.1 generates two tests for exercising `add(MyInput element)`:

```
Test 1:
        MyInput t0 = new MyInput(0);
        LinkedList THIS = new LinkedList();
        boolean RETVAL = THIS.add(t0);
Test 2:
        MyInput t0 = new MyInput(7);
        LinkedList THIS = new LinkedList();
        boolean RETVAL = THIS.add(t0);
```

Jtest also allows the user to configure whether to generate null values as method arguments. For the sake of simplicity in illustrative results, we configure Jtest 5.1 not to generate null argument values for LinkedList.

A list of arguments for a method consists of all arguments required for invoking the method. Two lists of arguments for a method are equivalent if the concrete state of each argument in the first list is equivalent to the concrete state of the corresponding argument in the second list. If an argument is of a primitive type, its concrete state is represented by its primitive values. If an argument is of Java built-in `String`, `Integer`, or another primitive-type wrapper, the concrete state of the argument is represented by its character strings or corresponding primitive value. If arguments are of other reference types, we use the WholeState technique (described in Section 3) for comparing their state equivalence.

We use the Rostra tool (developed in our previous work [23, 24]) to monitor the execution of the test class generated by Jtest and generate new tests based on collected method arguments. The pseudo-code of our test-generation algorithm is presented in Figure 3 (adapted from our previous work [23]). The test generation algorithm receives a set of third-party generated tests (e.g. Jtest-generated tests) and a maximum iteration number that specifies how many iterations we shall use to grow concrete object states. We first run these third-party generated tests and collect run time information from their execution; the collected runtime information includes the set of all nonequivalent non-constructor-method argument lists and nonequivalent object states exercised during the execution.

Then in the first iteration, the frontier set (containing the object states to be fully exercised) includes those nonequivalent states at exits of constructors exercised by the third-party tests. We iterate each object state in the frontier set and each argument list in the set of nonequivalent non-constructor-method argument lists exercised by the third-party tests. For each combination of an object state and an argument list, we construct a test by invoking the corresponding method with the argument list on the object state. We execute all constructed tests and collect runtime information. In the subsequent iteration, the frontier set includes those nonequivalent states exercised by the new tests but not exercised by any test in previous iterations. We continue the iterations until we have reached the maximum iteration number or the frontier set contains no object states.

```
Set testgen(Set thirdPartyTests, int maxIterNum) {
  Set newTests = new Set();
  RuntimeInfo runtimeInfo = runAndCollect(thirdPartyTests);
  Set nonEqArgLists = runtimeInfo.getNonEqArgsLists();
  Set frontiers = runtimeInfo.getAfterInitNonEqObjStates();
  for(int i=1;i<=maxIterNum && frontiers.size()>0;i++) {
      Set newTestsForCurIter = new Set();
      foreach (objState in frontiers) {
        foreach (args in nonEqArgLists) {
          Test newTest = makeTest(objState, args);
          newTestsForCurIter.add(newTest);
          newTests.add(newTest);
        }
      }
      runtimeInfo = runAndCollect(newTestsForCurIter);
      frontiers = runtimeInfo.getNewNonEqObjStates().
  }
  return newTests;
}
```

Figure 3: Pseudo-code of the test-generation algorithm.

For the LinkedList example, we configure the maximum iteration number as two. For illustration purpose, let us assume here that third-party tests contain only two tests (Tests 1 and 2) that we have shown in the beginning of this section. Then in the first iteration, we generate Tests 1 and 2; in the second iteration, we generate Tests 3 and 4 shown as below:

```
Test 3:
        MyInput t0 = new MyInput(0);
        LinkedList THIS = new LinkedList();
        boolean RETVAL = THIS.add(t0);
        MyInput t1 = new MyInput(7);
        boolean RETVAL1 = THIS.add(t1);
Test 4:
        MyInput t0 = new MyInput(7);
        LinkedList THIS = new LinkedList();
        boolean RETVAL = THIS.add(t0);
        MyInput t1 = new MyInput(0);
        boolean RETVAL1 = THIS.add(t1);
```

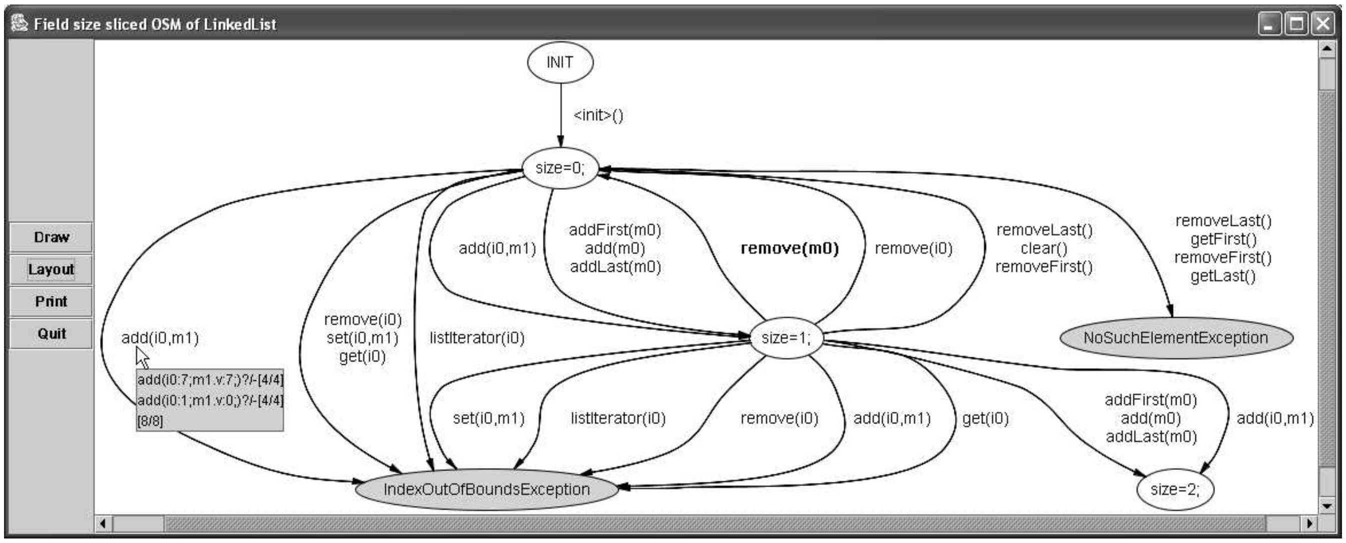
4.2 State Slicing

Given a concrete state and a member field of the class, we produce an abstract state represented by the value of the member field and the values of all those fields reachable from the member field if the member field is of a reference type. For example, in the end of Tests 1 and 2, the `THIS` object's concrete states are represented by the following object-field values:

```
Concrete object state at the end of Test 1:
 size=1;
 modCount=1;
 serialVersionUID=876323262645176354;
 header.element=null;
 header.next.element.v=0;
 header.next.next=header;
 header.next.previous=header;
 header.previous=header.next;

Concrete object state at the end of Test 2:
 size=1;
 modCount=1;
 serialVersionUID=876323262645176354;
 header.element=null;
 header.next.element.v=7;
 header.next.next=header;
 header.next.previous=header;
 header.previous=header.next;
```

When we slice these concrete object states by the `size` field, both abstract-state representations are "`size=1;`" and these two nonequivalent concrete states are mapped to the same abstract state. After we generate abstract states at the entry and exit of a method call, we generate a transition (characterized by the method call) from



```
add(i0:7;m1.v:7;)?/-[4/4]
add(i0:1;m1.v:0;)?/-[4/4]
[8/8]
```

Figure 4: A LinkedList OSM sliced by the `size` field

the abstract state at the method entry to the abstract state at the method exit. Then we can construct an abstract OSM from test executions. Figure 4 shows a LinkedList OSM sliced by the `size` field (displaying also exception states and transitions to them). Figure 5 shows a LinkedList OSM sliced by the `modcount` field (without displaying exception states or transitions to them). [4] We allow the user to configure whether to display exception states and transitions to them in a sliced OSM. By default, we do not display state-preserving transitions in a sliced OSM in order to present a succinct view. In Figure 4, the transition starting from the top "`INIT`" state is marked with `<init>()`, which represents a constructor call. In general, each transition edge in an OSM is marked with a simplified representation of the method name and signature that correspond to the method calls of the transition. When there are multiple nonequivalent argument lists of the same method transiting one state to another, we group them into one single transition edge. This grouping mechanism can be viewed as a form of abstraction on transitions. When the user move the mouse cursor over the edge, the details of method calls are displayed. For example, the leftmost edge in Figure 4 shows the simplified method name and signature for `add(`**`int`**` index, MyInput element)`: `add(i0, m1)`, where each parameter is represented as the combination of the first letter of its type name and its parameter order (starting from 0). The details of method calls in this left-most transition are:

```
add(i0:7;m1.v:7;)?/-[4/4]
add(i0:1;m1.v:0;)?/-[4/4]
[8/8]
```

where `m1.v` represents the `v` field of the second argument, argument values or argument's field values are shown following their argument names or argument's field names separated by "`:`", and different arguments or fields are separated by "`;`". For succinctness, we do not display the "not_null" value for a non-null reference-type field ("not_null" assignments are described in Section 3). A line of description for method calls is in the form of $m?/mr![tc/ec]$ where m is the method call name and argument values, mr is the return value if any (if a return is void or the method call throws an exception, we display the return value as "–" and we do not display "!"), tc is the transition count, and ec is the emission count (the descriptions of transition counts and emission counts are described in Section 3). In the bottom line of the detailed description, we summarize the total number of transition counts and emission counts for all the method calls in the transition. When the method calls in the transition exercise all existing argument lists for the method, we additionally display "ALL_ARG", such as in the details for a `remove(m0)` in Figure 5. To present a more succinct view, we group calls of different methods with the same starting state and ending state into a single transition edge if these method calls satisfy the following two properties: (1) the calls of each method exercise all existing argument lists for the method (displayed with "ALL_ARG"); (2) the calls of each method are deterministic (their transition counts are equal to their emission counts). For indeterministic transitions, we highlight their simplified method names and signatures in bold font. For example, one edge of `remove(m0)` is highlighted in central Figure 4. This indeterminism indicates that invoking `remove(m0)` on a linked list containing one element does not necessarily make the linked list empty. For example, one such case is to remove an element with the value of 0 from a linked list containing an element with the value of 7.

Extracted sliced OSM's provide succinct views for summarizing interesting state-transition behavior exhibited by a component. For example, by inspecting and exploring Figure 4, we can conveniently understand the conditions of throwing uncaught exceptions, which often indicate the sequencing constraints of using a component. For example, an `IndexOutOfBoundsException` is thrown when invoking `get(i0)` immediately after invoking a constructor. Previous research in inferring sequencing constraints [1, 22, 28] could be effective in inferring this simple constraint but might not be able to infer more complex constraints extracted by our approach. One such a complex constraint is that if we invoke a constructor, `add(m0)`, `removeLast()`, and finally `get(i0)`, an `IndexOutOfBoundsException` is thrown. The reasons are that previous research in inferring sequencing constraints does not consider the internal states of a component but only the sequence order among method calls invoked through a component interface.

By looking into the details of those transitions leading to the `IndexOutOfBoundsException` state, we can understand that if a method argument is an integer index to a linked list, it shall generally fall into the scope between zero and the size of the list. But

[4]We do not show the LinkedList OSM sliced by the `serialVersionUID` field in this paper because the class does not modify `serialVersionUID` and the extracted OSM is trivial.

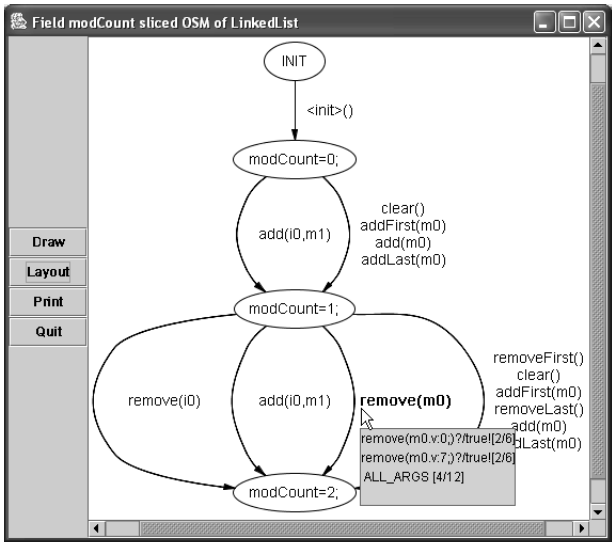


Figure 5: A LinkedList OSM sliced by the `modCount` field

one difference has caught our attention: `add(i0, m1)` in the left-most of Figure 4 is not grouped with other method calls with index arguments on the second-to-leftmost edge of Figure 4, such as `remove(i0)` and `set(i0, m1)`; this indicates that all argument lists for methods on the second-to-leftmost edge lead the "`size=0;`" state to the "`IndexOutOfBoundsException`" state, but not all argument lists for `add(i0, m1)` lead to the exception state. By inspecting their details, we found that, to avoid the exception, the `i0` argument for `add(i0, m1)` should satisfy `(0 <= i0 && i0 <= size())` but the `i0` argument for the methods on the second-to-leftmost edge should satisfy `(0 <= i0 && i0 < size())`. We also found that `listIterator(i0)` needs to satisfy the same constraint as `add(i0, m1)`. We have confirmed these small distinctions among exception-throwing conditions by browsing Java API documentation [17].

4.3 Structural Abstraction

When we slice two concrete object states in the end of Tests 1 and 2 by the `header` field, these two nonequivalent concrete object states are still mapped to two different abstract states. After we slice all exercised concrete object states by the `header` field, we reduce 11 concrete object states to 7 abstract states, whose corresponding OSM is still complex. Inspired by Korat's object graph isomorphism [3], we conduct *structural abstraction* by keeping only structural information among object fields but ignoring those primitive field values in a sliced state. The underlying rationale for this technique is that object states sharing the same object graph structure often exhibit certain common behavior. For example, after we apply structural abstraction on `header`-sliced states in the end of Tests 1 and 2, we produce the same abstract state as below:

```
header.element=null;
header.next.element.v=-;
header.next.next=header;
header.next.previous=header;
header.previous=header.next;
```

In the representation of abstract states, we replace all field values of primitive types with "–". In fact, we have found that the generated abstract states have a one-to-one correspondence with the states sliced by the `size` field. For example, the `header`-sliced state after structural abstraction in the end of Tests 1 and 2 corresponds to the "`size=1;`" state. Figure 6 shows a LinkedList OSM sliced by the `header` field after structural abstraction (without display-

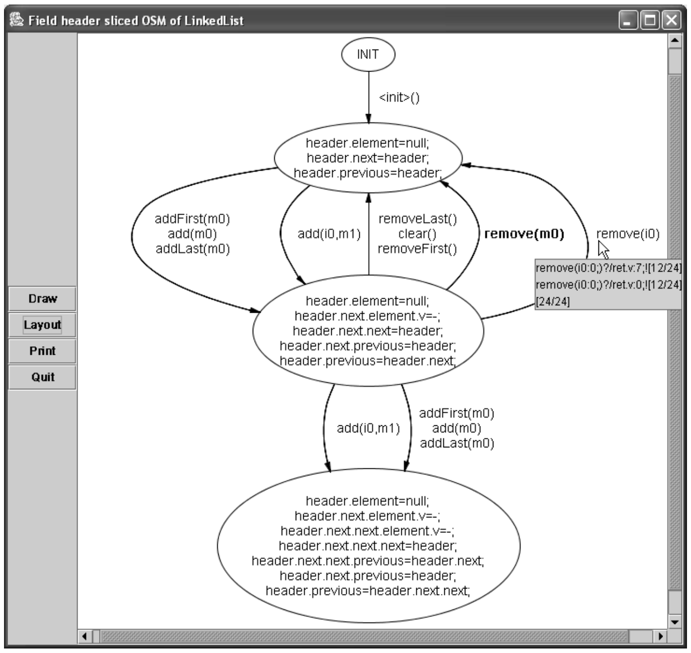


Figure 6: A LinkedList OSM sliced by the `header` field after structural abstraction

ing exception states or transitions to them). This OSM is especially useful for another implementation of a linked list that does not have a `size` field but computes the size on the fly from the `header` field when the size's value is needed. For other data structures such as a binary tree, one `size`-sliced abstract state might map to more than one sentinel-node-sliced abstract states after structural abstraction.

5. DISCUSSION AND FUTURE WORK

There are two main factors that affect our approach's usability in practice: member fields and generated tests. In our approach, member fields take the role of abstraction functions [14], which are used to specify state abstractions. In addition, like other dynamic inference techniques [1, 7, 11, 22, 27, 28], the quality or complexity of an extracted sliced OSM depends on the executed tests besides the characteristics of the used member field. Section 5.1 and 5.2 further discuss the factors of member fields and generated tests, respectively. Section 5.3 discusses other potential applications of our approach than the task of understanding component behavior.

5.1 Member Fields

Our approach uses a single member field as an abstraction function: different concrete states with the same value for the member field are abstracted to the same abstract state. Although we construct a sliced OSM for each member field, we might abstract away some aspects of concrete states that are central in understanding the behavior of a method in a sliced OSM. For example, in some classes, some member fields might be closely coupled and we might prefer to slice states by multiple member fields instead of a single member field. To provide tool supports for these cases, we can categorize member fields into groups based on field-access patterns by member methods using concept analysis [5]. Then we can slice states by these field groups and use sliced states to construct sliced OSM's. On the other hand, the state abstraction based on state slicing might not be high level enough; therefore, the resulting OSM's might be still too complicated for inspection.

In some cases, it might be difficult to infer a good abstraction function from the code itself by using various heuristics. Then in order to get satisfactory OSM's, we might need human inputs for defining indistinguishability properties [10] or other forms of abstraction functions to further abstract states. We expect that this way of getting human inputs in our approach shall be better for many types of programs than requiring upfront human inputs in traditional formal methods. First, we expect that programmers would be more willing to provide their inputs of abstraction functions after they have already seen OSM's extracted without their upfront inputs (some OSM's could have already been useful for them to understand parts of the component behavior). Second, we expect that it would be easier for programmers to formulate abstraction functions based on the crude OSM's extracted by our approach.

5.2 Generated Tests

There are two controllable configurations on the tests generated by our approach: method arguments and the maximum iteration number. When we use another third-party tool to generate more method arguments for a method but keep the same maximum iteration number as two, the sliced OSM's for LinkedList in Figure 4, 5, and 6 would be kept mostly the same (details associated with transitions might grow though) but the `header`-sliced OSM before structural abstraction would grow rapidly. When we keep the same method arguments but increase the maximum iteration number, the sliced OSM's in Figure 4, 5, and 6 would grow linearly. For example, in Figure 4, there will be new transitions starting from the bottom-right "`size=2;`" state similar to the ones starting from the "`size=1;`" state. In general, when there are more method arguments or higher maximum iteration numbers, the space of both concrete states and sliced states could grow. To address the scalability of the approach, programmers can configure fewer method arguments or lower maximum iteration numbers, or specify user-defined abstraction functions to further abstract states (discussed in Section 5.1).

If the generated tests used for OSM extraction are not of good quality, the quality of extracted sliced OSM's can be compromised. Static analysis techniques can be used to identify some insufficiency of generated tests for extracting sliced OSM's. For example, because Jtest 5.1 generates only an empty collection argument for `addAll(int index, Collection c)`, the `addAll` method is dynamically identified as a state-preserving method for all extracted sliced OSM's. Existing static techniques for method-purity analysis [2,16] can identify `addAll` not to be state preserving; then we can augment Jtest-generated tests with non-empty-collection arguments for `addAll`.

5.3 Other Applications

Although in this paper we primarily investigate the extraction of sliced OSM's to help understand component behavior, there are other promising applications of extracted OSM's. For example, we can extract sliced OSM's from existing generated tests to ease the task of test inspection. We can use extracted OSM's to guide test generation using existing finite-state-machine-based testing techniques [13], use new generated tests to further improve extracted OSM's, and then use new improved OSM's to generate more new tests and so forth. During iterations, any new generated tests violating existing inferred properties (e.g. OSM's) can be selected for inspection [26]. These iterations form a feedback loop between test generation and specification inference proposed in our previous work [25].

We can apply sliced OSM's in testing and verification by extrapolating unseen states and transitions based on observed states and transitions. Then the prescribed component behavior is not limited to observed one. For example, in Figure 4, we can predict the structure of transitions around the unseen "`size=3;`" state or other unseen states.

After we have extrapolated initial sliced OSM's, we can perform conformance checking between OSM's and the implementation, which is similar to conformance checking between abstract state machines and an implementation [8]. We can also explore ways of translating properties captured by OSM's to the forms understood by existing software model checking tools [4, 20] and use existing tools to verify programs against their extracted OSM's. Note that finding counterexamples does not necessarily expose bugs in programs but might expose insufficiency of originally generated tests for OSM extraction. These counterexamples can help generate new tests to augment existing generated tests.

Because we extract sliced OSM's from an implementation, if the implementation is faulty and the initial sliced OSM's exhibit wrong behavior, we might not expose faults by performing conformance checking between OSM's and the implementation. Therefore, before we extrapolate initial sliced OSM's, we might prefer human inspection on the initial sliced OSM's to make sure that the initial sliced OSM's exhibit expected behavior.

6. RELATED WORK

Our previous work develops the observer abstraction approach for extracting OSM's (called observer abstractions) from unit-test executions [27]. The observer abstraction approach uses the return values of observers invoked on a concrete object state as abstract state representation, whereas our new approach in this paper uses the values of a member field in a concrete object state as abstract state representation. Unlike the observer abstraction approach, our new approach does not require the availability of (good) observers. The complexity of an observer abstraction depends on the characteristics of its corresponding observers, whereas the complexity of a sliced OSM depends on the characteristics of its corresponding member field. Observer abstractions help investigate behavior related to the return values of observers and this type of behavior is not explored in our new approach. In the LinkedList example, in contrast to four sliced OSM's generated by our new approach, the observer abstraction approach generates 18 observer abstractions. One observer is `int size()`; therefore, the extracted `size()` observer abstraction is exactly the same as our `size`-sliced OSM.

From system-test executions, Whaley et al. dynamically extract Java component-interface models, each of which accesses the same field [22]. They statically determine whether a method is a state-modifying one. In their extracted models, they assume that the same state-modifying method transits an object to the same abstract state. This assumption makes the extracted models less accurate than our approach. Ammons et al. mine protocol specifications in the form of a finite state machine from system-test executions [1]. Although their approach uses data dependence to extract relevant API method calls, it does not use component internal states but use the sequence order among API method calls for learning models. Both Whaley et al. and Ammons et al.'s approaches usually require a good set of system tests for exercising component interfaces, whereas our approach receives a given component and generates a set of tests to exercise component's object states in a small scope. Because their approaches do not consider object state information but just sequence order among API method calls, applying Whaley et al.'s approach on our generated unit tests would yield a complete graph of methods that modify the same object field and applying Ammons et al.'s approach on our generated unit tests would yield a complete graph of all methods in the component interface.

Yang and Evans infer temporal properties in the form of the strictest pattern any two methods can have in execution traces [28]. Similar to Whaley et al. and Ammons et al.'s approaches, their approach considers only sequence order among method calls without considering internal states of a component, whereas our approach use sliced states to construct OSM's, which encoded more accurate sequencing constraints. In addition, their approach considers sequencing relationship between two methods, whereas our approach considers state-transition relationship among multiple methods.

Ernst et al. develop Daikon to dynamically infer likely invariants from test executions [7]. These invariants describe the observed relationships among the values of object fields, arguments, and returns of a single method in a component interface, whereas our sliced OSM's describe state-transition relationships among multiple methods in a component interface and use the values of fields reachable from a member field to represent object states. Henkel and Diwan discover algebraic specifications from the execution of automatically generated unit tests [11]. Their discovered algebraic specifications usually present a local view of relationships between two methods, whereas our sliced OSM's present a global view of relationships among multiple methods.

Corbett et al. develop Bandera to extract finite-state models from Java source code for model checking [4]. Given a property, Bandera's slicing component removes control points, variables, and data structures that are irrelevant for checking the property. For each member field of a component, our approach dynamically slices object states that are reachable from the member field and constructs a sliced OSM. Given a definition of an abstraction, Bandera's abstraction-based specializer transforms the source code into a specialized version by replacing concrete operations and tests on relevant concrete data with abstracted versions on abstract values. Our approach conducts structural abstraction on a sliced state by mapping all primitive values in the state to the same abstract value.

Grieskamp et al. allow the user to define indistinguishability properties to group infinite states in abstract state machines into equivalence classes, called hyperstates [10]. Their tool incrementally produces finite state machines by executing abstract state machines. Our approach use the values of a member field to group concrete object states into abstract states in a sliced OSM.

Kung et al. statically extract object state models from class source code and use them to guide test generation [12]. An object state model is in the form of a finite state machine: the states are defined by value intervals over object fields, which are derived from path conditions of method source; the transitions are derived by symbolically executing methods. Our approach dynamically extracts sliced OSM's from test executions and supports a much wider range of classes than Kung et al's approach. For example, Kung et al.'s approach could not extract any state models for the `header` field because `header`'s values cannot be characterized by value intervals, which are usually applicable for primitive numeric fields. Their approach could not extract any model for the `modeCount` field because there is no usable path condition for this integer field in the source code. Because of the code complexity, their approach would have difficulties in symbolically deriving transitions for the states extracted from the only path condition usable for their approach: `(size==0)`.

Turner and Robson use finite state machines to specify the behavior of a class [19]. The states in a state machine are defined by the values of a subset or complete set of object fields. The transitions are method names. Although both their specified finite state machines and our sliced OSM's are in a similar form, we automatically extract state machines from test executions, whereas they manually specify state machines for a class. Edwards develops an approach of generating tests based on flowgraphs extracted from a component's specifications [6]. A flowgraph is a directed graph where each node represents one method provided by the component and a directed edge from a node `n` to node `n'` represents the possibility that control may flow from `n` to `n'`. Our approach automatically extracts OSM's from test executions without requiring a priori specifications and our OSM's capture actual-state transition.

7. CONCLUSION

Lack of specifications for a component has posed the barrier to the reuse of the component in component-based software development. In this paper, we have proposed a new approach for automatically extracting sliced OSM's for component interfaces. Given a component such as a Java class, we generate a set of tests for the component. Then we slice exercised concrete object states by each member field of the component and construct OSM's based on the sliced states. These sliced OSM's provide useful state-transition information for inspection. These OSM's also have potential for component verification and testing.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback on an earlier version of this paper. This work was supported in part by the National Science Foundation under grant ITR 0086003. We acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

8. REFERENCES

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[2] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specifications. In *Proc. 6th Workshop on Formal Techniques for Java-like Programs*, June 2004.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *Proc. the 22nd International Conference on Software Engineering*, pages 439–448, 2000.

[5] U. Dekel and Y. Gil. Revealing class structure with concept lattices. In *Proc. 10th IEEE Working Conference on Reverse Engineering*, pages 353–365, 2003.

[6] S. H. Edwards. Black-box testing using flowgraphs: an experimental assessment of effectiveness and automation potential. *Software Testing, Verification and Reliability*, 10(4):249–262, 2000.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[8] Foundations of Software Engineering, Microsoft Research. Abstract state machine language. `http://research.microsoft.com/fse/AsmL`.

[9] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering.

Software: Practice and Experience, 30(11):1203–1233, Sept. 2000.

[10] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. International Symposium on Software Testing and Analysis*, pages 112–122, 2002.

[11] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. 17th European Conference on Object-Oriented Programming*, pages 431–456, 2003.

[12] D. Kung, N. Suchak, J. Gao, and P. Hsia. On object state testing. In *Proc. 18th International Computer Software and Applications Conference*, pages 222–227, 1994.

[13] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proc. The IEEE*, volume 84, pages 1090–1123, Aug. 1996.

[14] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[15] Parasoft. Jtest manuals version 5.1. Online manual, July 2004. `http://www.parasoft.com/`.

[16] A. Rountev. Precise identification of side-effect-free methods in Java. In *Proc. 20th IEEE International Conference on Software Maintenance*, pages 82–91, Sept. 2004.

[17] Sun Microsystems. Java 2 Platform, Standard Edition, v 1.4.2, API Specification. Online documentation, Nov. 2003. `http://java.sun.com/j2se/1.4.2/docs/api/`.

[18] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[19] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *Proc. International Conference on Software Maintenance*, pages 302–310, 1993.

[20] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, pages 3–12, 2000.

[21] M. Weiser. Program slicing. In *Proc. 5th International Conference on Software Engineering*, pages 439–449, 1981.

[22] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. the International Symposium on Software Testing and Analysis*, pages 218–228, 2002.

[23] T. Xie, D. Marinov, and D. Notkin. Improving generation of object-oriented test suites by avoiding redundant tests. Technical Report UW-CSE-04-01-05, University of Washington Department of Computer Science and Engineering, Seattle, WA, Jan. 2004.

[24] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.

[25] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software*, volume 2931 of *LNCS*, pages 60–69, 2003.

[26] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 40–48, 2003.

[27] T. Xie and D. Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proc. 6th International Conference on Formal Engineering Methods*, Nov. 2004.

[28] J. Yang and D. Evans. Dynamically inferring temporal properties. In *Proc. the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 23–28, 2004.

Formalizing Lightweight Verification of Software Component Composition

Stephen McCamant Michael D. Ernst
MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street
Cambridge, MA 02139 USA
smcc@csail.mit.edu, mernst@csail.mit.edu

ABSTRACT

Software errors often occur at the interfaces between separately developed components. Incompatibilities are an especially acute problem when upgrading software components, as new versions may be accidentally incompatible with old ones. As an inexpensive mechanism to detect many such problems, previous work proposed a technique that adapts methods from formal verification to use component abstractions that can be automatically generated from implementations. The technique reports, before performing the replacement or integrating the new component into a system, whether the upgrade might be problematic for that particular system. The technique is based on a rich model of components that support internal state, callbacks, and simultaneous upgrades of multiple components, and component abstractions may contain arbitrary logical properties including unbounded-state ones.

This paper motivates this (somewhat non-standard) approach to component verification. The paper also refines the formal model of components, provides a formal model of software system safety, gives an algorithm for constructing a consistency condition, proves that the algorithm's result guarantees system safety in the case of a single-component upgrade, and gives a proof outline of the algorithm's correctness in the case of an arbitrary upgrade.

1. INTRODUCTION

Previous work [12, 13] introduced a technique that seeks to identify unanticipated interactions among software components, before the components are actually integrated with one another. The technique compares the observed behavior of an old component to the observed behavior of a new component; it permits the upgrade only if the behaviors are compatible, for the way that the component is used in an application. The technique issues a warning when the behaviors of the new and old components are incompatible, but lack of such a warning is not a guarantee of correctness, nor is its presence a guarantee that the program's operation would be incorrect. The technique constructs *operational abstractions*, mathematical statements syntactically similar to specifications that describe a component's behavior and its expectations about the behavior of other components. For a given system of components, the technique constructs a consistency condition that relates the expectations of one module to how they might be satisfied by the behaviors of others. This combination of the abstractions according to the consistency condition is then passed to an automatic theorem prover (our prototype uses Simplify [4]), and the upgrade is approved only if the consistency condition is verified to hold. We have used our implementation to find behavioral inconsistencies in large software systems — for instance, differences between versions in the behavior of the Linux C library, as used by desktop applications.

In the case of an upgrade to a single, purely functional module, the consistency condition that our technique checks is similar to the classic condition of behavioral subtyping relating procedure pre- and postconditions [1, 3]. For upgrades to more complex systems with arbitrary numbers of components, bidirectional interactions among them, and components with internal state, the consistency condition is more complicated. This work refines the multi-component system model from [13] and gives an improved algorithm for constructing a consistency condition.

In order to decide whether our changes to the algorithm are really improvements, we need a standard by which to judge our technique. The major new work described here is a formalization of the consistency checking problem. We can use this formalization to verify that consistency checks have desirable logical properties. Specifically, we wish to verify that the consistency condition is sound relative to the abstractions that describe the behavior of individual components. If an upgrade is approved by virtue of satisfying a consistency condition, and the behavioral abstractions that are related via that condition are safe approximations of the components' actual behavior, then the actual upgrade in question is *safe* — that is, the upgraded system satisfies specific properties that the original one did. The algorithm improvements we describe eliminate unsound aspects of our previous algorithm, and using them we describe a strategy for a relative soundness proof (though we have not yet completed the proof in all details).

The remainder of this paper is organized as follows. Section 2 compares our technique with other approaches to component-based verification. Section 3 describes a model of the structure of a multi-component systems. Section 4 formalizes a simplified version of the upgrade safety problem. Section 5 proves that the consistency condition we use in the simplest case of upgrading a single component is indeed sound. Section 6 proposes a general algorithm for constructing consistency conditions and gives a proof outline of the algorithm's correctness. Section 7 concludes.

2. COMPARISON WITH OTHER WORK

Many other researchers share our goal of making component-based development safer and more efficient, as well as the general approach of verifying that components will interact correctly based on an abstraction of their behavior. However, our approach takes as its starting point a somewhat atypical combination of four theses. We argue that the abstractions describing components:

- should be stated in an expressive language at the same abstraction level as concrete interfaces
- should describe concrete implementations and the way they are exercised by real test suites

- should be compiled and compared automatically
- need not be sound over arbitrary executions

The following subsections discuss these points in turn.

2.1 An expressive language of abstractions

The operational abstractions that our technique uses to represent a component's behavior are expressed as statements in a first order logic whose atomic statements can refer to the same concrete values that program statements can, and include the same primitive operators as the language itself. The statements can express the same sorts of properties that a programmer might consider important, for instance as might be checked in a conditional or assertion statement. Matching the tool's understanding to the developer's has two benefits. First, it helps the tool find properties that might be important for correctness. Second, when user interaction is required, such as after a potential incompatibility has been flagged, it makes it easier for the developer to understand the problem.

Much of the most important early work on specifications and their combination, such as behavioral subtyping [11] and the Vienna Development Method [10] used expressive specification languages similar to our operational abstractions. More recent work has seen a trend toward less expressive representations, especially finite state ones such as regular languages [16] or labeled transition systems [15]. Besides being more amenable to automatic checking, such representations also focus verification effort on a more limited set of properties, such as those related to global temporal ordering. Such approaches necessarily neglect aspects of correctness in a program's local behavior, and cannot even express non-finite-state properties involving integers or unbounded data structures.

2.2 Using real implementations

The operational abstractions used in our technique are different from formal specifications in that they describe software as it has actually been implemented, rather than as it is intended to perform. While formal specifications can be useful as part of the design of a system, or to assign responsibility for deviations from an interface, they are unavailable for most real systems. In particular, some of the most productive uses of formal specifications take advantage of the ability to describe a component at a high level of abstraction, so as to capture only the aspects of its behavior most important to a global architecture. While it is possible to describe the complete correctness conditions of a component at the level of concrete inputs and outputs in a formal specification, doing so is prohibitively expensive for any but the most critical systems.

By contrast, our technique's use of operational abstractions is intended to leverage the investments that developers already make in implementation and testing, and to discover potential problems that would affect actual system executions. We presume that the developers of individual system modules have checked locally to a module, informally and/or via unit testing, that those modules behave as intended. The job of our technique is to propagate the characterization of behavior embodied in such local checks and cross-check it for consistency with the expectations held by other separately developed modules in a large system. Unlike dynamic contract-checking [14, 7], our technique is intended to be used before system integration, rather than during execution, and discovers inconsistencies without assigning blame to one component or another.

2.3 Automatic generation and comparison

Our tool automatically derives operational abstractions from a component's implementation, as it is exercised by representative uses such as a test suite. This approach takes advantage of developer effort already expended in development and in choosing which aspects of behavior to test. Such derivation is of course not applicable to a specification-first methodology, but dynamically inferred properties are increasingly used for verification; in addition to the axiomatic-semantics style properties we use [6], other researchers have applied similar techniques to infer algebraic specifications [8] and temporal properties [2, 19].

Many component-based verification techniques use behavioral abstractions that can be compared automatically, at least for finite scopes, by conceptually simple techniques such as model checking or approaches based on finite automata. In theory, the unbounded-state properties that make up our operational abstractions are more difficult to operate on, with many operations in fact undecidable. However, we have not found our use of an automated theorem prover to be a major bottleneck in our technique, for two reasons. First, because propositional logic is a standard abstraction, we can treat the theorem prover as a black box, and ignore its internal complexities. Second, the properties we wish to check tend to be straightforward deductions from a general statement to a more specific one, involving only simple arithmetic and relations between variables. In previous work, a more common approach has been to combine human direction and a proof assistant tool [20]; this can increase assurance relative to a completely manual proof, but does not necessarily reduce the effort required. Schumann and Fischer use an automated theorem prover with some specialized preprocessing [18] to compare specifications for a procedure reuse application, but the space of examples they consider is quite small.

2.4 Soundness and precision

Operational abstractions describe a component's behavior in specific contexts, namely those in which the component was tested. Our approach does not require the operational abstractions to be sound as statements describing a component's execution in any context. Even if our technique had access to a sound description of a component's general behavior, we would still want it to separately record the contexts in which a component was used, to be able to verify that a system uses only tested behavior. The real limitation of our approach is that we cannot necessarily make this distinction between properties that are true in general and those that hold only in a restricted context.

At the same time we give up soundness, however, we gain a dual benefit of precision [5]. By virtue of their construction, every property (from a particular grammar) that fails to appear in an operational abstraction can be traced back to at least one concrete execution in which it was false, and any property that held over each observed execution will appear in the abstraction. In exchange for restricting our attention to specific system executions, we enjoy accurate information about them, avoiding the over-approximation that can come with approaches that are sound.

3. A MULTI-COMPONENT MODEL

This section describes a model of software systems that handles complicated situations that arise in object-oriented systems, such as components with state, components that make callbacks, or a simultaneous upgrade to two components that communicate via the rest of a system. This model differs from that of [13] in separating control flow from data flow in some situations. This separation allows a more precise determination of what parts of a system influence others (Section 6.1) and a sound treatment of data flow through non-local state (Section 6.2.4).

We consider systems to be divided into *modules* grouping together code that interacts closely and is developed as a unit. Such modules need not match the grouping imposed by language-level features such as classes or Java packages, but we assume that any upgrade affects one or more complete modules. Our approach to consistency checking is modular, but not simply compositional; it also summarizes each module's observations of the rest of a working system, and uses them as a basis for comparison with a proposed upgrade.

3.1 Relations inside and among modules

Given a decomposition of a system into modules, we model its behavior with three types of relations. *Call and return relations* represent how modules are connected by procedure calls and returns. *Internal flow relations* represent the behavior of individual modules, in context: that is, the way in which each output of the module potentially depends on the module's inputs. *External summary relations* represent a module's observations of the behavior of the rest of the system: how each input to the module might depend on the behavior of the rest of the system and any previous outputs of the module.

3.1.1 Call and return relations

Roughly speaking, each module is modeled as a black box, with certain inputs and outputs. When module A calls procedure `f` in module B, the arguments to `f` are outputs of A and inputs to B, while the return value and any side effects on the arguments are outputs from B and inputs to A. In the module containing a procedure `f`, we use the symbol f to refer to the input consisting of the values of the procedure's parameters on entrance, and f' to refer to the output consisting of the return value and possibly-modified reference parameters. We use f_c and f_r for the call to and return from a procedure in the calling module. Collectively, we call these moments of execution *program points*. All non-trivial computation occurs within modules: calls and returns simply represent the transfer of information unchanged from one module to another.

3.1.2 Internal flow relations

Internal flow relations connect each output of a module to all the inputs to that module that might affect the output value. In a module M, $M(v|u_1, \ldots, u_k)$ is the flow relation from inputs u_1 through u_k to an output v. In some cases, it is also helpful to decompose a flow relation into a number of *flow edges,* one connecting each input to the output. An *independent output* $M(v)$ is one whose value is not affected by any input to the module.

Conceptually, the flow relation is a set of tuples of values at the relevant inputs and at the output, having the property that on some execution of the output point, the output values might be those in the tuple, if the most recent values at all the inputs have their given values. Because each variable might have a large or infinite domain, it would be impractical or impossible to represent this relation by a table. Instead, our approach summarizes it by a set of logical formulas that are (observed to be) always true over the input and output variables. The values that satisfy these formulas are a superset of those that occurred in a particular run. This representation is *not* merely an implementation convenience. Generalization allows our technique to declare an upgrade compatible when its testing has been close enough to its use, without demanding that it be tested for every possible input.

Flow relations capture the behavior of a module primarily in terms of relations over data in variables. However, a limited characterization of a system's control flow is required to correctly combine facts from different relations. To this end, we further model flow edges as being of two types: those that represent control flow as well as data flow information, or *control-flow edges* for short, and *data-flow edges* that represent the flow of data not mediated by control (say, communication via a shared variable). To represent conditional control flow, control-flow edges include an additional fact called a *guarding condition*. A flow edge from an input u to an output v does not imply that every execution of u is followed by some execution of v: for instance, u might be the entry point of a procedure that calls another procedure at v under some circumstances but not others. A guarding condition ϕ_g is a property that held on executions of u that were followed by executions of v, but did not hold on executions of u that were followed by another execution of u without an intervening v. If the control flow is unconditional, ϕ_g is simply "true."

In order to facilitate analysis of a model, we impose the restriction that the subgraph consisting of control-flow, call and return edges has no cycles. This restriction forbids mutual recursion between procedures when the procedures appear in different modules, but not the use of recursive procedures in the implementation of a module. Note that procedure calls in both directions between a pair of modules are not restricted as long as there can be no cycle of procedure invocations in different modules; for instance, callbacks are allowed. See Section 6.2.3 for further discussion of why we impose this restriction.

3.1.3 External summary relations

External summary relations are in many ways dual to internal flow relations. Summary relations connect each input of a module to all of the module outputs that might feed back to that input via the rest of the system. In a module M, we refer to the summary relation from outputs u_1 through u_k to an input v as $\overline{M}(v|u_1, \ldots, u_k)$. As a degenerate case, an *independent input* $\overline{M}(v)$ is one not affected by any outputs. The line over the M is meant to suggest that while this relation is calculated with respect to the interface of M, it is really a fact about the complement of M — that is, all the other modules in the system.

4. FORMALIZING THE MODEL

The key properties of our technique depend on the relationship between the abstract model and the concrete behavior of real components, but reasoning about the full complexities of languages such as Java, Perl or C would be difficult. Instead, this section presents a very simple module-structured language, and describes the meaning of the system model for that language, as well as giving a precise notion of soundness for systems in that language. In the context of this formalization it is then possible to unambiguously discuss whether a consistency checking technique is sound.

To concentrate on the most important aspects of the consistency checking problem, the formalization also differs in two key ways from our actual technique. First, our real technique has a broad goal of preserving the correct behavior of a system after an upgrade, as well as ensuring that an upgraded system relies only on tested behavior. To unify these notions and make them precise, the formalized language includes assertion statements, and we say that an upgraded system is safe if no assertions fail. Second, our real technique characterizes behavior by generalizing from facts that were observed to be true over finitely many executions. In the formalization, we imagine that these generalizations are always sound, so that descriptions of a component's behavior will be true for any inputs, and descriptions of the conditions under which behavior is safe in fact guarantee safety for any inputs. Because our actual implementation lacks this soundness property, soundness results about the formalization correspond only to relative soundness properties of the real system: guarantees about safety are only as reliable as the operational abstractions on which they are based.

4.1 Language

The statements of our simplified programming language have the following grammar. (C stands for code, and D stands for dynamic program point, which is discussed in Section 4.2.)

$$\begin{array}{lll} C & ::= & C; C \mid v := E \mid \texttt{if}\ P\ \texttt{then}\ C\ \texttt{else}\ C \\ & & \mid v := M.f(v_1, ..., v_k) \mid D \\ D & ::= & M.f.\text{DPP}(\texttt{enter}\ M.f) \\ & & \mid M.f.\text{DPP}(\texttt{exit}\ M.f) \\ & & \mid M.f.\text{DPP}(\texttt{call to}\ M.f\ \#n) \\ & & \mid M.f.\text{DPP}(\texttt{return from}\ M.f\ \#n) \end{array}$$

Predicates P and side-effect-free terms E include variable references and an arbitrary set of function and predicate symbols, such as the integer operations $+$, $\times$, and $<$. M and f range over the names of modules and procedures respectively.

Procedure definitions have the form $M.f(v_1, \ldots, v_k) : C$. Their semantics are defined by a substitution that transforms a program into one without procedure calls. A call $v_r = M.f(\alpha_1, \ldots, \alpha_k)$ originally appearing in a procedure $M_2.f_2$ rewrites to

$$\begin{array}{l} M_2.f_2.\text{DPP}(\texttt{call to}\ M.f\ \#i); \\ v'_1 := \alpha_1; \\ \ldots \\ v'_k := \alpha_k \\ M.f.\text{DPP}(\texttt{enter}\ M.f); \\ C[v'_1/v_1] \cdots [v'_k/v_k][r'/return]; \\ M.f.\text{DPP}(\texttt{exit}\ M.f); \\ v_r := r'; \\ M_2.f_2.\text{DPP}(\texttt{return from}\ M.f\ \#i) \end{array}$$

where i and the primed variables are fresh. v_1 through v_k are called the parameter variables, and α_1 through α_k are the argument variables. The return value of a procedure is signified by a distinguished variable *return*. Recursion is prohibited, as are calls between procedures in the same module (which can be simulated with ahead-of-time inlining). Note that this restriction on recursion is stronger than the one imposed in the real implementation, which forbids only recursion between modules. Our formalized language has no iteration constructs, and is far from Turing-complete, but we believe that the complications of loop verification and nontermination are orthogonal to the questions we wish to address with the formalization, so we have chosen to omit them.

The parameters to a procedure, and *return*, are local to it, and cannot be mentioned outside the procedure. Additional locals can be obtained by declaring parameters and ignoring their values. All other variables are associated with a particular module, and can only be mentioned there. Each module has a special variable *fail* which is initially zero, but set to 1 if any assertion fails. It can be mentioned only via a special syntactic sugar `assert`(P) which is otherwise equivalent to `if` P `then` *fail* $:=$ *fail* `else` *fail* $:= 1$. (For brevity in examples, we will sometimes abbreviate *return* as r and combine expressions and procedure calls in single statements.)

Modules may refer to other modules by name. A system is a collection of modules with a distinguished `main` procedure in one of the modules, such that all named references to other modules in a module can be satisfied by other modules in the system. An execution of the system is an execution of the main procedure, including the expansions of all called procedures (transitively), in which the initial values of all the variables are arbitrary, except that all the special *fail* variables are initially zero.

The semantics of the language are the usual ones, given by the following small-step relation $\mapsto$, which maps code and a store to either new code and a new store, or just a new store to signify termination.

$$\langle v := E, s\rangle \mapsto s[v := s(E)] \qquad \text{[assign]}$$

$$\langle D, s\rangle \mapsto s \qquad \text{[ppt]}$$

$$\frac{\langle C_1, s\rangle \mapsto \langle C'_1, s'\rangle}{\langle C_1; C_2, s\rangle \mapsto \langle C'_1; C_2, s'\rangle} \qquad \text{[seqProgress]}$$

$$\frac{\langle C_1, s\rangle \mapsto s'}{\langle C_1; C_2, s\rangle \mapsto \langle C_2, s'\rangle} \qquad \text{[seqElim]}$$

$$\frac{s(P)}{\langle \texttt{if}\ P\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \mapsto \langle C_1, s\rangle} \qquad \text{[ifTrue]}$$

$$\frac{\neg s(P)}{\langle \texttt{if}\ P\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \mapsto \langle C_2, s\rangle} \qquad \text{[ifFalse]}$$

A system execution is safe for a module M if at the end of execution, the *fail* variable of module M is still zero. A module is safe in a system if every execution is safe for the module, and a system is safe if every module in it is safe.

4.2 Program points and relations

The execution of a dynamic program point D marks a moment of execution; it has no other runtime effect, and may not appear in the original program. The name of a dynamic program point gives an event (in parentheses) and the procedure where the event occurs (before the 'DPP'). The values at a dynamic program point are the current values of all in-scope variables when the point expression evaluates.

Static program points S abstract over dynamic program points; for any procedure g in module M_1:

Static	Dynamic
$M\colon f$	$M.f.\text{DPP}(\texttt{enter}\ M.f)$
$M\colon f'$	$M.f.\text{DPP}(\texttt{exit}\ M.f)$
$M_1\colon M_2.f_c$	$M_1.g.\text{DPP}(\texttt{call to}\ M_2.f)$
$M_1\colon M_2.f_r$	$M_1.g.\text{DPP}(\texttt{return from}\ M_2.f)$

Static program points $M\colon f$ and $M_1\colon M_2.f_r$ are called input program points, and $M\colon f'$ and $M_1\colon M_2.f_c$ are output program points.

A flow relation $M(S^{\text{o}}|S^{\text{i}}_1, \ldots, S^{\text{i}}_k)$ consists of an output program point S^{o} and zero or more input program points S^{i}_j, all belonging to a module M, along with a formula ψ over all the variables of the given program points. In addition, one or more edges from inputs S^{i}_j to the output may be control-flow relations, with associated guarding conditions ϕ_j. ψ and each ϕ_j are together required to be sound in the following sense:

For any system containing M and other modules, a dynamic instance of the flow relation is a dynamic program point corresponding to the output point, along with a dynamic program point corresponding to each static input point, such that no later dynamic point for the same input occurs before the dynamic output point. The flow relation holds over a dynamic instance if ψ holds over the values of its variables at the dynamic points. A flow relation is required to hold over any dynamic instance in any system containing the module, for any system inputs. In addition, for each control-flow edge, it must be the case that every instance of the input program point S^{i}_j at which the guarding condition ϕ_j holds is followed later in the execution order, without any intervening instances of the output program point, by an instance of the output program point such that S^{i}_j and the output point are part of an instance of the relation.

A summary relation $\overline{M}(S^{\text{i}}|S^{\text{o}}_1, \ldots, S^{\text{o}}_k)$ consists of an input program point S^{i} and zero or more output program points S^{o}_j, all belonging to a module M, along with a formula ψ over the variables at all of the given program points. When the name of a summary relation appears in a logical formula, it stands for the formula ψ. ψ is required to soundly assure safety in the following sense:

For any system containing M and other modules, a dynamic instance of the summary relation is a dynamic program point corresponding to the input point, along with a dynamic program point corresponding to each static output point, such that no later dynamic point for the same output occurs before the dynamic input point. The summary relation holds over a dynamic instance if ψ holds over the values of its variables at the dynamic points. The summary relations of a module are required to have the property that if in any system, they all hold on each of their dynamic instances for any system input, then that execution must be safe for that module.

The variables in the formulas of flow and summary relations are named so that every name is qualified by the static program point it corresponds to; thus relations can refer separately to the value of a program variable at different program points.

A call relation consists of a procedure call program point, a procedure entrance point for the called procedure, and a formula that states that each formal parameter variable is equal to the corresponding actual argument variable. When the name of a call relation appears in a formula, it stands for this conjunction of equalities. Similarly a return relation consists of a procedure exit program point, a procedure return point for the procedure that the exit is the exit from, and a formula stating that the value of the return in the procedure returned to is equal to the value returned by the exiting procedure. When the name of a return relation appears in a formula, it stands for this equality.

5. SAFETY FOR A SINGLE COMPONENT UPGRADE

To illustrate the use of the formalism developed in the previous section, consider the simple case of an upgrade to a module that provides a single procedure without visible side effects. We will give our technique's consistency condition for such a system, and prove that it is sound. This result is analogous to Example 5 of [3], which proves that a similar condition between specifications is "reuse-preserving," based on a relational semantics of specifications. Our proof is more involved, because it addresses more explicit details such as the passing of procedure arguments. Explicitly formalizing these notions becomes important for more complicated systems (for instance, if a procedure might have multiple callers).

Consider a system consisting of two modules U and L. Library L contains a single procedure f, and U contains a single procedure m, which makes one or more calls to f. Furthermore, assume that f makes no use of any module-wide variables (except of course for uses of *fail* in assertions). We call this the single-component upgrade case.

We can model the system with two flow relations, $U(f_c)$ and $L(f'|f)$, and two dual summary relations, $\overline{L}(f)$ and $\overline{U}(f_r|f_c)$. Since every call to the procedure returns, the flow edge from f to f' is a control-flow edge with guarding condition "true." As a concrete example, one might imagine that the procedure f increments its argument, and that U happens to call f only with even integers; then $U(f_c)$ might be "f_c is even", $L(f'|f)$ might be "$f'.r = f.x + 1$", $\overline{L}(f)$ might be"$f.x$ is an integer", and $\overline{U}(f_r|f_c)$ might be "$f_r.r = f_c.x + 1 \wedge f_r.r$ is odd", while C and R would be simply $f.x = f_c.x$ and $f_r.r = f'.r$.

PROPOSITION 1. *If*

$$(U(f_c) \wedge C) \Rightarrow \overline{L}(f)$$

and

$$(U(f_c) \wedge C \wedge L(f'|f) \wedge R) \Rightarrow \overline{U}(f_r|f_c),$$

where C and R are the call and return relations for the call to and return from f, then the system of U and L is safe.

PROOF. First, we will check that the system is safe for L. Since L has only one summary relation, $\overline{L}(f)$, it suffices to check that $\overline{L}(f)$, which is a formula over the parameters to f, holds at each dynamic entrance to f (recall that f uses no module-wide variables, so the parameters to f are the only relevant variables for the execution of f). Now, each dynamic occurrence of $L\colon f$ is immediately preceded, except for intervening assignments of arguments to parameters, by a dynamic occurrence of $U\colon L.f_c$, by the definition of procedure expansion. Because $U(f_c)$ is a flow relation for U, on any execution, $U(f_c)$ will hold at each dynamic execution of $U\colon L.f_c$. Furthermore, the assignments of f's arguments to its parameters assure that C will hold at the occurrence of $L\colon f$, and since the parameters are disjoint from the arguments, the formula of $U(f_c)$ will continue to hold at that point. Thus, the formula $U(f_c) \wedge C$ will hold at each instance of $L\colon f$. Thus by assumption, $\overline{L}(f)$ will hold at each instance of $L\colon f$, completing the proof that L is safe.

Next, we'll check that the system is safe for U. Since U has only one summary relation, $\overline{U}(f_r|f_c)$, it suffices to check that $\overline{U}(f_r|f_c)$, which is a formula over the value returned by f given the arguments to f, perhaps along with other variables in m, holds for the actual arguments, the return value, and those other variables, for each dynamic execution of the return. Consider any particular dynamic instance of $\overline{U}(f_r|f_c)$, consisting of a call $U\colon L.f_c$ and a return $U\colon L.f_r$. By the structure of the procedure expansion, the instance of $U\colon L.f_c$ must be followed by an assignment of arguments to parameters, and an instance of $L\colon f$. Similarly, the instance of $U\colon L.f_r$ must be immediately preceded by a return assignment, and before that an instance of $L\colon f'$. Since, by the definition of a summary relation, the instance of $U\colon L.f_c$ was the most recent prior to the instance of $U\colon L.f_r$, and because only U calls f, it must also be that the instance of $L\colon f$ is the most recent prior to the instance of $L\colon f'$; thus, the instances of $L\colon f$ and $L\colon f'$ are related by the flow relation $L(f'|f)$. In other words, we know that $L(f'|f)$ holds over the parameters to and the return value from this dynamic invocation of f. Furthermore, $U\colon L.f_c$ and $L\colon f$ are separated only by the assignment of arguments to parameters, so C holds as a relation between the arguments and the parameters, and similarly R holds as a relation between the copies of the return value at $L\colon f'$ and at $U\colon L.f_r$. Finally, $U(f_c)$ is a flow relation for U, so it must hold at the same point $U\colon L.f_c$. In summary, we see that $U(f_c) \wedge C \wedge L(f'|f) \wedge R$ holds over the parameters, arguments, and return value of f, so by the assumed safety condition, $\overline{U}(f_r|f_c)$ also holds over that dynamic execution. Since we picked an arbitrary execution, $\overline{U}(f_r|f_c)$ holds for each dynamic invocation on any input, completing the proof that U is safe. □

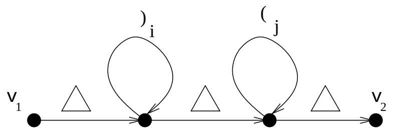
6. A MORE GENERAL CONDITION

The previous section described the consistency condition used by our technique when considering the simplest sort of component upgrade, and showed that it gave sound determinations of upgrade safety when used with sound abstractions of individual components. This section describes the algorithm for computing such conditions in arbitrary multi-component systems, and discusses the features that allow it to have the same soundness property. If desired, the algorithm can be performed separately for each summary relation in the model of a system, but we will describe it as checking all the relations together, in a series of three phases. First, for each summary relation it selects a subset of the model that is relevant to the expectations summarized in that relation; this process is analogous to the technique of slicing in program analysis. Second, it transforms the flow relations in the model so that they can be soundly combined to describe the behavior of modules working together. Finally, it combines the transformed flow relations in the subset of the model to construct a logical condition connecting the abstractions describing the behavior of various components to the expectations of the summary relation, so that if the condition holds, the summary expectations will be satisfied. The second and third phases are analogous to the construction of a verification condition in program verification, except that they operate using much larger atomic units of program behavior. The following subsections describe these three phases in turn.

In previous work [13] we described a simpler algorithm with the same purpose as the one described here, which combined the aforementioned three phases into one. However, the previous algorithm contained an ambiguity in its description, relating to the order in which the system graph was traversed, and the consistency conditions it produced could potentially lead to both false positive results (rejected safe upgrades) and false negatives (approved unsafe upgrades). In the present algorithm we focus on eliminating false negatives, to achieve soundness.

6.1 Selecting relevant relations

The set of data-flow relations that are relevant to a given summary relation can be determined using context-free language reachability [17] on the graph representing the model. Suppose that the edges of the model graph, excluding summary edges, are labelled as follows. Control-flow edges are labeled with $\triangle$. Procedure calls and returns are labelled with $(_i$ and $)_i$ respectively, where the indices i are chosen to be unique except that the call from and return to any particular site have the same index. Data-flow edges are replaced with sequences of three edges labeled with $\triangle$, connecting the original ends via two fresh vertices:



The fresh vertices are each adorned with a number of self-edges: one for each closing parenthesis $)_i$ on the first vertex, and one for each opening parenthesis $(_j$ on the second vertex. Now, let v be the input of a summary relation in such a graph. We say that a path from a node u to v is relevant if it is labelled by a word in the following context-free language:

$$\begin{array}{lcl} S & \rightarrow & R\,L \\ L & \rightarrow & Z \mid (_i\,L \mid B\,L \\ R & \rightarrow & Z \mid R\,)_i \mid R\,B \\ B & \rightarrow & Z \mid (_i\,B\,)_i \mid B\,B \\ Z & \rightarrow & \varepsilon \mid \triangle \end{array}$$

where productions with parentheses are repeated for all i. In other words, relevant paths include no mismatched parentheses; a $\triangle$ may appear anywhere. The set of all nodes u that start relevant paths can be determined by a dynamic-programming approach that enumerates all the triples consisting of two nodes and a nonterminal such that the path between the nodes can be labeled by the nonterminal. We say that an edge is relevant if it occurs on any relevant path. A relation is relevant if it contains any relevant edge (though in fact, either all or none of the edges in a relation will be relevant), or for an independent output if its node is on a relevant path.

The intent of this selection algorithm is to conservatively choose a subset of the system model whose behavior might affect the validity of a summary relation. The basic approach is similar to traditional interprocedural slicing: any path is considered feasible unless it violates the correct matching of procedure calls and returns. The treatment of data-flow edges, however, is non-traditional. Data-flow edges represent non-local dependencies between parts of a program, which need not obey the proper nesting of calls and returns: internal state might be set at one point in execution and read much later in an unrelated context. Thus, reachability across data-flow edges is granted an exemption from the usual matching of calls with returns, achieved via the data-flow edge rewriting shown above. Our treatment of state may be contrasted with the usual treatment of global variables in CFL-reachability-based program slicing [9], in which globals are threaded through procedures as extra parameters. While the traditional approach is potentially more precise, it relies on more detailed information about procedure implementations than is available in our framework.

Note that abstractly, this first phase of our algorithm is superfluous: one could obtain the correct results for each summary relation by using a consistency condition covering the entire system. However, including the first phase as described above has a number of practical benefits. First, as a matter of efficiency, existing automatic theorem provers are often unable to avoid consideration of supplied premises that are irrelevant to the statement being verified, especially when those premises include quantification. We would expect it to be more efficient to exclude irrelevant facts before passing them on to the theorem proving stage. Second, this slicing of the system model helps users of a tool track down and fix potential incompatibility warnings when they are generated. After a potential incompatibility is flagged, it is up to a user to decide which components must be modified or re-tested to allow the system to operate correctly. Knowing which components might be responsible for a failure reduces the scope of this search.

6.2 Transformations for sound composition

We aim to construct a consistency condition by conjoining formulas representing each relation in the relevant subgraph of the system model. We already assume that relations are sound with respect to the single module where they are found, but they must be changed to be sound over the larger domain of a combined system. Conjoining the formulas for each relation unmodified would lead to a consistency condition that might be satisfied even by an unsafe system. Recall that a flow relation from (input) program points $u_1, u_2, \ldots, u_k$ to an (output) program point v is a formula that holds at each dynamic instance of v, in terms of the values at the most recent preceding instances of each point u_i. In order to soundly combine relation formulas by conjunction, we must ensure that the variables in those formulas always consistently refer to the same set of values. In the following subsections, we explain where variable reference inconsistencies arise, and how we transform the relations to achieve sound combination.

6.2.1 Splitting relations into edges

Flow relations connect any number of inputs to a single output using formulas over the relevant variables at each program point. When combining information about multiple modules, though, it is more convenient to divide the relations into edges matching the graph structure of the model. Consider a relation from $u_1, \ldots, u_k$ to v. Introduce a fresh set of variables corresponding to all the variables at each of the points u_i, and rewrite any formula involving at least one of the u_i variables and a v variable to use the fresh copies of the variables from each u_i. Next, add formulas setting each original variable from u_i equal to its corresponding copy. Now, just these equations can be associated with each edge from a point u_i to v. The new set of equations represents the same relation between the variables at the points u_i and the variables at v that the original one did.

6.2.2 *Guarding conditional control flow*

Consider a control-flow edge from a point u to a point v. The 'meaning' of the edge is described with two formulas (as explained in Section 3.1.2). First, a relation ψ over the variables of both u and v describes data flow, holding for each occurrence of v over the variable values of that occurrence of v and the values at the most recent previous occurrence of u. Second, a guarding condition ϕ_g is a relation only over the variables of u, and holds on only on occurrences of u that are followed by an occurrence of v (without an intervening u). To construct an edge that can be used to soundly 'predict' the values at v given those at u, we combine these formulas into a new condition $\phi_g \Rightarrow \psi$. This new formula holds over every occurrence of u and the next following v: for any u, if u is not followed by v (without an intervening u), then ϕ_g will be false, making the implication true. On the other hand, if some u is followed by an occurrence of v, then that u is the most recent occurrence before v, so ψ holds over the values, and the implication is again true.

6.2.3 *Duplicating based on calling context*

Consider a procedure that is called from more than one module (say two); let e be its exit program point, and r_1 and r_2 the return program points for the two callers. The return relation between e and r_1 says that the return value seen by the caller is equal to the value returned, and similarly for the return relation between e and r_2. However, conjoining these relations would be unsound, because they are effectively referring to two different sets of values at e: one the values that will be returned to r_1, for the first caller, and the other only those to be returned to r_2. Given a property of the values returned to r_1 that distinguished them from the values returned to r_2, one might imagine using a technique similar to guarding to resolve this inconsistency, but that is not a feasible approach. Such a property may not even exist (if there are some values that could be returned to either caller), and even it did it exist it couldn't be determined in our modular approach, because it would require knowledge of all the calling modules.

Instead, the mismatch can be corrected by duplicating the program point e to create two points, e_1 connected to r_1 and e_2 connected to r_2, so that the variables at e_1 refer only to values on calls that return to r_1, and similarly for e_2. After this transformation, the return relations that equate e_1 with r_1, and e_2 with r_2, will be sound for any invocations of the restricted e_1 or e_2. Of course, if the program point e is split, we must also describe how the other edges ending at e are transformed. In the case of control-flow edges, we can repeatedly apply the same splitting technique to predecessor points until reaching the calls corresponding to the procedure returns; the net effect is to duplicate the representation of the procedure between the various call sites. For data-flow edges, see Section 6.2.4.

In practice, the duplication need not be performed step by step as it was just introduced. We can construct the right number of duplicates for every program point by simply traversing the system graph, maintaining a stack corresponding to the call stack of a system execution, and constructing one copy of a node for each unique stack contents (calling context) with which it might be reached. It is somewhat unfortunate that for soundness, our technique currently requires this extensive duplication of procedures called from multiple contexts. Because our model of the tested behavior of each module is collected without knowledge of the particular contexts where the module will be used, the context sensitivity provided by duplication should not be expected to significantly increase the precision of the technique's results. Rather, duplication seems necessary as a matter of soundness, for cases when two uses of a module are considered together, to avoid effectively assuming that a procedure's return value always took a single value, as would happen if it were represented by a single logical variable. Figure 1 shows an example where duplication appears necessary. Without duplication, one could conclude that the values returned by c and d are equal, and thus that the return value of b must be even, when in fact it must be odd. We are still looking for less extensive modifications that might achieve soundness while remaining essentially context-insensitive. The overhead incurred by duplication, though exponential in the worst case, should be modest in practice, because the number of modules comprising a system will be relatively small.

$$\begin{array}{rl}
A.m(x): & x := x \cdot x + 1;\ r := B.b(x);\ \texttt{assert}(r > 4 \cdot x) \\
B.b(y): & r := C.c(2 \cdot y) + D.d(2 \cdot y + 1) \\
C.c(v): & r := E.i(v) \\
D.d(v): & r := E.i(v) \\
E.i(x): & r := x + 1
\end{array}$$

$$\begin{array}{l}
(A(b_c) \wedge \text{Call}(B.b|A.b_c) \wedge B(c_c|b) \wedge \text{Call}(C.c|B.c_c) \\
\wedge\, C(i_c|c) \wedge \text{Call}((E.i)_c|C.i_c) \wedge (E(i'|i))_c \\
\wedge\, \text{Ret}(C.i_r|(E.i')_c) \wedge C(c'|i_r) \wedge \text{Ret}(B.c_r|C.c') \\
\wedge\, B(d_c|b) \wedge \text{Call}(D.d|B.d_c) \wedge D(i_c|d) \\
\wedge\, \text{Call}((E.i)_d|D.i_c) \wedge (E(i'|i))_d \wedge \text{Ret}(D.i_r|(E.i')_d) \\
\wedge\, D(d'|i_r) \wedge \text{Ret}(B.d_r|D.d') \wedge B(b'|c_r, d_r) \\
\wedge\, \text{Ret}(A.b_r|B.b')) \qquad\qquad \Rightarrow \overline{A}(b_r|b_c)
\end{array}$$

$$\begin{array}{l}
(A.b_c.y > 0 \wedge B.b.y = A.b_c.y \wedge B.c_c.v = 2 \cdot B.b.y \wedge C.c.v = B.c_c.v \\
\wedge\, C.i_c.x = C.c.v \wedge (E.i.x)_c = C.i_c.x \wedge (E.i'.r)_c = (E.i.x)_c + 1 \\
\wedge\, C.i_r.r = (E.i'.r)_c \wedge C.c'.r = C.i_r.r \wedge B.c_r.r = C.c'.r \\
\wedge\, B.d_c.v = 2 \cdot B.b.y + 1 \wedge D.d.v = B.d_c.v \wedge D.i_c.x = D.d.v \\
\wedge (E.i.x)_d = D.i_c.x \wedge (E.i'.r)_d = (E.i.x)_d + 1 \wedge D.i_r.r = (E.i'.r)_d \\
\wedge\, D.d'.r = D.i_r.r \wedge B.d_r.r = D.d'.r \wedge B.b'.r = B.c_r.r + B.d_r.r \\
\wedge\, A.b_r.r = B.B'.r) \qquad\qquad \Rightarrow A.b_r.r > 4 \cdot A.b_c.y
\end{array}$$

Figure 1: A small example of a consistency condition derived by the algorithm of Section 6. The three sections show code for a system, the form of the consistency condition as computed by the algorithm of Section 6, and the actual condition as passed to a theorem prover. The increment routine i of module E is called in different contexts by modules C (with an even argument) and D (with an odd argument). Duplication of the logical variables for procedure i is indicated by the notations $(\cdot)_c$ and $(\cdot)_d$.

6.2.4 *Mixing data-flow edges*

Data-flow edges must also be changed when the nodes they connect are duplicated, but they cannot be duplicated along with the nodes they connect in the way that control-flow edges are, because a data-flow edge does not correspond to any particular execution context. In fact, a data-flow edge might connect two nodes that are duplicated a different number of times. Our model is too abstract to determine which flows between copies of the source program point and copies of the target program point might occur (for instance, because it does not provide a total ordering of calls), so we instead conservatively assume that any flow might be possible. If an original source program point u is duplicated n times and a target point v is duplicated m times, we create nm copies of the data-flow edge originally connecting them, one connecting each duplicate of u to each duplicate of v. However, at each duplicate of v, the formulas for the edges are disjoined (rather than conjoined as formulas are otherwise). The effect is to express that each destination receives values flowing from at least one source.

6.3 Assembling a consistency condition

Once the relational model has been transformed as described in the previous subsection, assembling the consistency condition is straightforward. For a given summary relation, take the set of flow relations relevant to the summary input, as computed by CFL reachability. Let S be the set of all those relation formulas, as rewritten or duplicated according to the transformations above. Then the consistency condition states that the conjunction of all the formulas in S implies the summary relation formula.

In outline, the proof of the soundness of this technique is as follows, for a single summary relation. Suppose that the consistency condition holds; it is an implication of the form $(\bigwedge_i \phi_i) \Rightarrow \sigma$, where each ϕ_i is a transformed flow relation formula and σ is a summary relation formula. Each ϕ_i holds individually, due to the assumption that the original flow relations are sound and the fact that their transformations were soundness-preserving. Furthermore, the formulas ϕ_i use common variables consistently, so they can be legitimately conjoined, and their conjunction is true. Thus, by our assumption of the implication, σ must hold. By this argument, each summary relation σ in a system can be seen to hold, which guarantees, by the definition of a summary relation, that the system will be safe.

7. CONCLUSION

This research takes steps toward proving the soundness of a technique for verifying properties about a software system made up of components. One use of the technique is to verify that after some components of a software system have been upgraded, the system as a whole continues to behave as desired.

This paper makes five key contributions. First, we provided an abstract model of the behavior of software components. The model differs from previous models by distinguishing between control flow and data flow. Second, we gave a formalized version of the problem of checking consistency for a modular system in a simple, imperative language. Safe component composition is modeled by the success of arbitrary assertion statements. Third, we defined an algorithm that constructs a consistency condition. The consistency condition is a logical formula such that if the components satisfy its parts (which express expectations about component behavior), then the system as a whole behaves safely. This algorithm differs from previous work by applying to the new, more detailed model and by fixing an error in previous formulations. Fourth, we proved the correctness of the algorithm in the special case of a single-component upgrade: the condition that the algorithm generates suffices to ensure that an upgrade is safe. Fifth, we gave a proof outline of the correctness of the algorithm in the general case. Completing this proof is future work.

We made two types of changes to previous work. Some changes were motivated by the desire to prove correctness; for example, the proof required formalization of the safety condition. Other changes correct errors in previous work that were not apparent until revealed by our formalization and proof attempts. The result is a more detailed and correct technique for verifying the composition of software components.

Though our technique does not require its users to understand the complexities behind its operation, this research demonstrates that techniques from formal specification and verification can make possible a practical and lightweight tool to help software development.

Acknowledgment: This work was funded in part by NSF grant CCR-0133580, DARPA contract FA8750-04-2-0254, the Oxygen project, and an NDSEG fellowship.

8. REFERENCES

[1] P. America. Inheritance and subtyping in a parallel object-oriented language. In *ECOOP*, pages 234–242, June 1987.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, Jan. 2002.

[3] Y. Chen and B. H. C. Cheng. A semantic foundation for specification matching. In *Foundations of Component-Based Systems*, chapter 5, pages 91–109. Cambridge University Press, New York, NY, 2000.

[4] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, Palo Alto, CA, July 23, 2003.

[5] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9, 2003.

[6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, Feb. 2001.

[7] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *ESEC/FSE*, pages 229–236, Sept. 2001.

[8] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *ECOOP*, pages 431–456, July 2003.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, Jan. 1990.

[10] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, second edition, 1990.

[11] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, Nov. 1994.

[12] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE*, pages 287–296, Sept. 2003.

[13] S. McCamant and M. D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP*, pages 440–464, June 2004.

[14] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[15] S. Moisan, A. Ressouche, and J.-P. Rigault. Behavioral substitutability in component frameworks: A formal approach. In *SAVCBS*, pages 22–28, Sept. 2003.

[16] O. Nierstrasz. Regular types for active objects. In *OOPSLA*, pages 1–15, Sept./Oct. 1993.

[17] T. W. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, Nov./Dec. 1998.

[18] J. Schumann and B. Fischer. NORA/HAMMR: Making deduction-based software component retrieval practical. In *ASE*, pages 246–254, Nov. 1997.

[19] J. Yang and D. Evans. Dynamically inferring temporal properties. In *PASTE*, pages 23–28, June 2004.

[20] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM TOSEM*, 6(4):333–369, Oct. 1997.

Verification of Evolving Software[*]

Sagar Chaki Natasha Sharygina Nishant Sinha
chaki|natalie|nishants@cs.cmu.edu

ABSTRACT

We define the *substitutability* problem in the context of evolving software systems as the verification of the following two criteria: (i) previously established system *correctness properties* must remain valid for the new version of a system, and (ii) the *updated portion* of the system must continue to provide all (and possibly more) *services* offered by its earlier counterpart. We present a completely *automated procedure* based on learning techniques for regular sets to solve the substitutability problem for component based software. We have implemented and validated our approach in the context of the ComFoRT reasoning framework and report encouraging preliminary results on an industrial benchmark.

1. INTRODUCTION

Model checking [7] is a formal verification approach for detecting behavioral anomalies (including safety, reliability and security problems) in hardware and software systems. While model checking produces extremely valuable results, often uncovering defects that otherwise go undetected, there are several barriers to its successful integration into sofware development processes. In particular, model checking is hamstrung by scalability issues and is difficult for software engineers to use directly.

Most current research on model checking focuses on improving its scalability, and innovative techniques such as automated predicate abstraction and assume-guarantee reasoning have greatly improved the applicability of model checking to industrial-scale systems. However, there has been less progress on its transition from an academic to a practically viable discipline.

For instance, any software system inevitably evolves as designs take shape, requirements change, and bugs are discovered and fixed. While model checking is useful at each of these stages, it is usually applied to the entire system at every point irrespective of the amount of modification the system has actually undergone. The amount of time and effort required to verify an entire system can be prohibitive and repeating the exercise after each (even minor) system update discourages its use by practitioners.

In this article we present a framework that, while not affecting the initial model checking effort, is aimed at reducing dramatically the effort to keep analysis results up-to-date with evolving systems. More specifically, we make the following two contributions. First, we define the *substitutability* problem as the verification of the following two criteria: (i) previously established system *correctness properties* must remain valid for the new version of an evolving software, and (ii) the *updated portion* of the system must continue to provide all (and possibly more) *services* offered by its earlier counterpart. Second, we present a completely *automated procedure* to solve the substitutability problem in the context of component based software.

We will define our notion of components and their behaviors more precisely later. Intuitively, a behavior of a component involves a sequence of observable message-passing interactions with other components. We denote the set of behaviors of a component C by $\mathcal{B}(C)$. Also given two components C and C' we will write $C \preccurlyeq C'$ to mean that $\mathcal{B}(C) \subseteq \mathcal{B}(C')$. Suppose we are given an assembly of components: $\mathcal{A} = \{C_1, \ldots, C_n\}$, a safety property φ, and a new component, C_i^S, to be used in place of C_i. We assume that φ was proven to hold on the original assembly $\mathcal{A}$. We wish to check for the *substitutability* of C_i^S for C_i in $\mathcal{A}$ with respect to the property φ. More specifically, our aim is to develop a procedure that achieves the following goals:

1. **Containment.** Verify that $C_i \preccurlyeq C_i^S$, i.e., every behavior of C_i is also a behavior of C_i^S. To this end, we will construct a component C_i^F such that $\mathcal{B}(C_i^F) = \mathcal{B}(C_i) \cup \mathcal{B}(C_i^S)$. In particular, if $C_i \preccurlyeq C_i^S$, then C_i^F will be the same as C_i^S. If the check fails, we provide the developers with feedback regarding the differences between C_i and C_i^S. Assuming that the missing behaviors would be added to C_i^S subsequently, we proceed with C_i^F as a safe abstraction of the new component in the next phase.

2. **Compatibility.** Verify that the new assembly $\mathcal{A}' = \{C_1, \ldots, C_i^F, \ldots, C_n\}$ satisfies the safety property φ. Note that in general $\mathcal{B}(C_i^F) \supset \mathcal{B}(C_i)$ owing to additional behaviors from C_i^S. Hence $\mathcal{A}'$ might violate φ even though φ was satisfied by $\mathcal{A}$. Therefore, in our framework, compatibility must be explicitly verified.

[*]This work was done as part of the Predictable Assembly from Certifiable Components initiative at the Software Engineering Institute.

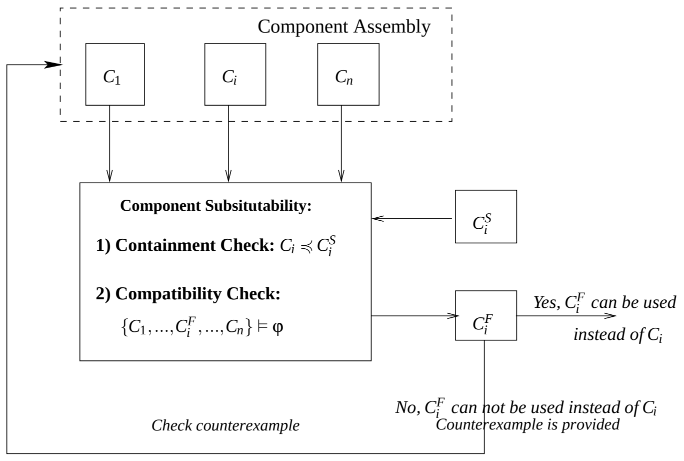

Component Assembly
C_1
C_i
C_n
Component Subsitutability:
1) Containment Check: $C_i \preccurlyeq C_i^S$
2) Compatibility Check:
$\{C_1, ..., C_i^F, ..., C_n\} \models \varphi$
C_i^S
C_i^F
Yes, C_i^F can be used instead of C_i
No, C_i^F can not be used instead of C_i
Counterexample is provided
Check counterexample

Figure 1: Overview of Component Substitutability.

We believe that our methodology is uniquely distinguished by the two phases it involves. **Containment** ensures that the substituted component satisfies the following criterion (CONT): it provides all services rendered by the original component C_i. If the new component C_i^S does not satisfy CONT, we generate a component which does, viz., C_i^F.

The new component C_i^S will usually be the result of design changes, bug fixes and other updates by a varied group of software professionals. Thus, it is unrealistic to expect C_i^S to always bear a specific relationship with the original component C_i. For instance, C_i^S will seldom refine C_i in the sense that all behaviors of the C_i^S are already there in C_i. We believe that in order to be viable, any approach to the substitutability problem must allow additional behaviors in C_i^S, and yet ensure that all old features of C_i continue to be supported. This is precisely the purpose of the containment phase which culminates in the construction of C_i^F. To the best of our knowledge, ours is the first framework to address this issue explicitly.

Compatibility guarantees that C_i^F can be safely integrated with the other components in the assembly. Recall that this must be checked explicitly since in general $\mathcal{B}(C_i^F) \supset \mathcal{B}(C_i)$. The compatibility check results in either a substitutable component C_i^F or produces a counterexample showing why the substitution of C_i^S is not feasible. The component C_i^F is such that: (i) it renders every service of C_i and yet (ii) the new assembly $\mathcal{A}' = \{C_1, \ldots, C_i^F, \ldots, C_n\}$ satisfies the safety property φ. The complete substitutability check procedure is outlined in Figure 1.

In addition to the computation of C_i^F, a major focus of the containment phase is to compute a set of behaviors in $\mathcal{B}(C_i) \setminus \mathcal{B}(C_i^S)$. Since these behaviors express features of C_i that are absent in C_i^S, they can be used to generate feedback for the developers. Such feedback can be of critical help by *localizing* the changes required to add the missing features back to C_i^S. We discuss this issue further in Section 5.4. We use automata-theoretic learning techniques for both the containment and compatibility phases of our approach. Specifically, we use techniques based on a learning algorithm for regular sets proposed by Angluin [2]. As we shall see later, our use of learning will aid in efficient feedback generation.

Finally, we employ state/event-based modeling techniques [4] in order to be able to model and reason about both the data and communication aspects of software. We use labeled Kripke structures (LKSs) to model, as well as to specify, software systems. This is important for our approach to be practically applicable to real-life component-based systems.

We implemented and validated our approach in the context of the Component Formal Reasoning Technology (ComFoRT) [10] reasoning framework being developed as part of the Predictable Assembly from Certifiable Components (PACC) [14] initiative at the Software Engineering Institute (SEI), Carnegie Mellon University (CMU). Specfically we implemented our substitutability framework as part of the model checking engine of ComFoRT, which is based on the C model checker MAGIC [5, 11] developed at CMU. In the rest of this article we will use the ComFoRT model checker and MAGIC synonymously.

The ComFoRT model checker employs *automated predicate abstracion* to extract finite models from concurrent C programs. Since abstract models often contain unrealistic behaviors, any counterexample obtained from an abstract model must be validated against the concrete system. If the counterexample is found to be spurious, the model must be refined and verification repeated. This iterative procedure is known as counterexample guided abstraction refinement (CEGAR) and implemented by MAGIC in a completely automated form. Furthermore, in the context of concurrent systems, MAGIC conducts both counterexample validation and abstraction refinement steps in a component-wise manner. Both predicate abstraction and automated CEGAR were critical for applying our technique to industrial component-based systems.

In summary, we believe that the presented component substitutability procedure has several advantages:

- Unlike conventional approaches, our methodology does not subscribe to the idea of *trace-theoretic refinement* while checking for substitutability. We believe that it is unduly restrictive to require a new component to directly refine its old counterpart in order be replaceable, and instead allow new components to have more behaviours. The *extra* behaviors are critical since they provide vendors with flexibility to implement new features into the product upgrades[1].

- Our technique identifies features of the old component C_i which are missing in the new component C_i^S. It also generates feedback to localize the modifications required in C_i^S to add the missing features back.

- Our method uses techniques based on learning algorithms for regular sets for accomplishing both phases of the substitutability check. This unified approach enables automatic verification of evolving software.

- Our technique supports component-wise abstraction, counterexample validation and abstraction refinement steps of the verification procedure and is thus expected to scale to large software designs.

This article is organized as follows. In Section 2 we discuss related work. Preliminary definitions and notations are

[1]Verification of these new features remains a responsibility of designers of the upgraded systems.

presented in Section 3 followed by a description of the L^* learning algorithm in Section 4. Details of our core component substitutability framework are presented in Section 5. Finally we present experimental results in Section 6 and conclude in Section 7.

2. RELATED WORK

This work relates to multiple projects targeting verification of component-based systems. In general, in contrast to our work, other projects often impose the restriction that every behavior of the new component must also be a behavior of the old component. In such a case the new component is said to *refine* the old component.

Alfaro et. al. [9, 6] define a notion of interface automaton for modeling component interfaces and show compatibility between components via refinement and consistency between interfaces. However, automated techniques for constructing interface automata from a component implementations are not presented. Labeled Kripke structures (LKSs) coupled with the interface alphabet as they are used in this work for constructing component abstractions are similar to interface automata. In contrast, this work is based on sound predicate abstraction techniques that automatically extract LKSs from component implementations. Also our work is not limited to showing refinement between the old component and the new one and therefore it suits more to realistic systems.

Ernst et. al. [13] suggest a technique for checking compatibility of multi-component upgrades. However, they restrict themselves to input/output specifications of components by abstracting away temporal information about the sequence of actions. They acknowledge that even though the abstraction is not sound, their approach is useful in detecting important problems. In contrast to that, since our work employs existential abstraction, our framework is sound. Another drawback of this related work is that due to the nature of the input/output abstraction that eliminates sequencing of actions, component specifications are not complete. This is not a problem in our work since predicate abstraction is over-approximation and preserves all possible behaviors of the original components. Another difference with our project is that [13] component consistency criteria imply that all behaviors of component upgrades are also behaviors of their old counterparts.

The compatibility check in the current work is automated following ideas of Cobleigh et. al. [8] of using learning for regular sets techniques. While [8] focuses on automating assume-guarantee reasoning, our work solves a more general problem of the component substitutability. We use compositional reasoning to discharge new behaviors of the component upgrades in new assemblies. Our approach differs from theirs in a number of ways. Firstly, we take care of state labeling information of LKSs by including both state and transition labels in the language definition of LKSs. Also in our case, the compatibility check is embedded in the abstraction-refinement framework for C programs, which makes verification tractable.

3. BACKGROUND AND NOTATION

DEFINITION 1 (FINITE AUTOMATA). *A non-deterministic finite automaton (NFA) is a 5-tuple $(S, S_0, \Sigma, \Delta, F)$ with S a finite set of states, $S_0 \subseteq S$ a set of initial states, Σ a finite alphabet, $\Delta \subseteq S \times \Sigma \times S$ a transition relation, and $F \subseteq S$ a set of final (accepting) states. The language of an NFA M is denoted by $L(M)$ and defined in the usual way. A deterministic finite automaton (DFA) is a NFA such that S_0 has exactly one element and Δ is a function from $S \times \Sigma$ to S.*

DEFINITION 2 (LABELED KRIPKE STRUCTURE). *A labeled Kripke structure (LKS for short) is a 6-tuple $(S, Init, AP, \mathcal{L}, \Sigma, \delta)$ with S a finite set of* states, *$Init \subseteq S$ a set of initial states, AP a finite set of (atomic) state propositions, $\mathcal{L} : S \to 2^{AP}$ a state-labeling function, Σ a finite set of events or actions (alphabet), and $\delta \subseteq S \times \Sigma \times S$ a transition relation.*

For any NFA, DFA (or LKS) with transition relation Δ (or δ), we write $q \xrightarrow{\alpha} q'$ to mean $(q, \alpha, q') \in \Delta$ (or $(q, \alpha, q') \in \delta$). We wish to define the language of an LKS in terms of that of an equivalent NFA. However since the states of an NFA are not labeled, we will have to transform the state labeling of the LKS to events in accordance with some scheme. Moreover, we would like to vary the alphabet of the resulting NFA by focusing on different sets of propositions. This idea is captured by an induced NFA.

DEFINITION 3 (INDUCED NFA). *The NFA induced by an LKS $M = (S, Init, AP, \mathcal{L}_M, \Sigma_M, \delta)$ is denoted by $NFA(M)$ and defined as: $(S \cup \{s_i\}, \{s_i\}, \Sigma_N, \Delta, S \cup \{s_i\})$ where $s_i \notin S$ is a new state, $\Sigma_N = (\Sigma_M \cup \{\tau\}) \times 2^{AP}$, and Δ is defined as follows:*

$$\forall s \in Init \,\centerdot\, s_i \overset{<\tau, \mathcal{L}_M(s)>}{\longrightarrow} s \in \Delta$$

$$\forall s \xrightarrow{\alpha} s' \in \delta \,\centerdot\, s \overset{<\alpha, \mathcal{L}_M(s')>}{\longrightarrow} s' \in \Delta$$

DEFINITION 4 (LKS LANGUAGE). *The language of an LKS M is denoted by $L(M)$ and defined as the language of the induced $NFA(M)$. Note that $L(M)$ is prefix-closed:*

$$\forall w \,\centerdot\, w \in L(M) \implies \forall w' \in prefix(w) \,\centerdot\, w' \in L(M)$$

DEFINITION 5 (ABSTRACTION). *Given two LKSs M_1 and M_2 we say that M_2 is an abstraction of M_1, denoted by $M_1 \preccurlyeq M_2$, iff $L(M_1) \subseteq L(M_2)$. Note that this concretizes our intuitive notion of abstraction being a form of behavioral containment since the set of behaviors of an LKS is captured by its language.*

DEFINITION 6 (PARALLEL COMPOSITION). *Let $M_1 = (S_1, Init_1, AP_1, \mathcal{L}_1, \Sigma_1, \delta_1)$ and $M_2 = (S_2, Init_2, AP_2, \mathcal{L}_2, \Sigma_2, \delta_2)$ be two LKSs. The parallel composition of M_1 and M_2, denoted by $M_1 \parallel M_2$, is the LKS $(S_1 \times S_2, Init_1 \times Init_2, AP_1 \cup AP_2, \mathcal{L}, \Sigma_1 \cup \Sigma_2, \delta)$, where $\mathcal{L}(s_1, s_2) = \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$, and δ is such that $(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')$ iff one of the following holds:*

1. *$\alpha \in (\Sigma_1 \setminus \Sigma_2) \cup \{\tau\}$ and $s_1 \xrightarrow{\alpha} s_1'$ and $s_2' = s_2$*
2. *$\alpha \in (\Sigma_2 \setminus \Sigma_1) \cup \{\tau\}$ and $s_2 \xrightarrow{\alpha} s_2'$ and $s_1' = s_1$*
3. *$\alpha \in (\Sigma_1 \cap \Sigma_2) \setminus \{\tau\}$ and $s_1 \xrightarrow{\alpha} s_1'$ and $s_2 \xrightarrow{\alpha} s_2'$*

In other words, LKSs must synchronize on shared actions (except τ) and proceed independently on local actions (and τ). This notion of parallel composition is derived from CSP [16].

Definition 7 (Components and Models). *In our framework a component is essentially a C program communicating with other components via blocking message passing. Since C programs are in general infinite state systems we will extract finite LKS models from components via predicate abstraction [5], and perform further analysis on these models. The data and message-passing aspects of a component C will be transformed conservatively into predicates and actions of its model M. Consequently, M is guaranteed to be a sound abstraction of C.*

Definition 8 (Assembly and Environment). *A component assembly $\mathcal{A}$ is a collection of components $\{C_1, \ldots, C_k\}$. For $1 \leq i \leq k$, let M_i be a model of C_i. Then the collection of models $\{M_1, \ldots, M_k\}$ is called a model assembly corresponding to $\mathcal{A}$ and denoted by $M_{\mathcal{A}}$. The environment of a component C_i with respect to $\mathcal{A}$ is a set $Env(C_i) \subseteq \mathcal{A}$ such that each component in $Env(C_i)$ communicates with C_i via message-passing. Similarly, the environment of a model M_i with respect to $M_{\mathcal{A}}$ is the model assembly corresponding to $Env(C_i)$.*

4. LEARNING REGULAR SETS

Central to our substitutability check procedure is the L^* inference algorithm for regular languages developed by Angluin [2] and later improved by Rivest et. al. [15]. In the rest of this article we will only concern ourselves with the original algorithm of Angluin. Let U be an unknown regular language over some alphabet Σ. In order to learn U, L^* needs to interact with a *minimally adequate teacher* MAT for U, which can answer two kinds of queries.

1. *Membership.* Given a word $\rho \in \Sigma^*$, MAT returns $true$ if $\rho \in U$ and $false$ otherwise.
2. *Candidate.* Given a DFA D, MAT returns $true$ if $L(D) = U$ and $false$ otherwise. If MAT returns $false$, it also returns a counterexample word w in the symmetric difference of $L(D)$ and U.

Given an unknown regular language U and a MAT for U, the L^* algorithm *iteratively* constructs a minimal DFA D such that $L(D) = U$. It maintains an observational table T where it records information about elements and non-elements of U. The rows of T are labeled by the elements of $S \cup S \cdot \Sigma$ where S is a prefix-closed set over Σ^*. The columns of T are labeled by the elements of a suffix-closed set E over Σ^*. Let us denote the set $S \cup S \cdot \Sigma$ by *Row*. Then the following condition always holds for T:

$$\forall s \in Row \centerdot \forall e \in E \centerdot T[s, e] = true \iff s \cdot e \in U$$

Additionally, for any $s \in Row$, let us define a function r_s as follows:

$$\forall e \in E \centerdot r_s(e) = T[s, e]$$

Then T is said to be *closed* and *consistent* if the following two conditions hold respectively:

$$\forall t \in S \cdot \Sigma \centerdot \exists s \in S \centerdot r_s = r_t$$

$$\forall s_1, s_2 \in S \centerdot r_{s_1} = r_{s_2} \implies \forall a \in \Sigma \centerdot r_{s_1 \cdot a} = r_{s_2 \cdot a}$$

L^* starts with a table T such that $S = E = \emptyset$ and in each iteration proceeds as follows. It first updates T using membership queries (starting with words of length at most one) till T is closed and consistent. Next L^* builds a candidate DFA $D(T)$ from T and makes a candidate query with $D(T)$. If the MAT returns $true$ to the candidate query, L^* returns $D(T)$ and stops. Otherwise, L^* updates T with all prefixes of the counterexample returned by MAT and proceeds with the next iteration. The complexity of L^* is expressed by the following theorem.

Theorem 1. *[2] If n is the number of states of the minimum DFA accepting U and m is the upper bound on the length of any counterexample provided by the MAT, then the total running time of L^* is bounded by a polynomial in m and n. Moreover, the observation table T is of size $O(m^2n^2 + mn^3)$.*

Optimizations. Note that in our case, the unknown language U is always prefix-closed since, by definition, the language of LKSs are prefix-closed. This allows us to augment L^* with some optimizations similar to those proposed by Berg et. al. [3]. A prefix-closed language L is characterized by the property that for a trace $\rho \in L$, all prefixes of ρ are in L. Conversely, for a trace $\rho \notin L$, no extension of ρ is in L.

Therefore, whenever L^* makes a membership query with ρ, we first look up all of ρ's prefixes in a *query cache*. The cache returns $false$ if any of the prefixes are present and marked $false$. Otherwise if ρ itself is present and marked $true$, the cache returns $true$. If none of the above cases hold, the query is passed on to the MAT. These optimizations yielded up to 20% speedup during our experiments (cf. Section 6).

5. COMPONENT SUBSTITUTABILITY

Recall that the substitutability problem involves two major phases: containment and compatibility. Suppose we are given an assembly of components: $\mathcal{A} = \{C_1, \ldots, C_n\}$ and an LKS φ such that $\mathcal{A} \preccurlyeq \varphi$. Also, we are given a new component C_i^S to be used in place of C_i. Our goal is to check for the substitutability of C_i^S for C_i in $\mathcal{A}$ while preserving all previous services as well as the validity of φ. Figure 2 shows the schematic diagram of our substitutability framework.

The complete substitutablility procedure occurs in a CEGAR-style loop. In each iteration of the loop, substitutablility checks are performed on abstract models instead of concrete components. Suppose M_i and M_i^S are the models of C_i and C_i^S respectively. Therefore we will check for the substitutability of M_i^S for M_i with respect to the property φ. Note that the final result is either a substitutable model M_i^F or a counterexample CE. However, CE may be spurious with respect to either C_i or C_i^S and therefore must be checked for validity against both. If CE is spurious we will refine M_i and/or M_i^S and repeat the CEGAR loop. Otherwise, we will report CE as an evidence of non-substitutablility of C_i^S and terminate.

In case we are able to prove substitutablility, we will also generate a set of traces in $L(M_i) \setminus L(M_i^S)$. These traces will then be used to provide constructive feedback to the developers (cf. Section 5.4).

5.1 Containment

The containment check accepts models M_i and M_i^S as inputs. In general, it might also be provided with an LKS B which captures a set of prohibited behaviors such

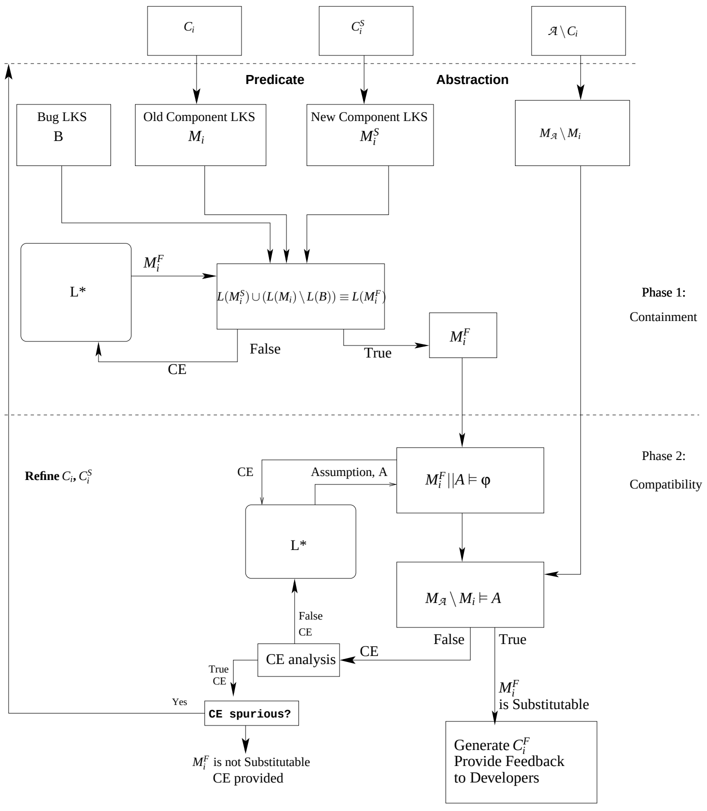

C_i
C_i^S
$\mathcal{A} \setminus C_i$
Predicate
Abstraction
Bug LKS
B
Old Component LKS
M_i
New Component LKS
M_i^S
$M_{\mathcal{A}} \setminus M_i$
L*
M_i^F
$L(M_i^S) \cup (L(M_i) \setminus L(B)) \equiv L(M_i^F)$
False
True
CE
M_i^F
Phase 1:
Containment
Phase 2:
Compatibility
Refine C_i, C_i^S
CE
Assumption, A
$M_i^F || A \models \varphi$
L*
$M_{\mathcal{A}} \setminus M_i \models A$
False
True
False CE
CE analysis
CE
True CE
Yes
CE spurious?
M_i^F is Substitutable
M_i^F is not Substitutable CE provided
Generate C_i^F Provide Feedback to Developers

Figure 2: Substitutability framework.

as previously detected bugs. Our goal in this phase is to apply a learning algorithm to build a new DFA M_i^F, which includes all the behaviors of M_i^S and M_i except those of B. In other words we wish to learn the language $U = L(M_i^S) \cup (L(M_i) \setminus L(B))$. Additionally we wish to compute a set of traces $\mathcal{F} \subseteq L(M_i) \setminus L(M_i^S)$ which can be subsequently used for feedback generation.

Recall that $NFA(M)$ denotes the NFA induced by an LKS M. Thus the alphabet over which the learning occurs is $\Sigma_{NFA(M_i)} \cup \Sigma_{NFA(M_i^S)}$. In addition, the membership and candidate queries are discharged as follows using a model checker.

Membership. In order to check for membership of a word ρ, the model checker checks whether it is accepted by either M_i or M_i^S and not accepted by B. Note that these three checks are performed separately without composing M_i, M_i^S and B.

Candidate. The candidate query for an intermediate candidate DFA D involves the following language equivalence check: $U = L(D)$. We avoid explicit computation of U (which would defeat the entire purpose of learning) while discharging the candidate query as follows. First we subdivide the candidate query into two subset checks: (i) $U \subseteq L(D)$ and (ii) $L(D) \subseteq U$. Next we perform the first check by verifying individually: (a) $L(M_i^S) \subseteq L(D)$ and (b) $L(M_i) \setminus L(B) \subseteq L(D)$. Finally we peform the second check in the following iterative manner.

1. Check if $L(D) \subseteq L(M_i^S)$. If the answer is *yes*, then $L(D) \subseteq U$ and we return *true*. Otherwise we get a trace $CE \in L(D) \setminus L(M_i^S)$.
2. Check if $CE \in L(M_i) \setminus L(B)$. If not, then $CE \in L(D) \setminus U$ and we return $false$ along with CE as counterexample. Otherwise $CE \in L(M_i) \setminus L(M_i^S)$.
3. Add CE to $\mathcal{F}$. Repeat from step 1 but look for counterexamples other than those already in $\mathcal{F}$. We achieve this by suitably modifying our model checker that performs step 1 above.

As mentioned previously, a key feature of our framework is to allow the new component to have more behaviors than the previous one. These extra behaviors might cause the new component assembly to violate the global property φ. Therefore the compatibility of the new component with the rest of the assembly must be verified separately. This forms the basis of the next phase in substitutability.

5.2 Compatibility

Recall that the global safety property is expressed as an LKS φ and that we write $M_1 \preccurlyeq M_2$ to mean $L(M_1) \subseteq L(M_2)$. On successful completion of the containment phase we obtain a DFA M_i^F such that $L(M_i^F) = L(M_i^S) \cup (L(M_i) \setminus L(B))$. We now need to verify that the component M_i^F is *compatible*, i.e, safe under the given environment $Env(M_i)$. In other words we need to check that $M_i^F \parallel Env(M_i) \preccurlyeq \varphi$. This is done by a combination of assume-guarantee style reasoning and learning similar to Cobleigh et. al. [8]. Hence we will not describe this phase in much detail but simply summarize the salient features as follows:

1. Learn an assumption DFA A for M_i^F such that $M_i^F \parallel A \preccurlyeq \varphi$ using L^* with a model checker as a MAT.
2. Check if $Env(M_i) \preccurlyeq A$. If so, return *true*. Otherwise a counterexample CE is obtained.
3. Check if $M_i^F \parallel CE \preccurlyeq \varphi$. If so, use CE to weaken the A and repeat from step 1. Otherwise, return false along with CE as the counterexample to the compatibility phase.

Note that since $L(M_i^F)$ contains traces from both $L(M_i)$ and $L(M_i^S)$, any counterexample CE returned by the above procedure must be checked against both C_i and C_i^S for spuriousness.

5.3 Why Learn?

Our use of techniques based on L^* provides us the following advantages:

- We can use our model checker to answer the membership and candidate queries and also to generate a counterexample in the event of the failure of a candidate query.
- Our use of learning during the containment phase enables us to compute a DFA for $L(M_i^S) \cup (L(M_i) \setminus L(B))$ without having to compose M_i, M_i^S or B. As a side-effect we are also able to generate a set of traces $\mathcal{F} \subseteq L(M_i) \setminus L(M_i^S)$ which can be subsequently used for feedback generation.
- L^* is incremental, computes the smallest DFA, and also enables us to proceed without precisely defining the language to be learned, e.g., during compatibility.
- Efficiency of L^* depends only on the length of individual counterexamples and the minimum automaton representation of the unknown language. This characteristic allows us to leverage the competency of the model checker in generating suitable counterexamples.

5.4 Feedback to Developers

In this section we present several approaches for providing feedback to developers that will enable them to add missing features back to C_i^S. Recall that upon successful completion of the substitutability check, we obtain a set of traces $\mathcal{F} \subseteq L(M_i) \setminus L(M_i^S)$. Let π be any trace in $\mathcal{F}$. Hence $\pi \in L(M_i)$. Recall that $L(M_i)$ was defined to be the language of the induced NFA $NFA(M_i)$. Since the actions of the NFA induced by any LKS M contain information about the propositional labeling of M, it is possible to retrieve this information from any trace of $NFA(M)$ and convert it back to a corresponding trace of M. Thus, in particular, we can obtain a trace ρ of M_i corresponding to π. The trace ρ constitutes our first level of feedback.

By itself, trace ρ provides limited assistance to a developer. While it shows a missing behavior, it does not relate this behavior to the new component C_i^S. Our next level of feedback attempts to improve this situation by identifying portions of the actual code for C_i^S which are relevant to the missing behavior expressed by Tr. One way to achieve this would be via a mapping *Map* from statements (or control points) of C_i to those of C_i^S. We intend to investigate the automated generation of *Map* as well as the fragments of C_i^S relevant to ρ as part of our future work.

Our most advanced form of feedback is aimed at eliminating completely the need for developers to modify C_i^S. Suppose we are given a trace ρ and the portions of C_i^S relevant to ρ as described earlier. Our goal is to automatically generate a modified version C_i^F of C_i^S such that C_i^F has all the features of C_i which were missing in C_i^S. Clearly, this is an extremely difficult problem in the general case and we must impose appropriate restriction in order to find effective solutions. For instance, we can restrict our changes to only branch statements and library routine calls. Additionally such code modifications can be made in the form of templates which can be inspected, and improved for performance if necessary, by the developers.

6. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We implemented our methodology for checking component substitutability in the ComFoRT framework developed at Carnegie Mellon Software Engineering Institute. The ComFoRT model checking engine is based on Magic model checking tool [5]. We used Magic capabilities to extract finite LKS models from C programs using predicate abstraction to construct abstract component models. The Magic model checker also serves as a *minimally adequate teacher* for the learning algorithms of the containment and compatability checks. Each of these checks instantiates its own L^* learner, which perform the task of learning their respective DFAs. If the compatibility check returns a counterexample, the counterexample validation and abstraction-refinement modules of Magic are employed to check for spuriousness and do refinement, if necessary.

We validated the component subsitutability framework while verifying upgrades of a benchmark provided to us by our industrial partner, ABB Inc. [1]. We verified part of an interprocess communication protocol (IPC-1.6) used to mediate communication in a multi-threaded robotics control automation system that must satisfy safety-critical requirements.

The IPC protocol provides multiple forms of communication including synchronous point-to-point, broadcast, publish/subscribe, and asynchronous communication, all of which are implemented in terms of messages passing between queues owned by different threads. The protocol handles the creation and manipulation of message queues, synchronizing access to shared data using various operating system primitives (e.g., semaphores and critical sections), and cleaning up internal state when a communication fails or times out.

We analyzed the portion of the IPC protocol that is used for synchronous communication among multiple threads. With this type of communication, a sender sends a message to a receiver and blocks until an answer is received or it times out. A receiver asks for its next message and blocks until a message is available or it times out. Whenever the receiver gets a synchronous message, it is then expected to send a response to the message's sender. The target of our verification was the IPC component that implements this communication scheduling, comprising of about 1500 lines of C code. We abstracted away communication with other IPC components by specifying external choice channels that were set to pass all possible inter-component inputs nondeterminstically.

We used a set of properties describing functionality of the verified portion of the IPC protocol. For example, we checked that no faults discovered during testing of older versions of the code are present in the current code. An example of such a property is an assertion that no messages are lost from the queue without being delivered to the receiver.

We upgraded the $WriteToQueue$ component of the IPC assembly by both adding and removing some behaviors. These modifications resulted in a violation of the containment check and hence a substitutable DFA was produced at the end of this check. This DFA was found to be smaller as compared to the old and new component LKSs. We believe this was the case with our examples since multiple states of the LKS obtained by predicate abstraction of a single control location in the C program accepted the language. Therefore, they were determined to be equivalent by the L^* algorithm and were collapsed into a single one in the DFA.

The compatibility check saves an order of magnitude in terms of memory (upto 50%) as compared to the verification of the composition of components directly. This is primarily because the assumptions generated during the assume-guarantee reasoning were of a small size as compared to the component LKSs, which in turn led to lesser sizes of product automata. Also, we observed verification time improvements of order of upto 10% by implementing the query cache for the membership queries in the learner. Another 10% improvement in time was obtained by doing the *prefix-closed* optimizations (cf. Section 4) during learning.

7. CONCLUSIONS AND FUTURE WORK

The current work proposes a solution to the component substitutability problem using a regular language inference along with a model checker. Although we provide an approach oriented towards the new component for verifying global assembly properties during its evolution, there are several ways to improve its efficiency. For example, we would like to use the results from verifying the previous assembly while checking the compatibility of the new component. Further, since the earlier assembly was verified to be correct, only the new behaviors of the evolved component should be considered during compatibility.

This work also brings out several interesting avenues of research. The L^* algorithm is a general framework for learning regular languages. However, our goal is to learn program behavior, which are earmarked by specific characteristics, e.g., the language is prefix-closed. We intend to investigate more such domain-specific characteristics of programs which increase the efficiency of the inference algorithm. We also aim to develop an algorithm for learning LKSs directly instead of appealing to its language definition in terms of NFA.

We have used an assume-guarantee framework for learning only safety properties in this work. In order to extend its scope to liveness properties, we plan to study the learning of $\omega - regular$ [12] languages. Finally, we plan to focus on providing improved feedback to developers which will enable them to add missing features and/or fix bugs in the updated components.

8. REFERENCES

[1] ABB Inc. `http://www.abb.com`.

[2] D. Angluin. Learning regular sets from queries and counterexamples. In *Information and Computation*, volume 75(2), pages 87–106, November 1987.

[3] T. Berg, B. Jonsson, M. Leucker, and M. Saksena. Insights to Angluin's learning. Technical Report 2003-039, Department of Information Technology, Uppsala University, Aug. 2003.

[4] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *Integrated Formal Methods*, pages 128–147, 2004.

[5] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of ICSE 2003*, pages 385–395, 2003.

[6] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, M. Jurdzinski, and F. Y. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th International Conference on Computer-Aided Verification*, pages pp. 428–441. Lecture Notes in Computer Science 2404, Springer-Verlag, 2002.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.

[8] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619. Springer-Verlag, April 2003.

[9] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*. ACM Press, 2001.

[10] J. Ivers and N. Sharygina. Overview of ComFoRT: A model checking reasoning framework. *CMU/SEI-2004-TN-018*, 2004.

[11] MAGIC. `http://www.cs.cmu.edu/~chaki/magic`.

[12] O. Maler and L. Staiger. On syntactic congruences for omega-languages. In *Symposium on Theoretical Aspects of Computer Science*, pages 586–594, 1993.

[13] S. McCamant and M. D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, Olso, Norway, June 16–18, 2004.

[14] PACC website. `http://www.sei.cmu.edu/pacc`.

[15] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Information and Computation*, volume 103(2), pages 299–347, April 1993.

[16] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, London, 1997.

Compositional Quality of Service Semantics

Richard Staehli
Simula Research Laboratory
P.O. Box 134
N-1325 Lysaker, Norway
richard@simula.no

Frank Eliassen
Simula Research Laboratory
P.O. Box 134
N-1325 Lysaker, Norway
frank@simula.no

ABSTRACT

Mapping QoS descriptions between implementation levels has been a well known problem for many years. In component-based software systems, the problem becomes how to predict the quality of a service from the quality of its component services. The weak semantics of many QoS specification techniques has complicated the solution. This paper describes a model for defining the rigorous QoS semantics needed for component architectures. We give a detailed analysis of compositional QoS relations in a video `Object-Tracker` and discuss how the model simplifies the analysis.

1. INTRODUCTION

The need for QoS management support in component architectures is well known [4]. While many aspects of QoS management have been investigated in the context of distributed systems, component architectures present new challenges. One of those new challenges is how to reliably predict QoS properties for a composition of components.

Component architectures such as the CORBA Component Model (CCM) guarantee that applications *assembled* from independently developed components will function correctly when deployed on any sufficiently provisioned implementation of the component architecture platform. We refer to this as the *safe deployment property.* Although current component standards have had good success in some domains, such as e-business, an application that performs well in one deployment may be unusable as load scales up or when connections are re-distributed across low-bandwidth connections. We use the term *QoS-sensitive application* to refer to an application that will commonly perform unacceptably if platform resources are scarce or if the deployment is not carefully configured and tuned for the anticipated load.

State-of-the-art middleware and component technologies provide QoS management APIs that force application developers to code deployment-specific knowledge into the application [1][6][8]. Rather than specifying an assembly of black-box components that will run on any standard platform, developers instead write deployment-specific configuration code that depends on knowledge of component and platform service implementations. This approach complicates development and fails to preserve the safe deployment property.

1.1 The QuA Project

The QuA project is investigating how a component architecture can preserve the safe-deployment property for QoS-sensitive applications [9]. We believe that *platform-managed* QoS is the only general solution that preserves the safe deployment property. This means that applications and application components should be written without knowledge of the runtime platform implementation and resource allocation decisions. Application specifications for accuracy and timeliness of outputs must refer only to the logical properties of the component interfaces.

Platform-managed QoS means that the platform must be able to reason about how end-to-end QoS depends on the quality of component services. The composition relationship is often non-trivial as in the case of a remote video conference where image quality may depend on both packet transport loss and video encoding.

We refer to the set of characteristics used to specify QoS as a quality model and the concepts used to define such characteristics as a QoS meta-model. To enable a component platform to reason about composition and conformance, we need a QoS meta-model that allows us to uniformly model every level of implementation as a service, from the application level down to physical resources, and that, for any service, allows us to define a quality model that does not depend on implementation. These properties allow the creation of QoS specification standards and developer provided mapping functions that can be used to derive composition QoS properties from independently developed components.

In the next section, we describe how the QuA QoS meta-model satisfies the above requirements. In Section 3 the model is applied recursively to define QoS models for a complex real-time video processing application and its components. We show how the model permits a precise mapping of component-composition QoS dependencies. Section 4 discusses related work and Section 5 presents our conclusions.

2. THE QUA QOS META-MODEL

The QuA QoS Meta-Model (QQMM) defines the semantics of our QoS specifications. It defines a generic model of a service and defines quality in a way that allows us to create precise and practical QoS models for any specific service type.

To begin, we need to be precise about what a *service* is. In common use, the term service means work done for another. In computing systems, we want the term to apply both to the work performed by a complex distributed application for its clients and to the work performed by the simplest of hardware resources, such as a memory address that stores a binary value for some computation.

Definition 1 A *service* is a subset of output messages and causally related inputs to some composition of objects.

This is consistent with the common definition, since the output messages represent the "work done" and the inputs represent the work request. An object's type semantics define which inputs cause which outputs.

Since network packet delivery is a service that is well known in the QoS literature, we will use it to illustrate our definitions. We can model an IP network port as a distributed port object that can accept `send(packet)` messages at multiple locations and emit `deliver(packet)` messages at an endpoint identified by the host and port number. This service is illustrated in Figure 1. We can define a service provided by this port object as consisting of only the `deliver(packet)` output events and causally related `send(packet)` input events for which the input came from source A. These packets are colored black in Figure 1.

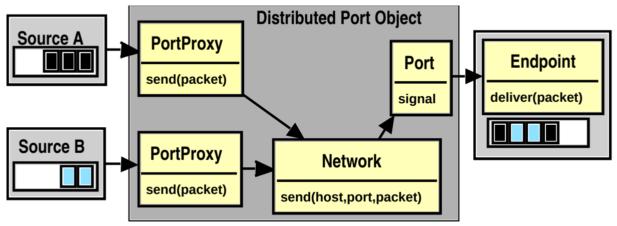


Figure 1: Example packet transport service.

In the QQMM definition of a service, we assume the most basic model of object-oriented analysis, where objects in some platform-defined identity space communicate by sending messages. The sending of a message is both an input event for the receiver and an output event for the sender. In the packet service, we can define the occurrence of the send event as precisely the moment when the send function is invoked and the occurrence of the deliver event as the precise moment when the received packet is made available in the output buffer. The QQMM makes no assumptions about object implementations and so we must allow that objects may receive input messages concurrently from multiple senders and send output messages concurrently to multiple receivers. This suits a general model of distributed computation.

As in the packet service example, other services may share the same object and even the same interface to that object, so long as they have a disjoint set of output events.

A service specification can be as simple as identifying an output interface for an object; the causally related inputs are implied by the semantics for the object type. For example, the semantics of the distributed port object dictate that every packet delivered has exactly one send event that caused it. The semantics also dictates that the value of the packet delivered should be the same as the packet that was sent.

Now, to reason about the behavior of a service, we need to talk about the history of input and output events.

Definition 2 A *message event trace* is a set of message values associated with the sending interface and the time it was sent.

A message value represents all content of the message, including its type or signature. The sending interface is the location of the mechanism used to send the message. In the packet service example, the sending interface used by `Source A` is the first `PortProxy` object, while the sending interface for packet service outputs is the `Endpoint` object. The time associated with an event comes from a local clock at the sending interface. We assume that clocks are synchronized, but acknowledge that when times from remote clocks are compared there will always be some uncertainty about how well they were synchronized.

The QQMM defines a message send to be a local and instantaneous event. All preparation for a message send is done in the sender and all processing of a message after the send event is done in the receiver. This property assures us that end-to-end processing time is consistently and properly accounted for. In the packet service example, all real delay occurs within one of the service components, including the `Network` object that encapsulates physical network access.

We use the term *input trace* to refer to the message event trace with all input events for a service, and *output trace* to refer to the message event trace with all output events for a service. The behavior of a service implementation is determined not only by the semantics of its declared interfaces, but also by the availability and scheduling of resources in the underlying platform. To define quality of service, we need to ask what is the ideal behavior of a service.

Definition 3 For a given input trace, the *ideal output trace* is generated when the service executes completely and correctly on an infinitely fast platform with unlimited resources.

That is, computation takes no time and results are obtained at the same time they are requested. Of course, events in an ideal output trace still only occur as frequently as the inputs that cause them!

Although we believe the QQMM can model QoS for nondeterministic computations, we ignore them for now to simplify the presentation. For deterministic computations, the semantics of a service's interfaces defines the ideal output trace as a function of an input trace. This means that the best possible quality for a service is always well defined.

In a real implementation of a service, the actual output trace will differ from the ideal in both the timing and value of message events. The causes for this deviation from ideal include finite CPU speed, queueing delays, and bandwidth reduction strategies. This is the stuff of QoS management.

We would like to view the possible output traces for a service as points in a behavioral space and consider distance from the ideal output trace as an error measurement, but to use this concept in QoS specifications, we must first define the dimensions of this behavioral space.

Consider again, the packet service example. Figure 2 shows the input trace as incoming messages on the left side

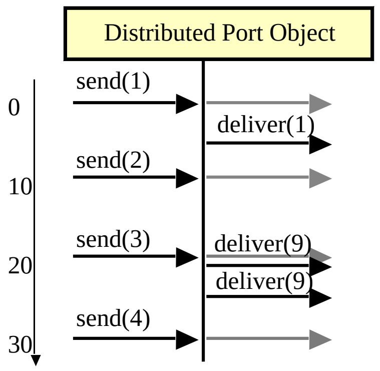


Figure 2: Ideal (shown in grey) versus actual (black) behavior.

of the service's timeline and the actual output trace as outgoing messages on the right. The ideal output trace is shown in grey. Assuming that this trace shows the entire lifetime of the service, we can see that each event in the ideal output trace can occur at a later time in the actual trace and may have its packet value corrupted or degraded in some way. The second and third packets have apparently been corrupted and it is not possible to be certain which is which without some knowledge of the service implementation. The fourth packet sent had not been delivered at the time of reckoning and may be considered lost.

The important point to observe about the packet service output is that every event can vary independently from the others in its delay from the ideal and in the change to its packet value. The QQMM generalizes from this example to make the assumption that the only possible values in an actual output trace that can differ from the ideal are the time of the output event and the values of message arguments. Each of these variables in the output trace represents an independent dimension of the behavioral space.

Let $\vec{X}$ be the vector of values that may differ in a trace X, ordered by event order in the ideal trace and by the order they occur in the message value. Then the difference between an actual output trace A and the ideal trace I is the difference vector $\vec{\delta} = \vec{A} - \vec{I}$.

For the packet service example, the values that may differ in the actual output trace are the event times and the packet values. If I is the ideal output trace for the example, the order of values in $\vec{I}$ is packet value followed by time for the first output, then for the second and so forth as shown in Equation 1.

$$\vec{I} = (1, 0, 2, 10, 3, 20, 4, 30) \quad (1)$$

If A is the actual output trace for the example, then the first two values of $\vec{A}$ are 1 and 5.

$$\vec{A} = (1, 5, 9, 21, 9, 24, 0, \infty) \quad (2)$$

As noted above, it is not possible to be certain which of the next two packets delivered corresponds to the second event in the ideal trace. This is an inescapable problem in QoS management: unreliable systems cannot be relied on to communicate enough information to unambiguously determine error. Instead, the difference vector can have many possible values depending on different interpretations of the correspondence between actual and ideal events [11]. Fortunately, it is usually safe to consider only the interpretation corresponding to the least error as it is unlikely that random faults would yield better service. So, we choose the following interpretation of $\vec{A}$ and the difference vector $\vec{\delta}$:

$$\vec{\delta} = (0, 5, 7, 11, 6, 4, -4, \infty) \quad (3)$$

Although this definition of the difference between actual and ideal allows us to define quality in a weak sense, i.e., actual traces in which every vector component is below a corresponding threshold value, it fails to tell us which of two output traces is better when each contain some components that are worse than the other. Another problem is that as the complexity of an ideal trace increases, through additional messages and more complex message structure, the number of ways in which actual traces may differ explodes. Fortunately, we frequently are concerned only with an overall measure of distance from the ideal. For example, we can ignore the individual values for event delay and instead monitor aggregate statistics such as maximum and median.

The QQMM concept of an error model provides a rigorous way to define the type of quality characteristics that are most useful for QoS management.

Definition 4 An *error model* $\vec{\epsilon} = \left(\epsilon_1(\vec{\delta}), ..., \epsilon_n(\vec{\delta})\right)$ is a vector of n functions that each map from a difference vector $\vec{\delta}$ to a real number such that $||\vec{\epsilon}(\vec{\delta})|| = 0$ when $||\vec{\delta}|| = 0$.

The notation $||\vec{\delta}||$ represents the magnitude of the vector $\vec{\delta}$. According to this definition, a function in an error model (error function) must be zero if there are no differences between the actual and the ideal. We also expect that the magnitude of an error model will generally increase as the difference from ideal grows larger, but we have found it difficult to formalize this requirement without being overly restrictive.

One trivial error model is the set of projection functions $\pi_i(\vec{\delta}) = \delta_i$; these are zero when $||\vec{\delta}|| = 0$ and increase as the respective component of $\vec{\delta}$ increases. But the value of our error model definition is that it allows us to construct a much simpler error space with the type of QoS characteristics commonly discussed in the QoS management literature. A point in this simplified error space represents an equivalence class of output traces that are the same distance from the ideal with respect to the error model.

To continue the packet service example, if we define the service as consisting of all `deliver(packet)` messages, then both the input trace and its associated ideal output trace may contain an unbounded number of events. To define an error model for this service we need to decide how to interpret packet delivery events that are expected but never happen, or that cannot be distinguished from other packet delivery events. For this error model, we consider a packet to be lost if either the time difference in $\vec{\delta}$ for its deliver event exceeds some limit, such as 20 seconds, or its value difference is non-zero. We associate a packet delivery event with an ideal delivery event if the the actual event happened after the ideal event and there is no other interpretation that offers fewer packet losses.

To allow us to model quality at time t during the service, we define $i(p, t, \vec{\delta})$ to be the sequence of consecutive values in $\vec{\delta}$ associated with ideal events in the interval $(t-p, t]$. Let $d(p, t, \vec{\delta})$ be the values in $i(p, t, \vec{\delta})$ associated with packets that were correctly delivered (not lost). A value of 10 seconds was chosen arbitrarily for the period p in this example.

We can then construct an error model with the following functions:

- $delay(t)$ is the mean of time differences in $d(10, t, \vec{\delta})$.
- $jitter(t)$ is the variance of time differences in $d(10, t, \vec{\delta})$.
- $loss(t)$ is the ratio of the number of packets lost in $i(10, t, \vec{\delta})$ to the number sent, or zero if none were sent.

These satisfy our definition of an error model, since each is zero when the difference values in $\vec{\delta}$ are zero and none decrease when a difference value in $\vec{\delta}$ increases.

Note that these definitions model only the error in the recent history of time t. To constrain error over the lifetime of a service we would need to use expressions like: for all time t, $delay(t) < 5$. We could define many other similar error models with different parameters or different functions. For example, it may be useful to model the 95th percentile of delay values. Still, the simple error model above is quite useful for specifying and measuring the quality of a packet delivery service and is similar to other common definitions of packet service QoS characteristics.

A point in this error space; say $delay(t) = 1$ second, $jitter(t) = 0.5$ second, and $loss(t) = 0.2$; corresponds to all output traces in which the mean delay for recent events before time t is one second, and so forth. This set of output traces forms a surface in the behavioral space that surrounds a neighborhood of the ideal output trace. Inside this neighborhood, all output traces are closer to the ideal and can be considered *better* at time t according to this error model.

We can use an error model to both quantify the *loss of quality* and to constrain it. Let $\epsilon(\vec{A} - \vec{I})$ be the tuple of error values for an error model ϵ, an actual trace A and the ideal trace for a service I. Let *limits* be a tuple of positive real numbers representing an upper bound for each of the functions in ϵ. Then we say a service with output trace A is acceptable if for each i, $|\epsilon_i(\vec{A} - \vec{I})| < limits_i$.

Since many error models can be defined for a given service, we would like some criteria to judge which error models are better than others. The QQMM allows us to formally define desirable properties of a good error model. For brevity, we suggest only informal definitions here. We say an error model is *sound* if any set of non-zero error limits can be satisfied by some set of actual output traces. An error model is *complete* if, for any output trace that is different from the ideal, we can find a set of non-zero error limits that would exclude this trace. An error model is *minimal* if no function can be removed without losing the ability to distinguish between some output traces. We say an error model M is more *expressive* than an error model N if it can define exactly the same sets of acceptable output traces as N, and then some more.

The example error model for the packet service appears to be sound, because any non-zero limits can be satisfied by output traces with non-zero delay, jitter, and loss. The error model also appears to be complete: for any time difference value $x > 0$ in some $i(10, t, \vec{\delta})$ with n successfully delivered packets, $x/(n + 1)$ is a non-zero delay limit that must be less than $|delay(t)|$ even if all $n - 1$ other time differences are zero, and if $v > 0$ is a value difference in such an interval with m packets, $1/(m+1)$ is a non-zero loss limit that must be less than $|loss(t)|$ even if all other packets are delivered in a timely manner without corruption.

The error model is also minimal, as each function models an independent facet of error. The error model could be made more expressive by adding functions, to distinguish packet corruption from loss for example.

3. APPLICATION TO AN OBJECT TRACKING SERVICE

In this section, we first define error models for a real application and its component services, and then show how error for the application service can be predicted from error in the component services. We apply QQMM to an example of a class of applications we refer to as real-time content-based video analysis. These applications must process the video data at least as fast as the video data is made available and perform analysis with acceptable accuracy.

Real-time content analysis is an active research field where efficient techniques have been found for problems such as multi-object detection and tracking. In such applications, pattern classification systems which automatically classify media content in terms of high-level concepts have been adopted. Such pattern classification systems must bridge the gap between the low-level signal processing services (filtering and feature extraction) and the high-level services desired by the end-user.

The design of such a pattern classification system must select analysis algorithms that balance requirements of accuracy against requirements of timeliness.

3.1 The object tracking service

In this example, we refer to a simple object tracking service as an `Object-Tracker`. The service is simple in the sense that it can track a single object on a stationary background (i.e. the camera remains still). This example is adapted from our earlier work on supporting quality constraints in real-time content-based video analysis [12].

Figure 3 shows the `Object-Tracker` as a component receiving input from a `VideoSrc` and sending output to an `EventSink`. The `Object-Tracker` is implemented as a composition of `Feature extractor` and `Classifier` components with multicast input to the feature extractors and publish/subscribe middleware for communications between the feature extractors and the classifier. The extractors and the classifier are further decomposed into functional and communication components.

The `VideoSrc` sends each frame of an uncompressed video stream to a filter component (not shown) that divides the frame into regions, publishing the data for each to a multicast channel for that region. The multicast channel is an efficient mechanism to communicate high-bandwidth data to multiple clients over a local area network. However, in this example, exactly one `Feature Extractor` receives the data for each frame region.

Each `Feature Extractor` calculates features from the set of frame data it has received. In this example, the `Feature Extractor` components compute a two dimensional array of motion vectors; one for each block of the frame region, where a block is a subdivision of the frame into fixed size rectangles of pixels. A motion vector indicates the direction

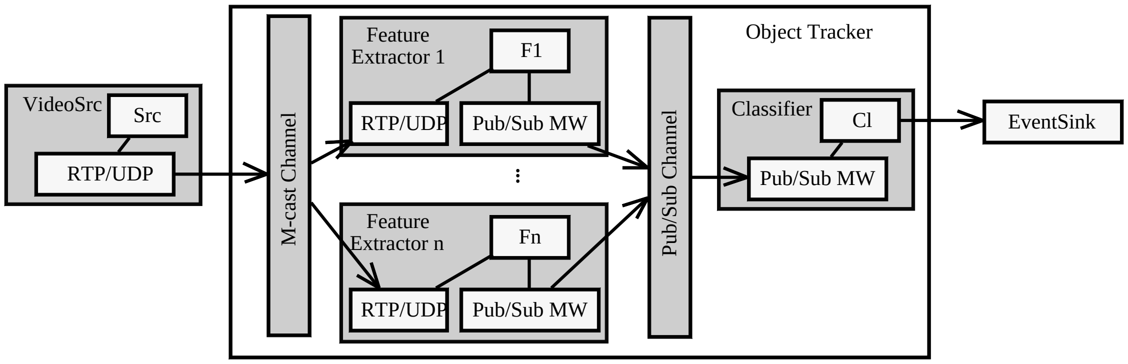


Figure 3: Functional decomposition of the `Object-Tracker`.

and distance that most pixels within a block appear to have moved from a previous frame.

The motion vectors and the associated frame number are then published by the `Feature Extractor` as event notifications on a channel for the frame region. The `Classifier` subscribes to these channels to receive motion vector data from multiple `Feature Extractor` components. In this example, the `Classifier` examines the history of motion vectors over several frames to identify and determine the center of a moving object.

We use Dynamic Bayesian Networks (DBNs) as a classifier specification language. DBNs [5] represent a particularly flexible class of pattern classifiers that allows statistical inference and learning to be combined with domain knowledge.

In order to automatically associate high-level concepts (e.g. object position) with the features produced through feature extraction, a DBN can be trained on manually annotated media streams. The goal of the training is to find a mapping between the feature space and high-level concept space, within a hypothesis space of possible mappings. After training the DBN can be evaluated by measuring the number of misclassifications on a manually annotated media stream not used in the training. This measured error rate can be used as an estimate of how accurately the DBN will classify novel media streams and can be associated with the classifier as meta data.

One important reason for using DBNs is the fact that DBNs allow features to be missing during classification at the cost of some decrease in accuracy. This fact is exploited in our work with the `Object-Tracker` to trade accuracy for timeliness as discussed below.

3.2 QoS role in planning service configuration

Even this simple `Object-Tracker` implementation requires many deployment configuration choices with associated QoS tradeoffs. Components may be deployed to different processors to work in parallel, increasing throughput and reducing delay. Throughput may also be increased by deploying pipeline components on different processors. But distribution may also increase communication overhead and add to delay.

Since the motion vector `Feature Extractor` operates locally on image regions, the performance can scale with the size of video processing task by distributing the work among a greater number of `Feature Extractor` components, each on its own processor. Similarly, the `Classifier` could be parallelized if classification should become a processing bottleneck [12]. In this paper, we consider only the case of parallelizing the `Feature Extractor`.

The amount of processing required for acceptable accuracy in object tracking is another tradeoff point. The level of accuracy provided by the `Object-Tracker` depends on the misclassification behavior (error rate) of the classifier algorithm, which in turn depends on the quality of the feature extraction input. Greater accuracy can be achieved by processing video data at a higher frame rate or higher resolution, but the increased processing requirements might increased delay in reporting the object location. Hence the configuration must be carefully selected based on both the desired level of accuracy and the tolerance for delay.

To reason about which configurations might satisfy the `Object-Tracker` QoS requirements, it must be possible to estimate what QoS the components will offer and how these QoS offers compose to satisfy the end-to-end requirements. To do this in practice, we exploit knowledge of the measured behavior of the components in the target physical processing environment.

3.3 Error model definitions

We model the `Object-Tracker` as a service that has uncompressed video frames as input and that produces a location event for every video frame as its output. A video frame number accompanies each frame, frame region, set of motion vectors, and each location output event so that there is no ambiguity about which frame these events are to be associated with. Each video frame is divided into $m \times n$ blocks. A location is represented as a block number in the video frame and indicates the center position of the tracked object.

For this service, the variables in the output trace are the time of the output and the location coordinates. It matters little whether the difference between actual and ideal location coordinates is in one axis or another, so we will refer to location as an aggregate value.

We would like to model three quality characteristics for this service: *latency*, *errorRate* and *period*. Let $i(p, t, \vec{\delta})$ be the values from the difference vector for the interval of period p up to time t as defined in the last section.

We can define the error model as follows:

- $\mathit{latency}(t)$ is the mean of time differences in $i(10, t, \vec{\delta})$.
- $\mathit{errorRate}(t)$ is the ratio of non-zero location differ-

ences in $i(10, t, \vec{\delta})$ to the total number of location values.

- *period*(t) is the maximum q such that location difference is non-zero for all values in $i(q, t', \vec{\delta})$, where $t - 10 \leq t' \leq t$.

The *latency*(t) is the mean of recent values for the elapsed time from when a frame containing a motion event is sent to the `Object-Tracker` until a causally related location event is output. The ideal latency is zero, but any real application will permit some latency greater than zero.

The *errorRate*(t) gives the fraction of the recent location difference values that were non-zero. The ideal error rate of zero may be achieved if all video blocks are processed and the moving object is similar to those used in classifier training.

The *period*(t) is the amount of time that may elapse before a updated and correct object location is reported. As with the *errorRate*, the ideal value of zero may be achieved with sufficient processing resources and good video input, but frame dropping will cause this error to increase.

Because the `Classifier` has the same output interface as the `Object-Tracker`, we can and should use the same error model. This shows a good feature of the QQMM: Error model definitions refer only to the output interface of a service and so an error model may be used for any service with the same output interface.

We model a `Feature Extractor` as a service that receives uncompressed video frames (or some region of a frame) as input and that produces a motion vector array and associated frame number as its output. The variables between actual and ideal output traces for this service are the time of output events and the motion vector array values.

For this example, we are not concerned with trading accuracy in the motion vectors against other quality dimensions so we again treat this complex data type as a single aggregate value in the following error model:

Let $i(p, t, \vec{\delta})$ be as defined in the last section.

- *latency*(t) is the mean time difference in $i(10, t, \vec{\delta})$.

- *errorRate*(t) is the ratio of non-zero motion vector array differences in $i(10, t, \vec{\delta})$ to the total number of motion vector array values.

These are the same definitions given for the functions of the same name in the previous error model except that here we reference the motion vector array as the output trace variable. Because we do not anticipate error in the deterministic algorithm for computing motion vectors, it might seem that *errorRate*(t) could be left out of our model, but this would leave our model incomplete and unable to express even the constraint that the motion vectors should be correct.

Our work with this application has been in a local area network environment where the remote communication has not been a bottleneck, but we understand that modeling the QoS of these communication links is necessary for future work. At this time, we have not defined error models for the the communication protocols or the component which divides the video frames into regions.

3.4 Modeling compositional QoS relations

To configure the `Object-Tracker` to perform with acceptable *errorRate*, *latency* and *period* requires an ability to predict these values for alternative configurations. Systems engineers accomplish this task with a combination of analysis and experimental observations of component behavior. The QQMM allows us to define an error model for each component service type that does not depend on its implementation, and thus can be used as a standard QoS specification model used by both component clients and independent component developers. Given such standards, a component developer can encode knowledge of the relation between component and composition QoS independently for each implementation. Thus, the QQMM enables a kind of QoS composition algebra: the algebraic operators are error prediction functions provided by the developer with each component implementation and the operands are the subcomponent QoS predictions.

As mentioned earlier, meta data measuring the *errorRate* for various configurations may be associated with the `Classifier`. For all of the components, measurements of processing time for a periodic task on a particular class of CPU and with a particular class of workload may be associated with the component type as meta data for use in estimating *latency*.

Another input to the *latency* prediction function is a model of the available computing resources in terms of both the number and class of CPUs, and the latency and bandwidth of communication between each pair of CPUs.

To simplify the estimation of *latency* for a service configuration, we assume that the communication between each pair of CPUs is contention free and that the latency and bandwidth are constant. These assumptions are valid for a significant class of distributed processing environments (e.g. dedicated homogeneous computers connected in a dedicated switched LAN) and allow us to ignore the complexity of communication contention, routing, etc., which are not the focus of this paper.

Given an allocation of components to processors, the processing period of each `Object-Tracker` component can be estimated. From the component QoS predictions and configuration values such as the classifier location output rate, the end-to-end error for this example of the `Object-Tracker` can be predicted.

The composition of *latency* for serial tasks, such as remote communication of video frame data and `Feature Extractor` processing of that frame, is the sum of the delays. The QQMM semantics ensure that this composition is seamless: that end-to-end accounting of time attributes every moment to exactly one service in the sequence. If the latency measurements follow these semantics than there is good reason to hope that the composition estimate will be good.

The serial composition of `Feature Extractor` and `Classifier` is not strictly serial processing however. The `Classifier` does not wait for all features from a frame to arrive before reporting a guess about the location, but instead is configured with its own periodic schedule to update hypotheses, including hypotheses about past. In this implementation, the composition trades the risk of an increase in the *errorRate* to avoid the high *latency* of waiting for every serial dependency. The end-to-end *latency* is held constant at runtime while the *errorRate* may vary.

The prediction of the *errorRate* can be made analytically from estimates of the availability of extracted features and knowledge of the classifier. The ARCAMIDE algorithm exploits the fact that DBNs allow features to be missing during classification to trade accuracy for timeliness [12]. Alternatives are generated for removing feature extractors from an initial configuration. The error prediction for the `Object-Tracker` is used to sort the alternatives, least increase in the *errorRate* first. This algorithm then tests each configuration to determine if the *latency* and *period* prediction will satisfy application requirements. In this sequence, the first service configuration which meets the *latency* and *period* requirements, must also meet the accuracy requirement, otherwise it is not possible to satisfy the requirements with the specified processing resources.

The *period* can be predicted from the absolute value of the difference between the classifier update period and the $VideoSrc$ frame period.

This example suggests that the error prediction for a composition can be a complex function of component interactions and component error. The QQMM semantics enable such error prediction functions to be written without knowledge of component implementations and thus, to predict error regardless of which component implementation is plugged in to the composition. However, we have only begun to define error models for real services and error predictions for compositions. Future work is needed to learn if there are important patterns for error prediction in compositions.

4. RELATED WORK

There has been little work published specifically addressing QoS models for component architectures. Researchers at BBN developed QuO (Quality of Service for Objects) [13] as a framework for management of QoS properties in CORBA applications. A more recent QuO paper introduced *qoskets* as a means for reusing adaptive QoS behaviors, but they have not yet adapted this work to a component architecture [10]. They specify QoS contracts between client and server objects, but do not address other service types such as a pipeline component that receives input from one object and sends output to another. Also, they do not focus on QoS semantics, so it is not clear that a description of QoS provided by one component would be understood correctly by an independent developer who would use such a component.

SLAng is a language for specifying service level agreements between very large grain components that may be owned and operated by separate commercial entities [3]. The authors acknowledge that SLAng's informal semantics for QoS characteristics is a weakness [2]. Our model is complementary and could provide a formal semantics and a method to expand the SLAng catalog with QoS characteristics for new service types.

Nahrstedt, et al., describe a QoS-aware middleware architecture designed to support QoS-sensitive applications [7]. They propose a QoS compiler to map from user-perceived QoS down to system-level QoS, but we understand that this compiler understands only a fixed set of QoS characteristics. They conclude that more research is needed to allow uniform specification of QoS for different application domains. We agree and we believe that QQMM provides a prerequisite common semantics independent of any particular middleware or component architecture.

5. CONCLUSIONS

In this paper we have described the QuA QoS Meta-Model (QQMM) for defining QoS semantics for an arbitrary service. The key features of the QQMM are a definition of a service that is based on component interface semantics and a definition of quality dimensions based on a metric space with ideal service behavior at the origin. Error models that conform to the QQMM define QoS dimensions that can be observed by a client without knowledge of the service implementation. This allows independent component developers to agree on a common standard for specifying client QoS requirements and component QoS offers. The QQMM allows us to analyze any implementation as a composition of component services and to define error models for these component services that fully account for loss of quality in the implementation. From this analysis, a component developer may be able to provide a mapping relation between QoS of component services and the QoS of a composition.

The QQMM provides clear guidelines for defining QoS measures with a rigorous semantics, but we have learned from initial experiments that it can be difficult to define precise error models that correspond to our intuition about quality dimensions. Despite this note of caution, we believe a strong semantics are a prerequisite for QoS management in component-base software engineering where representations of QoS properties come with "off-the-shelf" components.

5.1 Acknowledgment

Thanks to Jonathan Walpole for comments on an early draft and to the referees for their suggestions to improve the paper. This work was funded in part by a grant from the Research Council of Norway.

6. REFERENCES

[1] G. Coulson, G. S. Blair, M. Clarke, and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *ACM Distributed Computing Journal*, 15(2):109–126, 2002.

[2] D. Davide Lamanna and James Skene and Wolfgang Emmerich. Specification Language for Service Level Agreements, 2003. http://www.newcastle.research.ec-.org/tapas/deliverables/D2.pdf.

[3] Wolfgang Emmerich, D. Davide Lamanna, Giacomo Piccinelli, and James Skene. Method for service composition and analysis, 2003. http://www.newcastle.research.ec.org/tapas/deliverables/d3.pdf.

[4] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6), 2002.

[5] F. V. Jensen. Bayesian networks and decision graphs, 2001. Series for Statistics and Engineering and Information Science, Springer Verlag.

[6] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken. Specifying and measuring quality of service in distributed object systems. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, pages 20–22, Kyoto, Japan, 1998.

[7] Klara Nahrstedt, Dongyan Xu, Duangdao Wichadakul, and Baochun Li. QoS-aware middleware for ubiquitous computing. *IEEE Communications*

Magazine, 39(11):140–148, November 2001.

[8] I. Pyarali, D. Schmidt, and R. Cytron. Achieving end-to-end predictability of the TAO real-time CORBA ORB. In *Proceedings of the 8th IEEE Real-Time Technology and Applications Symposium*, San Jose, CA, 2002.

[9] Richard Staehli, Frank Eliassen. Component-Based Service Planning For Platform-Managed QoS. Submitted to Middleware 2004, 2004.

[10] R. Schantz, J. Loyall, M. Atighetchi, and P. Pal. Packaging quality of service control behaviors for reuse. In *Proceedings of ISORC 2002, The 5th IEEE International Symposium on Object-Oriented Real-time distributed Computing*, Washington, DC, 2002.

[11] Richard Staehli and Jonathan Walpole. Quality of service specifications for multimedia presentations. *Multimedia Systems*, 3(5/6), 1995.

[12] Viktor S. Vold Eide and Frank Eliassen and Ole-Christoffer Granmo and Olav Lysne. Supporting Timeliness and Accuracy in Distributed Real-Time Content-based Video Analysis. In *ACM Multimedia*, 2003.

[13] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3, 1997.

An Analysis Framework for Security in Web Applications

Gary Wassermann Zhendong Su
Department of Computer Science
University of California, Davis
{wassermg, su}@cs.ucdavis.edu

ABSTRACT

Software systems interact with outside environments (*e.g.*, by taking inputs from a user) and usually have particular assumptions about these environments. Unchecked or improperly checked assumptions can affect security and reliability of the systems. A major class of such problems is the improper validation of user inputs. In this paper, we present the design of a static analysis framework to address these input related problems in the context of web applications. In particular, we study how to prevent the class of SQL command injection attacks. In our framework, we use an abstract model of a source program that takes user inputs and dynamically constructs SQL queries. In particular, we *conservatively approximate* the set of SQL queries that a program may generate as a finite state automaton. Our framework then applies some novel checking algorithms on this automaton to indicate or verify the absence of security violations in the original application program. Work is in progress to build a prototype of our analysis.

1. INTRODUCTION

Web applications are designed to allow any user with a web browser and an internet connection to interact with them in a platform independent way. They are typically constructed in a two- or three-tiered architecture consisting of at least an application running on a web server, and a back-end database. Both components may have trust assumptions about their respective environments. The application may be designed with the assumption that users will only enter valid input as the programmer intended, in terms of both input values and ways of entering input. The back-end database may be set up with the assumption that the application will only send it authorized queries for the active user, in terms of both the types of actions the queries perform and the ranges of tuples the queries act on. All of these assumptions, if not checked properly, risk being violated, by malicious users.

Catching violations early (*e.g.*, at the application as opposed to at the database) is desirable in preventing malicious users from executing dangerous queries. However, the meta-programming aspect of these applications makes static checking difficult. A *meta-program* is a program in some source language that manipulates *object-programs*, perhaps by constructing object-programs or combining object-program fragments into larger object programs. In this sense, a Java/JDBC program or a CGI script that constructs SQL queries to retrieve information from a database is a meta-program. The source language is Java or Perl, and the target language is SQL.

1.1 SQL Command Injection

For web applications, one common class of security problems is the so-called *SQL command injection attacks* [8, 23]. We use a simple example to illustrate the problem. Many applications include code that looks like the following:

```
string query = "SELECT * FROM employee WHERE name
                        = '" + name + "'";
```

The user supplies the value of the `name` variable, and if the user inputs "John" (an expected value), then the `query` variable contains the string: `"SELECT * FROM employee WHERE name = 'John'"`. A malicious user, however, can input "`John' or 1=1--`," which results in the following query being constructed: `"SELECT * FROM employee WHERE name = 'John' OR 1=1--'"`. The "`--`" is the single-line comment operator supported by many relational database servers, including MS SQL Server, IBM DB2, Oracle, PostreSQL, and MySQL. In this way, the attacker can supply arbitrary code to be executed by the server and exploit the vulnerability.

Although the source language, *e.g.*, Java, may have a strong type system, it provides no guarantee about the dynamically generated SQL queries. Certainly direct string manipulation is a low-level programming model, but it is still widely used, and command injections do pose a serious threat both to legacy systems and to new code. A recent web-search easily revealed several sites susceptible to such attacks.

At the heart of command injections is an input validation problem, *i.e.*, to accept only certain expected inputs. Proper input validation turns out to be very difficult. Several techniques exist to address it, and we give an overview here. At a low level, input can either be filtered, so that "bad" inputs are rejected, or altered with the design of making all inputs "good." One suggested technique is to enumerate the strings that the programmer believes are necessary for an injection attack but not for normal use. If any of those strings appear as substrings in the input, either the input can be rejected, or they can be cut out, leaving usually nonsense or harmless code. Another common practice is to limit the length of input strings. More generally, inputs can be filtered by matching them against a regular expression and rejecting them if they do not match. An alternative is to alter input by adding slashes in front of quotes in order to prevent the quotes that surround literals from being closed within the input. Common ways to do this are with PHP's `addslashes` function and PHP's `magic_quotes` setting. Recent research efforts provide ways of systematically specifying and enforcing constraints on user inputs. PowerForms provides a domain-specific language to generate both client-side and server-side checks of constraints expressed as regular expressions [4]. Scott and Sharp propose using a proxy to enforce slightly more expressive constraints (*e.g.*, they can restrict numeric values of input) on individual user inputs [19]. A number of commercial products, such as Sanctum's AppShield [17] and Kavado's InterDo [11], offer similar strategies. One recent project proposes a type system to ensure that all data is "trusted"; that type system considers input to be trusted once it has passed through a filter [10]. Perl's "tainted mode" has a similar goal, but it operates at runtime [24].

All of these techniques are an improvement over unregulated input, but they have weaknesses. None of them can say anything about the syntactic structure of the generated queries, and all may still admit bad input. It is easy to miss important strings when enumerating "bad" strings, or to fail to consider the interactions between seemingly "safe" strings. Dangerous commands can be written quite concisely, so short strings are not necessarily "safe." Regular expression filters may also be under-restrictive. PHP's `addslashes` has led to some confusion because when used in combination with `magic_quotes`, the slashes get duplicated. Also, if a numeric input is expected and arbitrary characters can be entered, no quotes are needed to execute an injection attack. In the absence of a principled analysis to check these methods, they cannot provide security guarantees. Because vulnerabilities are known to be possible even when these measures are taken, black-box testing tools have been built. One from the research community is called WAVES (Web Application Vulnerability and Error Scanner) [9], and several commercial products also exist, such as AppScan [16], WebInspect [20], and ScanDo [11]. While testing can be useful in practice for finding vulnerabilities, it cannot be used to make guarantees either.

Other techniques deal with input validation by enforcing that all input will take the syntactic position of literals. Bind variables and parameters in stored procedures can be used as placeholders for literals within queries, so that whatever they hold will be treated as literals and not as arbitrary code. This is the most recommended practice because of increased security and performance. A recently proposed instruction set randomization for SQL in web applications has a similar effect [3]. It relies on a proxy to translate instructions dynamically, so SQL keywords entered as input will not reach the SQL server as keywords. These will not be acceptable solutions in the rare case when users are to be allowed to enter column names or anything more than literals. Also, these techniques guarantee that only the SQL code from the source program will be executed, but they cannot guarantee that those SQL queries will be "safe." There is currently no formal static verification technique to perform early detection of dangerous SQL commands in source code. Furthermore, although using stored procedures is less error-prone than string manipulation, many web applications have been written and continue to be written using string manipulation to construct dynamic SQL queries.

In this paper, we propose a static analysis framework to detect SQL command injection attacks. In our framework, we cast the SQL command injection problem as a version of the analysis of meta-programs [21] and propose a technique based on a combination of well-known automata-theoretic techniques [7], an extension of context-free language (CFL) reachability [15], and novel algorithms for checking automata for security violations [22].

1.2 Overview of Analysis Framework

In our framework, we assume that the user inputs are restricted with some regular expressions for input validation. The absence of such a filter means that all inputs are possible. The use of regular expressions makes possible the automatic generation of code for checking user inputs. We then statically verify that the regular expressions provide proper input checking such that no command injection is possible.

Our proposed analysis operates directly on the source code of the application. We consider Java programs in particular, but the programs can be written in any other language. Our analysis builds on top of two recent works on analysis of dynamically generated database queries, one for syntactic correctness [5] and one for type correctness [6].

Our analysis is split into two main steps. First, it starts with a conservative, dataflow-based analysis, similar to a pointer analysis [1], to approximate the set of possible queries that the program generates for a particular query variable at a particular program location. We take into account that the application programmer may check user input against a regular expression filter. The result for each query variable, *e.g.*, `query` in the earlier example, is a finite state automaton which represents a *conservative* set of possible string values that the variable can take at runtime.

In the second step, our analysis performs semantic checks on the generated automaton to detect security violations. In this step, two main checks are performed. First, we check access control against a given security policy specified by the underlying database. This also includes the detection of potential dangerous commands such as deleting a whole table. Second, we analyze the parts of the automaton corresponding to the `WHERE` clauses of the generated queries to check whether there is a tautology. The existence of a tautology indicates a potential vulnerability and the corresponding regular expressions need to be examined and perhaps redesigned. Knowing exactly which column the column names may refer to enables us to view the columns as variables in the object program. Whereas type checking of generated queries reasons about the types of these "variables," we reason about their values to check for tautologies in `WHERE` clauses. If no violations are detected, the soundness of our analysis guarantees that the original source-program does not produce any "threatening" SQL commands.

1.3 Paper Outline

The rest of the paper is structured as follows. We first present our analysis framework in detail (Section 2). In particular, we discuss the previous works on string analysis [5] (Section 2.1) and query structure discovery [6] (Section 2.2), followed by a discussion of the checks we perform

(Section 2.3). We then present an algorithm for tautology checking (Section 3) and discuss some current limitations of our approach and areas for future work (Section 4). Finally, we survey related work (Section 5) and conclude (Section 6).

2. ANALYSIS FRAMEWORK

In this section, we give a more detailed description of our analysis framework. We mentioned earlier that manual input filtering and validation of user inputs are error prone. In our framework, we suggest the use of regular expressions to filter user inputs. Our framework can then check the correctness of these regular expression specifications. Our analysis technique, however, is general and can also validate other input checking mechanisms, including the use of ad hoc input validation routines.

2.1 Abstract Model of Generated Queries

As the first step of our analysis, we build an abstract model of all the possible dynamically generated SQL queries by a source program. In particular, we consider programs written in Java.

This step of our analysis builds upon a string analysis of Java programs by Christensen *et al.* [5]. The string analysis approximates the set of possible strings that the program *may* generate for a particular string variable at a particular program location, which is called a *hotspot*. The string analysis represents the set of possible strings by generating a finite state automaton (FSA); that is, the set of strings the automaton accepts is a superset of the set of strings the program actually produces at that hotspot. For our purpose, the hotspots are the string variables that produce SQL query strings. For example, the string variable `query` at the statement:

```
        return statement.executeQuery(query);
```

is a hotspot for that program.

We can model regular expression filters, in the string analysis, as casts on the corresponding Java program variables; that is, all string values of a particular Java expression may be declared to be within a given regular expression. These casts can be thought of in much the same way as type casts in any typed programming language. We refer interested readers to Christensen *et al.*'s paper [5] for technical details on the string analysis.

Finally, the rest of our analysis requires that each FSA accepts only syntactically correct queries. We enforce this by first constructing an FSA which accepts an under approximation of the SQL language. By intersecting it with the FSA for the generated queries, we ensure syntactic correctness. (Section 4 discusses some limitations imposed by this approach.)

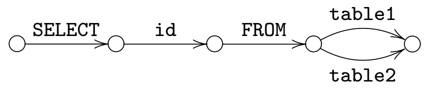
2.2 Syntactic Structure of Generated Queries

In order to analyze the FSA representation of database queries, we need to understand the queries' syntactic structure. We utilize aspects of earlier work on static type checking of generated queries [6] to discover the parsing structure of queries. Discovering the structure allows us to locate `WHERE` clauses to check for tautologies, for example. For individual programs, the structure is obtained by parsing the program according to the language's grammar. Our situation is different: instead of individual programs, we have an FSA which may accept a potentially infinite set of programs (database queries, in this context).

We use an extension of the context-free language (CFL) reachability algorithm [14, 15] to simulate parsing on the FSA. The CFL-reachability problem takes as inputs a context-free grammar G with terminals T and nonterminals N, and a directed graph A with edges labeled with symbols from $T \cup N$. Let S be the start symbol of G, and $\Sigma = T \cup N$. A path in the graph is called an *S-path* if its word is derived from the start symbol S. The CFL-reachability problem is to find all pairs of vertices s and t such that there is an S-path between s and t.

The SQL-language grammar and the generated FSA are inputs to the CFL-reachability algorithm. We need to extend the standard CFL reachability algorithm to record which edges in the graph led to the addition of each new edge to find the structure of every query accepted by the FSA. For example, it tells us not only that there is a `SELECT` statement starting at vertex s and ending at vertex t, but it also tells us every path between s and t that accepts a `SELECT` statement and whether each segment of each path is a `WHERE` clause, a column-list, or something else.

Having the complete structure of every query in the set enables the analysis to match each column name with the set of columns it may refer to. Note that because of the branching structure of the FSA, column names may refer to any of a set of columns, as in the following example:



To facilitate the next phase of analysis, we modify the FSA by adding transitions labeled with fully-qualified column identifiers (*e.g.*, `id.table1`) parallel to transitions labeled with column names. Further details regarding structure discovery can be found in Gould *et al.*'s paper [6].

2.3 Security Checking of Generated Queries

In the final step, we check for various security violations in the generated queries. We mention two of the main checks that one can perform.

2.3.1 Checking Access Control Policies

Access control policies grant entities permissions on resources. Our analysis checks the generated queries against some given access control policy for the database.

DBMSs usually use role-based access control (RBAC) [2], in which the entities are roles (*e.g.*, administrator, manager, employee, customer, etc.) and users act as one of these roles when accessing the database. The active role for each hotspot is an input to our analysis. The permissions include, for example, `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `DROP`, etc. The resources are tables and columns. As a result of "parsing" the FSA with CFL-reachability, we know for each column transition, all contexts (*e.g.*, `SELECT`, `INSERT`, etc.) in which it may appear in the generated queries. We use this to discover access control violations. For example, if the role `customer` does not have the `INSERT` permission on `id.table2`, even if `id.table2` is mentioned in a `SELECT` subquery of an `INSERT` statement, we will discover and flag the violation.

2.3.2 Detecting Tautologies

The second main check we perform on the generated SQL queries is to verify the absence of tautologies from all `WHERE` clauses. Generally, if an honest user wants to return all tuples for a query, the query will not have a `WHERE` clause. In the context of web applications, a tautology in a `WHERE` clause is an almost-certain sign of an attack, in which the attacker attempts to circumvent limitations on what web users are allowed to do.

Detecting generated tautologies is a non-trivial task. Earlier work on type checking dynamically generated queries [6] reasons about the types of constants, columns, and expressions. Tautology checking, on the other hand, has to reason about values, which is a much deeper semantic analysis than type checking.

To discover tautologies, we first extract the portions of the FSA that accept the conditional expressions in `WHERE` clauses, which we call *Boolean FSAs*. Detecting tautologies in acyclic portions of the FSA is straightforward because acyclic portions accept only a finite set of expressions. Cycles in the FSA make tautology detection challenging. We classify cycles as either *arithmetic* or *logical*, depending on the sort of expressions they accept. We conceptually view arithmetic portions of the FSA as network flow problems, and solve them using a decision procedure for first-order arithmetic. Logical loops cannot be handled this way. Instead, we "unroll" them the minimal number of times needed to ensure that if any tautology is accepted, at least one will be found. The next section explains tautology detection in more detail. If a tautology is discovered, we issue a warning.

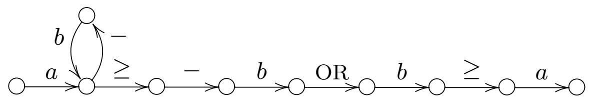
3. CHECKING FOR TAUTOLOGIES

For web applications, a tautology in the `WHERE` clause of a database query indicates a highly likely command injection problem. Perhaps the attacker wants to view all the information in a database, where only a subset is intended for any given user. In another setting, user names and passwords may be stored in a database so that the application authenticates users by querying the database to check for the supplied name and password. A tautology in such a query would nullify the authentication mechanism.

Checking for tautologies is challenging because tautologies may be non-trivial, such as "`(a > b) OR NOT ((b - 1 > c) AND (2 - b - c > -a - b))`." In fact, the general problem is undecidable because of the undecidability of solving Diophantine equations [13].

We restrict ourselves in this paper to discovering tautologies in linear arithmetic ("+" and "−" but no "×") over real numbers. Multiplication by a constant is within linear arithmetic, and we include it in our algorithm when it appears in an acyclic region of the FSA. However, for an FSA that has, for example, a loop over "× 2," if we attempt to include all multiplication by constants, we would characterize the multiplication as "× $2^{\mathtt{n}}$" for some `n`. Exponentiation with variables is difficult to reason about, so when multiplication appears in a cyclic region of the FSA or has variables as its multiplicands, we flag a warning. Columns of type `Integer` are approximated by real numbers. Relations over strings (*e.g.*, "'a'='a'") can be translated into questions over numbers, for example by mapping strings to numbers.

If the set of queries represented by the automaton is infinite, it is because the automaton has cycles. Cycles in the automaton come from both cyclic behavior in the source program, either from looping control structures or recursion, and repetition in regular expression filters (*e.g.*, "`*`"). Although cycles are finite structures, a single pass through a cycle may not reveal everything we need to know. Multiple passes through even a simple loop may be needed to discover a tautology. Consider the following example:



After two passes through the loop, the automaton accepts the string "`a - b - b` $\geq$ `- b OR b` $\geq$ `a`," which is semantically equivalent to "`a` $\geq$ `b OR b` $\geq$ `a`", a tautology.

3.1 Our Approach

In formulating an analysis to discover tautologies in the presence of cycles in a Boolean FSA, we first note a useful consequence of the syntactic-correctness property: the transitions of the Boolean FSA can be partitioned into four *transition types*. A transition of type:

(I) accepts part of an arithmetic expression ({`+`, `-`, `()`, `1`, `x`, ...}) before a comparison operator;

(II) accepts a comparison operator ($\{>, \geq, =, \leq, <, \neq\}$);

(III) accepts part of an arithmetic expressions after a comparison operator;

(IV) accepts a logical operator ({`AND`, `OR`, `NOT`}) or a parenthesis at the logical level.

This partitioning must be possible because, for example, if a transition that accepts a constant could be classified as both type I and type III, then the FSA would accept some string in place of a comparison expression which either had two comparison operators (*e.g.*, "...`x > 5 < 5`...") or none (*e.g.*, "...`AND 5 OR`..."). Consider also the notion of *parenthetic nesting* for each transition in a path as the number of logical (arithmetic) open parentheses minus the number of logical (arithmetic) closed parentheses encountered since the beginning of the path. Although a transition may be encountered on many different paths, it will always have the same parenthetic nesting. If this were not so, the FSA would accept some string with imbalanced parentheses.

Our analysis relies on this partitioning. Rather than trying to handle arbitrary cycles in the FSA uniformly, we classify each cycle as either *arithmetic*, if it only includes type I or type III transitions, or *logical*, if it includes type IV transitions. In order to handle each class of cycles without concerning ourselves with the other, we define an *arithmetic FSA* such that it can be viewed in isolation when we address arithmetic cycles and it can be abstracted out when we address logical cycles:

- The start state s immediately follows a type IV transition and immediately precedes a type I transition;
- The final state t immediately follows a type III transition and immediately precedes a type IV transition;
- All states and transitions are included that are reachable on some s-t path that has no type IV transitions.

The FSA fragment in Figure 1 has two arithmetic FSAs. The one defined by (s_1, t) includes all states and solid transitions in the figure. The one defined by (s_2, t) excludes the state s_1 and the x-transition. Finding the arithmetic

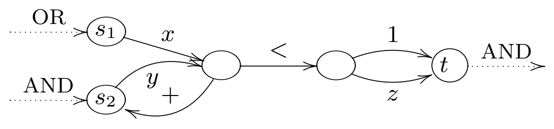
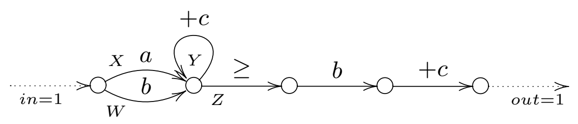


Figure 1: Example for arithmetic FSAs.



$$
\begin{array}{ll}
\exists W, X, Y, Z : & \} \text{ flow variables} \\
\quad 1 = X + W \quad \wedge & \\
\quad X + W + Y = Y + Z & \left.\right\} \text{ flow balance equations} \\
\quad \wedge \quad Z = 1 & \\
\forall a, b, c : & \} \text{ object-program variables} \\
\quad W \times (a) + X \times (b) + Y \times (c) \geq Z \times (b + c) & \} \text{ flow-comparison expression}
\end{array}
$$

Figure 2: Flow equations for arithmetic loops.

FSAs in a Boolean FSA is straightforward in our framework. The structure discovery from Section 2.2 adds a transition between each pair of states that accepts a comparison expression—these states become s and t in an arithmetic FSA. The structure discovery adds to the new transition references to the transitions that allowed it to be added—these transitions are followed to find the states and transitions between s and t.

For reasoning about comparison expressions, which arithmetic FSAs accept, we view arithmetic FSAs as network flow problems with single source and sink nodes and solve these problems using a construction in linear arithmetic (see Section 3.2). Boolean expressions are comparison expressions connected with logical operators (*e.g.*, "AND," "OR," "NOT"). We discover tautologies by unrolling logical loops a bounded number of times sufficient to ensure that if a tautology is accepted, we will find one. We simulate unrolling by repeating instances of the network flow problems. We determine the precise number of times to unroll based on the structure of the strong connections among arithmetic FSAs and the number of object-program variables in each arithmetic FSA (see Section 3.3).

3.2 Arithmetic Loops

We address arithmetic loops by casting questions about arithmetic automata as questions about network flows. We present the technique by the example in Figure 2. We consider the path taken as the FSA accepts a string to be a flow. Except at the entrance and exit states, each state's in-flow must equal its out-flow. In other words, if on an accepting path, a state is entered three times, then on the same path it must also be exited three times.

In order to capture this intuition, we label the incoming and outgoing transitions at each state where branching or joining occurs. In the example, we label four transitions as W, X, Y, and Z. The labels become the variable names for the *flow variables.* The value of a flow variable equals the number of times the corresponding transition was taken in some accepting path. For the start state, the final state, and each state with branching or joining, we write *flow balance equations.* The label of each transition entering that state appears on one side of the equation and the label of each transition leaving appears on the other. For the start and final states of the arithmetic automaton, we specify a value of "1" entering and leaving respectively.

Paths through the FSA accept expressions of constants and *object-program variables* (*i.e.*, column names, in the present context). A tautology is an expression true for all values of the variables, so we universally quantify the object-program variables named in the FSA.

Finally, we write *flow-comparison expressions* to link the flow through the FSA to the semantics of the accepted expression. In flow-comparison expressions, flow variables are multiplied by the expressions on their corresponding paths because each trip through a path adds the expression that labels the path to the accepted string. In Figure 2, $\{W, Y, Z \leftarrow 1; X \leftarrow 0\}$ makes the expression *true*, and corresponds to the string "`b + c` $\geq$ `b + c`." Additional expressions can prevent most false positives (*e.g.*, by preventing path variables from taking negative values), but we do not discuss them here due to space constraints.

Tarski's theorem [22] establishing the decidability of first-order arithmetic guarantees that expressions of this form are decidable when the variables range over real numbers. We state here a soundness result:

THEOREM 3.1. *If we do not discover a tautology then the FSA does not accept a tautology.*

Furthermore, when two or more arithmetic FSAs are linked in a linear structure by logical connectors (*e.g.*, "AND" or "OR"), we can merge in a natural way the equations we generate to model the arithmetic automata, and the soundness result holds for the sequence of automata:

THEOREM 3.2. *If we do not discover a tautology, then the linear chain of arithmetic FSAs does not accept a tautology.*

Incompleteness Allowing the variables to range over real numbers does leave a margin of incompleteness. If the flow variables take on non-integral values, they will not correspond to any path through the FSA. We discuss this further in Section 4.

3.3 Logical Loops

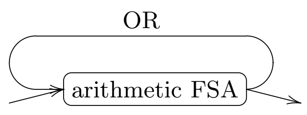


Figure 3: A simple logical loop.

Consider the simple abstract FSA in Figure 3. The arithmetic FSA might not accept any tautology, but two or more passes through the arithmetic FSA joined by "OR" may be a tautology.

Unfortunately, we cannot use equations to address logical loops as we did for arithmetic loops. If we did, the equations for arithmetic loops would not be expressible in first-order

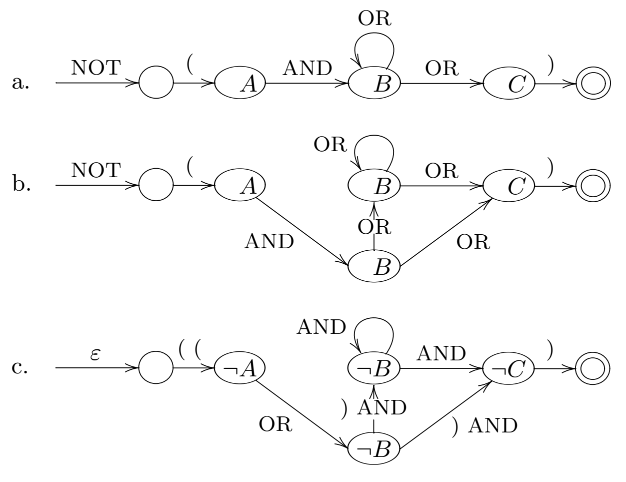


Figure 4: Removing "NOT" from a Boolean FSA.

arithmetic. Instead, we "unroll" the loop enough times that if the loop accepts some tautology, the unrolling must also accept some tautology. This section presents our technique for discovering tautologies in the presence of logical loops by explaining how we address transitions labeled with each of the four logical keywords: NOT, OR, AND, and (), in that order.

3.3.1 NOT-transitions

The first phase of the analysis takes as input a Boolean FSA F and transforms it into a Boolean FSA F', such that the sets of expressions that F and F' accept are logically equivalent, and F' has no transitions labeled "NOT." Figure 4 illustrates this transformation. Labeled states represent FSAs that accept comparison expressions (as in Figure 3). Because "AND" has a higher precedence than "OR," applying DeMorgan's law to a negated expression requires that parentheses be added to preserve the precedence in the original expression. However, because we are dealing with FSAs, not single expressions, adding parentheses along one path may lead to imbalanced parentheses on another path. To address this, the transformation first duplicates states that have differently labeled incoming or outgoing transitions. For example, the state B in the original Boolean FSA in Figure 4a has incoming transitions labeled "AND" and "OR," so it gets duplicated as in Figure 4b. The transformation then adds parentheses at transitions that terminate sequences of AND-transitions, flips the AND's and the OR's, and flags the states with "¬." When a state is flagged with "¬," the comparison operators in the arithmetic FSA get swapped with their opposites (*e.g.*, $< \rightleftarrows \geq$). Figure 4c shows the last step on the example.

3.3.2 OR-transitions

By Theorem 3.2, we can determine whether a linear boolean FSA accepts any tautologies. In this section, given an arbitrary Boolean FSA which has only OR-transitions, we generate a finite set of linear Boolean FSAs such that at least one accepts a tautology iff the original Boolean FSA accepts a tautology.

If all strongly connected components (SCCs) in an FSA are viewed as single states, the FSA is acyclic and all paths

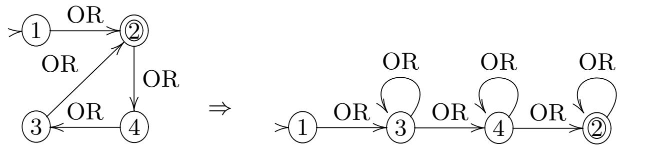


Figure 5: Transforming a complex looping structure into multiple self-loops.

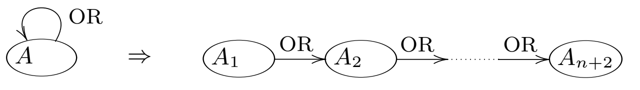


Figure 6: Logical-loop unrolling.

through it can be enumerated. The paths can be used to produce a finite set of linear FSAs, and iff the original FSA accepts a tautology, one of the linear FSAs accepts a tautology. The following lemma allows us to transform complex looping structures of SCCs into linear sequences of states with self-loops, as in Figure 5.

LEMMA 3.3. *Let F be a Boolean FSA with only OR-transitions which is linear except for SCCs. If F is transformed into F' by allowing only unique incoming and outgoing transitions for each state (so that F' is linear) and adding a self-loop to each state which was originally in an SCC, F accepts a tautology iff F' accepts a tautology.*

Lemma 3.3 follows directly from the commutative property of "OR." If we can determine the maximum number of times each state with a self-loop must be visited to discover a tautology, we can "unroll" the self-loops that number of times to produce linear Boolean FSAs. The following theorem yields this number:

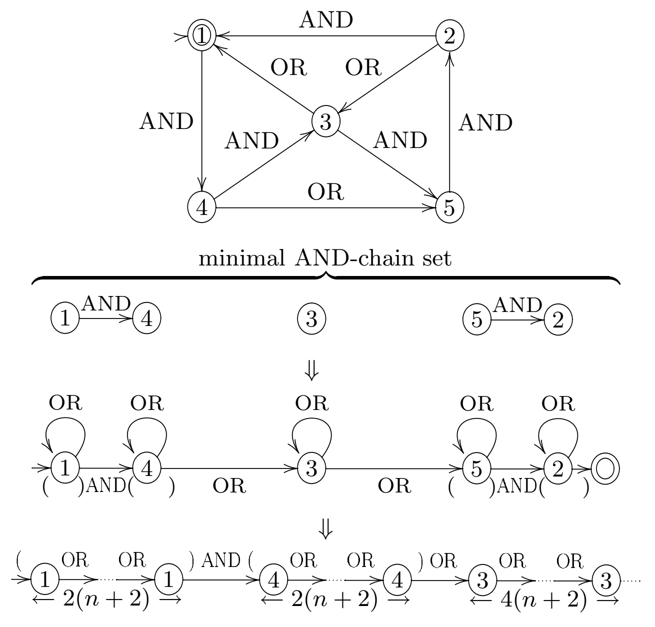
THEOREM 3.4. *Let T be an expression of the form $t_1 \vee \ldots \vee t_m$, where each t_i is a comparison of two linear arithmetic expressions. Let S map expressions to sets by mapping an expression E to the set of comparisons in E, so that $S(T) = \{t_1, \ldots, t_m\}$. T is a tautology, iff there exists some tautology T', such that $S(T') \subseteq S(T)$ and $|S(T')| \leq n + 2$, where n is the number of variables named in T.*

Due to space constraints we omit the proof of Theorem 3.4. The theorem is established through a connection between the maximum number of comparisons needed for a tautology and the maximum number of linearly independent vectors in n dimensions. Figure 6 illustrates how we use Theorem 3.4: if A represents an arithmetic FSA, and a total of n distinct program variables label the transitions of the FSA, the loop can be unrolled $n + 2$ times to guarantee that if the loop accepts all or part of a tautology, the unrolling does too.

3.3.3 AND-transitions

This section extends the algorithm from Section 3.3.2 to deal with AND-transitions. Because "AND" has a higher precedence than "OR," we cannot simply put self-loops on all states in an SCC. The following definitions will be useful in our algorithm:

DEFINITION 3.5 (AND-CHAIN). *An* AND-chain *is a sequence of states in an SCC connected sequentially by AND-transitions where OR-transitions in the SCC immediately*



AND
OR
OR
AND
AND
AND
AND
OR
minimal AND-chain set
AND
AND
⇓
OR
)AND(
OR
⇓
$\Leftarrow 2(n+2) \Rightarrow$
$\Leftarrow 2(n+2) \Rightarrow$
$\Leftarrow 4(n+2) \Rightarrow$

Figure 7: Forming a linear FSA from a strongly connected component to discover tautologies.

precede and follow the first and last states in the sequence respectively.

Definition 3.6 (Minimal AND-chain set). *The* minimal AND-chain set *of an SCC in a Boolean FSA is a subset S of the set of all AND-chains of an SCC, such that there are no pairs of AND-chains where the states in one AND-chain form a subset of the states in the other.*

Lemma 3.7. *Let F be a Boolean FSA with OR- and AND-transitions, and let F be linear except for SCC's which are entered and exited through OR-transitions. Let F be transformed into F' by replacing the SCC's with their minimal AND-chain sets, connecting them linearly with OR-transitions, and adding an OR-transition from the last to the first state of each AND-chain. F accepts a tautology, iff F' accepts a tautology.*

Lemma 3.7 follows first from the commutative property of "OR" because the order in which AND-chains occur does not influence whether or not a tautology is accepted. The minimal AND-chain set can be used because the conjunction of two non-tautologies can never form a tautology. An algorithm to construct this set finds all states in an SCC with incoming OR-transitions and from those states all acyclic paths which terminate at the first encountered state with an outgoing OR-transition. Figure 7 shows the minimal AND-chain set for an example SCC. In pathological cases this algorithm will discover an exponential number of AND-chains, but we expect this number to be small in practice.

Lemma 3.7 specifies a transformation from FSA F to F' such that F accepts a tautology iff F' accepts a tautology. The distributive property of "AND" can be used to transform F' into a linear FSA of states with self-loops and transitions with parentheses which accepts a tautology iff F' accepts a tautology. We create such an FSA directly from the AND-chains, as shown in Figure 7.

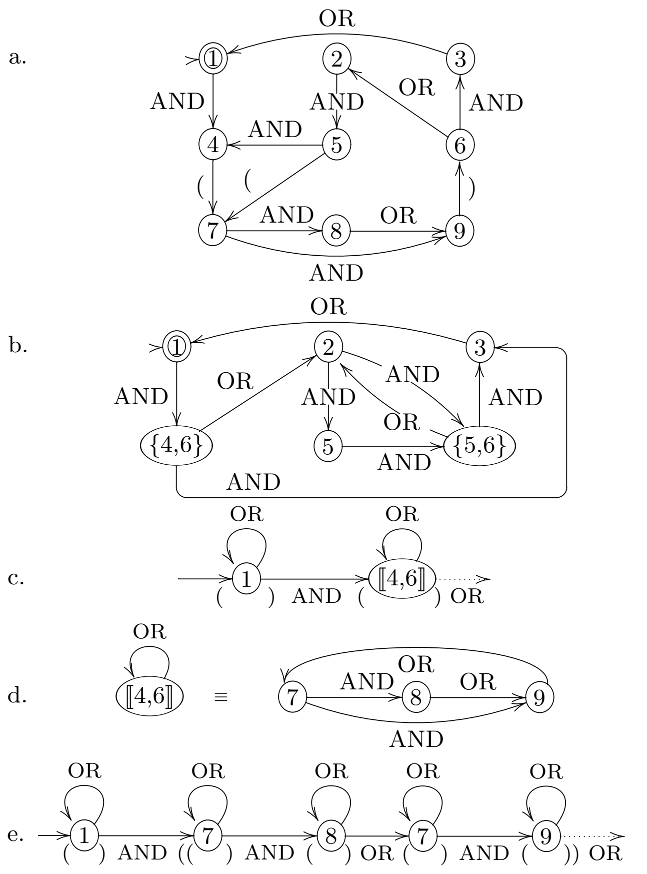


Figure 8: Forming a linear FSA from a strongly connected component with parentheses.

We can put an upper bound on the number of times each self-loop must be unrolled using Theorem 3.4. To find this number, we consider an example. Suppose an SCC has two AND-chains: (1) and (2)–(3). From these AND-chains we can construct a linear FSA F with self-loops as in Figure 7. We can also construct two sets of states where each set has exactly one state from each AND-chain: {(1), (2)} and {(1), (3)}. From these sets we can construct FSAs F_1 and F_2 where both F_1 and F_2 have only OR-transitions and the states have self-loops. The FSA F accepts a tautology iff F_1 and F_2 each accepts a tautology. The "only if" direction is straightforward. To prove the "if" direction, consider that if F_1 accepts the tautology "e_1 `OR` e_2," and F_2 accepts the tautology "e'_1 `OR` e_3," then F accepts "e_1 `OR` e'_1 `OR (`e_2`) AND (`e_3`)`." This expression in conjunctive normal form is "`(`e_1 `OR` e'_1 `OR` e_2`) AND (`e_1 `OR` e'_1 `OR` e_3`)`," a tautology. By Theorem 3.4 the self-loops in F_1 and F_2 need be unrolled at most $n+2$ times, where n is the number of variables that label the transitions in F_1 and F_2. A self-loop over state i in F must be unrolled $m(n+2)$ times, where m is the product of the numbers of states in the AND-chains which do not include state i. Figure 7 shows the final FSA with the unrollings of self-loops.

3.3.4 ()-transitions

This section extends the algorithm from Section 3.3.3 to deal with transitions labeled "(" and ")." Because parentheses have a higher precedence than “AND,” we discover AND-chains only among states and transitions of the FSA that have a common parenthetic nesting depth. Recall from Section 3.1 that parentheses must be balanced on all paths, and each state has a unique parenthetic nesting depth. Figure 8 illustrates this algorithm on the FSA in Figure 8a. Before the algorithm discovers AND-chains at depth i, it collapses pairs of states that enter/exit depth $i+1$ into single states, and temporarily removes all states and transitions at depths $> i$. For example, in Figure 8a, states (4) and (5) enter depth 1 and state (6) exits, so $\{4,6\}$ is one pair and $\{5,6\}$ is another pair. Figure 8b shows the FSA with collapsed states ($\{4,6\}$) and ($\{5,6\}$). The *meaning* of a collapsed states ($\{q_s,q_t\}$) is the sub-automaton that can be entered from state q_s and exited from state q_t, and is written ($[\![q_s,q_t]\!]$). The algorithm finds all AND-chains in the FSA, creates a linear FSA with self-loops (as in Figure 7), and replaces collapsed states with their meanings. Figure 8c shows only the beginning of this FSA in order to use the AND-chain (1)–($[\![4,6]\!]$) as an example. Figure 8d shows the sub-automaton that ($[\![4,6]\!]$) with a self-loop represents. In order to “unroll” the self-loop on ($[\![4,6]\!]$), the algorithm recurses on the represented sub-automaton. In this case, the sub-automaton has AND-chains (7)–(8) and (7)–(9). The algorithm produces a linear FSA with self-loops for this sub-automaton, and puts it in place of ($[\![4,6]\!]$). Figure 8e shows the result. When the FSA has no more collapsed states, the self-loops can be unrolled as in Figure 7.

The algorithm for analyzing Boolean FSAs is both sound and complete:

Theorem 3.8 (Soundness and Completeness). *Given a decision procedure for flow-comparison expressions, our algorithm discovers a tautology in an FSA F iff F accepts a tautology and accepts only syntactically correct expressions of comparisons of linear arithmetic expressions.*

Theorem 3.8 follows from Lemma 3.7 and the distributive property of “AND.” A tautology discovered in a linear FSA can be mapped back to a path in the original FSA for the purpose of a useful error message.

3.4 Complexity

The removal of NOT-transitions (Section 3.3.1) runs in time linear in the size of F, *i.e.*, $O(|F|)$, and expands F by a constant factor. The number of paths through F is exponential in the number of “acyclic” (cannot be reached from themselves) states in F, *i.e.*, $O(2^{|F_{\text{acyc}}|})$. Each path is a query to decision procedure. The number of AND-chains is exponential in the number of “strongly connected” (can be reached from themselves) states in F, *i.e.*, $O(2^{|F_{\text{sc}}|})$. The length of each path is bounded by either the number of acyclic states or the product of the number of AND-chains and the size of the alphabet, *i.e.*, $O(\max(|F_{\text{acyc}}|, 2^{2^{|F_{\text{sc}}|}}|\Sigma|))$. Therefore the number of queries is exponential and the size of each query is also exponential. For this analysis, we consider each query as being sent to an oracle.

Although in the worst case this algorithm runs in exponential time, we expect this to scale well because FSAs based on real-world programs typically do not have large and complex structures.

4. LIMITATIONS AND FUTURE WORK

In this section, we discuss some limitations of our current analysis and leave them for future work.

The first limitation lies in the way that we ensure syntactic correctness of the generated queries. The use of an FSA under-approximation of the SQL grammar may be too restrictive to remove some possible malicious queries from the represented set (Section 2.1). Based on the results from earlier work [5,6], we do not expect this in practice. We can also check for automata containment to make sure that the generated queries are syntactically correct.

The second limitation is our use of a decision procedure for first-order arithmetic over real numbers to solve our network flow problems (Section 3.2). It may be possible that the path variables could admit a tautology by taking on non-integral values which do not correspond to a path in the FSA. This makes our analysis incomplete. However, we do not view this as a serious limitation, because the analysis remains sound by modeling integer variables with real values. It is possible to address this by finding a decision procedure for the particular kind of constraints we have by exploiting their simple structure.

We do not yet have good ways to handle some operators, such as “`LIKE`” and “$\times$.” Generated constants pose similar problems for automata-based analyses. Questions about each of these is decidable in the absence of certain classes of loops, so loop unrolling algorithms, similar to the algorithm in Section 3.3, may provide good approximations.

Finally, to experimentally validate the effectiveness of our analysis framework, we are working on a prototype of the analysis and planning to apply it to some real-world examples.

5. RELATED WORK

In this section, we survey closely related work. Two previous projects are closely related to this work. The first is the string analysis of Christensen, Møller, and Schwartzbach [5]. Their string analysis ensures that the generated queries are syntactically correct. However, it does not provide any semantic correctness guarantee of the generated queries. The second, which builds on this string analysis, is on type checking of generated queries by Gould, Su, and Devanbu [6]. Their analysis takes the first step in the semantic checking of object-programs by ensuring that all generated queries are type-correct. Our analysis builds on these and goes a step further by checking deeper semantic properties.

Several tutorials are available on how to create web applications safely to avoid SQL command injection [8]. The only other research that we know of intended specifically for preventing command injection attacks uses instruction set randomization [3]. That technique relies on an intermediary system to translate instructions dynamically; our analysis is completely static, so it adds nothing to the run-time system. Several other techniques are mentioned in Section 1.

Several other automata-based techniques have been proposed with security in view, but they use automata in a fundamentally different way. For example, Schneider proposed formalizing security properties using security automata, which define the legal sequences of program actions [18]. In contrast, our analysis uses automata to represent values of variables at specified program points (hotspots).

To be put in a broader context, our research can be viewed as an instance of providing static safety guarantee for meta-programming [21]. Macros are a very old and established meta-programming technique; this was perhaps the first setting where the issue of correctness of generated code arose. Powerful macro languages comprise a complete programming facility, which enable macro programmers to create complex meta-programs that control macro-expansion and generate code in the target language. Here, basic syntactic correctness, let alone semantic properties, of the generated code cannot be taken for granted, and only limited static checking of such meta-programs is available. The levels of static checking available include none, syntactic, hygienic, and type checking. The widely used `cpp` macro pre-processor allows programmers to manipulate and generate arbitrary textual strings, and it provides no checking. The programmable syntax macros of Weise & Crew [25] work at the level of correct abstract-syntax tree (AST) fragments, and guarantee that generated code is syntactically correct with respect (specifically) to the C language. Weise & Crew macros are validated via standard type-checking: static type-checking guarantees that AST fragments (e.g., Expressions, Statements, etc.) are used appropriately in macro meta-programs. Because macros insert program fragments into new locations, they risk "capturing" variable names unexpectedly. Preventing variable capture is called hygiene. Hygienic macro expansion algorithms, beginning with Kohlbecker *et al.* [12] provide hygiene guarantees. Recent work, such as that of Taha & Sheard [21], focuses on designing type checking of object-programs into functional meta-programming languages. We do not introduce new languages or new language designs. In this particular work, our goal is to ensure that strings passed into a database from an arbitrary Java program are "non-threatening" SQL queries from the perspective of a given database security policy. We expect the general technique outlined in this paper can be extended to apply in other settings as well.

6. CONCLUSIONS

We have presented the design of the first static analysis framework for verifying a class of security properties for web applications. In particular, we have presented techniques for the detection of SQL command injection vulnerabilities in these applications. Our analysis is sound. We are currently working on an implementation of the analysis. Based on encouraging results from earlier work on syntactic and semantic checking of dynamically generated database queries and properties of the constructions presented in this paper, we expect our analysis to work well in practice and have a low false positive rate. Finally, we expect our analysis technique may be applicable in some other meta-programming paradigms.

7. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, University of Copenhagen, May 1994.

[2] M. Bishop. *Computer Security: Art and Science.* Addison Wesley Professional, 2002.

[3] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *ACNS*, 2004.

[4] C. Brabrand, A. Møller, M. Ricky, and M. I. Schwartzbach. Powerforms: Declarative client-side form field validation. *World Wide Web*, 2000.

[5] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. SAS'03*, pages 1–18, 2003. URL: `http://www.brics.dk/JSA/`.

[6] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. ICSE'04*, May 2004.

[7] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Language, and Computation.* Addison-Wesley, Reading, MA, 1979.

[8] M. Howard and D. LeBlanc. *Writing Secure Code.* Microsoft Press, 2002.

[9] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In *World Wide Web*, 2003.

[10] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *World Wide Web*, pages 40–52, 2004.

[11] Kavado, Inc. InterDo Vers. 3.0, 2003.

[12] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Conference on LISP and Functional Programming*, 1986.

[13] Y. Matiyasevich. Solution of the tenth problem of hilbert. *Mat. Lapok*, 21:83–87, 1970.

[14] D. Melski and T. Reps. Interconvertbility of set constraints and context-free language reachability. In *Proc. PEPM'97*, pages 74–89, 1997.

[15] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. POPL'95*, pages 49–61, 1995.

[16] Sanctum Inc. Web Application Security Testing-Appscan 3.5. URL: `http://www.sanctuminc.com`.

[17] Sanctum Inc. AppShield 4.0 Whitepaper., 2002. URL: `http://www.sanctuminc.com`.

[18] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[19] D. Scott and R. Sharp. Abstracting application-level web security. In *World Wide Web*, 2002.

[20] SPI Dynamics. Web Application Security Assessment. SPI Dynamics Whitepaper, 2003.

[21] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proc. PEPM'97*, 1997.

[22] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 1951.

[23] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison Wesley Professional, 2001.

[24] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl (3rd Edition).* O'Reilly, 2000.

[25] D. Weise and R. Crew. Programmable syntax macros. In *Proc. PLDI'93*, pages 156–165, 1993.

Synthesis of "correct" adaptors for protocol enhancement in component-based systems[*]

Marco Autili, Paola Inverardi, Massimo Tivoli
University of L'Aquila
Dip. Informatica
via Vetoio 1, 67100 L'Aquila
{marco.autili, inverard, tivoli}@di.univaq.it

David Garlan
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891
garlan@cs.cmu.edu

ABSTRACT

Adaptation of software components is an important issue in *Component Based Software Engineering* (CBSE). Building a system from reusable or *Commercial-Off-The-Shelf* (COTS) components introduces a set of problems, mainly related to compatibility and communication aspects. On one hand, components may have incompatible interaction behavior. This might require to restrict the system's behavior to a subset of safe behaviors. On the other hand, it might be necessary to enhance the current communication protocol. This might require to augment the system's behavior to introduce more sophisticated interactions among components. We address these problems by enhancing our architectural approach which allows for detection and recovery of incompatible interactions by synthesizing a suitable coordinator. Taking into account the specification of the system to be assembled and the specification of the protocol enhancements, our tool (called *SYNTHESIS*) automatically derives, in a compositional way, the glue code for the set of components. The synthesized glue code implements a software coordinator which avoids incompatible interactions and provides a protocol-enhanced version of the composed system. By using an assume-guarantee technique, we are able to check, in a compositional way, if the protocol enhancement is consistent with respect to the restrictions applied to assure the specified safe behaviors.

1. INTRODUCTION

Adaptation of software components is an important issue in *Component Based Software Engineering* (CBSE). Nowadays, a growing number of systems are built as composition of reusable or *Commercial-Off-The-Shelf* (COTS) components. Building a system from reusable or from COTS [14] components introduces a set of problems, mainly related to communication and compatibility aspects. Often, components may have incompatible interaction behavior. This might require to restrict the system's behavior to a subset of safe behaviors. For example, restrict to the subset of deadlock-free behaviors or, in general, to a specified subset of desired behaviors. Moreover, it might be necessary to enhance the current communication protocol. This requires augmenting the system's behavior to introduce more sophisticated interactions among components. These enhancements (i.e.: protocol transformations) might be needed to achieve dependability, to add extra-functionality and/or to properly deal with system's architecture updates (i.e.: components aggregating, inserting, replacing and removing).

We address these problems enhancing our architectural approach which allows for detection and recovery of incompatible interactions by synthesizing a suitable coordinator [6, 9, 11]. This coordinator represents a initial glue code. So far, as reported in [6, 9, 11], the approach only focussed on the restriction of the system's behavior to a subset of safe (i.e.: desired) behaviors. In this paper, we propose an extension that makes the coordinator synthesis approach also able to automatically transform the coordinator's protocol by enhancing the initial glue code. We implemented the whole approach in our *SYNTHESIS* tool [9, 15] (*http://www.di.univaq.it/tivoli/SYNTHESIS/synthesis.html*). In [15], which is a companion paper, we also apply *SYNTHESIS* to a real-scale context. Since in this paper we are focusing only on the formalization of the approach, in Section 5, we will simply refer to an explanatory example and we will omit implementation details which are completely described in [15].

Starting from the specification of the system to be assembled and from the specification of the desired behaviors, *SYNTHESIS* automatically derives the initial glue code for the set of components. This initial glue code is implemented as a coordinator mediating the interaction among components by enforcing each desired behavior as reported in [6, 9, 11]. Subsequently, taking into account the specification of the needed protocol enhancements and performing the extension we formalize in this paper, *SYNTHESIS* automatically derives, in a compositional way, the enhanced glue code for the set of components. This last step represents the contribution of this paper with respect to [6, 9, 11]. The enhanced glue code implements a software coordinator which

[*]This work is an extended and revisited version of [7].

avoids not only incompatible interactions but also provides a protocol-enhanced version of the composed system. More precisely, this enhanced coordinator is the composition of a set of new coordinators and components assembled with the initial coordinator in order to enhance its protocol. Each new component represents a wrapper component. A wrapper intercepts the interactions corresponding to the initial coordinator's protocol in order to apply the specified enhancements without modifying [1] the initial coordinator and the components in the system. The new coordinators are needed to assemble the wrappers with the initial coordinator and the rest of the components forming the composed system. It is worthwhile noticing that, in this way, we are readily compose-able; we can treat the enhanced coordinator as a new *composite* initial coordinator and enforce new desired behaviors as well as apply new enhancements. This allows us to perform a protocol transformation as composition of other protocol transformations by improving on the reusability of the synthesized glue code.

When we apply the specified protocol enhancements to produce the enhanced coordinator, we might re-introduce incompatible interactions avoided by the initial coordinator. That is, the enhancements do not hold the desired behaviors specified to produce the initial coordinator. In this paper, we also show how to check if the protocol enhancement holds the desired behaviors enforced through the initial coordinator. This is done, in a compositional way, by using an assume-guarantee technique [5].

The paper is organized as follows: Section 2 discusses related work. Section 3 introduces background notions helpful to understand our approach. Section 4 illustrates the technique concerning the enhanced coordinator synthesis. Section 5 formalizes the coordinator synthesis approach for protocol enhancement in component-based systems, and uses a simple explanatory example to illustrate the ideas. Section 6 formalizes the technique used to check the consistency of the applied enhancements with respect to the enforced desired behaviors. Section 7 discusses future work and concludes.

2. RELATED WORK

The approach presented in this paper is related to a number of other approaches that have been considered in the literature. The most closely related work is the scheduler synthesis for discrete event physical systems using supervisory control [3]. In those approaches system's allowable executions are specified as a set of traces. The role of the supervisory controller is to interact with the running system in order to cause it to conform to the system specification. This is achieved by restricting behavior so that it is contained within the desired behavior. To do this, the system under control is constrained to perform events only in strict synchronization with a synthesized *supervisor*. The synthesis of a supervisor that restrict behaviors resembles one aspect of our approach defined in Section 4, since we also eliminate certain incompatible behaviors through synchronized coordination. However, our approach goes well beyond simple behavioral restriction, also allowing augmented interactions through protocol enhancements.

Recently a reasoning framework that supports modular checking of behavioral properties has been proposed for the compositional analysis of component-based design [4, 10]. In [4], they use an automata-based approach to capture both input assumptions about the order in which the methods of a component are called, and output guarantees about the order in which the component calls external methods. The formalism supports automatic compatibility checks between interface models, where two components are considered to have compatible interfaces if there exists a legal environment that lets them correctly interact. Each legal environment is an adaptor for the two components. However, they provide only a consistency check among component interfaces, but differently from our work do not treat automatic synthesis of *adaptors* of component interfaces. In [10], they use a game theoretic approach for checking whether incompatible component interfaces can be made compatible by inserting a converter between them which satisfies specified requirements. This approach is able to automatically synthesize the converter. The idea they develop is the same idea we developed in our precedent works [6, 9, 11]. That is the restriction of the system's behavior to a subset of safe behaviors. Unlike the work presented in this paper, they are only able to restrict the system's behavior to a subset of desired behaviors and they are not able to augment the system's behavior to introduce more sophisticated interactions among components.

Our research is also related to work in the area of protocol adaptor synthesis [18]. The main idea of this approach is to modify the interaction mechanisms that are used to glue components together so that compatibility is achieved. This is done by integrating the interaction protocol into components. However, they are limited to only consider syntactic incompatibilities between the interfaces of components and they do not allow the kind of protocol transformations that our synthesis approach supports.

In other previous work, of one of the authors, we showed how to use formalized protocol transformations to augment connector behavior [13]. The key result was the formalization of a useful set of connector protocol enhancements. Each enhancement is obtained by composing wrappers. This approach characterizes wrappers as modular protocol transformations. The basic idea is to use wrappers to enhance the current connector communication protocol by introducing more sophisticated interactions among components. Informally, a wrapper is new code that is interposed between component interfaces and communication mechanisms. The goal is to alter the behavior of a component with respect to the other components in the system, without actually modifying the component or the infrastructure itself. While this approach deals with the problem of enhancing component interactions, unlike this work it does not provide automatic support for composing wrappers, or for automatically eliminating incompatible interaction behaviors.

In other previous work, by two of the authors, we showed how to apply protocol enhancements by dealing with components that might have syntactic incompatibility of interfaces [16]. However the approach described in [16] is limited only to consider deadlock-free coordinator.

3. BACKGROUND

In this section we discuss the background needed to understand the approach that we formalize in Section 4.

3.1 The reference architectural style

[1] This is needed to achieve compose-ability in both specifying the enhancements and implementing them.

The starting point for our work is the use of a formal architectural model of the system representing the components to be integrated and the connectors over which the components will communicate [12]. To simplify matters we will consider the special case of a generic layered architecture in which components can request services of components below them, and notify components above them. Specifically, we assume each component has a top and bottom interface. The top (bottom) interface of a component is a set of top (bottom) ports. Connectors between components are synchronous communication channels defining top and bottom ports.

Components communicate by passing two types of messages: notifications and requests. A notification is sent downward, while a request is sent upward. We will also distinguish between two kinds of components (i) *functional components* and (ii) *coordinators*. Functional components implement the system's functionality, and are the primary computational constituents of a system (typically implemented as COTS components). Coordinators, on the other hand, simply route messages and each input they receive is strictly followed by a corresponding output. We make this distinction in order to clearly separate components that are responsible for the functional behavior of a system and components that are introduced to aid the integration/communication behavior.

Within this architectural style, we will refer to a system as a *Coordinator-Free Architecture* (CFA) if it is defined without any coordinators. Conversely, a system in which coordinators appear is termed a *Coordinator-Based Architecture* (CBA) and is defined as *a set of functional components directly connected to one or more coordinators, through connectors, in a synchronous way.*

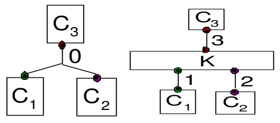


Figure 1: A sample of a CFA and the corresponding CBA

Figure 1 illustrates a CFA (left-hand side) and its corresponding CBA (right-hand side). C_1, C_2 and C_3 are functional components; K is a coordinator. The communication channels identified by 0, 1, 2 and 3 are connectors.

3.2 Configuration formalization

To formalize the behavior of a system we use *High level Message Sequence Charts* (HMSCs) and *basic Message Sequence Charts* (bMSCs) [1] specification of the composed system. From it, we can derive the corresponding CCS (*Calculus of Communicating Systems*) processes [8] (and hence *Labeled Transitions Systems* (LTSs)) by applying a suitable adaptation of the translation algorithm presented in [17]. HMSC and bMSC specifications are useful as input language, since they are commonly used in software development practice. Thus, CCS can be regarded as an internal specification language. Later we will see an example of derivation of LTSs from a bMSCs and HMSCs specification (Section 5.1).

To define the behavior of a *composition* of components, we simply place in parallel the LTS descriptions of those components, hiding the actions to force synchronization. This gives a CFA for a set of components.

We can also produce a corresponding CBA for these components with equivalent behavior by automatically deriving and interposing a "*no-op*" coordinator between communicating components. That coordinator does nothing (at this point), it simply passes events between communicating components (as we will see later the coordinator will play a key role in restricting and augmenting the system's interaction behavior). The "*no-op*" coordinator is automatically derived by performing the algorithm described in [6, 9, 11].

Formally, using CCS we define the CFA and the CBA for a set of components $C_1, .., C_n$ as follows:

Definition 1. Coordinator Free Architecture (CFA) $CFA \equiv (C_1 \mid C_2 \mid ... \mid C_n) \backslash \bigcup_{i=1}^{n} Act_{C_i}$ where for all $i = 1, .., n$, Act_{C_i} is the action set of the CCS process C_i.

Definition 2. Coordinator Based Architecture (CBA) $CBA \equiv (C_1[f_1^0] \mid C_2[f_2^0] \mid ... \mid C_n[f_n^0] \mid K) \backslash \bigcup_{i=1}^{n} Act_{C_i}[f_i^0]$ where for all $i = 1, .., n$, Act_{C_i} is the action set of the CCS process C_i, and f_i^0 are relabelling functions such that $f_i^0(\alpha) = \alpha[i/0]$ for all $\alpha \in Act_{C_i}$; K is the CSS process corresponding to the automatically synthesized coordinator.

By referring to [8], $\mid$ is the *parallel composition* operator and $\backslash$ is the *restriction* operator. There is a finite set of visible actions $Act = \{a_i, \overline{a}_j, b_h, \overline{b}_k, ...\}$ over which α ranges. We denote by $\overline{\alpha}$ the action complement: if $\alpha = a_j$, then $\overline{\alpha} = \overline{a}_j$, while if $\alpha = \overline{a}_j$, then $\overline{\alpha} = a_j$. By $\alpha[i/j]$ we denote a substitution of i for j in α. If $\alpha = a_j$, then $\alpha[i/j] = a_i$. Each $Act_{C_i} \subseteq Act$. By referring to Figure 1, 0 identifies the only connector (i.e: communication channel) present in the CFA version of the composed system. Each relabelling function f_i^0 is needed to ensure that the components $C_1, ..C_n$ no longer synchronize directly. In fact by applying these relabelling functions (i.e.: f_i^0 for all i) each component C_i synchronizes only with the coordinator K through the connector i (see right-hand side of Figure 1).

3.3 Automatic synthesis of failure-free coordinators

In this section, we simply recall that from the MSCs specification of the CFA and from a specification of desired behaviors, the old version of *SYNTHESIS* automatically derives the corresponding deadlock-free CBA which satisfies each desired behavior. This is done by synthesizing a suitable coordinator that we call failure-free coordinator. Informally, first we synthesize a "no-op" coordinator. Second, we restrict its behavior by avoiding possible deadlocks and enforcing the desired behaviors. Each desired behavior is specified as a *Linear-time Temporal Logic* (LTL) formula (and hence as the corresponding *Büchi Automaton*) [5]. Refer to [6, 9, 11] for a formal description of the old approach and for a brief overview on the old version of our *SYNTHESIS* tool.

4. METHOD DESCRIPTION

In this section, we informally describe the extension of the old coordinator synthesis approach [6, 9, 11] that we formalize in Section 5 and we implemented in the new version of the *SYNTHESIS* tool. The extension starts with

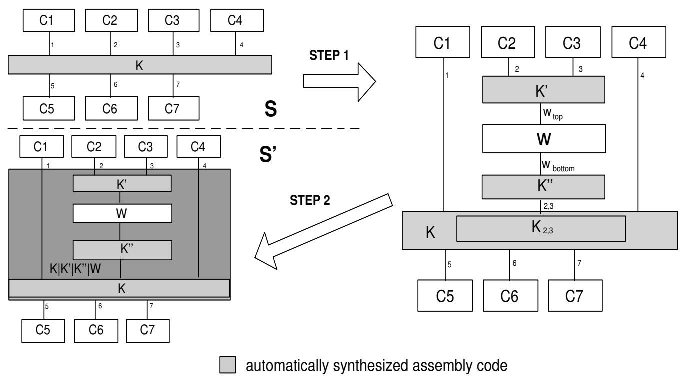


Figure 2: 2 step method

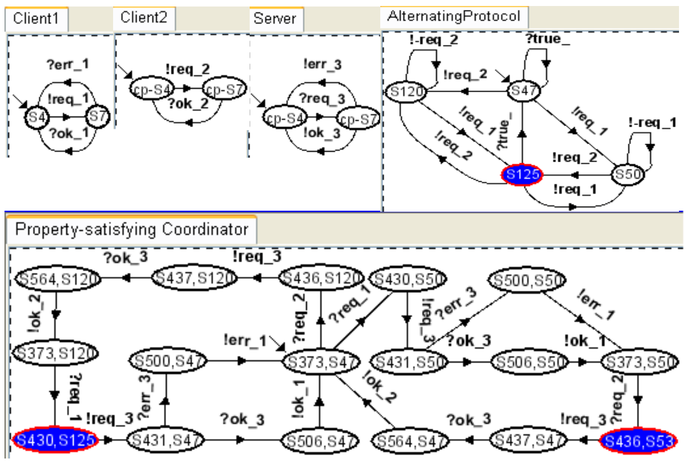


Figure 3: *LTSs* specification of S and Büchi Automata specification of P

a deadlock-free CBA which satisfies specified desired behaviors and produces the corresponding protocol-enhanced CBA.

The problem we want to face can be informally phrased as follows: *let P be a set of desired behaviors, given a deadlock-free and P-satisfying CBA system S for a set of black-box components interacting through a coordinator K, and a set of coordinator protocol enhancements E, if it is possible, automatically derive the corresponding enhanced, deadlock-free and P-satisfying CBA system S'.*

We are assuming a specification of: i) S in terms of a description of components and a coordinator as LTSs, ii) P in terms of a set of Büchi Automata, and of iii) E in form of bMSCs and HMSCs specification. In the following, we discuss our method proceeding in two steps as illustrated in Figure 2.

In the first step, by starting from the specification of P and S, if it is possible, we apply each protocol enhancement in E. This is done by inserting a wrapper component W between K (see Figure 2) and the portion of S concerned with the specified protocol enhancements (i.e.: the set of $C2$ and $C3$ components of Figure 2). It is worthwhile noticing that we do not need to consider the entire model of K but we just consider the *"sub-coordinator"* which represents the portion of K that communicates with $C2$ and $C3$ (i.e.: the *"sub-coordinator"* $K_{2,3}$ of Figure 2). $K_{2,3}$ represents the *"unchangeable"*[2] environment that K *"offers"* to W. The wrapper W is a component whose interaction behavior is specified in each enhancement of E. Depending on the logic it implements, we can either built it by scratch or acquire it as a pre-existent *COTS* component (e.g. a data translation component). W intercepts the messages exchanged between $K_{2,3}$, $C2$ and $C3$ and applies the enhancements in E on the interactions performed on the communication channels 2 and 3 (i.e.: connectors 2 and 3 of Figure 2). We first decouple K (i.e.: $K_{2,3}$), $C2$ and $C3$ to ensure that they no longer synchronize directly. Then we automatically derive a behavioral model of W (i.e.: a LTS) from the bMSCs and HMSCs specification of E. We do this by exploiting our implementation of the translation algorithm described in [17]. Finally, if the insertion of W in S allows the resulting composed system (i.e.: S' after the execution of the second step) to still satisfy each desired behavior in P, W is interposed between $K_{2,3}$, C_2 and C_3. To insert W, we automatically synthesize two new coordinators K' and K''. In general, K' always refers to the coordinator between W and the components affected by the enhancement. K'' always refers to the coordinator between K and W. By referring to Section 3.1, to do this, we automatically derive two behavioral models of W: i) W_TOP which is the behavior of W only related to its *top* interface and ii) W_BOTTOM which is the behavior of W only related to its *bottom* interface.

[2]Since we want to be readily compose-able, our goal is to apply the enhancements without modifying the coordinator and the components.

In the second step, we derive the implementation of the synthesized glue code used to insert W in S. This glue code is the actual code implementing K'' and K'. By referring to Figure 2, the parallel composition K_{new} of K, K', K'' and W represents the enhanced coordinator.

By iterating the whole approach, K_{new} may be treated as K with respect to the enforcing of new desired behaviors and the application of new enhancements. This allows us to achieve compose-ability of different coordinator protocol enhancements (i.e.: modular protocol's transformations). In other words, our approach is compositional in the automatic synthesis of the enhanced glue code.

5. METHOD FORMALIZATION

In this section, by using an explanatory example, we formalize the two steps of our method. For the sake of brevity we limit ourselves to formalize the core of the extended approach. Refer to [2] for the formalization of the whole approach.

In Figure 3, we consider screen-shots of the *SYNTHESIS* tool related to both the specification of S and of P. *Client*1, *Client*2 and *Server* are the components in S. *Property-satisfying Coordinator* is the coordinator in S which satisfies the desired behavior denoted with *AlternatingProtocol*. In this example, *AlternatingProtocol* is the only element in the specification P. The CBA configuration of S is shown in the right-hand side of Figure 1 where C_1, C_2, C_3 and K are *Client*1, *Client*2, *Server* and *Property-satisfying Coordinator* of Figure 3 respectively.

Each LTS describes the behavior of a component or of a coordinator instance in terms of the messages (seen as I/O actions) exchanged with its environment[3]. Each node is a state of the instance. The node with the incoming arrow (e.g.: the state $S4$ of $Client1$ in Figure 3) is the starting state. An arc from a node n_1 to a node n_2 denotes a transition from n_1 to n_2. The transition labels prefixed by "!" denote output actions (i.e.: sent requests and notifications), while the transition labels prefixed by "?" denote input actions (i.e.: received requests and notifications). In each transition label, the symbol "_" followed by a number denotes the identifier of the connector on which the action has been performed. The filled nodes on the coordinator's LTS denote states in which one execution of the behavior specified by the Büchi Automaton *AlternatingProtocol* has been accomplished.

Each Büchi Automaton (see *AlternatingProtocol* in Figure 3) describes a desired behavior for S. Each node is a state of S. The node with the incoming arrow is the initial state. The filled nodes are the states accepting the desired behavior. The syntax and semantics of the transition labels is the same of the LTSs of components and coordinator except two kinds of action: i) a universal action (e.g.: $?true\_$ in Figure 3) which represents any possible action[4], and ii) a negative action (e.g.: $!-req\_2$ in Figure 3) which represents any possible action different from the negative action itself[5].

$Client1$ performs a request (i.e.: action $!req\_1$) and waits for a erroneous or successful notification: actions $?err\_1$ and $?ok\_1$ respectively. $Client2$ simply performs the request and it never handles erroneous notifications. $Server$ receives a request and then it may answer either with a successful or an erroneous notification [6].

AlternatingProtocol specifies the behavior of S that guarantees the evolution of all components. It specifies that $Client1$ and $Client2$ must perform requests by using an alternating coordination protocol. More precisely, if $Client1$ performs an action req (the transition $!req\_1$ from the state $S47$ to the state $S50$ in Figure 3) then it cannot perform req again (the loop transition $!-req\_1$ on the state $S50$ in Figure 3) if $Client2$ has not performed req (the transition $!req\_2$ from the state $S50$ to the accepting state $S125$ in Figure 3) and viceversa.

In Figure 4.(a), we consider the specification of E as given in input to the *SYNTHESIS* tool. In this example, the *RETRY* enhancement is the only element in E.

$Client1$ is an interactive client and once an erroneous notification occurs, it shows a dialog window displaying information about the error. The user might not appreciate this error message and he might lose the degree of trust in the system. By recalling that the dependability of a system reflects the users degree of trust in the system, this example shows a commonly practiced dependability-enhancing technique. The wrapper *WR* attempts to hide the error to the user by re-sending the request a finite number of times. This is the *RETRY* enhancement specified in Figure 4.(a).

[3]The environment of a component/coordinator is the parallel composition of all others components in the system.
[4]The prefixed symbols "!" or "?", in the label of a universal action, are ignored by *SYNTHESIS*.
[5]The prefixed symbols "!" or "?", in the label of a negative action, are still interpreted by *SYNTHESIS*.
[6]The error could be either due to an upper-bound on the number of request that *Server* can accept simultaneously or due to a general transient-fault on the communication channel.

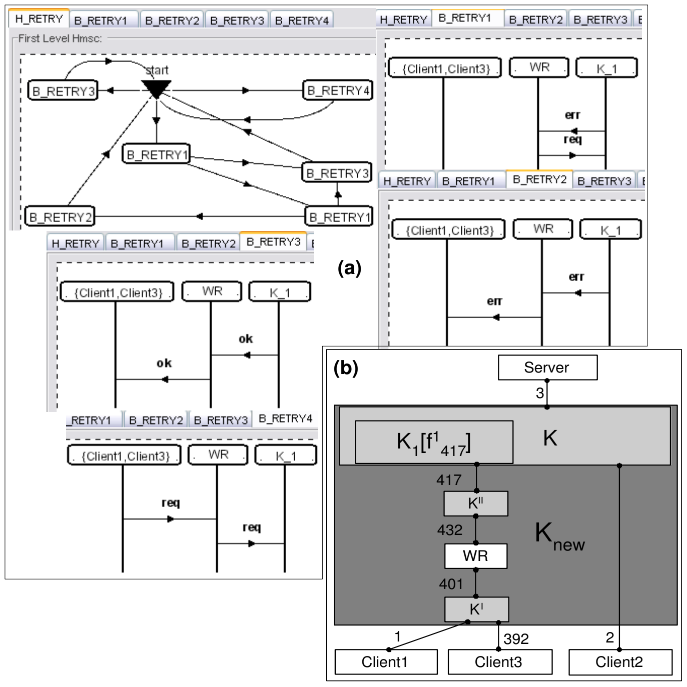


Figure 4: *bMSCs* and *hMSC* specification of *E*: *RETRY* enhancement

The wrapper *WR* re-sends at most two times. Moreover, the *RETRY* enhancement specifies an update of S obtained by inserting $Client3$ which is a new client. In specifying enhancements, we use *"augmented"*-bMSCs. By referring to [1], each usual bMSC represents a possible execution scenario of the system. Each execution scenario is described in terms of a set of interacting components, sequences of method call and possible corresponding return values. To each vertical lines is associated an instance of a component. Each horizontal arrow represents a method call or a return value. Each usual HMSC describes possible continuations from a scenario to another one. It is a graph with two special nodes: the starting and the ending node. Each other node is related to a specified scenario. An arrow represents a transition from a scenario to another one. In other words, each HMSC composes the possible execution scenarios of the system. The only difference between *"augmented"*-bMSCs and usual bMSCs is that to each vertical line can be associated a set of component instances (e.g.: $\{Client1, Client3\}$ in Figure 4.(a)) rather than only one instance. This is helpful when we need to group components having the same interaction behavior.

5.1 First step: wrapper insertion procedure

By referring to Section 4, each enhancement MSCs specification (see Figure 4.(a)) is in general described in terms of the *sub-coordinator* K_1 (i.e.: K_1 in Figure 4.(a)), the wrapper (*WR*), the components in S ($Client1$) and the new components ($Client3$). The LTS of the *sub-coordinator* is automatically derived from the LTS of the coordinator in S (K) by performing the following algorithm:

Definition 3. $L_{j,..,j+h}$ *construction algorithm*
Let L be the LTS for a component (or a coordinator) C,

we derive the LTS $L_{j,..,j+h}$, $h \geq 0$, of the *behavior of C on channels j,..,j+h* as follows:

1. set $L_{j,..,j+h}$ equal to L;
2. for each loop (ν, ν) of $L_{j,..,j+h}$ labeled with an action $\alpha = a_k$ where $k \neq j, .., j+h$ do:
   remove (ν, ν) from the set of arcs of $L_{j,..,j+h}$;
3. for each arc (ν, μ) of $L_{j,..,j+h}$ labeled with an action $\alpha = a_k$ where $k \neq j, .., j+h$ do:
   - remove (ν, μ) from the set of arcs of $L_{j,..,j+h}$;
   - if μ is the starting state then
     set ν as the starting state;
   - for each other arc (ν, μ) of $L_{j,..,j+h}$ do:
     replace (ν, μ) with (ν, ν);
   - for each arc (μ, ν) of $L_{j,..,j+h}$ do:
     replace (μ, ν) with (ν, ν);
   - for each arc (μ, υ) of $L_{j,..,j+h}$ with $\upsilon \neq \mu, \nu$ do:
     replace (μ, υ) with (ν, υ);
   - for each arc (υ, μ) of $L_{j,..,j+h}$ with $\upsilon \neq \mu, \nu$ do:
     replace (υ, μ) with (υ, ν);
   - for each loop (μ, μ) of $L_{j,..,j+h}$ do:
     replace (μ, μ) with (ν, ν);
   - remove μ from the set of nodes of $L_{j,..,j+h}$;
4. until $L_{j,..,j+h}$ is a non-deterministic LTS (i.e.: it contains arcs labeled with the same action and outgoing the same node) do:
   - for each pair of loops (ν, ν) and (ν, ν) of $L_{j,..,j+h}$ labeled with the same action do:
     remove (ν, ν) from the set of arcs of $L_{j,..,j+h}$;
   - for each pair of arcs (ν, μ) and (ν, μ) of $L_{j,..,j+h}$ labeled with the same action do:
     remove (ν, μ) from the set of arcs of $L_{j,..,j+h}$;
   - for each pair of arcs $((\nu, \mu)$ and $(\nu, \upsilon))$ or $((\nu, \nu)$ and $(\nu, \upsilon))$ of $L_{j,..,j+h}$ labeled with the same action do:
     - remove (ν, υ) from the set of arcs of $L_{j,..,j+h}$;
     - if υ is the starting state then
       set ν as the starting state;
     - for each ingoing arc *in* in υ, outgoing arc *out* from υ and loop l on υ do:
       move the extremity on υ of *in*, *out* and l on ν;
     - remove υ from the set of nodes of $L_{j,..,j+h}$.

Informally, the algorithm of Definition 3 *"collapses"* (steps 1,2 and 3) linear and/or cyclic paths made only of actions on channels $k \neq j, .., j+h$. Moreover, it also avoids (step 4) possible *"redundant"* non-deterministic behaviors[7].

By referring to Figure 5, the LTS of K_1 is the *LTS Restricted Coord*.

In general, once we derived $K_{j,..,j+h}$, we decouple K from the components $C_j, .., C_{j+h}$ (i.e.: *Client*1) connected through the connectors $j, .., j+h$ which are related to the specified enhancement (i.e.: the connector 1). To do this, we use the decoupling function defined as follows:

[7]These behaviors might be a side effect due to the collapsing.

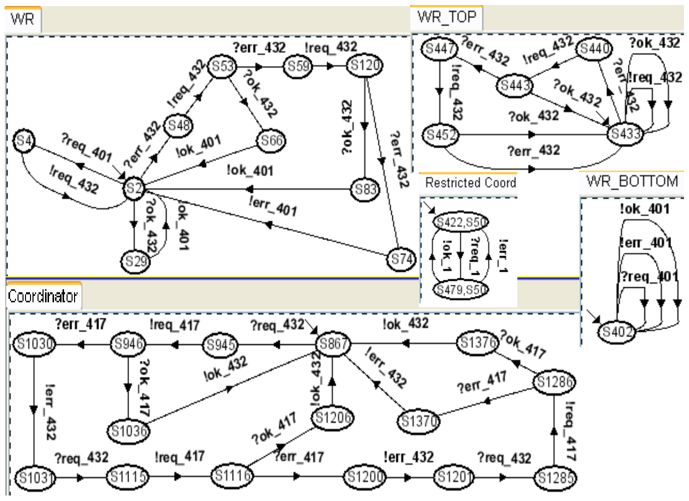


Figure 5: LTSs of wrapper, sub-coordinator and K''

Definition 4. Decoupling function
Let Act_K be the set of action labels of the coordinator K, let be $ID = \{j, \ldots, j+h\}$ a subset of all connectors identifiers[8] of K and let be $\delta \neq j, .., j+h$ a new connector identifier, we define the *"decoupling"* function $f_{\delta}^{j,..,j+h}$ as follows:

- $\forall\ a_i \in Act_K$, if $i \in ID$ then: $f_{\delta}^{j,..,j+h}(a_i) = a_{\delta}$;

The unique connector identifier δ is automatically generated by *SYNTHESIS*. In this way we ensure that K and *Client*1 no longer synchronize directly. In [15], we detail the correspondence between the decoupling function and components/coordinator deployment.
Now, by continuing the method described in Section 4, we derive the LTSs for the wrapper (see *WR* in Figure 5) and the new components (the *LTS* of *Client*3 is equal to the LTS of *Client*1 in Figure 3 except for the connector identifier). We recall that *SYNTHESIS* does that by taking into account the enhancements specification and by performing its implementation of the translation algorithm described in [17]. It is worthwhile noticing that *SYNTHESIS* automatically generates the connector identifiers for the actions performed by *WR* and *Client*3. By referring to Section 3.1, *WR* is connected to its environment through two connectors: i) one on its top interface (i.e.: the connector 432 in Figure 4.(b)) and ii) one on its bottom interface (i.e.: the connector 401 in Figure 4.(b)). Finally, as we will see in detail in Section 6, if the insertion of *WR* allows the resulting composed system to still satisfy *AlternatingProtocol*, *WR* is interposed between $K_1[f_{417}^1]$, *Client*1 and *Client*3. We recall that $K_1[f_{417}^1]$ is K_1 (see *Restricted Coord* in Figure 5) renamed after the decoupling. To insert *WR*, *SYNTHESIS* automatically synthesizes two new coordinators K' and K''. *Coordinator* in Figure 5 is the LTS for K''. For the purposes of this paper, in Figure 5, we omit the LTS for K'. K'' is derived by taking into account both the LTSs of $K_1[f_{417}^1]$ and *WR_TOP* (see Figure 5) and by performing the old synthesis approach [6, 9, 11]. While K' is derived analogously to the LTSs of *Client*1, *Client*3 and *WR_BOTTOM* (see Figure 5). The LTSs of *WR_TOP* and *WR_BOTTOM*

[8]By referring to Section 5, the connectors identifiers are postfixed to the labels in Act_K.

are WR_{432} and WR_{401} respectively. By referring to Definition 3, WR_{432} and WR_{401} model the behavior of WR on channels 432 and 401 respectively. The resulting enhanced, deadlock-free and *Alternating Protocol*-satisfying system is $S' \equiv ((Client_1 \mid Client_2 \mid Client_3 \mid Server \mid K_{new}) \setminus (Act_{Client1} \cup Act_{Client2} \cup Act_{Server} \cup Act_{Client3}))$ where $K_{new} \equiv ((K[f^1_{417}] \mid WR \mid K' \mid K'') \setminus (Act_{WR} \cup Act_{K_1[f^1_{417}]}))$ (see Figure 4.(b)). In [2], we proved the correctness and completeness of the approach.

5.2 Second step: synthesis of the glue code implementation

The parallel composition K_{new} represents the model of the enhanced coordinator. By referring to Section 4, we recall that K is the initial glue code for S and WR is a COTS component whose interaction behavior is specified by the enhancements specification E. That is, the actual code for WR and K is already available. Thus, in order to derive the code implementing K_{new}, *SYNTHESIS* automatically derives the actual code implementing K' and K''. Using the same technique described in [6, 9, 11, 15], this is done by exploiting the information stored in the nodes and arcs of the LTSs for K' and K''. More precisely, the code implementing K' and K'' reflects the structure of their LTSs which describe state machines. For the sake of brevity, here, we omit a detailed description of the code synthesis. Refer to [6, 9, 11, 15] for it. In [9, 15], we validated and applied *SYNTHESIS* for assembling *Microsoft COM/DCOM* components. The reference development platform of the current version of *SYNTHESIS* is *Microsoft Visual Studio 7.0* with *Active Template Library*.

6. CHECKING ENHANCEMENT CONSISTENCY

In this section we formalize a compositional technique to check if the applied enhancements are consistent with respect to the previously enforced desired behaviors. In other words, given the Büchi Automata specification of a desired behavior P_i, given the deadlock-free and P_i-satisfying coordinator K and given the MSCs specification of an enhancement E_i, we check (in a compositional way) if the enhanced coordinator K_{new} still satisfies P_i ($K_{new} \models P_i$).

In general, we have to check $((K[f^{j,..,j+h}_\delta] \mid WR \mid K' \mid K'') \setminus (Act_{WR} \cup Act_{K_{j,..,j+h}[f^{j,..,j+h}_\delta]})) \models P_i$. By exploiting the constraints of our architectural style, it is enough to check $((K[f^{j,..,j+m}_\delta] \mid K''_\delta) \setminus Act_{K_{j,..,j+m}[f^{j,..,j+m}_\delta]}) \models P_i$ where $\{j,..,j+m\}$ is the set of channel identifiers which are both channels in $\{j,..,j+h\}$ and in the set of channel identifiers for the action labels in P_i. In order to avoid the state explosion phenomenon we should decompose the verification without composing in parallel the processes $K[f^{j,..,j+m}_\delta]$ and K''_δ. We do that by exploiting the *assume-guarantee paradigm* for compositional reasoning [5].

By recasting the typical proof strategy of the *assume-guarantee paradigm* in our context, we know that if $\langle A\rangle K[f^{j,..,j+m}_\delta]\langle P_i\rangle$ and $\langle true\rangle K''_\delta\langle A\rangle$ hold then we can conclude that
$\langle true\rangle\ ((K[f^{j,..,j+m}_\delta] \mid K''_\delta) \setminus Act_{K_{j,..,j+m}[f^{j,..,j+m}_\delta]})\ \langle P_i\rangle$ is true. This proof strategy can also be expressed as the following inference rule:

$$\frac{\langle true\rangle K''_\delta\langle A\rangle \qquad \langle A\rangle K[f^{j,..,j+m}_\delta]\langle P_i\rangle}{\langle true\rangle((K[f^{j,..,j+m}_\delta]|K''_\delta)\setminus Act_{K_{j,..,j+m}[f^{j,..,j+m}_\delta]})\langle P_i\rangle}$$

where A is a LTL formula (and hence it is modeled as a Büchi Automaton). We recall that, in S, K already satisfies P_i. Once we applied E_i to obtain the enhanced system S', A represents the assumptions (in S') on the environment of K that must be held on the channel δ in order to make K able to still satisfy P_i. Without loss of generality, let $\{j,..,j+m\}$ be $Channels_{P_i} \cap \{j,..,j+h\}$ where $Channels_{P_i}$ is the set of channel identifiers for the action labels in P_i; then A is the Büchi Automaton corresponding to $K_{j,..,j+m}[f^{j,..,j+m}_\delta][f_{env}]$ where $f_{env}(?\alpha) = !\alpha$ and $f_{env}(!\alpha) = ?\alpha$. For the example illustrated in Section 5, K''_δ and A are the Büchi Automata corresponding to K''_{417} (i.e.: $K2$ showed in Figure 6) and to $K_1[f^1_{417}][f_{env}]$ (*Assumption* showed in Figure 6) respectively.

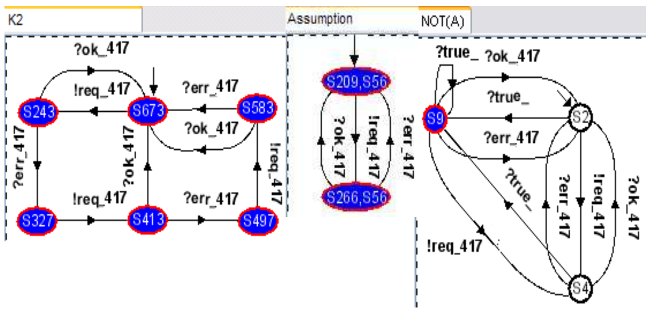


Figure 6: Büchi Automata of K''_{417}, $K_1[f^1_{417}][f_{env}]$ and $\overline{K_1[f^1_{417}][f_{env}]}$

In general, a formula $\langle true\rangle M\langle P\rangle$ means $M \models P$. While a formula $\langle A\rangle M\langle P\rangle$ means if A holds then $M \models P$. In our context, P is modeled as the corresponding Büchi Automaton B_P. M is modeled as the corresponding LTS. By referring to [5], to a LTS M always corresponds a Büchi Automaton B_M. With $L(B)$ we denote the language accepted by B. Exploiting the usual automata-based model checking approach [5], to check if $M \models P$ we first automatically build the product language $L_{M\cap P} \equiv L(B_M) \cap \overline{L(B_P)}$ and then we check if $L_{M\cap P}$ is empty.

Theorem 1. Enhancement consistency check
Let P_i be the Büchi Automata specification of a desired behavior for a system S formed by $C_1,..,C_n$ components; let K be the deadlock-free and P_i-satisfying coordinator for the components in S; let E_i be the MSCs specification of a K-protocol enhancement; let K'' be the adaptor between K and the wrapper implementing the enhancement E_i; let δ the identifier of the channel connecting K'' with K; let $\{j,..,j+m\}$ the set of channel identifiers which are both channels in the set of channel identifiers for the action labels in P_i and in the set of channels identifiers affected by the enhancement E_i; and let f_{env} be a relabeling function in such a way that $f_{env}(?\alpha) = !\alpha$ and $f_{env}(!\alpha) = ?\alpha$ for all $\alpha \in Act_K$, if $L_{K''_\delta \cap K_{j,..,j+m}[f^{j,..,j+m}_\delta][f_{env}]} = \emptyset$ then $((K[f^{j,..,j+m}_\delta] \mid K''_\delta) \setminus Act_{K_{j,..,j+m}[f^{j,..,j+m}_\delta]}) \models P_i$ and hence E_i is consistent with respect to P_i.

PROOF. Let A be the Büchi Automaton corresponding to $K_{j,..,j+m}[f_\delta^{j,..,j+m}][f_{env}]$, if $L_{K''_\delta \cap K_{j,..,j+m}[f_\delta^{j,..,j+m}][f_{env}]} = \emptyset$ then $K''_\delta \models A$. That is, $\langle true \rangle K''_\delta \langle A \rangle$ holds. Moreover, by construction of A, $\langle A \rangle K[f_\delta^{j,..,j+m}]\langle P_i \rangle$ holds too. By applying the inference rule of the *assume-guarantee paradigm*, $\langle true \rangle ((K[f_\delta^{j,..,j+m}] \mid K''_\delta) \backslash Act_{K_{j,..,j+m}[f_\delta^{j,..,j+m}]}) \langle P_i \rangle$ is true and hence $((K[f_\delta^{j,..,j+m}] \mid K''_\delta) \setminus Act_{K_{j,..,j+m}[f_\delta^{j,..,j+m}]}) \models P_i$. □

By referring to Theorem 1, to check if E_i is consistent with respect to P_i, it is enough to check if $\langle true \rangle K''_\delta \langle A \rangle$ holds. In other words, it is enough to check if K'' provides K with the environment it expects (to still satisfy P_i) on the channel connecting K'' to K (i.e.: the connector identified by δ). In the example illustrated in Section 5, $RETRY$ is consistent with respect to *AlternatingProtocol*. In fact, by referring to Figure 6, $NOT(A)$ is the Büchi Automaton for $\overline{K_1[f^1_{417}][f_{env}]}$ (i.e.: for $\overline{A}$ of Theorem 1) and $K2$ is the Büchi Automaton for K''_{417} (i.e.: for K''_δ of Theorem 1). By automatically building the product language between the languages accepted by $K2$ and $NOT(A)$, *SYNTHESIS* concludes that $L_{K''_{417} \cap K_1[f^1_{417}][f_{env}]} = \emptyset$ and hence that $((K[f^1_{417}] \mid K''_{417}) \setminus Act_{K_1[f^1_{417}]}) \models$ *AlternatingProtocol*. That is $RETRY$ is consistent with respect to *AlternatingProtocol*.

7. CONCLUSION AND FUTURE WORK

In this paper, we combined the approaches of protocol transformation formalization [13] and of automatic coordinator synthesis [6, 9, 11] to produce a new technique for automatically synthesizing failure-free coordinators for protocol enhanced in component-based systems. The two approaches take advantage of each other: while the approach of protocol transformations formalization adds compose-ability to the automatic coordinator synthesis approach, the latter adds automation to the former. This paper is a revisited and extended version of [7]. With respect to [7], the novel aspects of this work are that we have definitively fixed and extended the formalization of the approach, we have implemented it in our *"SYNTHESIS"* tool and we have formalized and implemented the enhancement consistency check.

The key results are: (i) the extended approach is compositional in the automatic synthesis of the enhanced coordinator; that is, each wrapper represents a modular protocol transformation so that we can apply coordinator protocol enhancements in an incremental way by re-using the code synthesized for already applied enhancements; (ii) we are able to add extra functionality to a coordinator beyond simply restricting its behavior;(iii) this, in turn, allows us to enhance a coordinator with respect to a useful set of protocol transformations such as the set of transformations referred in [13]. The automation and applicability of both the old (presented in [6, 9, 11]) and the extended (presented in this paper and in [15]) approach for synthesizing coordinators is supported by our tool called *"SYNTHESIS"* [9, 15].

As future work, we plan to: (i) develop more user-friendly specification of both the desired behaviors and the protocol enhancements (e.g., UML2 Interaction Overview Diagrams and Sequence Diagrams); (ii) validate the applicability of the whole approach to large-scale examples different than the case-study treated in [15] which represents the first attempt to apply the extended version of *"SYNTHESIS"* (formalized in this paper) in real-scale contexts.

8. REFERENCES

[1] Itu-t reccomendation z.120. message sequence charts. (msc'96). Geneva 1996.

[2] M. Autili. Sintesi automatica di connettori per protocolli di comunicazione evoluti. Tesi di laurea in Informatica, Universitá dell'Aquila - April,2004 - http://www.di.univaq.it/tivoli/AutiliThesis.pdf.

[3] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin. Supervisory control of a rapid thermal multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, July 1993.

[4] L. de Alfaro and T. Heinzinger. Interface automata. In *ACM Proc. of the joint 8th ESEC and 9th FSE, 2001.*

[5] O. G. Edmund M. Clarke, Jr. and D. A. Peled. *Model Checking.* The MIT Press, 2001.

[6] P. Inverardi and M. Tivoli. *Software Architecture for Correct Components Assembly - Chapter in: Formal Methods for the Design of Computer, Communication and Software Systems: Software Architecture.* Springer, LNCS 2804, Sept. 2003.

[7] M.Autili, P.Inverardi, and M.Tivoli. Automatic adaptor synthesis for protocol transformation. In *WCAT04.*

[8] R. Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.

[9] M.Tivoli, P.Inverardi, V.Presutti, A.Forghieri, and M.Sebastianis. Correct components assembly for a product data management cooperative system. In *proceedings of the Int. Symposium CBSE7. May,2004.* Springer, LNCS 3054.

[10] R. Passerone, L. de Alfaro, T. Heinzinger, and A. L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proc. of ICCAD, 2002.*

[11] P.Inverardi and M.Tivoli. Failure-free connector synthesis for correct components assembly. In *Proceedings of SAVCBS'03.*

[12] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[13] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *proceeding of the 25th ICSE'03 - Portland, OG (USA)*, May 2003.

[14] C. Szyperski. *Component Software. Beyond Object Oriented Programming.* Addison Wesley, 1998.

[15] M. Tivoli, M. Autili, and P. Inverardi. Synthesis: a tool for synthesizing correct and protocol-enhanced adaptors. submitted for publication - Aug,2004 - http://www.di.univaq.it/tivoli/LastSynthesis.pdf.

[16] M. Tivoli and D. Garlan. Coordinator synthesis for reliability enhancement in component-based systems. Carnegie Mellon University, C.S.Dep. - Tech.Rep. - http://www.di.univaq.it/tivoli/CMUtechrep.pdf.

[17] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *ACM Proceedings of the joint 8th ESEC and 9th FSE*, Vienna, Sep 2001.

[18] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, march 1997.

Monitoring Design Pattern Contracts

Jason O. Hallstrom
Computer Science
Clemson University
Clemson, SC 29634, USA
jasonoh@cs.clemson.edu

Neelam Soundarajan, Benjamin Tyler
Computer Science and Engineering
Ohio State University
Columbus, OH 43210, USA
{neelam, tyler}@cse.ohio-state.edu

ABSTRACT

Design patterns allow system designers to reuse well established solutions to commonly occurring problems. These solutions are usually described informally. While such descriptions are certainly useful, to ensure that designers precisely and unambiguously understand the requirements that must be met when applying a given pattern, we also need formal characterizations of these requirements. Further, system designers need tools for determining whether a system implemented using a given pattern satisfies the appropriate requirements. In [18], we described an approach to specifying design patterns using formal *contracts*. In this paper, we develop a monitoring approach for determining whether the pattern contracts used in developing a system are respected at runtime.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Verification—*Runtime Monitoring*; D.1.m [**Programming Techniques**]: Patterns—*AOP*

General Terms

Design, Reliability, Verification

Keywords

Design patterns, Aspect-oriented programming, Runtime monitoring of contracts

1. INTRODUCTION

Design patterns [2, 8, 10, 17] have, over the last decade, fundamentally changed the way we think about the design of large software systems. Using design patterns not only helps designers exploit the community's collective wisdom and experience as captured in the patterns, it also enables others studying the system in question to gain a deeper understanding of how the system is structured, and why it behaves in particular ways. And as the system evolves over time, the patterns used in its construction provide guidance on managing the evolution so that the system remains faithful to its original design, ensuring that the original parts and the modified parts interact as expected. Although they are not components in the standard sense of the word, patterns may, as has been noted, be the real key to reuse since they allow the reuse of design, rather than mere code. But to fully realize these benefits, we must ensure that the designers have a thorough understanding of the precise requirements their system must meet in applying a given pattern, as well as automated or semi-automated ways of checking whether the requirements have been satisfied. To that end, the work we present in [18] describes an approach to specifying design patterns precisely using formal *contracts*. Our goal in this paper is to extend that work, and to develop a runtime monitoring approach that allows system designers to determine whether the patterns used in constructing a system have been applied correctly. We use an *aspect-oriented programming* [12, 11] approach to achieve this goal.

Consider the Observer pattern [8], illustrated in Fig. 1, which will be our case-study. There are two *roles* [15] in this pattern, Subject and Observer. The purpose of the pattern is to allow a set of objects that have enrolled to play the Observer role to be *notified* whenever the state of the object playing the Subject role changes, so that each of the observers[1] can update its state to be *consistent* with the new state of the subject. Also clear from Fig. 1 is the fact

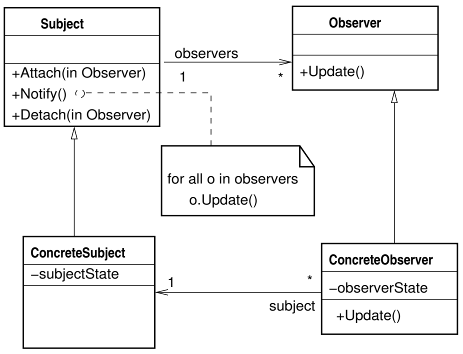


Figure 1: Observer Pattern

that the Notify() method of Subject will invoke the Update() method on each observer. What is not clear is when Notify() will be called and by whom. The informal description [8] states, "... subject notifies its observers whenever a change occurs that could make its observers' state inconsistent with

[1]We use names starting with uppercase letters, such as Subject, for roles; and lowercase names, such as subject, for the individual objects that play these roles. We also use names starting with uppercase letters for patterns. Occasionally, the name of a pattern is also used for one of its roles, as in the case of the Observer role of the Observer pattern. In such cases, the context will make clear whether we are talking about the role or the pattern.

its own." But it is not clear how the subject will know when its state has become inconsistent with that of one or more observers. Indeed, what does it mean to say that the subject state has become *inconsistent* with that of an observer? In other words, what exactly are the requirements that the designer must ensure are met in order to apply this pattern as intended? The pattern contracts described in [18] provide precise answers to these questions. We will consider the requirements specified by these contracts in Section 2.

Next consider the question of runtime monitoring. In standard specification-based testing/monitoring [1, 3, 13], we typically consider the behavior of the methods of a *single* class. For this, we instrument the class in question to see that the pre- and post-conditions of the class methods are satisfied at appropriate points. But in the case of patterns, we are dealing not with individual classes, but with multiple classes. Indeed, the focus is usually on the interactions and interrelations among the classes, rather than on the behaviors of the classes in isolation.

A natural solution is to use *aspects* [12, 11], since aspects allow us to deal with *crosscutting* concerns. We will define an *abstract aspect* for Observer that implements the monitoring functionality common across all applications of the pattern. Corresponding to any particular application in an actual system, we will define a *concrete subaspect* that tailors the monitoring functionality as appropriate to the application in question. An abstract aspect captures the requirements embodied in a given pattern's *contract*, and a concrete subaspect captures the specializations embodied in a *subcontract* of the pattern's contract. In a sense, we can consider the aspects used in the current paper as aspect-versions of the contracts presented in [18]. (Although, [18] did not consider the notion of a subcontract.) Indeed, hereafter, we refer to the abstract aspect as a *contract*, and the concrete subaspect as a *subcontract*. We will see how a contract-subcontract pair can be used to monitor a system to see if it applies a given pattern faithfully.

There is an inherent risk in formalizing patterns in that their hallmark flexibility may be lost [16]. For the case of Observer, if we adopt one definition for the notion of *consistency* between the Subject and Observer states, the pattern may not be usable in systems that have a different notion of this concept; or we may have to come up with multiple contracts, one for each possible notion of consistency. Clearly, this would be undesirable. As we will see, our contract for Observer, while precisely capturing the pattern requirements, will also retain all of the flexibility contained in the pattern.

Hannemann and Kiczales [9] show how patterns can be *implemented* as aspects. They argue that the code for a given pattern should be collected within an aspect, rather than being distributed among different classes. By contrast, the aspect we develop in Section 2 *monitors* a system to check whether it satisfies the requirements that any designer implementing the Observer pattern must meet. This raises the question, does the Hannemann-Kiczales implementation of the pattern meet our contract? It must, if our contract is truly general. As we will show, we can indeed define a subcontract for this implementation of Observer, in exactly the same way as we do for a more 'standard' implementation of the pattern in Section 3. This is remarkable because when developing the contract for Observer, we only had in mind standard class-based implementations of the pattern, and we tried to ensure that our contract would be appropriate for all such implementations. Here we had a very different kind of implementation, and our contract turned out to be appropriate for this implementation as well. Further, and somewhat to our surprise, when we ran our contract and subcontract against this aspect-based implementation of the pattern, a contract violation was reported! It turns out, as we will see, that there is a minor error in the implementation in [9].

In Section 2, we develop the contract for Observer, present a simple system built using the pattern, define the subaspect corresponding to the pattern as used in this system, show how the aspect and subaspect allow us to monitor the system at runtime, and discuss the monitoring results. In Section 3, we outline how a subaspect corresponding to the implementation of [9] can be defined, and discuss the monitoring results. In the next section, we discuss related work. In Section 5, we summarize our approach, and provide pointers to future work.

2. PATTERN CONTRACTS

Since we use *AspectJ* [11] to develop the pattern contracts and subcontracts, we begin with a brief summary of some of the essential parts of *AspectJ*. Three key concepts of the language are join points, pointcuts, and advice. A *join point* identifies a particular point in the execution of a program; for example, a call to a particular method of a particular class, or a call to a particular constructor of a particular class. A *pointcut* is a way of grouping together a set of join points that we want to treat in a particular fashion; for example, calls to *all* methods of a given class. The pointcut construct enables us to collect *context*: for example, the object on which the method in question was applied, or the additional arguments that were passed to the method. Finally, the *advice* associated with a given pointcut specifies the code that needs to be executed at runtime when control reaches any of the join points that match the pointcut. There are three distinct types of advice. Consider a method call. The associated *before* advice, if any, will be executed before the method is executed. The *after* advice, if any, will be executed after the method is executed. We do not use the third kind of advice, the *around* advice.

2.1 Observer Contract

The aspect that defines the Observer contract appears in Figures 2, 3, and 4. The following notes explain the lines with the corresponding numbers in the figures.

1. The interfaces Subject and Observer correspond to the two roles of the pattern. Note that unlike in Fig. 1, there are no methods such as Notify() in these interfaces. The pointcuts of *AspectJ*, as we will see, provide a more general way of introducing these.

2. ObserverPatternContract, an *abstract aspect*, captures the requirements to be checked for all applications of Observer.

3. The information needed to monitor the system will be maintained in three variables: xSubjectsObservers maps[2] each object that enrolls as a subject to the objects that are enrolled to be observers of that subject,

[2]This should be a WeakHashMap to allow garbage collection to proceed normally.

```
protected interface Subject { }                // see note (1)
protected interface Observer { }
public abstract aspect ObserverPatternContract {      note (2)
  //Aux. variables:                                       (3)
    private Map xSubjectsObservers = new HashMap();
    private Set xUpdateCalls = new HashSet();
    private Map xrecordedStates = new HashMap();
  //Auxiliary functions:                                  (4)
    abstract protected String xSubjectState(Subject s);
    abstract protected String xObserverState(Observer o);
    abstract protected boolean
          xModified(String s1, String s2);
    abstract protected boolean
          xConsistent(String s, String o);
  //Pointcuts:                                            (5)
    abstract protected pointcut subjectEnrollment(Subject s);
    abstract protected pointcut
          attachObs(Subject s, Observer o);
    abstract protected pointcut
          detachObs(Subject s, Observer o);
    abstract protected pointcut Notify(Subject s);
    abstract protected pointcut Update(Subject s,Observer o);
    abstract protected pointcut subjectMethods(Subject s);
```

Figure 2: Observer Contract (part 1 of 3)

and is initially empty. xUpdateCalls is used to keep track of the observers that are updated when the corresponding subject state changes. xrecordedStates is used to save, for each subject, the state that its observers have been most recently notified of.

4. We use auxiliary functions to represent pattern concepts that vary among applications. As we noted earlier, the pattern requires that observers become *consistent* with the subject state when they are updated, but the notion of consistency will vary from one system to another. Similarly, the pattern requires the observers to be notified when the subject state is *modified*, but what modification of the subject state means will vary from system to system. xModified() and xConsistent() allow us to specify these requirements precisely, while allowing for variation among different systems.

   Given two subject states, xModified() tells us if the second state should be considered 'modified' from the first. Since this function is *abstract*, the pattern contract will not define it; instead, the subcontract will provide a definition tailored to the system in question. Similarly, xConsistent(), given a subject state and an observer state, tells us whether the latter is *consistent* with the former. This, too, is abstract, since the notion of consistency varies from system to system.

   For simplicity, rather than working with the actual states of the subjects and observers, we assume that we have functions xSubjectState() and xObserverState() that will encode the states into Strings. Naturally, such encodings will depend on the system: hence these are *abstract*, to be suitably defined in the subcontract.

5. Next we have the pointcuts that identify the points at which the system should be *interrupted* at runtime, either to save information needed by the contract, or to check if the contract requirements are being met.

   subjectEnrollment is the pointcut that represents the points at which an object enrolls to play the Subject role. The only argument here is the object enrolling. attachObs and detachObs correspond to an object attaching or detaching, respectively. The arguments for these two pointcuts are the subject and observer involved.

   Next we have Notify, which corresponds to the points at which a given subject's observers are notified following a change in the state of the subject (as defined by the xModified() function). The Update pointcut corresponds to an individual observer being updated to become consistent (as defined by xConsistent()) with the modified (or rather, xModified()) subject state. The final pointcut, subjectMethods, corresponds to all the methods of the class playing the Subject role.

```
  //Advice for Subject enrollment:                         (6)
    after(Subject s): subjectEnrollment(s) {
      Set obSet = new HashSet();
      xSubjectsObservers.put(s,obSet);
      xrecordedStates.put(s,xSubjectState(s)); }
  //Advice for attaching Observer:                         (7)
    before(Subject s, Observer o): attachObs(s,o) {
      xUpdateCalls.clear(); }
    after(Subject s, Observer o): attachObs(s,o) {
      if (!xUpdateCalls.contains(o)) { System.out.println(
      "Update not called on attaching Observer"); }
      Set obSet = (Set)xSubjectsObservers.get(s);
      obSet.add(o); xSubjectsObservers.put(s,obSet); }
  //Advice for detaching Observer:                         (8)
    before(Subject s, Observer o): detachObs(s,o) {
      Set obSet = (Set)xSubjectsObservers.get(s);
      obSet.remove(o); xSubjectsObservers.put(s,obSet); }
    //No "after" advice for detachObs.
```

Figure 3: Observer Contract (part 2 of 3)

Let us now consider the advice corresponding to the various pointcuts[3].

6. The advice for subjectEnrollment adds the enrolling object to xSubjectsObservers with an empty set of observers, and saves its current state as its recorded state. As there are no observers for this subject, we can vacuously say that they have all been informed of its current state.

7. When a new observer attaches to a subject, we must ensure that it is updated. As we noted in [18], this point has been overlooked in many informal descriptions of the pattern. If this is not done, the observer's state may be inconsistent with the subject state until the point when the subject is next modified.

   To check this, the before advice clears xUpdateCalls. As we will see below, the advice for the Update pointcut adds the observer being updated to xUpdateCalls.

[3]Java's collection classes rely on Object.equals() to locate items. We assume the default implementation of equals(), which tests for equality based on the *identity* of the objects.

Hence, in the after advice of attachObs, we require xUpdateCalls to contain this observer. If it does not, that indicates that the observer was *not* updated when it enrolled, and we output a message to that effect[4].

8. Detachment of an observer simply requires eliminating it from the set of objects enrolled to observe the subject. It is possible that in the actual system, nothing is done at this point, i.e., the designer might have decided to continue updating the object whenever the subject's state is modified. This will not violate our contract; and it is consistent with the intent of the pattern since the pattern requires that all enrolled observers be updated, not that others should *not* be[5].

```
//Advice for Notify:                                    (9)
 before (Subject s) : Notify(s) {
  xUpdateCalls.clear();
  xrecordedStates.put(s,xSubjectState(s)); }
 after (Subject s) : Notify(s) {
  Set obSet = (Set)xSubjectsObservers.get(s);
  if (!xUpdateCalls.containsAll(obSet)) {
   System.out.println("Some Observers not notified
       of change in Subject!"); }  }
//Advice for Update:                                    (10)
 before (Subject s, Observer o) : Update(s,o)
  { xUpdateCalls.add(o); }
 after (Subject s, Observer o) : Update(s,o) {
  if (!xConsistent(xrecordedStates.get(s),
       xObserverState(o))) { System.out.println(
       "Observer not properly updated!"); }  }
//Advice for Subject's methods:                         (11)
 after(Subject s): subjectMethods(s) {
  if (xModified(xrecordedStates.get(s),xSubjectState(s))){
   System.out.println("Observers not notified
       of change in Subject!"); }   }
}
```

Figure 4: Observer Contract (part 3 of 3)

9. The before advice for Notify updates xrecordedStates for the subject since its observers are about to be notified of its state change. And xUpdateCalls is cleared so in the after advice we can check that *all* of its observers have been notified. If not, we print a suitable message.

10. The before advice of Update adds the observer to the set of observers being updated. In the after advice, we check that the state of the observer is consistent with the subject state. This checks that the system code that is supposed to update the observer is working correctly, at least as judged by the definition of xConsistent(). If the condition is not satisfied, it may be an error in the subcontract, rather in the monitored system. We clearly need to identify such errors and correct them, and such checks help with that task.

11. The final advice corresponds to the methods of the class playing the role of Subject. For any such method, there are three possibilities.

    First, the method execution did not change the subject state (according to xModified()). Hence, the final state should match the recorded state of the subject, assuming that this condition was satisfied at the start of the method. (If this were not the case, an earlier error would already have been caught.)

    Second, the method execution changed the subject state and called the appropriate operations to notify/update the observers. This would have triggered the advice associated with the Notify pointcut, and the advice associated with Update for each observer. Those two advices would have checked that all observers were updated, and would also have saved, in xrecordedStates, the state of the subject at that time. So the final subject state would match that in xrecordedStates.

    Third, the method changed the subject state, but did not notify the observers. Or perhaps the method changed the subject state, notified the observers, and then *again* changed the subject state, and this time did not notify the observers. In both cases, the if-condition of the after advice would be true, and we would get the appropriate error message.

It is worth stressing that by specifying the auxiliary functions and pointcuts as *abstract*, we have ensured that all of these can be defined, in the subcontract, as appropriate to the particular system. But at the same time, the checks in the various pieces of advice ensure that the essential intent of the pattern is not violated. Thus, the contract precisely specifies the pattern's requirements without in any way compromising flexibility.

2.2 A Simple System Using Observer

Fig. 5 presents TCL, a simple system that uses Observer. Instances of the Time class play the Subject role. Instances of Clock and LazyPerson play the Observer role; these two classes implement the TimeObserver interface. The Time class maintains a hash set of objects that enroll (via its attach() method) to observe the time. When the time changes, which only happens in the tickTock() method, the object calls its notifyObs() operation, which invokes the update() operation on each of its observers. In the main() method, we create aTime (a Time object), aClock (a Clock object), and bob (a LazyPerson object), attach the latter two to aTime, invoke tickTock() a few times on aTime, and then check the state of bob. TCL is a fairly standard, if simple, example of a system built using the Observer pattern.

2.3 Observer Subcontract for TCL

The subaspect, appropriate to TCL, that defines the subcontract of our pattern contract appears in Fig. 6[6].

12. We use the declare parents mechanism of *AspectJ* to state that Time implements the Subject interface of

[4]We should note that in our actual contract, we have additional checks. For example, in the before advice for this pointcut, we check that this object has not already enrolled as an observer for this subject. We also check that s has enrolled as a Subject. We omit some of these details.

[5]If we wish to disallow detached observers from being updated, the contract can be suitably modified: in the after advice of Notify, check that obSet.containsAll(xUpdateCalls) evaluates to true; i.e., for any subject, the set of updated observers must equal the set of attached observers.

[6]For readability, we use "$\wedge$", rather than the standard "&&", to denote the 'and' operation.

```
interface TimeObserver { public void update(Time t); }
class Clock implements TimeObserver {
  protected int hour = 12, minute = 0;
  public void update(Time t) {
    hour = t.getHour(); minute = t.getMinute(); }
  public String ClockTime() {
    return("The time is: " + hour + ":" + minute); }
}
class LazyPerson implements TimeObserver {
    protected boolean isSleepy = true;
    public void update(Time t) { isSleepy = t.isAm(); }
    public boolean readyToRiseNShine(){ return (!isSleepy); }
}
class Time {
  protected HashSet observers = new HashSet();
  protected int hour = 0, minute = 0, second = 0;
  public void attach(TimeObserver o) {
     observers.add(o); o.update(this); }
  public void detach(TimeObserver o) { observers.remove(o);}
  protected void notifyObs() {
    for (Iterator e = observers.iterator() ; e.hasNext() ;) {
       ((TimeObserver)e.next()).update(this); } }
  public int getHour() { // Return hour in 12-hour mode. }
  public int getMinute() { . . . }
  public int getSecond() { . . . }
  public boolean isAm() { . . . }
  public void tickTock() {
    // Update hour, etc. appropriately. Code omitted.
    // In our actual system, this function sets the Time to
    // a random (legal) value.
    notifyObs(); }
  public static void main(String[] args) {
    Time aTime = new Time(); Clock aClock = new Clock();
    LazyPerson bob = new LazyPerson();
    aTime.attach(bob); aTime.attach(aclock);
    aTime.tickTock(); aTime.tickTock(); aTime.tickTock();
    System.out.println(aClock.ClockTime());
    if (bob.readyToRiseNShine()) {
      System.out.println("Bob is ready to face another day!");}
    else { System.out.println("Too early for Bob!"); }  }
}
```

Figure 5: Time-Clock-LazyPerson (TCL) System

```
public aspect TCLContract extends ObserverPatternContract{
  declare parents: Time implements Subject;                (12)
  declare parents: TimeObserver extends Observer;
  //Pointcuts:                                             (13)
    protected pointcut attachObs(Subject s, Observer o):
      call(void Time.attach(TimeObserver))
            ∧ target(s) ∧ args(o);
    protected pointcut detachObs(Subject s, Observer o):
      call(void Time.detach(TimeObserver))
            ∧ target(s) ∧ args(o);
    protected pointcut subjectEnrollment(Subject s):
      call(Time.new()) ∧ target(s);
    protected pointcut subjectMethods(Subject s):
      call(* Time.*()) ∧ target(s);
    protected pointcut Notify(Subject s):
      call(void Time.notifyObs()) ∧ target(s);
    protected pointcut Update(Subject s, Observer o):
      call(void TimeObserver.update(Time))
            ∧ target(o) ∧ args(s);
  //Aux. functions:                                        (14)
  protected String xSubjectState(Subject s) {
    //s must be of type Time; return the time as a String. }
  protected String xObserverState(Observer o) {
    //o must be of type Clock or LazyPerson; use getClass()
    //to check, and return state encoded as a String. }
  protected boolean xModified(String s1, String s2) {
    //Return true if the times encoded in s1 and s2 are
    // equal, else false. }
  protected boolean xConsistent(String s, String o) {
    //Check if o encodes a Clock state or a LazyPerson state.
    //For a LazyPerson, return true if isSleepy agrees with hour
    //in the Time state encoded in s being between 0 and 11.
    // Similarly if o encodes a Clock. }
}
```

Figure 6: TCL Subcontract

the pattern contract (Fig. 2), and that TimeObserver is an extension of the Subject interface.

13. Next we provide definitions for the abstract pointcuts of the base contract. Thus, attachObs is defined as a call to the attach() method of Time, since that is the method that Time's observers are required to use to enroll as observers. detachObs, Notify, and Update are equally direct. In each case, we use the target and args constructs of *AspectJ* to bind the parameters of the pointcut with the appropriate entities from the actual (join) point in the system.

    In TCL, there is no explicit enrollment of a Time object as a subject; instead, it becomes a subject upon construction. We define the subjectEnrollment pointcut accordingly. subjectMethods captures *all* the methods of the Time class. Note that if in a future modification of the system, new methods are added to Time, those methods will also be captured by this pointcut, and will be required to abide by the requirements of the pattern contract, as captured by clause (11) in Fig. 4.

14. Next we define the auxiliary functions. xSubjectState() encodes the time represented by the the given Time object. xObserverState() is similar, but has to handle two types of observer objects, Clock and LazyPerson. xModified() determines whether the times encoded in its two arguments are equal. xConsistent(), depending on whether the state encoded in the second argument is of type Clock or LazyPerson, compares the value of either isSleepy, or hour and minute in that argument to the time in the first argument.

These definitions are dictated by the TCL system. If we considered another system that had different classes playing the Subject and/or Observer roles, or did the *notification*, *update*, etc. in other ways, we would have to define another subcontract tailored to that system. But for another system that uses the same classes as TCL, and does the notification, etc., in the same manner as TCL, we can use the same subcontract.

2.4 Results of Runtime Monitoring

We can now compile the abstract aspect that captures the Observer contract (Figs. 2, 3, 4), the subaspect that captures the subcontract for this system (Fig. 6), and the actual system code (Fig. 5) using the *AspectJ* compiler. The compiler will do the necessary *code weaving* [12, 11]. When the resulting byte code is executed, if there are no problems, that is, if all the requirements of the pattern contract/subcontract are met, the system will run as usual (if a bit slower than usual). However, in order to check that the monitoring was indeed progressing appropriately, we inserted additional output statements in the various pieces of advice, as well as in the tickTock() method, to help us track the progress of the system. A portion of the output from a sample run appears in Fig. 7 (the line numbers were inserted by hand).

```
1:  Tick-tock!
2:     before Notify(Time:11:42:06)
3:     before Update(Time:11:42:06, Clock:5:48am)
4:     after subjectMethods(Time:11:42:06)
5:     after subjectMethods(Time:11:42:06)
6:     after subjectMethods(Time:11:42:06)
7:     after Update(Time:11:42:06, Clock:11:42am)
8:     before Update(Time:11:42:06, LazyPerson:true)
9:     after Update(Time:11:42:06, LazyPerson:true)
10:    after Notify(Time:11:42:06)
11:    after subjectMethods(Time:11:42:06)
12: Tick-tock!
13:    before Notify(Time:17:09:06)
14:    . . .
19:    before Update(Time:17:09:06, LazyPerson:true)
20:    after Update(Time:17:09:06, LazyPerson:true)
21:    *** Observer not properly updated!
22:    * Subject: Time:17:09:06; Observer: LazyPerson:true
```

Figure 7: Sample Monitored Run of TCL System

Line 1 indicates that tickTock() was called, which resulted in Time.notifyObs() being called, which resulted in the Notify pointcut being entered, with the aTime value at this point being as stated (line 2). Next (line 3), Update on aClock was called. (Note that the clock reading is incorrect in this line because we have not yet done the update.) Updating aClock requires three calls to the Time methods for getting the hour, minute, and am/pm information. In each case, the after advice of the subjectMethods pointcut was executed. The advice did not report any problems, since at the start of Notify, xrecordedStates had already been updated for this Time object. The outputs from the after advice for these three calls appear in lines 4, 5, and 6. Finally, the update() operation finished, and the output from the after advice (line 7) shows that the clock was properly updated.

Next, notifyObs invoked update() on the bob object. During this run, we inserted an error in the system by replacing the code of LazyPerson.update() with an empty body; this update() operation did not invoke any operation of Time. Hence, immediately following the output from the before advice of Update (line 8), we have the output from the after advice (line 9). But there was no error reported, because the value of bob.isSleepy happened to have the correct value. In the next call to tickTock(), the error was reported (lines 21, 22). Thus, without any changes in the code of TCL, we were able to monitor the system to see if it met the appropriate pattern requirements. For a more complex system built using several patterns, we would define the appropriate contract and subcontract for each, and would compile all of them against the system source code.

3. MONITORING ALTERNATE PATTERN IMPLEMENTATIONS

As required by the pattern, notifyObs() in Time, and update() in Clock and LazyPerson, are all concerned with updating the observers when the state of the Time object changes. Hannemann and Kiczales [9] argue that such code is better written as an aspect, thereby *localizing* this code in a single module. They present an aspect that implements Observer. The aspect contains the code for *notifying* the observers of a given subject when the subject state changes. This naturally involves calling an update() operation on each observer; this operation is flagged as abstract since it will depend on the class of the observer. Further, they define an abstract pointcut, subjectChange, intended to capture all the methods of the Subject class that might result in the subject state being modified. This portion of their aspect looks as in Fig. 8.

```
abstract protected pointcut subjectChange(Subject s);
abstract protected void updateObserver(
  Subject s, Observer o);
after (Subject s): subjectChange(s) { notifyHandler(s); }
public void notifyHandler(Subject s) {
  Iterator i = ((Set)perSubjectObservers.get(s)).iterator();
  if (i==null) { System.out.println("Trouble 1"); }
  else { while (i.hasNext()) {
            updateObserver(s, (Observer)i.next()); } }
```

Figure 8: Partial AOP Implementation of Observer

We have made a slight change in their code; we have written the after advice for subjectChange as a call to notifyHandler(). In the original version, notifyHandler() is not introduced; instead, the advice simply contains the code that appears in the body of our notifyHandler(). The reason for this change is that in defining the subcontract corresponding to this implementation of Observer, we need to define the execution of this after advice as our Notify pointcut, but *AspectJ* does not provide a construct that will allow us to do so[7]. Therefore, we introduce the notifyHandler() method corresponding to this advice, and use this method to define the Notify pointcut.

The aspect in [9] also defines the code shown in Fig. 9, for adding and removing an observer. The code for adding an observer adds the object to the set corresponding to the subject; the code for removing an observer removes it from this set. Here, too, we have made a change. If the map does not contain an entry for the subject, that means the object is not currently enrolled. We must then add it (paired with a set consisting of just this observer) to the map. This is the point where the object is enrolling as a Subject. So this point should, in our subcontract, be captured by the subjectEnrollment pointcut. To achieve this, we have introduced an empty method, subEnroll(), inserted a call to it in

[7]Recent versions of *AspectJ* seem to include such constructs.

```
public void addObserver(Subject s, Observer o) {
  Set obSet = (Set)perSubjectObservers.get(s);
  if (obSet == null) {obSet = new HashSet(); subEnroll(s);}
  obSet.add(o); perSubjectObservers.put(s,obSet); }
public void removeObserver(Subject s, Observer o) {
  Set obSet = (Set)perSubjectObservers.get(s);
  obSet.remove(o); perSubjectObservers.put(s,obSet); }
public void subEnroll(Subject s) { ; }
```

Figure 9: AOP Implementation of Observer (cont'd)

addObserver(), and will define the subjectEnrollment pointcut (in the subaspect) as a call to subEnroll().

Let us now turn to the subcontract, presented in Fig. 10, corresponding to this implementation of Observer. Due to space limitations, we present only some key portions of the subaspect.

```
protected pointcut attachObs(Subject s, Observer o):
  call(void HKObserver.addObserver(Subject, Observer))
        ∧ args(s,o);
protected pointcut subjectEnrollment(Subject s):
  call(void HKObserver.subEnroll(Subject)) ∧ args(s);
protected pointcut Notify(Subject s):
  call(void HKObserver.notifyHandler(Subject))
        ∧ args(s);
```

Figure 10: Subcontract for AOP Implementation

As we noted above, introducing the subEnroll() method allows us to define an appropriate pointcut for subject enrollment. Similarly, introducing notifyHandler() allows us to define the Notify pointcut. The attachObs pointcut is defined directly in terms of the addObserver() method.

We next ran this implementation (along with the concrete Subject and Observer classes defined in [9]) using our pattern contract and subcontract. Surprisingly, the system printed a message indicating that an observer was not properly updated. Further analysis showed that the addObserver() code (Fig. 9) does not meet the requirement of the pattern contract (Fig. 3, line (7)) that requires observers to be updated upon attachment. Thus, our original contract is general enough to be used to monitor such novel implementations of patterns.

4. RELATED WORK

A number of authors have recognized the importance of describing patterns precisely. The work in [20, 4], for example, improves the traceability of design patterns in design documentation by developing UML extensions. Other authors have more directly addressed the requirements question. Eden *et al.* use a higher-order logic formalism [7, 5] to encode patterns as formulae. The primitives of the logic include classes, methods, and the relations among them. While the approach seems to capture the structural properties of interest, it provides only limited support for behavioral properties. Mikkonen [14] specifies behavioral properties of patterns using an action system, the guarded commands of which operate over abstract models and relations. Taibi *et al.* combine these two approaches to capture both structural and behavioral properties.

There does not seem to be much work focused explicitly on monitoring design pattern specifications. In [19], the authors discuss issues in testing software created using patterns that rely heavily on the use of dynamic binding and dynamic dispatch, but the question of testing whether the patterns are being used correctly is not considered. Techniques for *implementing* design patterns may be worth mentioning. Much of this work targets the development of pattern repositories encoding individual patterns that can be applied to an existing design automatically [6, 21]. More relevant to our work, however, is the aspect-based implementation approach of Hanneman and Kiczales [9] discussed earlier.

5. DISCUSSION

The goal of our work was to develop a monitoring approach for determining whether design pattern requirements are satisfied at runtime. As patterns cut across class boundaries, the requirements to be checked are also cross-cutting. An AOP-based approach was therefore a natural choice. The monitoring code common across all applications of a given pattern is implemented as an abstract aspect; the parts that vary among applications are expressed over abstract functions and pointcuts. These functions and pointcuts are defined in a subaspect corresponding to a particular application of the pattern. The abstract aspect and subaspect combined form the complete monitoring code for the system in question.

Our monitors are fairly robust. Consider, for example, the requirements defined for subjectMethods. Suppose a designer, as part of evolving a system, adds a new method to the class that plays the Subject role, and that this method modifies the state of the object. Even if the new method respects the invariants of the class, problems will arise if the designer neglects to call notifyObs() after performing the modifications, as this will leave the object inconsistent with its observers. Such maintenance errors will be detected by monitoring the new system without any changes to our aspect-based monitor.

Our future work aims to investigate the applicability of our monitoring approach to other types of design patterns. In particular, we plan to investigate more complex patterns, such as those used in concurrent and networked systems.

6. ACKNOWLEDGEMENTS

We would like to thank the anonymous referees for their detailed comments on the first draft of this paper.

7. REFERENCES

[1] R. Binder. *Testing object-oriented systems.* Addison-Wesley, 1999.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: A system of patterns.* Wiley, 1996.

[3] Y. Cheon and G. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *Proc. of ECOOP 2002*, pages 231–255. Springer-Verlag LNCS, 2002.

[4] J. Dong. UML extensions for pattern compositions. *J. of Object Technology*, 3:149–161, 2002.

[5] A. Eden. A visual formalism for object-oriented architecture. In *Proceedings, Integrated Design and Process Technology (IDPT-2002)*, June 2002.

[6] A. Eden, J. Gil, Y. Hirshfeld, and A. Yehudai. Toward a mathematical foundation for design patterns. Technical Report 004, Tel Aviv University, 1999.

[7] A. Eden, A. Yehudai, and J. Gil. Precise specification and automatic application of design patterns. In *Automated Software Engineering*, pages 143–152, 1997.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable OO Software.* Addison-Wesley, 1995.

[9] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proc. of OOPSLA*, pages 161–173. ACM, 2002.

[10] R. Johnson. Components, frameworks, patterns. In *Symposium on Software Reusability*, 1997.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proc. 15th ECOOP*, pages 327–353. Springer, 2001.

[12] C. Lopes, B. Tekinerdogan, W. de Meuter, and G. Kiczales. Aspect oriented programming. In *Proc. of ECOOP'98*. Springer, 1998.

[13] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[14] T. Mikkonen. Formalizing design patterns. In *Proceedings of 20th ICSE*, pages 115–124. IEEE Computer Society Press, 1998.

[15] T. Reenskaug. *Working with objects.* Prentice-Hall, 1996.

[16] D. Riehle. Composite design patterns. In *Proc. of OOPSLA*, pages 218–228. ACM, 1997.

[17] D. Riehle and H. Zullighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.

[18] N. Soundarajan and J. Hallstrom. Responsibilities and rewards: specifying design patterns. In *Proc. of Int. Conf. on Software Engineering (ICSE)*, 2004.

[19] W. Tsai, Y. Tu, W. Shao, and E. Ebner. Testing extensible design patterns in object-oriented frameworks through scenario templates. In *Proc. of COMPSAC*, pages 166–171, 1999.

[20] J. Vlissides. Notation, notation, notation. *C++ Report*, April 1998.

[21] S. Yau and N. Dong. Integration in component-based software development using design patterns. In *24th Ann. Int. Computer Software Applications Conf.*, Taipei, Taiwan, October 2000.

DEET for Component-Based Software

Murali Sitaraman
Durga P. Gandi
Computer Science
Clemson University
Clemson, SC 29634-0974, USA
+1-864-656-3444
murali@cs.clemson.edu

Wolfgang Küchlin
Carsten Sinz
Universität Tübingen,
W.-Schickard Institut für Informatik
Tübingen, Germany
+49-7071-29.77047
kuechlin@informatik.uni-tuebingen.de

Bruce W. Weide
Computer Science and Engineering
The Ohio State University
Columbus, OH 43210, USA
+1-614-292-1517
weide.1@osu.edu

Abstract

The objective of DEET (*Detecting Errors Efficiently without Testing*) is to detect errors automatically in component-based software that is developed under the doctrine of *design-by-contract*. DEET is not intended to be an alternative to testing or verification. Instead, it is intended as a complementary and cost-effective prelude. Unlike testing and run-time monitoring after deployment, which require program execution and comparison of actual with expected results, DEET requires neither; in this sense, it is similar to formal verification. Unlike verification, where the goal is to prove implementation correctness, the objective of DEET is to show that an implementation is defective; in this sense, it is similar to testing. The thesis is that if there is an error in a component-based software system either because of a contract violation in the interactions between components, or within the internal details of a component (e.g., a violated invariant), then it is likely—but not guaranteed—that DEET will find it quickly. DEET is substantially different from other static checking approaches that achieve apparently similar outcomes. Yet it builds on a key idea from one of them (Alloy): Jackson's *small scope hypothesis*. Among other things, the DEET approach weakens full verification of component implementation correctness to static checking for errors, in a systematic way that makes it clear exactly which defects could have been detected, and which could have been overlooked.

Keywords

Design-by-contract, error detection, SAT solvers, software component, specification, static analysis, static checking.

1. INTRODUCTION

This paper describes a new approach to detecting errors in component-based software that is developed using the popular paradigm known as *design-by-contract*, and presents results from early experience with a prototype tool. We call the approach DEET for *Detecting Errors Efficiently without Testing*. DEET has the potential to be effective and efficient, and the potential to "scale up" to large component-based software systems. It is intended to offer the following important benefits over early testing:

- DEET can analyze one component at a time in a modular fashion, i.e., it can detect mismatches between a component implementation and its contract, even in isolation from the rest of a component-based system.
- Since DEET does not require program execution or inlining of called procedures, it does not depend on code or even stub availability for other components.
- DEET can detect substitutability bugs, i.e., contract violations that are literally undetectable by testing. Such bugs arise from situations where a particular implementation of a component requires less or delivers more than its contract specifies, and where the correctness of the larger system relies on such incidental behavior of that particular implementation.
- DEET is automated and does not require manual input selection.
- When an error is detected, DEET can pinpoint the origin of the error in the source code. In particular, it can detect internal contract violations among participating components in a larger system—and assign blame. This property makes it suitable for debugging component-based software.

DEET bears some resemblance to other static analysis/checking tools[1], e.g., Alloy [38] and ESC [18]. Section 2 explores connections with these two systems in particular. Section 3 explains the steps of the DEET approach with a detailed example. Section 4 discusses other related work, and Section 5 summarizes the paper.

2. ESC, ALLOY, AND DEET

From the synopsis of features given in the introduction, it may appear that DEET is essentially the same as ESC or Alloy—two well-known efforts in the same general direction. In fact, while DEET shares some common objectives with these approaches, it is complementary in nearly every respect, as explained in this section. Only one technical detail from these systems has been consciously adapted for use in DEET: Jackson's *small scope hypothesis* [37], which is discussed in Section 3.2.3.

2.1 Objectives, Context, and Assumptions

ESC and Alloy seek incremental improvements to current software engineering practice, focusing on "real" languages

[1] But should not be confused with N, N-diethyl-m-toluamide. ABC News (7/3/02) summarized a *New England Journal of Medicine* report as follows: "DEET Is Best Bug Repellent."

and "real" programmers—constraints that impose technical complications which have proved unexpectedly troublesome. The defects that can be detected in practice with these tools are limited by (1) the failure of the "real" programming language to ensure by fiat certain desirable properties of programs, to prevent programming practices that complicate reasoning about software behavior, and in general to have semantics that facilitates modular reasoning; (2) the assumption that "real" programmers are unwilling or even unable to write full specifications of intended functional behavior, and that they will write only certain kinds of annotations that capture part of that intent; and (3) an interest in a tool that can be used with any software—component-based or not—that can be written in the "real" language.[2]

By contrast, DEET is part of a long-term plan to explore the foundations of future software engineering practice as it *could* be. The overall project goal is not to live within the shackles of current practice, but rather to remove them. DEET's context includes (1) a combined specification and implementation language (Resolve [66]) that is expressly designed to support modular reasoning, while still permitting the development of "real" software by strictly disciplined use of "real" languages such as C++ [34]; (2) a recognition, based on teaching experience [71], that tomorrow's software engineers can be taught to understand and even to write formal-language specifications, just as they can be taught to write formal-language implementations; and (3) a focus on component-based software. The project vision is to have an automatic verifier for functional (and performance) correctness of component-based systems. The purpose of an intermediate tool such as DEET is for software engineers to find errors quickly before attempting full verification, as this is likely to remain more efficient than full verification and hence potentially give real-time feedback.

2.2 References and Aliasing

ESC deals with, among other things, "nil-dereference errors" [18]. Among other things, it introduces a "downward closure" rule for *modifies* clauses in contracts. This is used to account for situations where aliases to instance fields of a class could impact an object's abstract state via unintended side effects. This, in turn, leads to a "rep visibility requirement" and another annotation construct, the *depends* clause, that is "a key ingredient of our solution to the problem" [18]. Nonetheless, it is admitted that "one problem in this area that has stumped us is a form of rep exposure that we call abstract aliasing"[18]; see also [17]. The potential for aliasing technically does not always prevent modular reasoning, but the above measures help illustrate that it seriously complicates matters [79].

The Alloy approach "targets properties of the heap" [77] to detect errors in implementations of linked data structures and null dereferences. The extent to which the Alloy approach can scale up to other properties remains an open question: "We expect that the tool will work well for modular analysis of even quite complex classes; how well it scales for analyses amongst classes and whether it will be economical enough for everyday use remains to be seen" [42].

Resolve has value semantics for all variables; there is no aliasing because the language does not permit it [49]. Resolve includes reference-free abstractions for lists, trees, etc., and programs that use these components are not burdened with the complication of reasoning about references or aliasing. However, it remains possible to write programs for situations where explicit aliasing improves efficiency and would be exploited in a language like Java. Using specifications of pointer-like behavior [47], it is possible to reason about these programs formally and to find errors in them using the DEET approach (although the current prototype does not handle this). Techniques used in ESC and/or Alloy to deal with aliasing may prove helpful in such situations, but our initial focus is on typical Resolve programs, in which pointers are not needed or used. And in any case, the use of value semantics for all variables distinguishes DEET from ESC and Alloy.

2.3 Undefined or Invalid Variable Values

ESC reportedly has been successful in detecting failure-to-initialize defects. Indeed, this seems to be one of its primary uses: "… our experience has been that many ESC verifications can be successfully completed with almost no specifications at all about the contents and meanings of abstract types, other than the specification of validity" [18].

The Alloy approach is tied to the Alloy Annotation Language [42] and "is designed for object model properties: namely what objects exist, how they are classified grossly into sets, and how they are related to one another. It is not designed for arithmetic properties…" [37].

In Resolve, every variable has an initial value upon declaration. A variable is never "undefined" or "invalid" and there is no question about whether it "exists", so there is no need for DEET to detect such errors. DEET, rather than avoiding specifications of "the contents and meanings of abstract types" and "arithmetic properties" (of arithmetic types), is intended to find defects related to fully specified component behavior.

2.4 Contracts and Other Assertions

ESC includes contract specification syntax, and most reported examples involve these constructs. However, ESC does not treat such assertions as complete contract specifications, but merely as partial statements of intent (e.g., as specifying only the property that a variable has been initialized). Moreover, ESC seeks to avoid making programmers write loop invariants, for example, because they can be "pedagogical and heavy-handed" [18] and sometimes can be produced automatically. For instance, in a definite **for** loop whose index i ranges from 1 to 10, the invariant $1 \leq i \leq 10$ can be generated by a tool, allowing ESC to detect some range errors without the programmer having to write a loop invariant.

Alloy requires no annotations beyond the property to be checked, and the assertions it checks are not necessarily parts of formal contracts, although the authors "are hopeful that it will extend to the analysis of code in terms of abstract sets and relations specified in an API" [37]. So, the current Alloy approach "expands calls inline" [77] rather than relying on contract specifications, hence requires special translation to handle recursive calls. Alloy relies on loop-unrolling to avoid the need for programmer-supplied loop invariants.

DEET expects full contract specifications for components, and additional internal assertions inherent in the Resolve syntax: loop invariants, representation invariants, and abstraction relations. This is necessary to study the impact of having

[2] ESC handles not just sequential programs but a class of synchronization errors in multi-threaded programs. Alloy and DEET so far are limited to sequential programs.

complete specifications on the quality of checking that can be achieved. If it turns out that experience with DEET suggests that future software engineers would be better off by learning to write specifications than by avoiding them, then it becomes our obligation to teach them how to write specifications. Moreover, because we focus explicitly on component-based software, DEET reasonably expects component libraries to have the specifications and internal assertions needed for full verification. This is because the extra cost of developing these annotations can be amortized over many component uses. The Resolve component catalog is an existence proof that a nontrivial library of fully specified components can be developed.

2.5 Soundness and False Alarms

All the tools here might fail to detect errors. However, they differ in why this is so. In ESC, there is no characterization of which errors might have eluded detection. Moreover, ESC is unsound in the usual logical sense because "verification condition generation is unsound". The verification condition produced is fed to ESC's own refutation-based general theorem prover that is claimed to be "sound, as far as we know". ESC can also produce false alarms, or "spurious warnings". The claim is that neither of these apparent technical shortcomings is a big problem if, on balance, the system in practice finds interesting classes of defects that actually do exist [18].

The idea of Alloy is that if an error is not detected, it is because the "scope", or subset of situations considered in the analysis, has been limited. Specifically, the Alloy approach generates a composite verification condition, and then limits the number of heap cells for each type and the number of loop iterations for each loop to turn this condition into a propositional formula that can be fed to a back-end SAT-solver, which tries to refute it. Any given error might have a witness only outside this scope. Alloy is designed not to give false alarms, i.e., it is not supposed to report errors where none exist [37].

DEET is much like Alloy in this regard, with two major exceptions. First, the soundness of verification condition generation has been established for most constructs of Resolve [22, 32, 67, 68]. The overall approach is still that the verification condition needed for full verification is generated from the relevant specifications and code. But then the scopes of all variables are restricted—not based on the number of heap cells in a data representation, but on the possible abstract mathematical model values for the variables involved. As with Alloy, this allows the verification condition to be recast as a propositional *error hypothesis* that can be fed to a back-end off-the-shelf SAT-solver in an effort to produce a witness to a defect. So DEET, like Alloy, can fail to detect an error if there are no witnesses to that error in the analyzed scope. But DEET does not report errors in code that could be verified as correct.

In overall structure and in many technical details, DEET seems to be a closer cousin of Alloy than of ESC. Nonetheless, there are other significant technical differences between DEET and Alloy. For example, DEET's verification condition generator automatically accounts for the "path conditions" associated with various execution paths. The Alloy approach is based on generating a control flow graph for the program to account for the various execution paths, and is cleverly optimized to do this [77]. A proposal in [42] suggests that loop-free code with method invocations can be handled by generating verification conditions using a logic similar to that used in ESC [17].

Two other differences between DEET and its predecessors are important. In the process of looking for errors, DEET generates the verification condition that would be needed to prove correctness. Proofs of this assertion can be attempted with human-assisted theorem provers (e.g., PVS [60]) when DEET finds no errors. And using the foundations for an extended system for specification and verification of performance (both time and space) [46, 69, 70], in principle DEET might be extended to detect errors relative to performance contracts.

3. DEET APPROACH

This section explains the DEET approach. As in [37], we choose a simple list example to explain in detail, because only with a concise example is it possible to illustrate concretely the variety of technical issues involved. The example helps to demonstrate the additional advantages of both the DEET and Alloy approaches over testing alone. Finally, it highlights some key differences between the DEET and Alloy approaches, because it involves recursive code that is a client of a *List* component contract, rather than being an implementation of a list method that has direct access to the data representation.

```
Concept List_Template( type Entry );
  uses Std_Integer_Fac, String_Theory;

  Type List is modeled by (
     Left, Right: Str(Entry)
    );
    exemplar P;
    initialization
      ensures|P.Left| = 0 and |P.Right| = 0;

  Operation Insert( alters E: Entry;
                            updates P: List );
    ensures P.Left = #P.Left and
      P.Right = ⟨#E⟩ * #P.Right;

  Operation Remove( replaces R: Entry;
                            updates P: List );
    requires |P.Right| > 0;
    ensures P.Left = #P.Left and
      #P.Right = ⟨R⟩ * P.Right;

  Operation Advance ( updates P: List );
    requires |P.Right| > 0;
    ensures |P.Left| = |#P.Left| + 1 and
      P.Left * P.Right = #P.Left * #P.Right;
    …
end List_Template;
```

Figure 1: A Specification of *List_Template*

3.1 Example: A Defective Implementation

Figure 1 shows a skeleton of a contract specification for a *List_Template* component in a dialect of Resolve [66]. In the specification, the value space of a *List* object (with position) is modeled mathematically as a pair of strings of entries: those to the "left" and those to the "right" of an imaginary "fence" that separates them. Conceptualizing a *List* object with a position makes it easy to explain insertion and removal at the fence. A sample value of a *List* of *Integer*s object, for example, is the ordered pair (<3,4,5>, <4,1>). Insertions and removals are explained as taking place between the two strings, specifically at the left end of the right string.

Formally, the declaration of type *List* introduces the mathematical model and, using an example *List* variable *P*, states that both the left and right strings of a *List* are initially empty. A *requires* clause serves as an obligation for a caller, whereas an *ensures* clause is a guarantee from a correct implementation. In the ensures clause of *Insert*, for example, *#P* and *#E* denote the incoming values of *P* and *E*, respectively, and *P* and *E* denote the outgoing values. The infix operator * denotes string concatenation, the outfix operator ⟨•⟩ denotes string construction from a single entry, and the outfix operator |•| denotes string length.

```
 Enhancement Reversal_Capability for
                                List_Template;
   Operation Reverse( updates P: List );
     requires |P.Left| = 0;
     ensures P.Left = Rev(#P.Right) and
             |P.Right| = 0;
  end Reversal_Capability;
```

Figure 2: Specification of a List Reversal Operation

```
 Realization Recursive_Realiz for
                        Reversal_Capability;
   Recursive Procedure Reverse(
                                updates P: List );
     decreasing |P.Right|;
     var E: Entry;
     if ( Right_Length(P) > 0 ) then
       Remove(E, P); Reverse(P); Insert(E, P);
     end;
   end Reverse;
end Recursive_Realiz;
```

Figure 3: A Defective Implementation of *Reverse*

An interesting aspect of the *Insert* specification is that its behavior is relational. The semantics of *alters* mode for the formal parameter *E* is that the result value of entry *E* is undetermined. This under-specification allows implementations not to have to make expensive copies of non-trivial type parameters, which is an important issue in the design of generic abstractions. It is well known that copying references, while efficient, introduces aliasing and complicates reasoning [33, 49, 79]. The present specification is more flexible. It allows the entry to be moved or swapped into the container structure (efficiently, i.e., in constant time, by manipulating references "under the covers") and thus potentially to alter it, without introducing aliasing [30]. Correspondingly, the *Remove* operation is specified to remove an entry from *P*, and it *replaces* the parameter *R*. Operation *Advance* allows the list insertion position (fence) to be moved ahead. The rest of the specification is in [68]; but it is not needed to understand this example.

Figure 2 contains the specification of an operation to reverse (the right string of) a list. Here, *Rev* denotes the mathematical definition of string reversal. Figure 3 shows an (incorrect) recursive implementation. It uses the *List* operations given in Figure 1. To demonstrate termination, the recursive procedure has a progress metric using the keyword *decreasing*.

3.2 DEET Steps to Detect Errors

3.2.1 Generation of a Symbolic Reasoning Table

As a first step in modular static analysis—either to prove correctness or to find errors—a *symbolic reasoning table* is generated [68]. The soundness and relative completeness of the approach that justifies this step are established in [32]. Figure 4 contains a table for the code in Figure 3. A key observation is that this table can be produced mechanically from the information in Figures 1, 2, and 3, as explained in [32, 68] and summarized below. In the table, each program *State* is numbered. For each state, the *Assume* column lists verification assumptions and the *Confirm* column lists the assertions to be proved. The *Path Condition* column denotes under what condition a given state will be reached.

Reasoning table generation involves the profligate use of variable names, because each program variable name is extended with the name of the state to denote the value of the variable in that state. *P1*, for example, denotes the value of variable *P* in state 1. To prove that the procedure for *Reverse* is correct, we assume that its precondition is true in the initial state and must confirm that its postcondition is true in the final state. For modular analysis, we rely only on the behavioral contracts of the called operations (i.e., *Insert* and *Remove*). In particular, for the calling code to be correct, we must be able to confirm that the precondition of a called operation is true in the state before the call; then we may assume that the postcondition is true in the state after the call. The recursive call to *Reverse* is treated just like any other call. However, before the recursive call, we additionally need to confirm that the progress metric decreases.

State	Path Condition	Assume	Confirm
0		|P0.Left| = 0	
	`if ( Right_Length(P) > 0 ) then`		
1	|P0.Right| > 0	P1 = P0	|P1.Right| > 0
	`Remove(E, P);`		
2	|P0.Right| > 0	P2.Left = P1.Left ∧ P1.Right = <E2> * P2.Right	|P2.Left| = 0 ∧ |P2.Right| < |P0.Right|
	`Reverse(P);`		
3	|P0.Right| > 0	E3 = E2 ∧ P3.Left = Rev(P2.Right) ∧ |P3.Right| = 0	
	`Insert(E, P);`		
4	|P0.Right| > 0	P4.Left = P3.Left ∧ P4.Right = <E3> * P3.Right	
	`end;`		
5.1	|P0.Right| = 0	P5 = P0	P5.Left = Rev(P0.Right) ∧ |P5.Right| = 0
5.2	|P0.Right| > 0	P5 = P4	

Figure 4: A Reasoning Table for the *Reverse* Procedure

The path condition in a given state serves as an antecedent for the implications that are the actual assertions to be assumed and confirmed in that state. In other words, assume/confirm entries apply only when the path condition holds.

3.2.2 Generation of Error Hypotheses

To prove the correctness of the code, then, entails confirming each obligation in the last column, using the assumptions in the states above and including the state where the obligation arises (but, critically for soundness, not the states below it in the table [32]). Rather than attempting the non-trivial process of verification using a general theorem-proving tool, DEET instead looks for a witness to a bug in the code. In particular, it attempts to find values for the variables that satisfy all relevant assumptions but that fail to satisfy something that needs to be confirmed. This is done by conjoining the assumptions and the negation of the assertion to be confirmed, and then seeking a satisfying assignment for the variables in this *error hypothesis*—a witness to a bug.

To illustrate the idea, consider the assertions that need to be confirmed in state 5 (arising from the postcondition of *Reverse*). In particular, consider the recursive case when the path condition *|P0.Right| > 0* holds. The code is defective if there is a set of assignments to the variables that satisfies the assertion in Figure 5. In the figure, the conjunct numbered I is the path condition, conjuncts II through VII are assumptions from states 0 through 5, and conjunct VIII is the negation of the assertion to be confirmed in state 5.

Error hypothesis generation also can be mechanized. There are four error hypotheses for the present example, one each corresponding to the confirm clauses in states 1 and 2, and two for state 5 (one for the base case 5.1 and one for the recursive case 5.2). If a satisfying assignment exists for an error hypothesis arising from an intermediate state (e.g., state 1 or 2 here), then the code fails to live up to its part of the contract for an operation it calls. It is possible that the error hypothesis arising from the final state at the end of the code (in state 5 in the table) cannot be satisfied, even though intermediate errors (e.g., violation of preconditions of called operations) are found. The code still should be deemed defective under design-by-contract because the calling code violates a requirement of a called operation.

```
(|P0.Right| > 0) ∧
  I
(|P0.Left| = 0) ∧
  II
(P1 = P0) ∧                                  III
(P2.Left = P1.Left ∧
  P1.Right = <E2> * P2.Right) ∧              IV
(E3 = E2 ∧ P3.Left = Rev(P2.Right) ∧
  |P3.Right| = 0) ∧                          V
(P4.Left = P3.Left ∧
  P4.Right = <E3> * P3.Right) ∧              VI
(P5 = P4) ∧                                  VII
(¬ (P5.Left = Rev(P0.Right) ∧
  |P5.Right| = 0))                           VIII
```

Figure 5: Error Hypothesis for Confirm Clause 5.2

3.2.3 Restriction of Scope

The search for a witness to an error hypothesis relies on Jackson's small scope hypothesis (where "scope" is, loosely speaking, a measure of the size of the input space to be searched). Jackson notes that even though, for any given scope, one can construct a program with a bug whose detection requires a strictly larger scope, in practice, many bugs will be detectable in small scopes [37]. If a bug is found within a small scope, then the code is not consistent with the verification conditions. If none is found in the given scope, then there are no inconsistencies in that scope; yet, inconsistencies might exist in a larger scope.

For DEET, we have explored restricting the scopes of participating variables by restricting their mathematical spaces, instead of placing bounds on loop iterations or heap cells. It is reasonable to begin with the most stringent restrictions. In the example, for instance, we start by looking for a witness to the error hypothesis in which all variables of type *Entry* have exactly one value, and in which strings of type *Entry* are either empty or contain just a single *Entry* with that value. Without loss of generality, we use *Z0* to stand for the single value of type *Entry*. This in turn restricts the scope of the search for strings to the two-element set {*Str_Empty*, *Str_Z0*}, where *Str_Empty* denotes the empty string and *Str_Z0* denotes the string <*Z0*>.

These restrictions on scope lead to a (possibly large, but finite) propositional formula corresponding to each error hypothesis generated from the code and the specifications, e.g., the one in Figure 6. Each satisfying assignment for this formula identifies a particular witness to a particular error hypothesis. To conserve space, we have shown only a part of the formula to use as a means of explaining how it can be generated. In the conjuncts listed in Figure 6, the names of all (Boolean) variables can be generated automatically. The variable *P0_Left_equals_Str_Empty* being true, for example, denotes that the left string of the program variable *P* in state 0 is equal to the empty string. In addition to the variables that correspond directly to the symbols in Figure 5, variable names corresponding to mathematical expressions involving string length, reverse, and concatenation are needed as well. Given this, the first two conjuncts in Figure 6 correspond directly to those in Figure 5.

To assert that *P1 = P0* (conjunct III in Figure 5), the formula has to assert that the left strings of the two lists are equal and that the right strings are equal. However, each string may have only one of two values because of scope restriction: *Str_Empty* or *Str_Z0*. The left strings of *P0* and *P1* will be equal if they are both *Str_Empty* or if they are both *Str_Z0*. This observation leads to conjuncts in III in Figure 6. The rest of the conjuncts are derived similarly. A list of additional conjuncts needs to be generated to complete the propositional formula generation, and only some of these additional conjuncts are shown in Figure 6. For example, we need to assert that the right string of a list cannot be both empty and contain a single entry (although it could be longer), i.e.:

```
(¬ P0_Right_equals_Str_Empty ∨
 ¬ P0_Right_equals_Str_Z0)
```

The formula needs to make this assertion for the left and right strings of each *List* variable in each state. Another set of assertions is based on mathematical string length, e.g.:

```
(Len_P0_Right_equals_Zero ⇔
 P0_Right_equals_Str_Empty)
```

Other sets of assertions are generated for string reversal and concatenation within the restricted scope. Notice that similar conjuncts for, e.g., reversal of the left string of a list, are not generated because they do not arise in the conjuncts corresponding to the assertions in Figure 5. The complete formula is at:

http://www.cs.clemson.edu/~resolve/reports/RSRG-03-05.pdf

```
(¬Len_P0_Right_equals_Zero)                   I
( Len_P0_Left_equals_Zero)
  II
((P1_Left_equals_Str_Empty ∧
P0_Left_equals_Str_Empty)                     III
      ∨ (P1_Left_equals_Str_Z0 ∧
P0_Left_equals_Str_Z0)) ∧
  ((P1_Right_equals_Str_Empty ∧
P0_Right_equals_Str_Empty)
      ∨ (P1_Right_equals_Str_Z0 ∧
P0_Right_equals_Str_Z0))
((P2_Left_equals_Str_Empty ∧
P1_Left_equals_Str_Empty)                     IV
      ∨ (P2_Left_equals_Str_Z0 ∧
P1_Left_equals_Str_Z0)) ∧
  ((P1_Right_equals_Str_Empty ∧
    Cat_E2_P2_Right_equals_Str_Empty)
        ∨ (P1_Right_equals_Str_Z0 ∧
              Cat_E2_P2_Right_equals_Str_Z0))
(E3_equals_Z0 ∧ E2_equals_Z0) ∧               V
  ((P3_Left_equals_Str_Empty ∧
          Rev_P2_Right_equals_Str_Empty)
      ∨ (P3_Left_equals_Str_Z0 ∧
              Rev_P2_Right_equals_Str_Z0)) ∧
  (Len_P3_Right_equals_Zero)
((P4_Left_equals_Str_Empty ∧
P3_Left_equals_Str_Empty)                     VI
      ∨ (P4_Left_equals_Str_Z0 ∧
P3_Left_equals_Str_Z0)) ∧
  ((P4_Right_equals_Str_Empty ∧
        Cat_E3_P3_Right_equals_Str_Empty)
      ∨ (P4_Right_equals_Str_Z0 ∧
              Cat_E3_P3_Right_equals_Str_Z0))
((P5_Left_equals_Str_Empty ∧
P4_Left_equals_Str_Empty)                     VII
      ∨ (P5_Left_equals_Str_Z0 ∧
P4_Left_equals_Str_Z0)) ∧
  ((P5_Right_equals_Str_Empty ∧
       P4_Right_equals_Str_Empty)
      ∨ (P5_Right_equals_Str_Z0 ∧
P4_Right_equals_Str_Z0))
(¬ ((( P5_Left_equals_Str_Empty ∧
  VIII
        Rev_P0_Right_equals_Str_Empty) ∨
      (P5_Left_equals_Str_Z0 ∧
         Rev_P0_Right_equals_Str_Z0)) ∧
      (Len_P5_Right_equals_Zero)))
```

Additional Assertions

Unique Values (sample: P0.Right)

```
(¬ P0_Right_equals_Str_Empty ∨
 ¬ P0_Right_equals_Str_Z0)
```

String Length (sample: |P0.Right|)

```
(Len_P0_Right_equals_Zero ⇔
 P0_Right_equals_Str_Empty)
```

String Reverse (sample: Rev(P0.Right))

```
(Rev_P0_Right_equals_Str_Empty ⇔
 P0_Right_equals_Str_Empty) ∧
(Rev_P0_Right_equals_Str_Z0 ⇔
 P0_Right_equals_Str_Z0)
```

*String Concatenate (sample: <E2> * P2.Right)*

```
(¬ Cat_E2_P2_Right_equals_Str_Empty) ∧
(Cat_E2_P2_Right_equals_Str_Z0 ⇔
     (E2_equals_Z0 ∧
       P2_Right_equals_Str_Empty))
```

Figure 6: Selected Conjuncts Corresponding to Figure 5

The number of variables in the formula is bounded by the product of the size of the restricted scope, the number of program variables and expressions in the original verification conditions, and the number of rows in the tracing table (i.e., the number of lines of code). The number of conjuncts depends on the mathematical models and the assertions involved, along with the number of generated variables.

3.2.4 Error Detection

The example illustrates that the formulas generated during this process are not in conjunctive normal form (CNF). We do not convert them to CNF, but rather apply a SAT-solver that can handle arbitrary propositional formulas [41]; other state-of-the-art SAT solvers such as BerkMin [28] or Chaff [54] could be used by converting the formulas to CNF. The solver we have used, developed by the co-authors at Tübingen, is based on a Davis-Putnam-style [16] algorithm. It can handle formulas involving several thousand variables. For example, when the formula in Figure 6 was (translated into the required input format and) supplied to this solver, it produced the assignment given in Figure 7 within a fraction of a second. In addition, it concluded that this is the *only* solution.

```
Len_P0_Left_equals_Zero
P0_Left_equals_Str_Empty
P0_Right_equals_Str_Z0
Rev_P0_Right_equals_Str_Z0
P1_Left_equals_Str_Empty
P1_Right_equals_Str_Z0
P2_Left_equals_Str_Empty
E2_equals_Z0
P2_Right_equals_Str_Empty
Cat_E2_P2_Right_equals_Str_Z0
Rev_P2_Right_equals_Str_Empty
P3_Left_equals_Str_Empty
E3_equals_Z0
P3_Right_equals_Str_Empty
Cat_E3_P3_Right_equals_Str_Z0
Len_P3_Right_equals_Zero
P4_Left_equals_Str_Empty
P4_Right_equals_Str_Z0
P5_Left_equals_Str_Empty
P5_Right_equals_Str_Z0
```

Figure 7: Only Solution (*true* Vars) for Formula in Figure 6

The solution gives the value of each program variable in each state. For example, the following variables are true in the witness: *P0_Left_equals_Str_Empty*, *P0_Right_equals_Str_Z0*, *P5_Left_equals_Str_Empty*, and *P5_Right_equals_Str_Z0*. This corresponds to a *List* input value of *P* = (< >, <*Z0*>) and an output value of *P* = (< >, <*Z0*>). The code is defective because the output value as required by the specification is *P* = (<*Z0*>, < >). A problem with the code is identified here with a severely restricted scope because the lengths of the left and right strings resulting from the code and specification do not match. (If no satisfying assignments were found, the scopes would have to be enlarged and the process repeated.)

A key benefit of the modular error detection approach is that it is relatively easy to debug the code from the given solution. Based on the finding in Figure 7, especially with the help of a tool to improve the presentation, the programmer of *Reverse* can infer how to fix the code. In particular, based on the input that revealed a defect (*P0*), it is easy to see that the program is erroneous when it is given a list *P.Left* = < > and *P.Right* = <*Z0*>. The assignment from the SAT solver gives the values of each variable in each state, making it relatively easy to debug.

3.3 Effectiveness and Efficiency of DEET

DEET should need to deal with a large number of statements only rarely, because it examines not just one component, but only one component operation, at a time. Still, to check scalability in this dimension, we mechanically generated an error hypothesis formula for a "synthetic" procedure body with 2000 statements, using operation specifications similar to the ones given in the example [72]. The resulting formula involved 6000 variables and twice as many conjuncts. The solver found two solutions (witnesses to errors) in less than 2 seconds on a 1.2 MHz Athlon PC.

Much more experimentation is needed with this and other solvers before we can reach any conclusions on the effectiveness or efficiency of DEET. There is significant potential for further improvements to take advantage of the kinds of formulas that arise from the DEET process, including parallelization and specialized computer algebra techniques.

4. OTHER RELATED WORK

The idea of error detection within a small "scope"—borrowed by DEET from Alloy—differs from most related work in fundamental ways, as noted in [29, 37, 42, 77], and we summarize only additional differences here.

The benefits of static analysis are widely acknowledged, even more so recently as a result of the extensive work in model checking research and industrial practice [10, 14, 36]. Though model checking has its origins in hardware verification, an impressive collection of results spans a spectrum of programming languages and software systems. Given that it is difficult to summarize even the most important work in this area, we discuss only a representative sample.

Finite-state systems are the focus, though there have been efforts to extend model checking to minimize the impact of this inherent limitation (e.g., [5]). Holzman has employed SPIN to detect numerous bugs in the PathStar processing system developed in C. Java Pathfinder at NASA has been used successfully to locate a variety of heap-related errors [31]. To limit the search space, Bandera, a tool for analyzing Java code, employs user-supplied abstractions [15, 58] whereas Smith *et al.* have described a system that assists in property specification [74]. The fundamental difference between DEET and such uses of model checkers is in the way a finite-state model of program execution is devised, i.e., by combining Jackson's small scope hypothesis with assertions that arise from verification conditions that are generated from the code and component contract specifications.

Symbolic execution of programs, where concrete inputs used in testing are replaced with symbolic values to generate constraints between inputs and outputs, have been used for debugging and testing [12, 45] and verification [19]. Early work on symbolic execution was limited by its inability to handle complex types, loops, and dynamic data structures. Coen *et al.* have shown that symbolic execution can be useful for verification of safety-critical properties in an industrial setting, but this requires severe limitations to be placed on the code [13]. More recently, using symbolic execution for model checking, the SLAM project [1] has shown how to handle recursive calls in C code. Khurshid *et al.* have addressed properties of the heap and dynamic data structures [43]. Unlike these efforts, whose focus is on verification, PREfix is a tool based on symbolic execution for error detection [6]. While the tool has been shown to reveal errors in large-scale C/C++ systems, it cannot handle properties such as invariants and it can produce false alarms.

With user-supplied loop invariants (similar to the DEET approach for handling loops), in [39] Jensen *et al.* have discussed how to prove heap-related properties and find counterexamples. Their program has been shown to be quite effective in practice. Their work differs from traditional pointer analyses because they can answer more questions that can be expressed as properties in first-order logic. While this work focuses on linear linked lists and tree structures, more recently Moller and Schwartzbach have extended the results to all data structures that can be expressed as "graph types" [53]. There is also significant work in shape analysis, including recent work on parametric shape analysis that allows more questions to be answered concerning heaps [62]. Ramalingam *et al.* describe how to check client conformance with component constraints [61] using abstract interpretation. The goals and methods of these related efforts are quite different from ours because our focus is on the total correctness of component-based software based on design-by-contract, not on verifying heap properties.

Ernst provides an overview of the complementary merits of dynamic and static analysis approaches for error detection in [24]. While the benefits of writing assertions and using them to detect errors in software are widely known [26, 78], assertion checking is especially useful in component-based software development to detect contractual violations among collaborating components [2, 8, 21, 27, 52]. Eiffel is among the earliest systems to popularize runtime assertion checking [52]. iContract, a contract-checking tool for Java programs, has similar objectives [20]. Using an executable industrial-strength specification language, AsmL, Barnett *et al.* describe a system for dynamic checking [2]. Cheon and Leavens have used JML for writing assertions and for runtime assertion checking of component-based Java programs [7, 8, 9]. The benefit of contract checking in commercial development of a component-based C++ software system is described in [34]. Use of wrappers to separate contract-checking code from underlying components is described in [21, 22]. However, runtime checking is difficult to modularize, requires that implementations of not just the unit being checked but all reused components be available, detects only errors that arise from particular implementations rather than their contracts (so substitutability bugs are not revealed), and requires manual input selection—all problems that DEET avoids.

There is considerable work on making SAT solvers efficient. But that work is orthogonal to DEET, which is intended to use an off-the-shelf solver (i.e., based only on its functional specification). Experimentation with different solvers for DEET is necessary to develop an effective tool because of potentially significant performance differences among solvers.

5. SUMMARY

The ultimate objective of formal verification techniques is to prove that a piece of code (in our case, a software component) is correct with respect to its specification. Experience shows, however, that before attempting to prove correctness, it is usually cost-effective to look for behavioral errors that can be found by simpler means. DEET is our first effort toward a modular, static analysis approach for discovering errors of this sort, including some that are not revealed by testing—which is the usual approach to finding code defects—or by existing static analysis/checking tools. Some aspects of the DEET approach have been automated at the time of writing, and others are work in progress.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grant CCR-0113181. We thank the reviewers for observing, quite correctly, that some related work deserved a more detailed discussion than in our original submission.

REFERENCES

1. T. Ball and S. K. Rajamani. The SLAM toolkit. *CAV* 2001, pp. 260-264.
2. M. Barnett, W. Grieskamp, C. Kerer, W. Schulte, C. Szyperski, N. Tillmann, and A. Watson. Serious specification for composing components. In *Proc. Sixth ICSE Workshop on Component-Based Software Engineering*, May 2003, pp. 31-36.
3. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *Tools and Algorithms for the Analysis and Construction of Systems* (TACAS'99), LNCS 1579, Springer-Verlag, 1999.
4. W. Blochinger, C. Sinz, and W. Küchlin. Parallel propositional satisfiability checking with dynamic learning. *Parallel Computing,* 29(7), 2003, pp. 969–994.
5. T. Bultan, R.Gerber, and W. Pugh, Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4), July 1999.
6. W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7), 2000, pp. 775–802.
7. Y. Cheon and G.T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Magnusson, B., editor, *ECOOP 2002 – Object-Oriented Programming, 16th European Conference, Malaga, Spain, Proceedings*, LNCS 2374, Springer-Verlag, Berlin, June 2002, pp. 231-255.
8. Y. Cheon and G.T. Leavens. A runtime assertion checker for the Java modeling language (JML). In *Proc. Int'l Conf. Software Engineering Research and Practice*, CSREA Press, June 2002, pp. 322-328.
9. Y. Cheon and G.T. Leavens, M. Sitaraman, and S. H. Edwards. *Model variables: Cleanly supporting abstraction in design by contract*. Technical Report 03-10a, Department of Computer Science, Iowa State University, September 2003; available from archives.cs.iastate.edu.
10. D. Clarke, O. Grumberg and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency - Reflections and Perspectives*. LNCS 803, Springer-Verlag, 1994.
11. D. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7{34}, 2001.
12. L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3), September 1976, pp. 215-222.
13. A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. Using symbolic execution for verifying safety-critical systems. In *Proc. 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2001, pp. 142–151.
14. D. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In Gerard Berry, Hubert Comon, and Alan Finkel, editors, *Proc. Computer Aided Verification*, LNCS 2102, Springer-Verlag, 2001, pp. 435-453.
15. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, H. Zheng, H. Bandera: extracting finite-state models from Java source code. *Proceedings of the 22nd International Conference on Software Engineering*, Limeric, Ireland, 2000.
16. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM 7*, 1960, 201-215.
17. D. L. Detlefs, K. R. M. Leino, and G. Nelson. *Wrestling with Rep Exposure*. Research Report 156, Compaq Systems Research Center, July, 1998.
18. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. *Extended Static Checking.* Research Report 159, Compaq Systems Research Center, December, 1998.
19. L. K. Dillon. Using symbolic execution for verification of Ada tasking programs. *ACM Transactions on Programming Languages and Systems*, 12(4), 1990, pp. 643-669.
20. A. Duncan and U. Hölzle. *Adding Contracts to Java with Handshake*. Technical Report TRCS98-32, Univ. of California at Santa Barbara, Dec. 1998.
21. S. H. Edwards, G. Shakir, M. Sitaraman, B.W. Weide, and J. Hollingsworth. A framework for detecting interface violations in component-based software. In *Proc. 5th Int'l Conf. Software Reuse*, IEEE, June 1998, pp. 46-55.
22. S. H. Edwards, M. Sitaraman, B.W. Weide, and J. Hollingsworth. Contract-Checking Wrappers for C++ Components. *IEEE Trans. On Software Engineering*, 2004, to appear.

23. G. W. Ernst, R. J. Hookway, and W. F. Ogden. Modular verification of data abstractions with shared realizations. *IEEE Trans. Software Eng.*, 20(4), Apr. 1994, 288-307.

24. M. D. Ernst. Static and dynamic analysis: synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, Portland, OR, May 2003, pp. 24-27.

25. J. Esparza, A. Kucera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. *Information and Computation*, 186(2), November 2003, pp. 355–376.

26. R.B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *Proc. 8th European Software Engineering Conference*, ACM Press, New York, NY, 2001, pp. 229–236.

27. R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proc. ACM SIGPLAN 2001 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2001, pp. 1-15.

28. E. Goldberg, E. and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proc. Design, Automation, and Test in Europe Conference and Exposition (DATE)*, IEEE Computer Society Press, 2002, 131-149.

29. O. Grumberg, D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, v.16 n.3, pp.843-871, May 1994.

30. D.E. Harms and B.W. Weide. Copying and swapping: influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5), 1991, pp. 424-435.

31. K. Havelund and T. Pressburger. Model checking Java programs Using Java Pathfinder. *International Journal on Software Tools for Technology Transfer,* 2(4), Springer-Verlag, April 2000.

32. W. D. Heym. *Computer Program Verification: Improvements for Human Reasoning.* Ph.D. Dissertation, Department of Computer and Information Science, The Ohio State University, Columbus, OH, 1995.

33. J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. *The Geneva Convention On The Treatment of Object Aliasing*, http://gee.cs.oswego.edu/dl/aliasing/aliasing.html, 1997.

34. J.E. Hollingsworth, L. Blankenship, and B.W. Weide. Experience report: Using RESOLVE/C++ for commercial software. In *Proc. ACM SIGSOFT 8th Int'l Symposium on the Foundations of Software Engineering*, ACM, Nov. 2000, pp. 11-19.

35. H.Hoos. SAT-encodings, search space structure, and local search performance. *Proc. 16th Intl. Joint Conf. On Artificial Intelligence (IJCAI'99)*, Stockholm, Sweden, Morgan Kaufmann,1999, pp. 296–303.

36. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997, pp.279-295.

37. D. Jackson and M. Vaziri. Finding bugs with a constraint solver. *ACM SIGSOFT Software Engineering Notes*, Sept. 2000, pp. 14-25.

38. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2), April 2002, pp.256-290.

39. J. L. Jensen, M. E. Jorgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, 1997.

40. C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

41. A. Kaiser. *A SAT-based Propositional Prover for Consistency Checking of Automotive Product Data.* Technical Report WSI-2001-16, W.-Schickard Institut für Informatik, Universität Tübingen, Tübingen, Germany, 2001.

42. S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. *Procs. 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, Seattle, WA, 2002.

43. S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Procs. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2003)*, Warsaw, Poland, April 2003.

44. S. Khurshid, D. Marinov, I. Shlyakhter, and D. Jackson. A case for efficient solution enumeration. *Procs. 6th International Conference on Theory and Applications of Satisfiability Testing* (SAT), Portofino, Italy, May 2003.

45. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, vol. 19 (7), July 1976, 385-394.

46. J. Krone, W. F. Ogden, and, M. Sitaraman. *Modular Verification of Performance Constraints.* Technical Report RSRG-03-04, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, May 2003, 25 pages; available at www.cs.clemson.edu/~resolve.

47. G. Kulczycki, M. Sitaraman, W. F. Ogden, and J. E. Hollingsworth, *Component Technology for Pointers: Why and How*, Technical Report RSRG-03-03, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, April 2003, 19 pages; available at http://www.cs.clemson.edu/~resolve.

48. G. Kulczycki, M. Sitaraman, W. F. Ogden, and G. T. Leavens, *Preserving Clean Semantics for Calls with Repeated Arguments*, Technical Report RSRG-04-01, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, April 2003, 35 pages; available at http://www.cs.clemson.edu/~resolve.

49. G. Kulczycki. *Direct Reasoning.* Ph.D. Dissertation, Department of Computer Science, Clemson University, Clemson, SC, May 2004.

50. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, 37(5), 2002, pp. 246-257.

51. D. Marinov and S. Khurshid. TestEra: a novel framework for automated testing of Java programs. *Procs. 16th IEEE Conference on Automated Software Engineering* (ASE), San Diego, CA, 2001.

52. B. Meyer, *Object-oriented Software Construction*, 2nd Edition, Prentice Hall, Upper Saddle River, NJ, 1997.

53. A. Moller and M. I. Schwartzbach, The pointer assertion logic engine. *ACM SIGPLAN Notices*, 36(5), May 2001, pp.221-231.

54. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*. ACM, 2001, 530-535.

55. P. Muller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency, Computation Practice and Experience*, 15, 2003, pp. 117-154.

56. J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 2001.

57. J.W. Nimmer and M.D. Ernst. Invariant inference for static checking: an empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, Charleston, SC, November 2002, pp. 11-20.

58. C. Pasareanu, M.B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, 2001.

59. S. Prestwich. Local search on SAT-encoded coloring problems. *Proc. 6th Intl. Conf. On Theory and Applications of Satisfiability Testing (SAT 2003),* Santa Margherita Ligure, Italy, Springer, 2003, pp. 105–119.

60. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification of fault-tolerant architectures: prolegomena to the design of PVS. *IEEE Trans. Software Engineering*, 21(2), Feb. 1995, 107-125.

61. D. Ramalingam, A. Warshavsky, J. Field , D. Goyal, M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. *ACM SIGPLAN Notices*, 37(5), May 2002.

62. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Tran. on Programming Languages and Systems 24*, 3 (2002), 217-298.

63. C. Sinz, W. Blochinger, and W. Küchlin. PaSAT - parallel SATchecking with lemma exchange: implementation and applications. In H. Kautz und B. Selman, Hrsg., *LICS'2001 Workshop on Theory and Applications of Satisfiability Testing (SAT'2001)*, Electronic Notes in Discrete Math., 9, Elsevier, Boston, MA, June 2001.

64. C. Sinz, T. Lumpp, J. Schneider, and W. Küchlin. Detection of dynamic execution errors in IBM System Automation's rulebased expert system. *Information and Software Technology*, 44(14), November 2002, pp. 857–873.

65. C. Sinz. Verifikation regelbasierter Konfigurationssysteme. Dissertation, Fakultät für Informations- und Kognitionswissenschaften, Universität Tübingen, 2003.

66. M. Sitaraman and B.W. Weide. Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes 19*, 4 (1994), pp. 21-67.

67. M. Sitaraman, B. W. Weide, and W. F. Ogden. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Transactions on Software Engineering,* 23(3), March 1997, pp. 157-170.

68. M. Sitaraman, S. Atkinson, G. Kulczycki, B.W. Weide, T. Long, P. Bucci, S. Pike, W. Heym, and J.E. Hollingsworth. Reasoning about software-component behavior. In *Proceedings of the 6th International Conference on Software Reuse*, LNCS 1844, Springer-Verlag, 2000, pp. 266-283.

69. M. Sitaraman. Compositional performance reasoning. *Procs. Fourth ICSE Workshop on Component-Based Software Engineering: Component-Certification and System Prediction*, Toronto, CA, May 2001.

70. M. Sitaraman, J. Krone, G. Kulczycki, W. F. Ogden, and A. L. N. Reddy. Performance specification of software components. *ACM SIGSOFT Symposium on Software Reuse*, May 2001.

71. M. Sitaraman, T. J. Long, B. W. Weide, J. E. Harner, and L. Wang. A formal approach to component-based software engineering: education and evaluation. In *Procs. of the International Conference on Software Engineering*, IEEE, Toronto, Canada, May 2001, pp. 601-609.

72. M. Sitaraman, D. P. Gandi, W. Küchlin, C. Sinz, and B. W. Weide. *The Humane Bugfinder: Modular Static Analysis Using a SAT Solver*. Technical Report RSRG-03-05, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, May 2003, 18 pages; available at http://www.cs.clemson.edu/~resolve.

73. M. Sitaraman, B. W. Weide, and W. F. Ogden. *Design, Specification, and Analysis of Software Components*. CS 372 Course Notes, Clemson University, Clemson, SC 29634-0974, 2003.

74. R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. PROPEL: an approach supporting property elucidation. *Proceedings of the 24th International Conference on Software Engineering*, May 2002.

75. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995, pp. 121–189.

76. M. Vardi. On the complexity of modular model checking. In *Proc. 10th IEEE Symposium on Logic in Computer Science*, 1995, pp. 101-111.

77. M. Vaziri and D. Jackson. Checking heap-manipulating procedures with a constraint solver. *TACAS'03*, Warsaw, Poland, 2003.

78. J. M. Voas. How assertions can increase test effectiveness. *IEEE Software 14*, 2 (Feb. 1997), pp. 118-122.

79. B.W. Weide and W.D. Heym. Specification and verification with references. In *Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems*, ACM, 2001.

80. B.W.Weide. Component-based systems. In *Encyclopaedia of Software Engineering*, ed. J. J. Marciniak, John Wiley and Sons, 2001.

81. J. M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 29(9), Sep. 1990, pp. 8-24.

SAVCBS 2004
POSTER ABSTRACTS

SPECIFICATION
& VERIFICATION
OF COMPONENT-
BASED SYSTEMS

UML Automatic Verification Tool (TABU)[*]

M. Encarnación Beato
Escuela Universitaria de Informática
Universidad Pontificia de Salamanca
Salamanca, Spain
ebeato@upsa.es

Manuel Barrio-Solórzano
Facultad de Informática
Universidad de Valladolid
Valladolid, Spain
mbarrio@infor.uva.es

Carlos E. Cuesta
Facultad de Informática
Universidad de Valladolid
Valladolid, Spain
cecuesta@infor.uva.es

ABSTRACT

The use of the UML specification language is very widespread due to some of its features. However, the ever more complex systems of today require modeling methods that allow errors to be detected in the initial phases of development. The use of formal methods make such error detection possible but the learning cost is high.

This paper presents a tool which avoids this learning cost, enabling the active behavior of a system expressed in UML to be verified in a completely automatic way by means of formal method techniques. It incorporates an assistant for the verification that acts as a user guide for writing properties so that she/he needs no knowledge of either temporal logic or the form of the specification obtained.

Keywords

Formal methods, automatic verification, UML active behaviour, formal UML verification

1. INTRODUCTION

The Unified Modeling Language (UML) [3, 5] has unquestionable advantages as a visual modeling technique, and this has meant that its applications have multiplied rapidly since its inception. To the characteristics of UML itself must be added numerous tools that exist in the market to help in its use (Rational Rose, Argo UML, Rhapsody ...). However, unfortunately, none of them guarantee specification correctness.

However, it is widely accepted that error detection in the early phases of development substantially reduces cost and development time, as the errors detected are not transmitted to or amplified in later phases. It would thus be very useful to have a tool that would allow the integration of this semi-formal development method with a formal method to enable system verification. This paper presents a tool —TABU (*Tool for the Active Behaviour of UML*)— to carry out this integration by providing a formal framework in which to verify the UML active behaviour.

The tool uses SMV [4] (*Symbolic Model Verifier*) like formal specification, as it has the adequate characteristics for representing the active behaviour of a specification in UML. The main reason for this is that it is based on labeled transition systems and because it allows the user's own defined data types to be used, thus facilitating the definition of variables. It also uses symbolic model checking for the verification, which means that the test is automatic, always obtains an answer and more importantly, should the property not be satisfied generates a means of identifying the originating error.

The tool carries out, with no intervention on the user's part, a complete, automatic transformation of the active behaviour specified in UML into an SMV specification, focusing mainly on reactive systems in which the active behaviour of the classes is represented through state diagrams, while activity diagrams are used to reflect the behaviour of class operations. XMI [6] (*XML Metadata Interchange*) is used as the input format, thus making it independent of the tool used for the system specification.

On the other hand, the tool has a versatile assistant that guides the user in writing properties to be verified using temporal logic. The verification is carried out in such a way that the user needs no knowledge of either formal languages or temporal logic to be able to take advantage of its potential; something which has traditionally been difficult to overcome when deciding on the use of formal methods. In addition, notions of the form of the specification obtained are unnecessary: that is, knowledge of the internal structure of variables or modules obtained is not required for verification. Figure 1 is a graphical representation of the tool's architecture, the engineer only need knowledge of UML and the system studied, the tool obtain automatically the formal representation in SMV from textual representation in XMI. Parallel, a wizard helps to write properties to verified using LTL (*Linear Temporal Logic*), moreover if the property is not satisfied, the tool shows a counterexample trace.

The rest of the paper shows the functionalities of the tool illustrated through a case study. It is analysed in terms of two main aspects of the tool: how to obtain a formal specification from the UML diagrams, and how the assistant helps and guides in verifying properties. This is followed by a review of the work in the same field from the literature and, finally, the conclusions are presented along with possible fu-

[*]Supported by Junta de Castilla y León (Spain) in the Research Project VA117/03

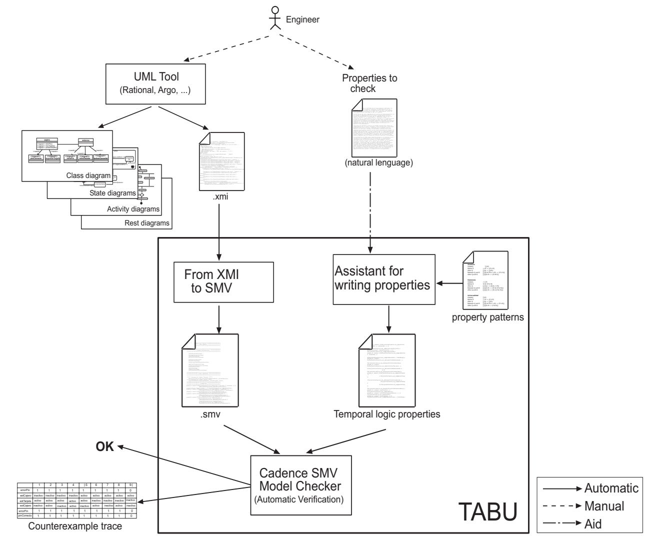


Figure 1: Tool architecture

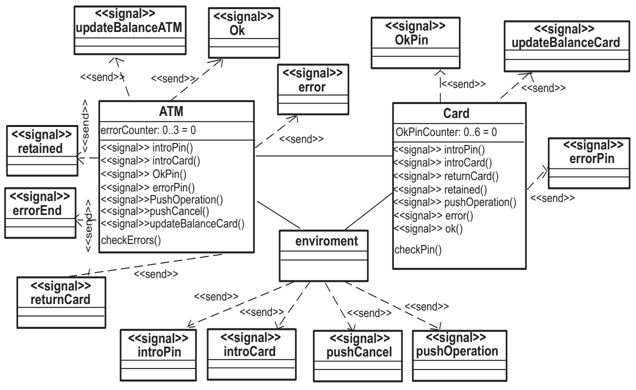


Figure 2: Class diagram

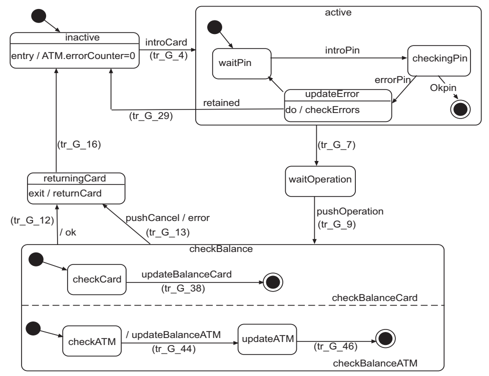


Figure 3: State diagram of the ATM class

ture work.

2. FROM UML TO SMV

The tool input is a UML specification which has been formatted using the XMI exchange syntax. From this input, a SMV specification is automatically generated. Three kinds of diagram are taken into account when transforming the active behaviour from UML into SMV: class, state and activity diagrams. The first provides information concerning the elements that make up the system and their relationships, while the second and third provide information about the behaviour, through time, of each of those elements.

In order to show how the tool works we use the example of an automatic teller machine (figures 2, 3, 4, the diagrams of card class have been omitted by fault of space), both because it is a very well known example, and because it incorporates in its specification most of the existing building blocks of statemachine and activity diagrams.

The following is the basic description of the system. First of all, the user introduces the credit card followed by a pin number. The system checks whether it is correct and, if not it allows the user to try again. If the user introduces three consecutive wrong pin numbers, the card will not be returned to the user. Once the right pin is introduced, the user will be allowed to push the operation button. This operation updates the card information including the available left-over. At any time, the user can push the cancel button that will make the card to be returned and an error signal to be generated.

3. CLASS DIAGRAM

The fundamental concept taken as our starting point is that of the active class. The system is specified in terms of active classes which are associated to the reception of signals. The behaviour of each active class is reflected in a different SMV module, which in turn is instantiated in the main module by each of the class objects.

Each SMV module, representing a class, needs the signals the class receives as its input parameters, and those the class emits as output parameters. Thus, the said signals are reflected in the class diagram using the stereotypes <<send>> and <<signal>> as shown in figure 2.

Here, the signals `okPin`, `errorPin` and `updateBalanceCard` correspond the the signals emitted by the `Card` class, while **`introPin`**, **`introCard`** and **`returnCard`** are the ones it receives.

An additional class called `environment` also has to be included. It has no associated behaviour and contains details of the signals produced outside the system and which are input signals.

4. STATE MACHINES

Behaviour of each of the active objects is reflected through state machine and activity diagrams. To correctly control the evolution of a state machine, the state it is in at any given moment must be known. This is achieved by using a separate variable to store this information for each machine.

In addition, the fact that combined states, both sequential and concurrent, may appear within a machine means that additional variables are needed in order to deal with the submachines. These will be dealt with following the same reasoning as for the main machine, with the exception of the peculiarities they possess with respect to activation and deactivation.

As for the evolution machines, the SMV operator `next` is used. This represents the value taken by the variable in the following step. The state machine is initiated using the `init` operator. As far as the machine for the ATM class is concerned (see Figure 3), the SMV representation of the outermost machine behaviour is as shown below:

```
/***** Statemachine for state: ATM *****/
st_ATM:{checkBalance,waitOperation,active,inactive,returningCard};
/***** Evolution of statemachine for class: ATM*****/
   init(st_ATM) := inactive;
   next(st_ATM) := case {
      tr_G_9 : checkBalance;
      tr_G_7 : waitOperation;
      tr_G_4 : active;
      tr_G_29 | tr_G_16: inactive;
      tr_G_12 | tr_G_13: returningCard;
      default : st_ATM;    };
```

Where `tr_G_9`, `tr_G_7`, `tr_G_4`... represent the firing of transitions `tr_G_9`, `tr_G_7`, `tr_G_4`... The block `default` represents the behaviour where there is no change of state, that is, when no transition present in the machine is fired and it remains in the same state during the following step.

A similar reasoning has been used for the behaviour of the submachines, based on having a different machine for each sequential composite state and for each region of a concurrent composite state. By doing so, the behaviour associated to the concurrent composite state `checkBalance` of Figure 3 is represented in terms of the following machines:

```
/***** Statemachine for state: checkBalanceATM *****/
   st_checkBalanceATM :{updateATM,checkATM,FINAL,DontKnow};
 /***** Statemachine for state: checkBalanceCard *****/
   st_checkBalanceCard :{checkCard,FINAL,DontKnow};
/** Evolution statemachine state: checkBalanceATM ***/
   init(st_checkBalanceATM) := DontKnow;
   next(st_checkBalanceATM) := case {
      tr_G_12 | tr_G_13 : DontKnow;
      tr_G_9 : checkATM;
      tr_G_44 : updateATM;
      tr_G_46 : FINAL;
      default : st_checkBalanceATM;   };
/** Evolution statemachine state: checkBalanceCard ***/
   init(st_checkBalanceCard) := DontKnow;
   next(st_checkBalanceCard) := case {
      tr_G_12 | tr_G_13: DontKnow;
      tr_G_9 : checkCard;
      tr_G_38 : FINAL;
      default : st_checkBalanceCard;   };
```

Where `DontKnow` is the state of a machine which is deactivated. Deactivation can take place either because of the firing of transition `tr_G_13`, the cancel button is pushed, or because both submachines reach the final state and `tr_G_12` is fired by termination. Its syntax is the following:

```
  tr_G_12:=in_checkBalance & in_FINALcheckBalanceATM &
           in_FINALcheckBalanceCard;
```

5. ACTIONS

The evolution of an active object can lead to different actions, including sending signals and modifying the value of class attributes.

With regard to sending signals, it can happen in any of the following situations: (1) the firing of a transition, if the signal is among the transition effects; (2) the activation of a state, if the signal is among its entry actions; and (3) the deactivation of a state, if the signal is among its exit actions. Taking into account that both state activation and deactivation are due to the firing of some transition, signal evolution can be represented in a similar way to state machine evolution.

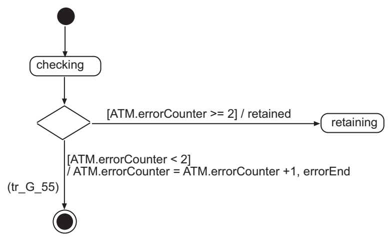

checking
[ATM.errorCounter >= 2] / retained
retaining
[ATM.errorCounter < 2]
/ ATM.errorCounter = ATM.errorCounter +1, errorEnd
(tr_G_55)

Figure 4: Activity diagram for the activity checkErrors

As for modifying the value of an attribute, very much the same philosophy can be followed. This means that it will be specified through the use of the SMV operators `init` and `next`. Attributes will be initialised with `init` if they have an initial value in the class diagram, whereas their evolution (`next`) will depend on the firing of transitions. For instance, the SMV behaviour for the attribute `errorCounter` in class `ATM`, which keeps track of how many wrong consecutive pin numbers have been introduced, is the following (see Figures 2, 3 and 4).

```
/***** Attribute: errorCounter *****/
   ATM_errorCounter: 0..3;
   init(ATM_errorCounter):=0;
   next(ATM_errorCounter) := case {
      tr_G_55: ATM_errorCounter +1;
      tr_G_29 | tr_G_16: 0;
      default : ATM_errorCounter;    };
```

6. VERIFICATION

Having obtained a system specification in a formal language with a solid mathematical basis means that it is possible to check whether the system complies with certain desirable properties. As with the formal specification methods, the increasing complexity of software systems requires the development of new verification methods and tools to carry it out either automatically or semi-automatically.

In our tool, verification is carried out using the SMV tool model checker. With this, it is possible to make the verification process completely automatic. That is, given a property, a positive or negative reply is always obtained.

The property must be expressed in a temporal logic present in SMV, CTL (*Computation Tree Logic*) or LTL (*Linear Temporal Logic*). This property writing is not a trivial problem. To write them correctly, advanced knowledge of logics and the type of specification obtained from the system is necessary. Our tool overcomes this problem as it has an assistant that guides the user through the writing of properties until the property to be verified is finally obtained following the appropriate syntax.

Our starting point was the pattern classification proposed by Dwyer et al [2] to which our own cataloguing of the different properties to be automatically verified has been added.

6.1 Property patterns

The property writing assistant is based on the pattern scheme proposed by Dwyer et al [2] where it is established a first classification between patterns of occurrence and order. Most of the properties of a system to be verified in practice, fit in with one of these two categories.

Occurrence patterns describe properties with respect to the occurrence of a state or signal during the evolution of a system. These include absence (never), universality (always), existence (sometimes) and bounded existence (appearing a certain number of times). Order patterns establish properties with respect to the order in which they occur. They include: precedence (`s` precedes `p`), response (`s` responds to `p`), and combinations of both: chain precedence (`s` and `t` precede `p` or `p` precedes `s` and `t`), chain response (`s` and `t` respond to `p` or `p` responds to `s` and `t`), and constrain chain (`s` and `t` without `z` respond to `p`).

On the other hand, each kind of pattern has a scope of application which indicates the system execution on which it must be verified. There are five basic scopes: Global (the entire program execution), Before R (the execution up to a given property), After Q (the execution after a given property), Between Q and R (any part of the execution from a given property to another given property) and after Q until R (like between but the designated part of the execution continues even if the second property does not occur).

6.2 Property classification

The different properties to be verified have been catalogued to establish limits for the scopes (Q and R) and to specify the order of properties when more than one must be determined (`s`, `t` o `z`), so that the user does not need to know or understand the structure of the specification obtained in SMV to carry out verification The established property types are:

- A state machine is in a particular state.
- An object activity is in a particular state.
- A signal or event is produced.
- Value comparison of an attribute.

The tool will automatically generate the property in the adequate format, in accordance with the chosen option and the selected pattern and scopes. Once we have the properties to be verified, it is possible, using the tool itself, to execute the SMV *model checker* to carry out the verification. If the property is not satisfied, it generates a trace showing a case where it is not verified.

For example, for the automatic teller machine, it would be possible to verify that the card is never retained; this means that the signal `retained` never happens (pattern: absence, scope: global) .

As expected, the result of the checker is false. If the generated counterexample trace is analyzed (see next table), it can be seen that the card is retained when there has been 2 wrong pin numbers and again errorPin is generated.

Step	**62**	**63**	**64**	**|:65**
st_ATM	active	active	active	active
st_active	checkingPin	updateError	updateError	updateError
st_checkErrors	DontKnow	checking	BRANCH	retaining
errorPin	**1**	0	0	0
ATM_errorCounter	2	2	2	2
retained	0	0	0	**1**

7. CONCLUSIONS AND FUTURE LINES OF WORK

This paper presents a tool whose main aim is to integrate formal methods with semi-formal ones in such a way as to be transparent for the user. It verifies the UML active behaviour using SMV. Although this is not a new idea —there are other works that use formal methods to verify the behaviour of UML specifications [7, 1, 8]— as far as we know at the present time, nowhere are activity and state diagrams jointly verified, using the former to represent the behaviour of the class operations.

However, the most innovative characteristic of the tool is that, in spite of using the potential of temporal logic to verify systems, the user need have no knowledge of all the technical intricacies of such logics. Most of the former related works do not verify automatically, except vUML [7] although it doesn't fully exploit the use of temporal logics, implementing a limited verification based on checking that it is impossible to reach error states. These error states are introduced by the user in the diagrams, so the diagrams are more complicated.

It should also be pointed out, though it has not been discussed here through lack of space, that the representative elements of both state and activity diagrams are included in this approach, something that cannot be said of other contributions in this field, in which few of the characteristics provided by UML (history states, deferred events, transitions fired by termination...) are dealt with.

As for future lines of work, some kind of treatment of the traces obtained in the verification when the property is not satisfied would seem to be of great interest. More precisely, that the representation of the traces should be visual instead of written, by using either some of the UML diagrams or an animated representation of the state and activity machines which could help the user to locate the error source very quickly.

8. ADDITIONAL AUTHORS

Additional authors: Pablo de la Fuente (Universidad de Valladolid) email: `pfuente@infor.uva.es`)

9. REFERENCES

[1] A. Darvas, I. Majzik, and B. Beny. Verification of UML Statechart Models of Embedded Systems. In *Proc. 5th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS 2002)*, IEEE Computer Society TTTC, pages 70–77, Abril 2002.

[2] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering*, Mayo 1999.

[3] J. R. G. Booch and I. Jacobson. *The Unified Modeling Language.* Addison-Wesley, 1999.

[4] K. L. McMillan. *Symbolic Model Checking. An approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, Mayo 1992.

[5] OMG. *UML 2.0 Diagram Interchange Final Adopted Specification.* OMG Document pct/03-09-01, 2003.

[6] OMG. *XML Metadata Interchange (XMI) v 2.0.* OMG Document 2003-05-02, 2003.

[7] I. Porres. *Modeling and Analyzing Software Behavior in UML.* PhD thesis, Department of Computer Science, Åbo Akademi University, Noviembre 2001.

[8] W. Shen, K. Compton, and J. Huggins. A Toolset for Supporting UML Static and Dynamic Model Checking. pages 147–152. IEEE Computer Society, 2002.

Integration of Legacy Systems in Software Architecture

Maria Wahid Chowdhury
Department of Computer Science
University of Victoria
PO Box 3055, STN CSC
Victoria, BC, Canada V8W 3P6
Email: mwchow@uvic.ca
Phone no: 250-477-9420

Muhammad Zafar Iqbal
Professor & Head,
Computer Science and Engineering Department,
Shah Jalal University of Science and Technology
Sylhet-3114, Bangladesh
Email: i_am_zafar@yahoo.com
Phone: 880-821-713491(Ext: 154)

ABSTRACT

Most Companies have an environment of disparate legacy systems, applications, processes and data sources. Maintaining legacy systems is one of the difficult challenges that modern enterprises are facing today. The commercial market provides a variety of solutions to this increasingly common problem of legacy system modernization. However, understanding the strengths and weaknesses of each modernization technique is paramount to select the correct solution and the overall success of a modernization effort. This paper examines the strengths and the weaknesses of several modernization techniques in order to help engineers to select the right technique to modernize a legacy system.

Categories and Subject Descriptors
D.3.3 [Management]: Life Cycle

General Terms
Design.

Keywords
Legacy system, reengineering, integration.

1. INTRODUCTION

Software systems are critical assets for companies and incorporate key knowledge acquired over the life of an organization. Companies spend a lot of money on software systems. To get a return on that investment, these software systems must be usable for a number of years. The lifetime of software systems is very variable though many large systems remain in use for many years. Organizations rely on the services provided by these systems and any failure of these services would have a serious effect on the day to day running of business. These old systems have been given the name legacy systems.

Legacy systems incorporate a large number of changes continuously to reflect evolving business practices. Repeated modification has a cumulative effect on system complexity. Usually, a legacy system has to pass through many developers evolving over decades to satisfy new requirements. These systems are matured, heavily used, and constitute massive corporate assets. Today, legacy systems must be designed to be capable of integrating with other applications within the enterprise. However, scrapping legacy systems and replacing them with more modern software involves significant business risk. Replacing a legacy system is a risky business strategy for a number of reasons [4]: a) There is rarely a complete specification of the legacy system. Therefore, there is no straightforward way of specifying a new system, which is functionally identical to the system that is in use, b) Business processes and the ways in which legacy systems operate are often inextricably inter-twined. If the system is replaced, these processes will also have to change, with potentially unpredictable costs and consequences, c) Important business rules may be embedded in the software and may not be documented elsewhere, d) New software development is itself risky because there may be unexpected problems with a new system. In general, a legacy system has following characteristics:

1. High maintenance cost.
2. Complex structure.
3. Obsolete support software.
4. Obsolete hardware.
5. Lack of technical expertise
6. Business critical.
7. Backlog of change requests.
8. Poorly documented.
9. Embedded business knowledge.
10. Poorly understood.

2. ARCHITECTURE DESIGN CONSTRAINTS AND ISSUES

A legacy system significantly resists modification and evolution to meet new and constantly changing business requirements. Legacy systems have not been designed to accommodate changes because of the following reasons:

- The Legacy system was designed for the immediate needs. When constructed, it was not expected that it would be in service for many years.
- There may be some constraints (as for example: low

cost) that were satisfied by the development of legacy system.

Before making any change, it is necessary to assess the feasibility of making changes and to determine the impact of the changes on the rest of the system. Due to the complex structure of legacy systems, they require considerable effort to understand. The challenge in the integration of legacy systems is to understand the functionality, design, operation and performance of the system and to anticipate the types of changes that will be required over the integration steps. After years of maintaining, upgrading and enhancing the legacy system, the user manuals and system design documentation are often out of date, inaccurate, and fail to reflect the current system's capabilities and operations. As a result legacy system architectures are often poorly documented. This emerges as a new kind of problem when integrating a legacy system into an overall system architecture design and specification.

Architectural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among alternatives (an architectural style). [1]. Some of the constraints found in integrating legacy system are on how to deal with components, connectors, semantics or topology.

Perhaps the most obvious component constraint relates to the component types allowed by the architectural style. There are also a number of constraints when one component needs or manages to collaborate with other components. This includes collaborator location and availability, information such as transfer protocol, data format, schema and content (including method signatures and interfaces) as well as architectural assumptions. Furthermore, there may be constraints on the types of access that the components must provide, e.g. interface access to the database, to the application logic, or to internal objects of the component. We must be careful about incompatible data and file formats, hardware incompatibility, software dependency on hardware, proprietary protocols and networks when integrating a legacy system. Other problems posed by legacy systems are the absence of clear interfaces, and insufficient encapsulation.

In summary, the most important issues to consider when integrating a legacy system are:

1. Data: how data is going to be integrated. That is, identify and link records on the same subject or other entity in disparate systems. One solution is to use metadata. However, this leads to one problem; because the same metadata can have different meanings in different applications, companies must develop custom interfaces between applications. Another approach is to perform data integration at the semantic level (based on actual content, not the metadata).
2. Connectivity to each component in the architecture.
3. Routing of messages between components.
4. Validation and transformation of data into and out of each application.
5. Interfacing with each application based on its own syntactical and semantic requirements.
6. There exist some security issues that must be addressed. We must pay attention to the legacy system's security mechanisms.
7. Conformity to organizational and business process structures. The legacy system must be adapted to new business policies.

The above issues make us wonder: What is the format in which data is interchanged? How does the application interpret a message it receives? What is the impact of changing a message definition?

3. ARCHITECTURE DESIGN STRATEGIES

There are basically two approaches to reuse legacy systems: reengineering and integration. Re-engineering means re-structuring. Re-structuring a legacy system's code requires that the system and code are well documented and/or can be automatically analyzed and transformed by an automatic process. Re-engineering a system is slow. Integration is faster and cheaper than re-engineering. To integrate a legacy system, we must define the role of each subsystem, define interfaces for each subsystem, and build an object wrapper for each subsystem.

An integration strategy can be intrusive or non-intrusive. An integration strategy that requires knowledge of the internals of a legacy system is called intrusive (white box) integration, while integration strategy that requires the knowledge of external interfaces of a legacy system is called non-intrusive (black box) integration. A connection to an application system is considered non-intrusive if an existing entry or exit point is used. If application source code is modified, the connection is considered intrusive. Intrusive connections are used when custom coding is developed to handle specific application needs or to increase performance. Non-intrusive connections are recommended for use if the information required from the application is already available from an existing interface and the transaction volume is low to moderate. Intrusive integration requires an initial reverse engineering process to gain an understanding of the internal system operation. After the code is analyzed and understood, intrusive integration often includes some system or code restructuring. There are two major approaches for legacy systems integration: application integration and data integration.

3.1. Application integration

The guiding philosophy behind this approach is that applications contain the business logic of the enterprise, and the solution lies in preserving that business logic by extending the application's interfaces to interoperate with other or sometimes newer applications. There are some major classes of application integration solutions given below:

User Interface Modernization: The user interface (UI) is the most visible part of a system. Modernizing the UI improves usability and is greatly appreciated by final users. A common technique for UI modernization is Screen scraping, as shown in Figure 1, consists of wrapping old, text-based interfaces with new graphical interfaces. [2]

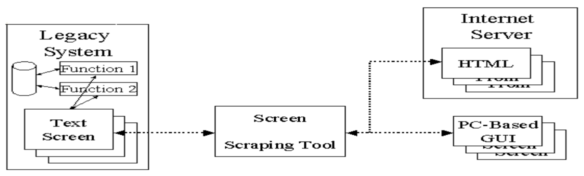


Figure 1. Legacy System Wrapping Using Screen Scrapping

Point-to-point integration: In Point-to-point integration, communication channels are developed between each pair of applications. Such a solution is expensive, because the number of interfaces required grows exponentially. With n applications, n*(n - 1) interfaces may be required since each application may need an interface with other application. The impact of minor changes in communication requirements and that of adding a new application is significant. Maintenance is clearly a problem due to the number of nodes.

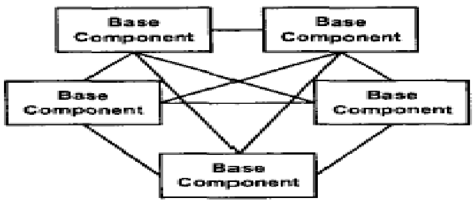


Figure 2 (a). point-to-point integration

Message routers: Point-to-point integration exponentially increases the number of interfaces. This can be reduced to a linear increase through the use of middleware – message-oriented or based on the Common Object Request Broker Architecture (CORBA).

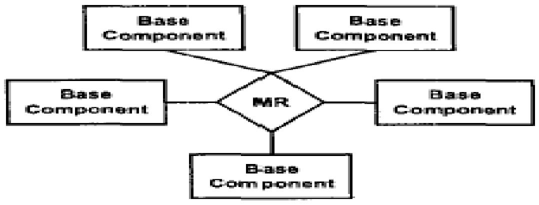


Figure 2(b). message router

The solution requires interfacing each application to the message bus through an adapter. Each application has only one programmatic interface, the message bus. Applications communicate by publishing a message to the bus, which delivers message to those who subscribe. Subscription topics of queues let subscribers receive only messages they are interested in. The Middleware product may also provide value-added services such as guaranteed delivery, certified delivery, transactional messaging, message transformation (using brokers) and so on. [1]

CGI integration: The Common Gateway Interface (CGI) is a standard for interfacing external applications with information servers, such as HTTP or Web servers. Legacy integration using the CGI is often used to provide fast web access to existing assets including mainframes and transaction monitors. [2]

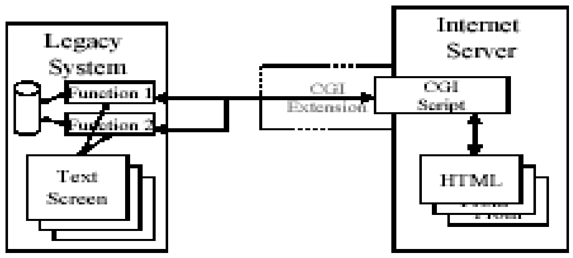


Figure 3. CGI Integration

3.2. Data Integration

The guiding philosophy behind integration of data is that the real currency of the enterprise is its data. The implied business logic in the data and metadata can be easily manipulated directly by applications in the new architecture of the enterprise. Some data integration solutions are described below:

XML Integration: The Extensible Markup Language (XML™) is a broadly adopted format for structured documents and data on the Web. [2]

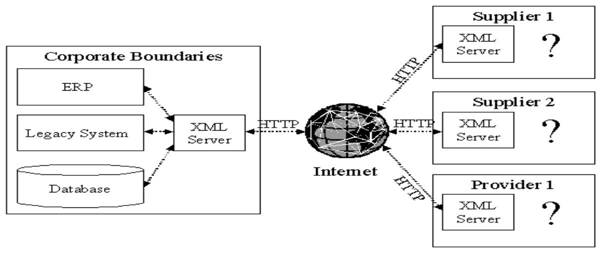


Figure 4. XML Wrapping

XML is a simple and flexible text format derived from standard generalized markup language (SGML) (ISO 8879) and developed by the World Wide Web Consortium® (W3C). XML is expanding from its origin in document processing and becoming a solution for data integration.

Data replication: Database replication is the process of copying and maintaining database objects in multiple databases that make up a distributed database system.

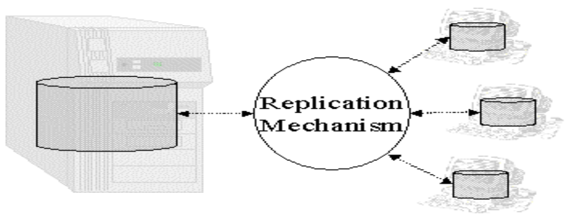


Figure 5. Data Replication

Replication provides users with fast, local access to shared data and greater availability to applications because alternative data access options exist. Even if one site becomes unavailable, users can continue to query, or even update, data at other locations. Database replication is often used to enable decentralized access to legacy data stored in mainframes.

4. EXAMPLE OF GENERIC ARCHITECTURES

Java J2EE Connector architecture: Java J2EE Connector architecture defines a standard set of services that allow developers to quickly connect and integrate their applications with virtually any back-end enterprise information system. These services are supplied as "plug-in" connectors.

Sun ONE: Sun Open Net Environment (Sun ONE) is Sun's standards-based software vision, architecture, platform, and expertise for building and deploying Services on Demand. The network is all about servicing the communities, stockholders, customers, and employees.

OMG MDA: Computing infrastructures are expanding their reach in every dimension. New platforms and applications must interoperate with legacy systems. MDA is a new architectural approach that provides companies with the tools necessary to integrate all the various middleware technologies (such as CORBA, EJB, XML, SOAP and .NET). MDA addresses the complete life cycle of designing, implementing, integrating and managing applications and data using open standards. MDA provides an architecture that assures portability, cross platform interoperability, platform independence, domain specificity, and productivity.

B2B: B2B integration or B2Bi is basically about the secured coordination of information among businesses and their information systems.

EAI: As the need to meet increasing customer and business partner expectations for real-time information continued to rise,

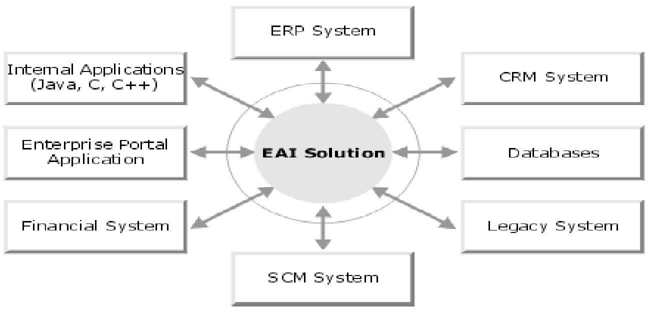


Figure 6. Enterprise Application Integration

companies are forced to link their disparate systems to improve productivity, efficiency, and, ultimately, customer satisfaction. EAI is the process of creating an integrated infrastructure for linking disparate systems, applications, and data sources across the corporate enterprise.

CORBA: CORBA allows applications to communicate with one another no matter where they are located or who has designed them. With CORBA, users gain access to information transparently, without them having to know what software or hardware platform it resides on or where it is located on an enterprises' network. This characteristic makes CORBA an excellent technology to integrate legacy systems.

XML: XML improves the web functionality by providing more flexible and adaptable information identification (tags).

SOAP: The Simple Object Access Protocol (SOAP) is a standard that specifies how two applications can exchange XML documents over HTTP.

Java RMI: Java Remote Method Invocation (RMI) enables the programmer to create distributed Java technology-based applications in which methods of remote java objects can be invoked from other Java virtual machines, possibly on different hosts. Java RMI is well suited to be used in the application level of integration.

JDBC: JDBC technology is an API that lets user access to virtually any tabular data source from the Java programming language. The JDBC API allows developers to take advantage of the Java platform's "Write Once, Run Anywhere^{TM}" capabilities for industrial strength, cross-platform applications that require access to enterprise data.

DCOM: The Distributed Component Object Model (DCOM) is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner.

5. CONCLUSION

There are different approaches to the modernization of legacy assets including reengineering (white-box) and wrapping (black-box). Before starting any legacy modernization effort, every possible option should be considered and business and strategic factors also need to be considered for ensuring long-term success. Present-day systems are the potential source of future legacy problems. To eliminate future legacy problems from present-day systems, systems should be built by using modular engineering and configurable infrastructure.

6. REFERENCES

[1] Architectural Integration Styles for Large-Scale Enterprise Software system, By Jonas Anderson, Pontus Johnson, Department of industrial and Control Systems. Royal Institute of Technology, Sweden.

[2] A Survey of Legacy System Modernization Approaches: http://www.sei.cmu.edu/publications/documents/00.reports/00tn003.html

[3] OMG Model Driven Architecture, http://www.omg.org/mda/

[4] Software Engineering, Sixth Edition, By: Ian Sommervile.

Toward Specification and Composition of BoxScript Components

H. Conrad Cunningham
Computer & Information Science
University of Mississippi
(662) 915-5358
cunningham@cs.olemiss.edu

Yi Liu
Computer & Information Science
University of Mississippi
(662) 915-7602
liuyi@cs.olemiss.edu

Pallavi Tadepalli
Computer & Information Science
University of Mississippi
(662) 915-7602
pallavi@cs.olemiss.edu

ABSTRACT

BoxScript is a Java-based, component-oriented programming language whose design seeks to address the needs of teachers and students for a clean, simple language. This paper briefly describes BoxScript and presents the authors' preliminary ideas on specification of components and their compositions.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *class invariants*, *formal methods, programming by contract.*

General Terms

Design, Languages, Verification.

Keywords

Component, composition, specification, BoxScript.

1. INTRODUCTION

The goal of component-oriented programming is to enable a software system to be built quickly and reliably by assembling separately developed software components to form the system. The system should be flexible enough to be readily adapted to changing requirements by replacing, adding, or removing components. The concepts and languages that support this approach should be taught to students in computing science and software engineering programs.

In 2002 the first author taught an advanced software engineering class focused on Component Software in which the second and third authors were students [4]. The class used an approach to design similar to the "UML Components" approach of Cheesman and Daniels [2]. For the programming projects, the class used the Enterprise JavaBeans (EJB) component model and technology. EJB is a component model for building server-side, enterprise-class applications [11]. The complexity of the EJB technology meant it was not ideal for use in an academic course. The technology got in the way of teaching the students how to "think in components" cleanly. The students had to map their designs into the EJB technology [2, 7] and struggle to master enough of the technology to complete their term projects.

As a result, the second author undertook the design of a simple, component-oriented language with features that support its use in teaching. This language, called BoxScript, is described in section 2. Section 3 discusses the authors' preliminary ideas on how to specify BoxScript components and their composition formally and section 4 summarizes and identifies areas for further work.

2. BOXSCRIPT

BoxScript is a Java-based, component-oriented programming language whose design seeks to address the needs of teachers and students for a clean, simple language. The component concept is shown in Figure 1 [5].

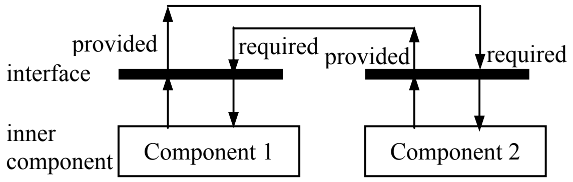


Figure 1. Components and Their Interconnections

A component is called a *box*. A box is a strongly encapsulated module that hides its internal details while only exposing its interfaces. There are two types of interfaces. A *provided interface* describes the operations that a box implements and that other boxes may use. A *required interface* describes the operations that the box requires and that must be implemented by another box. A BoxScript interface is represented syntactically by a Java interface, that is, by a set of related operation signatures. Each occurrence of an interface in a box has an *interface handle*, which identifies that occurrence uniquely within the box, and a *type*, which is the Java interface type. Each box has a corresponding box description (`.box`) file that gives the needed declarations.

An *abstract box* is a box that describes the provided and required interfaces but does not implement the provided interfaces. An abstract box should be implemented by concrete boxes, i.e., atomic or compound boxes. Figure 2a shows an abstract box description `DateAbs`. Its provided interface `DayCal` calculates the day of the week for a date. Figure 2b shows another abstract box description `CalendarAbs`, which has one provided and one required interface. Its provided interface `Display` takes the time range and displays the calendar accordingly.

```
abstract box DateAbs
{  provided interface DayCal Dc;
     //Dc is handle of interface DayCal
}
```

Figure 2a. DateAbs.box

```
abstract box CalendarAbs
{  provided interface Display Dis;
   required interface DayCal  DayC;
}
```

Figure 2b. CalendarAbs.box

An *atomic box* is the basic element in BoxScript. It does not contain any other boxes. It must supply an implementation for each provided interface, that is, a Java class that `implements` the interface. The description of an atomic box gives the box name and, if appropriate, the name of the abstract box it implements. It also gives its provided and required interfaces by listing their interface types and handles. Figure 3a and 3b show atomic box descriptions `Date` and `Calendar`. `Date` implements `DateAbs` and `Calendar` implements `CalendarAbs`.

```
box Date implements DateAbs
{  provided interface DayCal Dc; }
```

Figure 3a. Date.box

```
box Calendar implements CalendarAbs
{ provided interface Display Dis;
  required interface DayCal  DayC;
}
```

Figure 3b. Calendar.box

A *compound box* is a box composed from other boxes. It does not implement its provided interfaces, but uses the implementations provided by its constituent boxes. Each constituent box is given an identifier, called its *box handle*, to enable it to be uniquely identified as a participant within the composition. The box description for a compound box not only supplies the information given in the atomic box, but also specifies (1) the boxes from which this compound box is composed, (2) the sources of its provided and required interfaces, and (3) the connection information that describes how the constituent box interfaces are "wired" together. To provide flexibility, a compound box can be declared to be `composed from` either concrete or abstract boxes. `BuildCalendar` (in Figures 4a and 4b) is composed from abstract boxes `DateAbs` and `CalendarAbs`. When we configure `BuildCalendar`, we substitute concrete boxes such as `Date` and `Calendar` for the corresponding abstract boxes.

```
abstract box BuildCalendarAbs
{  provided interface Display D;}
```

Figure 4a. BuildCalendarAbs.box

```
box BuildCalendar implements
                    BuildCalendarAbs
 {  composed from DateAbs boxD,
          CalendarAbs boxC;
  // boxD is box handle for DateAbs
  // boxC is box handle for CalendarAbs
   provided interface
       Display D from boxC.Dis;
   connect boxC.DayC to boxD.Dc;
 }
```

Figure 4b. BuildCalendar.box

BoxScript uses the box handles to expose and connect the interfaces of the constituent boxes. The `composed from` declaration in `BuildCalendar` assigns `boxD` and `boxC` as the box handles for `DateAbs` and `CalendarAbs`, respectively. The `provided interface` and `required interface` declarations give the types of the interfaces, their interface handles, and their sources. The source is a box handle and interface handle associated with a constituent box. In `BuildCalendar`, interface handle `D` identifies an interface of type `Display` that is mapped to interface handle `Dis` of the box with box handle `boxC` (i.e., `CalendarAbs`). The `connect` statement connects a required interface of one box to a provided interface of another. In `BuildCalendar`, the required interface with handle `DayC` of the box with box handle `boxC` (i.e., `boxC.DayC`) is connected to the provided interface with handle `Dc` of the box with box handle `boxD` (i.e. `boxD.Dc`).

The composition of boxes into a compound box hides all provided interfaces that are not explicitly exposed and must expose every required interface that is not wired to a provided interface of a box within the composition. In the example, provided interface `Display` is exposed. Figures 5a and 5b illustrate the composition process.

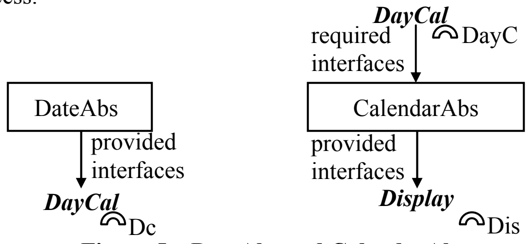


Figure 5a. DateAbs and CalendarAbs

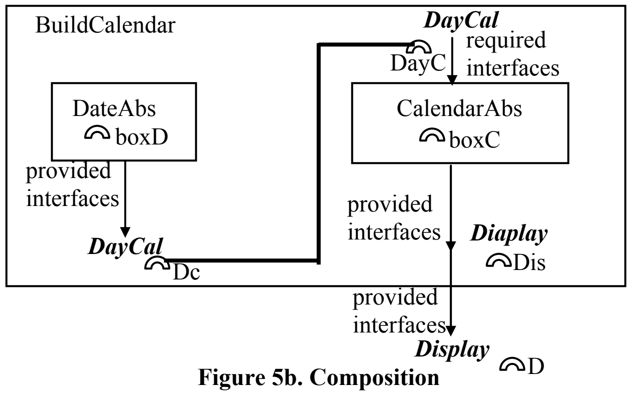


Figure 5b. Composition

Atomic and compound boxes may either be standalone or implementations of abstract boxes. All the implementations of an abstract box are *variants* of the abstract box. The intention is that one variant can be safely substituted for another. When one box substitutes for another, the substitute must *satisfy* the specification of the original box. A variant's provided interfaces should supply at least the operations of the abstract box and the variant's required operations should be at most those of the abstract box.

3. SPECIFICATION

In BoxScript, as in the Cheesman-Daniels approach [2], one basic unit for specifying functionality is the interface. An interface is a set of operation signatures (name, parameter types and order, and return value types) that are related. BoxScript uses Java interfaces for its interface types.

In the Cheesman-Daniels approach, the semantics of an interface is specified in terms of an *interface information model* [2], which is expressed graphically as a UML type (class) diagram augmented by Object Constraint Language (OCL) [12] invariants. For BoxScript, we simplify the presentation and consider the information model to consist of a pair `(V,I)`, where `V` is a set of abstract variables representing the *abstract state* of the component instance associated with the interface and `I` is an *invariant* representing the valid values of the abstract state.

An *invariant* is an assertion that must be kept true in all states of a box that are visible to its clients [6]. We attach invariants to an interface to specify the unchanging properties of the objects that implement the interface. In the model, symbol `I` denotes the conjunction of all the invariants attached to an interface.

We specify the semantics of an individual operation using *precondition* and *postcondition* assertions. A precondition expresses the requirements that any call of the operation must satisfy. That is, it gives valid values of the operation arguments and the interface's abstract state from which the operation can be safely called. A postcondition expresses properties that are ensured in return by the execution of the call. It gives the results of the operation in terms of the arguments and abstract state. We require any operation that is called with the precondition true to terminate eventually with the postcondition true.

To provide precise specification about the relationships of operations calls to each other, we can include *history sequences* [3], which record the sequence of operation calls. This allows assertions about the sequences to appear in the invariants, preconditions, and postconditions.

A box interface `x` *extends* box interface `y` (syntactically) if and only if type(`x`) = type(`y`) or type(`y`) extends type(`x`) in the Java type system. That is, all the operation signatures in `y` also appear in `x`, but `x` may have additional operations. Type extension does not allow either covariant or contravariant changes to operations.

Box interface `x` satisfies interface `y` when `x` provides at least the operations required by `y` and the operations of `x` have an equivalent meaning to the matching operations in `y`. More formally, box interface `x` *satisfies* box interface `y` if and only if:

- `x` *extends* `y`
- `I(x) & C(x,y) ⇒ I(y)`

-
  ```
  (∀m : m ∈ y :
     (pre(y,m) & C(x,y)& I(y) ⇒ pre(x,m))
   & (post(x,m)& C(x,y)& I(x) ⇒ post(y,m)))
  ```

Above, `I(x)` refers to the invariant for `x` and `pre(x,m)` and `post(x,m)` refer to the precondition and postcondition, respectively, for operation `m` on interface `x`. Assertion `C(x,y)` is a coupling invariant that relates the equivalent aspects of the interface information models for `x` and `y`.

The above definition of *satisfaction* is motivated by Meyer's treatment of inheritance in the design by contract approach (and the Eiffel language) [8] and the concept of a coupling invariant in program and data refinement [9].

The second basic unit of specification is the box. A box is a program module that encapsulates some functionality behind its provided interfaces. A client of the box may call an operation on a provided interface. To carry out this operation, a box may invoke operations on its required interfaces, each of which is connected to a provided interface of some box. The specification for a provided interface must be satisfied by the implementation of the box; the specification for a required interface must be satisfied by a provided interface of some box.

A *box's information model* is formed by joining the information models of its provided interfaces. It may have new abstract state variables and a *box invariant* that defines the validity of the box's state. For a box `B`, let `I(B)` be its box invariant, `C(B)` be the coupling invariant that ties it to the interface information models, and `prov(B)` be the provided interfaces. For any box `B`, it must be the case that:

```
(∀p: p ∈ prov(B): I(p)) & C(B) ⇒ I(B)
```

An *atomic box* must supply implementations for its provided interfaces as a cluster of Java classes. The implementations of the interfaces within an atomic box may interact directly with each other and share internal state. A provided interface thus must preserve the invariants of all the box's provided interfaces [10]. A convenient way to achieve this is for all of the provided interfaces of an atomic box to have the same information model `(V,I)`.

A *compound box* composes one or more other boxes to form the "larger" box. As is the case with any box, a compound box has a specification as described above. It has a box information model (i.e., abstract state and box invariant) and interface specifications for the provided and required interfaces. The box invariant ties together the information models of the provided interfaces to form the information model for the compound box.

As with the atomic box, a compound box must provide implementations for its provided interfaces and it may use its required interfaces in doing so. However, unlike the atomic box, the compound box defers the implementation of a provided interface to one of its constituent boxes. The interface handle in the compound box is either the same as in the constituent box or it may be an *alias* that is linked to an interface of the constituent box. Similarly, a required interface of the compound box may be a required interface of one or more constituent boxes. A constituent box may have provided interfaces that are not exposed by the compound box. However, a required interface of a constituent box must either be exposed outside the compound box or be satisfied by some provided interface within the compound box. Thus the box invariant for a compound box must relate the properties expected for its interfaces to the related properties of the corresponding interfaces of constituent boxes.

More formally, for any compound box `B`, the following must hold:

- `(∀p: p ∈ prov(B): I(p)) & C(B) ⇒ I(B)`

-
  ```
  (∀p: p ∈ prov(B):
       (∃D,q: D ∈ const(B) & q ∈ prov(D)
           & q = alias(B,p): q satisfies p))
  ```

-
  ```
  (∀D,r: D ∈ const(B) & r ∈ req(D):
       (∃s: s ∈ req(B) & r = alias(B,s):
           s satisfies r) OR
       (∃E,q: E ∈ const(B) & q ∈ prov(E)
           & connected(B,r,q): q satisfies r))
  ```

In the above, `const(B)` denotes the set of boxes that are composed to form compound box `B`, `alias(B,q)` is the function that maps an interface `q` of compound box `B` to an interface in a constituent box, `req(B)` is the set of required interfaces of box `B` and `connected(B,r,p)` is an assertion that required interface `r` is connected to provided interface `p`. This information is available from the box description. The box invariant may be used in showing that one interface within the box satisfies another.

Consider a valid relationship between a concrete box `B` and an abstract box `A` that it *implements*. Clearly, if abstract box `A` specifies the presence of a provided interface `p`, then concrete box `B` *must* have a provided interface that satisfies `p`. If concrete box `B` has a required interface `r`, then abstract box `A` *must* specify a required interface that satisfies `r`. In terms of operations, the provided interfaces of `B` should supply at least the operations of `A`, and the required operations of `B` should be at most those of `A`. A similar situation occurs if we consider an abstract box that *extends* another abstract box.

More formally, box `B` *satisfies* box `A` if and only if:

- `I(B) & C(A,B)` $\Rightarrow$ `I(A)`
- `(∀p: p ∈ prov(A): (∃q: q ∈ prov(B): handle(q) = handle(p) & q` *satisfies* `p))`
- `(∀r: r ∈ req(B):(∃s: s ∈ req(A): handle(r) = handle(s) & s` *satisfies* `r))`

Above, `C(A,B)` denotes a coupling invariant for the refinement of the information model when replacing `A` by `B`. In particular, `C(A,B)` serves as the coupling invariant for showing that the interfaces of `B` have the needed satisfaction relationship with the corresponding interfaces of `A`. The notation `handle(p)` refers to the interface handle of interface `p`.

A compound box may be composed of abstract boxes. At runtime, an instance of a variant of the abstract box is configured into the instance of the compound box. As noted above, the variant must satisfy the specification for the abstract box it implements. That is, the variant is the same as the abstract box from the perspective of its specification. Thus the box invariant of the compound box can transparently address the different variants.

4. CONCLUSION

BoxScript is a Java-based, component-oriented programming language that is under development by the authors. Its design seeks to address the needs of teachers and students by providing a simple and clean language, yet one that can be used to solve practical problems. It introduces a notation for components and their composition but uses the Java language (which is familiar to most students) to express the internal details of components.

This paper briefly describes the concepts of BoxScript and presents the authors' preliminary ideas on formal specification of BoxScript components and their compositions. Although formal specification and verification were not design goals for BoxScript, its relatively simple design, which is based on strongly encapsulated modular units, seems to be amenable to the application of formal techniques. The ideas outlined in this paper do seem promising, but considerable work is needed to elaborate the formalism and experiment with the pragmatics of the approach. In particular, several examples need to be worked out to demonstrate the concepts and techniques. It will also be helpful to adapt the BoxScript approach to enable use of techniques and tools such as those associated with the Java Modeling Language (JML) [6].

The approach sketched in this paper is likely insufficient to capture the full semantics of calls to the required interfaces, in particular, calls that may lead to reentrance into the calling box (e.g., call-backs). The greybox approach [1] or a similar technique may be needed to enable verification of compound boxes.

This paper approaches specification of program semantics in a manner that is language-oriented, that is, somewhat bottom-up and compositional. The ideas should also be addressed from a software engineering perspective, seeking techniques that can be applied effectively in a more top-down, decompositional manner.

5. ACKNOWLEDGMENTS

This work was supported, in part, by a grant from Acxiom Corporation titled "The Acxiom Laboratory for Software Architecture and Component Engineering (ALSACE)." The design and implementation of BoxScript is part of a Liu's dissertation project under the direction of Cunningham. We thank the reviewers for their many helpful comments on this paper.

6. REFERENCES

[1] M. Büchi and W. Weck. *The Greybox Approach: When Blackbox Specifications Hide Too Much*, Technical Report No. 297a, Turku Centre for Computer Science, Finland, August 1999.

[2] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software***,** Addison Wesley, 2001.

[3] H. C. Cunningham and Y. Cai. "Specification and Refinement of a Message Router," In *Proceedings of the Seventh International Workshop on Software Specification and Design*, IEEE, December 1993.

[4] H. C. Cunningham, Y. Liu, P. Tadepalli, and M. Fu. "Component Software: A New Software Engineering Course," *Journal of Computing Sciences in Colleges*, Vol. 18, No. 6, pp. 10-21, June 2003.

[5] W. Fleisch. "Applying Use Cases for the Requirements Validation of Component-Based Real Time Software," In *Proceedings of the Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing,* p. 75, IEEE, 1999.

[6] G. T. Leavens and Y. Cheon. "Design by Contract with JML," draft paper, Iowa State University, August 2004.

[7] Y. Liu and H. C. Cunningham. "Mapping Component Specifications to Enterprise JavaBeans Implementations," In *Proceedings of the ACM Southeast Conference*, pp. 177-181, April 2004.

[8] B. Meyer. *Object-Oriented Software Construction*, Second Edition, Prentice Hall, 1997.

[9] C. Morgan. *Programming from Specifications*, Prentice Hall International, 1994.

[10] P. Müller. *Modular Specification and Verification of Object-Oriented Programs,* Lecture Notes in Computer Science 2262, Springer-Verlag, 2002.

[11] I. Singh, B. Stearns, M. Johnson, and the Enterprise Team. *Designing Enterprise Applications with the J2EE^{TM} Platform*, Second Edition. Addison Wesley, 2002.

[12] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.

Hierarchical Presynthesized Components for Automatic Addition of Fault-Tolerance: A Case Study*

[Extended Abstract]

Ali Ebnenasir
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA
ebnenasi@cse.msu.edu

Sandeep S. Kulkarni
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA
sandeep@cse.msu.edu

ABSTRACT

We present a case study for automatic addition of fault-tolerance to distributed programs using presynthesized distributed components. Specifically, we extend the scope of automatic addition of fault-tolerance using presynthesized components for the case where we automatically add *hierarchical* components to fault-intolerant programs. Towards this end, we present an automatically generated diffusing computation program that provides nonmasking fault-tolerance. Since presynthesized components provide reuse in the synthesis of fault-tolerant distributed programs, we expect that our method will pave the way for automatic addition of fault-tolerance to large-scale programs.

Keywords

Fault-tolerance, Automatic addition of fault-tolerance, Formal methods, Program synthesis, Distributed programs

1. INTRODUCTION

In this paper, we present a case study for automatic addition of presynthesized fault-tolerance components to distributed programs using a software framework called Fault-Tolerance Synthesizer (FTSyn) [5]. Specifically, we use FTSyn to add distributed components with *hierarchical* topology to a diffusing computation program to provide recovery in the presence of faults. Presynthesized fault-tolerance components provide *reuse* in the synthesis of fault-tolerant distributed programs from their fault-intolerant version. Such reuse is particularly beneficial in dealing with the exponential complexity of synthesis [7]. Also, fault-tolerance components provide an abstraction that simplifies the reasoning about the fault-tolerance and functional concerns.

The FTSyn framework incorporates the results of [9] where the synthesis algorithm automatically specifies and adds presynthesized fault-tolerance components, namely *detectors* and *correctors*, to fault-intolerant programs during the synthesis of their fault-tolerant version. It is shown in the literature [6] that such components are necessary and sufficient for the *manual* design of fault-tolerant programs. As a result, we expect to benefit from their generality in *automatic* addition of fault-tolerance as well.

In [9], the synthesis algorithm is applied to programs where the underlying communication topology between processes is linear. In this paper, we show how we add *hierarchical* presynthesized components to distributed programs. Specifically, we add tree-like structured components to a diffusing computation program where processes are arranged in an out-tree, where the indegree of each node is at most one.

This case study shows that the synthesis method presented in [9] handles presynthesized components (respectively, distributed programs) with different topologies. Also, we extend the scope of synthesis for the case where we simultaneously add multiple presynthesized components to the program being synthesized. Moreover, the use of presynthesized components provides a theoretical foundation for automated development of component-based systems where we reason about the correctness of each individual component and the composition of components.

The organization of the paper. In Section 2, we present preliminary concepts. In Section 3, we describe how we formally represent a hierarchical fault-tolerance component. Subsequently, in Section 4, we show how we automatically add a hierarchical component to a diffusing computation program. Finally, we make concluding remarks and discuss future work in Section 5.

2. PRELIMINARIES

In this section, first, we present basic concepts in Subsection 2.1. Then, in Subsection 2.2, we represent the formal

*This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

problem statement of adding fault-tolerance components to programs (adapted from [9]). In Subsection 2.3, we give an informal overview of the synthesis method presented in [9].

2.1 Basic Concepts

We specify programs in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [1]. The definition of faults and fault-tolerance is adapted from Arora and Gouda [2] and Kulkarni and Arora [7]. The issues of modeling distributed programs is adapted from [7, 4].

Program. A program p is specified by a finite set of variables, say $V = \{v_0, v_2, .., v_q\}$, and a finite set of processes, say $P = \{P_0, \cdots, P_n\}$, where q and n are positive integers. Each variable v_i is associated with a finite domain of values D_i $(1 \le i \le q)$. A state of p is of the form: $\langle l_0, l_2, .., l_q \rangle$ where $\forall i : 0 \le i \le q : l_i \in D_i$. The state space of p, S_p, is the set of all possible states of p.

A process, say P_j $(0 \le j \le n)$, in p is associated with a set of program variables, say r_j, that P_j can read and a set of variables, say w_j, that P_j can write. Also, process P_j consists of a set of transitions of the form (s_0, s_1) where $s_0, s_1 \in S_p$.

A state predicate of p is any subset of S_p. A state predicate S is closed in the program p iff (if and only if) $\forall s_0, s_1 : (s_0, s_1) \in p : (s_0 \in S \Rightarrow s_1 \in S)$. A sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of p iff the following two conditions are satisfied: (1) $\forall j : j > 0 : (s_{j-1}, s_j) \in p$, and (2) if $\langle s_0, s_1, ... \rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in p$. A finite sequence of states, $\langle s_0, s_1, ..., s_n \rangle$, is a computation prefix of p iff $\forall j : 0 < j \le n : (s_{j-1}, s_j) \in p$; i.e., a computation prefix need not be maximal. The projection of program p on state predicate S, denoted as $p|S$, consists of transitions $\{(s_0, s_1) : (s_0, s_1) \in p \wedge s_0, s_1 \in S\}$.

Distribution issues. We model distribution by identifying how read/write restrictions on a process affect its transitions. A process P_j cannot include transitions that write a variable x, where $x \notin w_j$. Given a single transition (s_0, s_1), it appears that all the variables must be read to execute that transition. For this reason, read restrictions require us to *group* transitions and ensure that the entire group is included or the entire group is excluded. For example, in a program with two Boolean variables a and b and a process P_r that cannot read b, the transition from the state $\langle a = 0, b = 0 \rangle$ to $\langle a = 1, b = 0 \rangle$ can be included iff the transition from $\langle a = 0, b = 1 \rangle$ to $\langle a = 1, b = 1 \rangle$ is also included. The grouping of these two transitions makes the value of b irrelevant for P_r.

Specification. A specification is a set of infinite sequences of states that is suffix-closed and fusion-closed. Suffix closure of the set means that if a state sequence σ is in that set then so are all the suffixes of σ. Fusion closure of the set means that if state sequences $\langle \alpha, s, \gamma \rangle$ and $\langle \beta, s, \delta \rangle$ are in that set then so are the state sequences $\langle \alpha, s, \delta \rangle$ and $\langle \beta, s, \gamma \rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and s is a program state.

Following Alpern and Schneider [1], we let the specification consist of a safety specification and a liveness specification. For a suffix-closed and fusion-closed specification, the safety specification can be specified as a set of bad transitions [6] that a program is not allowed to execute, that is, for program p, its safety specification is a subset of $S_p \times S_p$.

Given a program p, a state predicate S, and a specification *spec*, we say that p satisfies *spec* from S iff (1) S is closed in p, and (2) every computation of p that starts in a state in S is in *spec*. If p satisfies *spec* from S and $S \neq \{\}$, we say that S is an invariant of p for spec.

We do not explicitly specify the liveness specification in our algorithm; the liveness requirements for the synthesis is that the fault-tolerant program eventually recovers to its invariant from where it satisfies its specification.

Faults. A class of faults f for a program p with state space S_p, is a subset of the set $S_p \times S_p$. A sequence of states, $\sigma = \langle s_0, s_1, ... \rangle$, is a computation of p in the presence of f (denoted $p[]f$) iff the following three conditions are satisfied: (1) every transition $t \in \sigma$ is a fault or program transition; (2) if σ is finite and terminates in s_l then there exists no program transition originating at s_l, and (3) the number of fault occurrences (i.e., transitions) in σ is finite.

We say that a state predicate T is an f-span (read as fault-span) of p from S iff the following two conditions are satisfied: (1) $S \Rightarrow T$ and (2) T is closed in $p[]f$.

Nonmasking fault-tolerance. Given a program p, its invariant, S, its specification, *spec*, and a class of faults, f, we say p is nonmasking f-tolerant for *spec* from S iff the following two conditions hold: (i) p satisfies *spec* from S; (ii) there exists a state predicate T such that T is an f-span of p from S, and every computation of $p[]f$ that starts from a state in T has a state in S.

2.2 Problem Statement

In this subsection, we adapt the problem statement presented in [9] where the authors add presynthesized fault-tolerance components to a program p, with state space S_p, invariant $S \subseteq S_p$, specification *spec*, and faults f, in order to synthesize a fault-tolerant program p' with the new invariant S' in the new state space $S_{p'}$. Since each component has its own set of variables, we expand the state space of p to $S_{p'}$ by adding a fault-tolerance component to it.

To create a projection from the states and the transitions of p' to the states and the transitions of p, we define an onto function H: $S_{p'} \rightarrow S_p$, which can be applied on the domain of states, state predicates, transitions, and groups of transitions.

Now, since we require p' not to include new behaviors in the absence of faults, the invariant S' cannot contain states s'_0 whose image $H(s'_0)$ is not in S. Otherwise, in the absence of faults, p' will include computations in the new state space $S_{p'}$ that do not have corresponding computations in p. Hence, we have $H(S') \subseteq S$. Likewise, we require p' not to contain a transition (s'_0, s'_1) in $p'|S'$ that does not have a corresponding transition (s_0, s_1) in $p|H(S')$ (where $H(s'_0) = s_0$ and $H(s'_1) = s_1$). Otherwise, p' may create a new way for satisfying *spec* in the absence of faults. Therefore, the problem of adding fault-tolerance components to programs is as follows:

The Addition Problem.
Given p, S, *spec*, f, with state space S_p such that
p satisfies *spec* from S,
  $S_{p'}$ is the new state space due to adding fault-tolerance components to p,
  $H : S_{p'} \rightarrow S_p$ is an onto function,
Identify p' and $S' \subseteq S_{p'}$ such that
  $H(S') \subseteq S$,
  $H(p'|S') \subseteq p|H(S')$, and
  p' is nonmasking f-tolerant for *spec* from S'. □

2.3 The Synthesis Method

In this subsection, we present an informal overview of the synthesis method presented in [9]. We note that the presentation of this subsection suffices for this paper, however, the interested reader may refer to [9] for a formal presentation.

To deal with the exponential complexity [7] of synthesizing distributed programs, the synthesis algorithm presented in [9] provides a hybrid approach where it uses heuristics (developed in [8]) along with presynthesized fault-tolerance components. Specifically, the algorithm of [9] first uses heuristics under distribution restrictions to add recovery from a specific deadlock state s_d. If the heuristics fail then the synthesis algorithm adds presynthesized correctors to resolve the deadlock state s_d (cf. Section 3 for a formal definition of detectors/correctors). To add a presynthesized component (i.e., detectors/correctors), the synthesis algorithm automatically (i) specifies the required component; (ii) extracts the necessary component from an existing component library; (iii) ensures that the components do not *interfere* with the program execution, i.e., the program and the presynthesized components satisfy their specifications in the presence of each other, and (iv) adds the components.

To automatically specify and add the required components during the synthesis of a distributed program p with n processes $\{P_1, \cdots, P_n\}$, the synthesis algorithm of [9] introduces a high atomicity processes P_{high_i} corresponding to each P_i $(1 \leq i \leq n)$. Each P_{high_i} is allowed to read all program variables and has the write abilities of P_i. At the outset of the synthesis, process P_{high_i} has no actions to execute, where an *action* atomically updates program variables when a specific condition holds. For a specific deadlock state s_d, the synthesis algorithm determines whether there exists a high atomicity process P_{high_i} that can add recovery from s_d, given its high atomicity abilities. Since high atomicity processes have read access to all program variables, they may add recovery actions whose guards are global state predicates; i.e., *high atomicity actions*.

If P_{high_k}, for some $1 \leq k \leq n$, succeeds in adding high atomicity recovery from s_d then the synthesis algorithm automatically specifies and extracts the desired detectors for the refinement of the added high atomicity recovery actions. If the presynthesized detectors do not *interfere* with program execution then the refinement will be successful. Otherwise, the synthesis algorithm of [9] fails to add recovery to s_d.

3. SPECIFYING HIERARCHICAL COMPONENTS

In this section, we describe the specification and the representation of hierarchical fault-tolerance components (i.e., detectors and correctors). Specifically, we concentrate on detectors and we consider a special subclass of correctors where a corrector consists of a detector and a write action on the local variables of a process. We have adapted the specification of detectors from [6].

Specification. Let X and Z be state predicates. Let 'Z detects X' be the problem specification. Then, 'Z detects X' stipulates that

- (*Safety*) When Z holds, X must hold as well.
- (*Liveness*) When the predicate X holds and continuously remains *true*, Z will eventually hold and continuously remain *true*. $\square$

We represent the safety specification $spec_d$ of a detector as a set of transitions that a detector is not allowed to execute.

$$spec_d = \{(s_0, s_1) : (Z(s_1) \ \wedge \ \neg X(s_1))\}$$

The Representation of Hierarchical Detectors. We focus on the representation of a detector with a tree-like structure as a special case of hierarchical detectors. The hierarchical detector d consists of n elements d_i $(0 \leq i < n)$, its safety specification $spec_d$, its variables, and its invariant U. The element d_0 is placed at the root of the tree and other elements of the detector are placed in other nodes of the tree. Let $i \preceq j$ denote the parenting relation between nodes d_i and d_j, where d_i is the parent of d_j. Each node d_i has its own detection predicate X_i and witness predicate Z_i represented by a Boolean variable y_i. The siblings of a node can detect their detection predicate in parallel. However, the truth-value of the detection predicate of each node depends on the truth-value of its children. In other words, node d_i can witness if all its children have already witnessed. The element d_i can read/write the y values of its children and its parent $(0 \leq i < n)$. Moreover, each element d_i is allowed to read the variables that P_i can read. We present the *template* action of the detector d_i as follows $((0 \leq i, j, k < n) \wedge (\forall r : j \leq r \leq k : i \preceq r))$:

$$DA_i : \ (LC_i) \ \wedge \ (y_j \wedge \cdots \wedge y_k) \ \wedge \ (y_i = false) \longrightarrow \ y_i := true;$$

Using action DA_i $(0 \leq i < n)$, each element d_i of the hierarchical detector witnesses (i.e., sets the value of y_i to *true*) whenever (i) the condition LC_i becomes *true*, where LC_i represents a local condition that d_i atomically checks (by reading the variables of P_i), and (ii) its children $d_j, \cdots, d_k$ have already witnessed. The above action is an abstract action that should be instantiated by the synthesis algorithm during the synthesis of a specific program in such a way that the program and the detector do not interfere. We represent the invariant of the hierarchical detector by the predicate U, where

$$U = \{s : (\forall i : (0 \leq i < n) : (y_i(s) \Rightarrow (\forall j : i \preceq j : LC_j))\}$$

Note that $y_i(s)$ represents the value of y_i at the state s.

4. CASE STUDY: DIFFUSING COMPUTATION

In this section, we present an overview of synthesizing a nonmasking diffusing computation program by adding presynthesized components. The synthesized program provides the same behavior as the nonmasking diffusing computation program manually designed in [3]. For reasons of space, we omit the actions of the synthesized program and refer the reader to [10].

The diffusing computation (DC) program (adapted from [3]) consists of four processes $\{P_0, P_1, P_2, P_3\}$ whose underlying communication is based on a tree topology. The process P_0 is the root of the tree. Processes P_1 and P_2 are the children of P_0 (i.e., $(0 \preceq 1) \wedge (0 \preceq 2)$) and P_3 is the child of P_2 (i.e., $2 \preceq 3$). Starting from a state where every process is green, P_0 initiates a diffusing computation throughout the tree by propagating the red color towards the leaves. The leaves reflect the diffusing computation back to the root by coloring the nodes green. Afterwards, when all processes become green again, the cycle of diffusing computation repeats.

When the root process (i.e., the node whose parent is itself) is green, it starts a session of diffusing computation by changing its color to red and toggling its session number, which is a binary value. If a process P_j $(0 \leq j \leq 3)$ is green and its parent is red and its session number is not the same as its parent then it copies the color and the session number of its parent to propagate the wave of diffusing computation. If a process P_j $(0 \leq j \leq 3)$ is red and all its children are green and have the same session number as P_j then P_j changes its color to green to reflect back the wave of diffusing computation.

In each session of diffusing computation, every process P_j meets one of the following requirements: (i) P_j and its parent have both started participating; (ii) P_j and its parent have both completed the current session of diffusing computation; (iii) P_j has not started participating in the current session whereas its parent has, and (iv) P_j has completed participating in the current session whereas its parent has not. These requirements identify the program invariant.

Fault transitions can perturb the color and the session number of the processes. Also, faults may perturb the underlying communication topology of the program by changing the parenting relationship in the tree.

Intermediate Nonmasking Program. The faults may perturb the state of the DC program to the states where the program may fall in a non-progress cycle or reach a deadlock state. For example, faults may perturb the program to states where all processes are green and P_0 is no longer the root process. No program action will be enabled from such states; i.e., deadlock states. To add recovery from such states, FTSyn assigns a high atomicity process P_{high_j} to each process P_j $(0 \leq j < 4)$. A process P_{high_j} may add high atomicity recovery actions to resolve some deadlock states.

Adding Presynthesized Detectors. To refine the guard of high atomicity actions, FTSyn automatically identifies the interface of the required component. The component interface is a triple $\langle X, R, i\rangle$, where X is the detection predicate of the required component, R is a relation that represents the topology of the required component, and i is the index of the process that performs the local write action after the detection of X. Using the interface of the required presynthesized component, the synthesis algorithm queries an existing library of presynthesized components. The synthesis algorithm automatically instantiates an instance of the template action presented in Section 3 with the appropriate local condition. The local conditions are automatically identified based on the set of readable variables of each process.

Interference-freedom. The interference-freedom requires the synthesized program to provide recovery in the presence of faults, and satisfy the specification of the DC program in the absence of faults. Currently, FTSyn reduces the interference-freedom requirements to the satisfiability problem and automatically verifies them using SAT solvers. Although the synthesized nonmasking program is correct by construction, we verified the synthesized program using the SPIN model checker to gain more confidence on the implementation of FTSyn.

Complexity. The verification of interference-freedom and the addition of presynthesized components can be done in polynomial time in the state space of program-component composition (cf. [9] for proof).

5. CONCLUSION AND FUTURE WORK

In this paper, we presented a case study for adding presynthesized fault-tolerance components to programs using a hybrid synthesis method (presented in [9]) that combines heuristics presented in [8] with pre-synthesized detectors and correctors. Specifically, we showed how we add presynthesized detectors and correctors [6] to fault-intolerant distributed programs that have *hierarchical* topology. This case study extends the scope of synthesis using presynthesized components to the cases where we (i) use hierarchical components, and (ii) simultaneously add multiple components. Currently, except the extraction of the components from an existing library of presynthesized components, we automatically perform other steps of the synthesis (e.g., component specification, interference-freedom verification). As an extension to this work, we plan to apply efficient component extraction techniques where we identify the appropriate components during synthesis. Also, we plan to extend this work to programs with higher number of processes and more complicated topologies.

6. REFERENCES

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[2] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[3] A. Arora, M. G. Gouda, and G. Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems. *Journal of High Speed Networks*, 5(3):293–306, 1996.

[4] P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM TOPLAS (a preliminary version of this paper appeared in PODC96)*, 23(2), March 2001.

[5] A. Ebnenasir and S. S. Kulkarni. FTSyn: A framework for automatic synthesis of fault-tolerance. `http://www.cse.msu.edu/~ebnenasi/research/tools/ftsyn.htm`.

[6] S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.

[7] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, page 82, 2000.

[8] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of byzantine agreement. *Symposium on Reliable Distributed Systems*, 2001.

[9] S. S. Kulkarni and A. Ebnenasir. Adding fault-tolerance using pre-synthesized components. *Technical report MSU-CSE-03-28, Department of Computer Science, Michigan State University, East Lansing, Michigan, USA. A revised version is available at* `http://www.cse.msu.edu/~sandeep/auto_component_techreport.ps`, 2003.

[10] S. S. Kulkarni and A. Ebnenasir. Hierarchical presynthesized components for automatic addition of fault-tolerance: A case study. Technical Report MSU-CSE-04-41, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, September 2004.

Using Wrappers to Add Run-Time Verification Capability to Java Beans

Vladimir Glina
Department of Computer Science
Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
vglina@vt.edu

Stephen Edwards
Department of Computer Science
Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
edwards@cs.vt.edu

ABSTRACT

Because of limited information exchange between component providers and users, both these parties should perform component verification. Java Modeling Language, a notation which allows writing of behavioral specifications for Java programs, can be used for verification purposes. This paper shows that placing JML specifications in separate wrappers distributed in the binary form alongside components gives component buyers an additional value. The wrapper can serve for Java components verification on the user's side, verification checks can be enabled and disabled on per-class or per-package basis at run-time, and there is no performance overhead when they are disabled, unlike the traditional variant when checking code generated from JML specifications is placed directly into the underlying class bytecode. The paper describes wrapper design for Java Beans run-time verification and discusses advantages and challenges of it.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*programming by contract, assertion checkers, class invariants;* F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, invariants, assertions;* D.2.3 [Software Engineering]: Coding Tools and Techniques—*object-oriented programming;* D.2.5 [Software Engineering]: Testing and Debugging—*debugging aids;* D.3.2 [Programming Languages]: Language Classifications – *design languages, constraint and logic languages*

General Terms

Languages, Verification.

Keywords

JML, Java Beans, run-time verification, design by contract, events handling

1. INTRODUCTION

Construction of software from commercial off-the-shelf (COTS) components is getting more and more popular. But limited information exchange between component providers and component users is a serious problem for it. Because of that, both component provider and user must perform component verification [1].

Assertion checking is an effective means to improve quality of software verification [2]. One of assertion checking tools is Java Modeling Language (JML), a notation for formal specification of behavior and interfaces of Java classes and methods. JML implements the Design by Contract (DBC) software development principle [3]: JML specifications (pre-conditions, post-conditions, and class invariants) placed in special comments within Java source code are transformed by the JML compiler into run-time checks. Originally that checks were placed directly into Java program bytecode. It created substantial performance overhead, so checks were removed before shipping software and did little for customers.

This paper discusses using JML-based assertion checking wrappers for verification and specification of Java Beans. The paper states that for component users this approach adds value to components by providing the following:

- checking the quality of connections between components;
- saving component user's time on producing tests;
- distribution checks in binary form alongside beans so that checks can be included without access to source code;
- run-time enabling or disabling checks on per-class or per-package basis;
- avoiding performance overhead when checks are excluded.

2. ASSERTION CHECKING WRAPPER IMPLEMENTATION

Assertion checking wrapper design for Java classes was proposed in [4]. The main idea of wrapper design for Java Beans is the same. Checking code is moved to a separate class, so that there are two classes extending the same interface: an unwrapped original class and the wrapper class which only provides assertion checking and calls methods of the unwrapped class to realize all other functionality. Objects of either class are created by a class factory, which allows separating the decision which class to instantiate from the object requesting the instance. Users can

```
public class MyBean implements PropertyChangeListener {
   protected /*@ spec_public @*/ int NonNegativeValue;
   //@ requires newValue >= 0;
   public void setNonNegativeValue( int newValue ){
      // implementation goes here …
   }
   // …
}
```

Figure 1. A fragment of the `MyBean` bean source code

enable or disable assertions on per-class basis, at run-time.

Wrapper-based design uses a customized version of the JML compiler `jmlc` which automatically generates four class files from the original source code shown in Figure 1. These classes, as Figure 2 shows, are:

- the implementation class providing original functionality (`MyBeanImplementation`);
- the wrapper class containing assertion checks (`MyBeanWrapper`);
- an interface that both the classes mentioned above implement (`MyBean`);
- a factory class (`MyBeanFactory`).

Figure 3 shows how inheritance is addressed. If `MyBean` inherits from `GeneralBean`, all the generated classes related to `MyBean` inherit from the corresponding classes related to `GeneralBean`. It means that if a class has JML specifications, all its superclasses are to be transformed by the JML compiler regardless of whether they have specifications or not.

As Figure 4 shows, the interface just re-declares all the public methods of the original bean class.

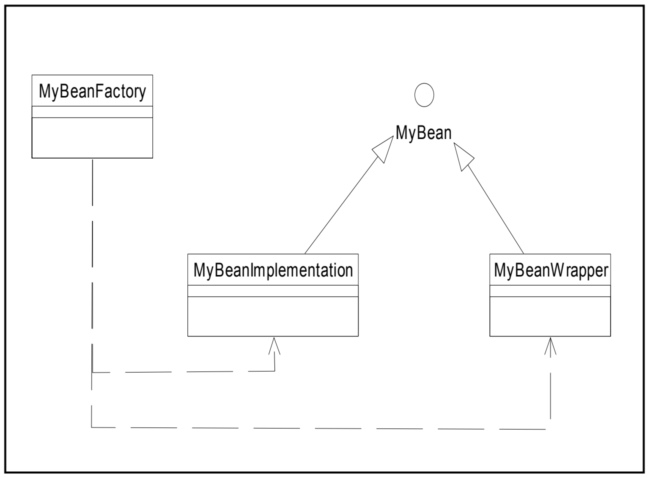


Figure 2. Transformation of the original class

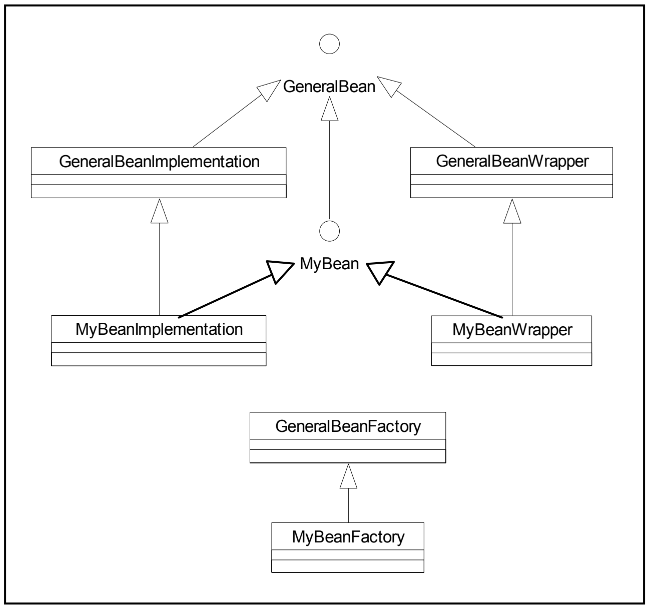


Figure 3. Inheritance: `MyBean` inherits from `GeneralBean`

In Figure 5, the wrapper corresponding to our sample bean is shown. The wrapper class masks the implementation class and adds assertion checks before and after every method. To do that, it holds the reference to the wrapped instance of the implementation class in the `wrappedObject` field. All the methods of the original class have corresponding methods in the wrapper class. Being called, the wrapper class methods at first perform pre-condition checks, then call the corresponding methods of the implementation class to perform core behavior, and after that make post-condition checks. The `isEnabled` object defines whether to perform particular checks at run-time. There is one such an object for every wrappable class and one similar object for each Java package. Thanks to these objects, it is possible to activate and deactivate run-time verification on per-class or per-package basis without access to source code, using a graphic control panel displaying the tree which maps the Java package nesting structure.

Figure 6 shows the factory class code. For every constructor in the original class, there is a corresponding factory method in the factory class. The factory queries its `isEnabled` field to decide what version of the object, wrapped or unwrapped, to create.

The implementation class has the same code as the original class

```
public interface MyBean {
   public void setNonNegativeValue( int newValue );
   // …
}
```

Figure 4. The interface that both wrapped and unwrapped classes implement

```
public class MyBeanWrapper implements MyBean {
  MyBeanImplementation wrappedObject;
  public static CheckingPrefs isEnabled = null;
  public void setNonNegativeValue( int newValue ) {
    if( isEnabled.precondition() ) {
      // the actual precondition check
      checkPre$setNonNegativeValue$MyBean( newValue );
    }
    wrappedObject.setNonNegativeValue( newValue );
  }
  // …                                                }
```

Figure 5. The wrapper class

shown in Figure 1 except it gets another name:

```
public class MyBeanImplementation {
  // implementation goes here…
}
```

When a method returns an object, the object becomes wrapped when created.

A method of a wrapped object can have an exceptional postcondition for type *T* which describes what must hold true for the method to throw an exception of type of *T* (or a subtype of *T*). After a wrapped object method throws an exception, the wrapper checks if the corresponding exceptional postcondition is present and observed, and then the exception is rethrown. Otherwise, the assertion checking fails.

Non-public method calls and the same class method calls are checked in the same way as in the case of checking wrappers for regular Java classes [4].

To enable using wrapped object with existing code which does not use assertion checking and is probably available in the binary form only, the authors of [4] are implementing a custom class loader that can transform bytecode at load time if checking wrappers are used.

```
public class MyBeanFactory {
  public static CheckingPrefs isEnabled = null;
  public static MyBean instantiate() {
    MyBean result = new MyBeanImplementation();
    if ( isEnabled != null && isEnabled.wrap() ) {
      result = new MyBeanWrapper( result, isEnabled );
    }
    return result;
  }
}
```

Figure 6. The factory class

```
public static Object instantiate( ClassLoader cls, String
beanName, BeanContext beanContext, AppletInitializer
initializer) throws java.io.IOException,
ClassNotFoundException {
  // …
  if( result == null ) {
    // No serialized object, try just instantiating the class
    Class cl;
    try {
      if( cls == null ) {
        cl = Class.forName( beanName );
      }
      else {
        cl = cls.loadClass(beanName);
      }
    }
    catch (ClassNotFoundException ex) {
      if (serex != null) {
        throw serex;
      }
      throw ex;
    }
  }
}
```

Figure 7. Original `instantiate` method

Assertion checking wrappers for Java Beans require less change in coding practices in comparison with the ones for regular Java classes. In the latter case, developers have to get used to accessing attributes of classes through getters and setters that are added into the interface besides the methods defined in the original class, whereas properties of Beans can not be accessed other than through accessors. Also, all Java Beans are usually accessed through interfaces, not concrete classes.

Nevertheless, certain changes in Java Beans run-time environment, as well as in coding practice, are required for assertion checking wrappers implementation. Typically, a user can instantiate a bean either by using operator `new`, or by calling one of `the java.beans.Beans.instantiate` methods. The latter variant is equivalent to call of the method

```
java.beans.Beans.instantiate( ClassLoader
cls, String BeanName, BeanContext
BeanContext, AppletInitializer initializer
),
```

a fragment of which is shown in Figure 7, with some (or none) arguments set to null. After wrapper design implementation, an attempt to instantiate a bean using `new` will result in a compile-time error because the name of the unwrapped bean class is now belongs to the interface. The implementation of the `instantiate` method itself is to be changed in the place responsible for bean instantiation when there is no serialized object. The changes are shown in bold in Figure 8.

```
public static Object instantiate( ClassLoader cls, String
beanName, BeanContext beanContext, AppletInitializer
initializer) throws java.io.IOException,
ClassNotFoundException {
   //…
   if( result == null ) {
     Class cl;
     String factoryName = beanName.concat( "Factory" );
     try {
       if( cls == null  ) {
         cl = Class.forName( factoryName );
       }
       else {
         cl = cls.loadClass( factoryName );
       }
       Method instantiation = cl.getMethod( "instantiate",
                                            null );
       result = instantiation.invoke( cl, null );
     }
     catch( ClassNotFoundException ex ) {
       if (serex != null) {
         throw serex;
       }
       throw ex;
     }
}
```

Figure 8. The modified version of the instantiate method

3. TOWARDS SPECIFICATION OF JAVABEANS BEHAVIOR

The most substantial problem on the way of using design-by-contract tools for software components specification is dealing with concurrency, callbacks, and event handling. In this paper, we will describe how to use JML to check contracts on events a Java Bean is registered for.

As an example, let us suppose that there is a bean called `TickGenerator` which models a generator of electric impulses. For us it is enough to know this bean has the following properties: `IsPowerOn` of type `boolean` and `NumberOfTicks` of type `int`. When `IsPowerOn == true`, `NumberOfTicks` periodically increases. A bean user can revert the `IsPowerOn` value by clicking the corresponding button on the bean graphical interface. `IsPowerOn` and `NumberOfTicks` are bounded properties. The `MyBean` bean we dealt with at the beginning of the paper has registered itself with `TickGenerator` to be notified about changes of the properties values. The events handling logic of `MyBean` and the corresponding JML specifications are shown in Figure 9.

Of course, `MyBean` has very artificial events handling that is easy to describe. Nevertheless, it shows that JML specifications can be of benefit for events handling verification. The future work

```
public class MyBean implements PropertyChangeListener {
protected /*@ spec_public @*/ int NonNegativeValue;
  // …
  /*@  requires evt.getPropertyName() == “isPowerOn”;
    @  ensures  NonNegativeValue == 0;
    @ also
    @  requires evt.getPropertyName() == “numberOfTicks”;
    @  ensures  NonNegativeValue % 2 == 1;
    @*/
  public void propertyChange( PropertyChangeEvent evt )     {
     int n;
     if( evt.getPropertyName().equals(“isPowerOn” ) {
       n = 0;
     }
     else {
       if( evt.getPropertyName().equals(“numberOfTicks” ) {
         n = 4 * n + 1;
       }
     }
     setNonNegativeValue( n );
  }
}
```

Figure 9. Specification of the Bean event handling

includes specifying event broadcasting, interaction of a group of components where one of them is listening for others, and beans serving in complicated environments that are hard to formalize (for instance, beans working with the FTP protocol).

4. ACKNOWLEDGMENTS

Our thanks to ACM SIGCHI for allowing us to modify templates they had developed.

5. REFERENCES

[1] Beydeda, S., and Gruhn, V. The Self-Testing COTS components (STECC) Strategy – a new form of improving component testability. *Proceedings of the 29th Euromicro Conference(EUROMICRO'03)* (Belek-Antalya, Turkey, September 1-6, 2003). IEEE Computer Society, Los Alamitos, CA, 2003, 107 – 114.

[2] J.M.Voas. Quality time: How assertions can increase test effectiveness. *IEEE Software,* 14, 2 (Feb. 1997), 118-119.

[3] G.Leavens, Y.Cheon. Design by Contract with JML. ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf, draft, 2004.

[4] Tan, R., Edwards, S., “An Assertion Checking Wrapper Design for Java”, *Proceedings of the Specification and Verification of Component-Based Systems workshop (SAVCBS'03)*, (Helsinki, Finland, September 1-2, 2003). Technical Report #03-11, Dept. of Computer Science, Iowa State University Ames, IA, 2003, 29-34.

Integrating Specification and Documentation in an Object-Oriented Language

[Extended Abstract]

Jie Liang
Department of Computing and Software
McMaster University
Hamilton ON Canada, L8S 4K1
liangj2@mcmaster.ca

Emil Sekerinski
Department of Computing and Software
McMaster University
Hamilton ON Canada, L8S 4K1
emil@mcmaster.ca

1. INTRODUCTION

This paper reports on the integration of specification and documentation features into an object-oriented programming language and its compiler. The goal of this integration is to improve software quality, in particular correctness, extensibility, and maintainability in a uniform and coherent manner. The language taken is Lime, an action-based concurrent object-oriented language developed at McMaster University. The concurrency aspect of Lime is motivated by the observation that concurrency is increasingly used to improve responsiveness of programs. Concurrency in Lime is expressed by attaching *actions* to objects. This eliminates the conceptual distinction between objects and threads. For the theory behind this approach and an implementation scheme the reader is referred to [12].

This paper focuses on features that are being added in order to improve software quality. We argue that specification and documentation means need to be integrated in a programming language. The documentation can be easier kept up-to-date if there is no need to switch between the programing and documentation environments; outdated documentation is a common problem [11]. If specifications are "first-class citizens", then the means for structural checks, composition, reuse, and documentation can be extended to specifications, in addition to offering the possibility for behavioral checks. We argue that for object-oriented programs to support specification, a strict separation of subclassing (code sharing) and subtyping (substitutability) is needed. Such a separation allows each class to serve as a superclass (be reused) or of a supertype (be implemented) and any child class to either inherit the implementation of the parent class, the behavioral specification, or both. Behavioral specifications are expressed by preconditions, postconditions, and invariants. These and other intermediate annotations can be written using quantifiers and other standard mathematical notation, and can be checked at run-time. The associated documentation tool generates a description of the interface of each class that includes the preconditions and postconditions of the methods, the class invariant, the subtype hierarchy, and the subclass hierarchy. Mathematical symbols in the source file are represented by Unicode characters. The compiler, LimeC, generates code for the Java Virtual Machine and the documentation tool, LimeD, generates HTML. The behavioral specifications are embedded in the generated JVM files. When inheriting from a class of a different compilation unit, both LimeC and LimeD extract these specifications from the object files of classes; the source code and separate documentation are not needed.

SAVCBS'04 Newport Beach, California, USA

2. INTEGRATING SPECIFICATIONS

The interface specification languages in the Larch family [13], JML [2, 3] and Eiffel [10] specify the behavior of their modules by Hoare-style correctness assertions.

Design by Contract (DBC), proposed by Meyer for Eiffel [10], is a formal technique for dynamically checking specification violation during run-time. The idea behind DBC is that a class and its client have a "contract" with each other. The client must guarantee certain conditions before calling a method defined by the class, and in return the class guarantees certain properties that will hold after the call. In Eiffel, the contracts are defined by program code, and are translated into executable code by the compiler. Thus, any violation of the contract can be detected immediately during run-time.

The lack of assertions and design by contract features in Java has led to some languages and run-time assertion checking tools, such as Alloy Annotation Language (AAL) [6], Jass [1], and iContract [7]. AAL is a language for annotating Java code based on the Alloy modeling language.

JML, which stands for "Java Modeling Language," is a behavioral interface specification language (BISL) designed to specify Java modules. JML adds assertions to Java by writing them as special comments (/*@ ... @*/ or //@ ...). It is based on the use of preconditions, postconditions and invariants. JML uses Java's expression syntax to write the predicates used in assertions. Java expressions lack some expressiveness that makes more specialized assertion languages convenient for writing behavioral specifications; JML solves this problem by extending Java's expressions with some specification constructs, such as quantifiers.

Lime integrates assertions as programming language constructs, as Eiffel does; for offering a trade-off between checking for correctness and efficiency, assertion checking can be selectively enabled and disabled. Lime offers *class invariants*, *precondition*, and *postconditions* to specify module behavior. In addition, Lime has *assert* statements which specify a constraint on an intermediate state in a method body.

To have more expressiveness for writing behavioral specifications, Lime allows the following constructs in expressions:

- Boolean operators $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$. Note that $\Leftrightarrow$ means the same as $=$ for expressions of type *boolean*; however, $\Leftrightarrow$ has a lower precedence.
- *old e* for referring to a value in the pre-state. It is used in postconditions to indicate an expression whose value is taken from the pre-state of a method call. For example, *old(x + y)* denotes the value of *x + y* evaluated in the pre-state of a method call.
- *result* for referring to the value or object that is being returned by a method. It is used in a method's postcondition. Its type is the return type of the method.
- linear quantifications $* \, x \, | P \bullet E$, where $*$ is a quantifier operator; x is the bound variable; P is the range; E is the body of the quantification; $*$ is $\forall$, $\exists$, Σ, Π, MAX, MIN, or NUM. The range has the form $E_{low} \preceq x \preceq E_{up}$, where $\preceq$ is either $<$ *or* $\leq$.

To make the source code and generated documentation more readable and meaningful, we use a number of mathematical symbols. With the aid of Unicode and UTF-8 encoding, these mathematical symbols can be parsed by the compiler and displayed on any word processor that supports UTF-8 enconding for editing source code and inside generated documentation on web browsers.

Since Java and JVM do not support behavioral specifications, we need a way to store the preconditions, postconditions and invariants in a Java *class* file. When handling inheritance and separate compilation, we only get information about other compilation units from their Java class file. The reason is that in some situation such as using a library class, we may not have the source code. The class invariant, preconditions and postconditions are therefore stored in Java class files as constant strings. They are extracted as constant values from the constant pool by LimeC and LimeD.

3. TYPES AND CLASSES

Inheritance is a language mechanism that allows new object definitions to be based on existing ones. A new class inherits the properties of its parents, and may introduce new properties that extend, modify or defeat its inherited properties. Subtyping and subclassing are conceptually different views of inheritance: Subtyping is related to specification and interface inheritance; subclassing is a mechanism for implementation reuse.

Cook [4] points out that in most strongly-typed object-oriented languages subtyping are subclasses are combined and equated, and inheritance is basically restricted to satisfy the requirements of subtyping. It has been argued that this eliminates several important opportunities for code reuse [8, 10]. Currently, only a few languages, such as POOL-I, Theta, PolyTOIL and Sather, support separating subtyping and subclassing to some degree.

$\tau \;\; extend \;\; \sigma$

1. τ inherits every non private attribute a_σ of σ: $\tau.A \supseteq \sigma.A$.
2. For any non private method m_σ of σ there is a corresponding method m_τ of τ, such that
   - m_τ has m_σ's signature: $m_\tau.Sig = m_\sigma.Sig$.
   - m_τ has m_σ's implementation: $m_\tau.Imp = m_\sigma.Imp$.

Figure 1: Definition of *extend*

$\tau \;\; implement \;\; \sigma_1, \sigma_2, ..., \sigma_n$

1. τ preserves invariants of all supertypes $\sigma_1, \sigma_2, ..., \sigma_n$: $\tau.I \Rightarrow \bigwedge_{i=1}^{n} \sigma_i.I$.
2. τ inherits all non private attributes from all supertypes $\sigma_1, \sigma_2, ..., \sigma_n$:
   $\tau.A \supseteq \bigcup_{i=1}^{n} \sigma_i.A$.
3. For any non private method m_{σ_i} of each supertype σ_i there is a corresponding method m_τ of τ, such that
   - m_τ has m_{σ_i}'s signature ($m_\tau.Sig = m_{\sigma_i}.Sig$).
   - m_τ weakens preconditions: $m_\tau.Pre \Leftarrow \bigvee_{i=1}^{n}(m_{\sigma_i} \in \sigma_i.M \wedge m_{\sigma_i}.Pre)$.
   - m_τ strengthens postconditions: $m_\tau.Post \Rightarrow \bigwedge_{i=1}^{n}(m_{\sigma_i} \in \sigma_i.M \Rightarrow m_{\sigma_i}.Post)$.

Figure 2: Definition of *implement*

Syntactic subtyping can be extended to behavioral subtyping. The essence of behavioral subtyping is summarized by Liskov and Wing's subtype requirement [9]:

> Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

We propose an inheritance mechanism that strictly separates subclassing from subtyping and makes inheritance more flexible. Any class in Lime can act as a superclass or a supertype. A class contains a syntactic interface, the specified behavior, and an implementation. A child class has the choice of inheriting either the behavioral specification, the implementation, or both.

A Lime class definition consists of a class invariant (*I*), a set of attributes (*A*) and a set of methods (*M*). We model a class as a triple $\langle I, A, M \rangle$. A method is composed of a signature (*Sig*), behavioral specification and implementation (*Imp*). The method signature includes name, access, return and parameters' types. The behavioral specification consists of a precondition (*Pre*) and postcondition (*Post*). The implementation is the source code of the method body. We model a method as a quadruple $\langle Sig, Pre, Post, Imp \rangle$.

Lime uses the *extend* clause to handle single subclassing (Figure 1) and the *implement* clause to handle multi-subtyping (Figure 2). The case of combined subclassing and subtyping is expressed by the *inherit* clause. The subtyping definitions follows that of [9]; JML uses the generalization of [5]. For example, the class header

class Sub extend Sup2 implement Sup1

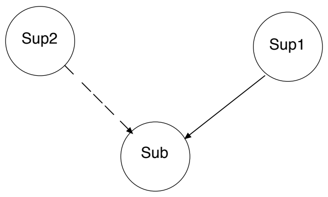


Figure 3: Inheritance graph in Lime

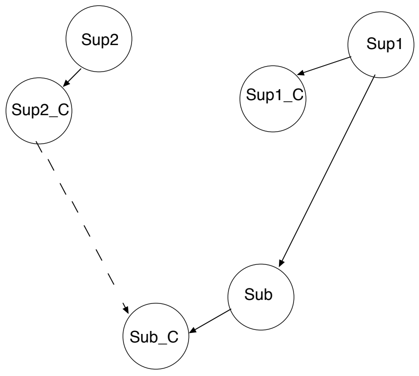


Figure 4: Inheritance graph of generated Java classes

builds an inheritance relation shown by the inheritance graph in Figure 3. Solid and dashed arcs are used to represent subtype and subclass relationships, respectively.

We sketch how the inheritance relationship is implemented in the generated executable Java class file. Java supports single inheritance of classes and multiple inheritance of interfaces that can only contain method signatures and constant static variables. For each Lime source file, we generate two Java class files. One stores a Java class that contains all the information in the original Lime file, and its name ends with "*_C*", the other stores a Java interface that still uses its original name. The graph in Figure 4 shows the inheritance relationship among the generated Java classes. In Java, it would be legal to assign an instance of *Sub_C* to a variable declared as of type *Sup1* or *Sup2*. According to our definition of *extend*, *implement* and *inherit*, class *Sub* is *Sup1*'s subclass, not subtype. The compiler checks whether the variable being assigned is of a supertype of the instance's class. From the inheritance graph view, this amounts to checking whether there exists a path that is composed of all solid arcs between two types.

In the following example, class *Polygon* is only a subclass of *Rectangle*, not a subtype. It can reuse the code in class *Rectangle* such as method *boundingBox*. It also overrides method *move* and *area*. Quantifier $\forall$ is used for specifying the behavior of method *move*. In the initialization, *MAX* and *MIN* are used for calculation.

```
abstract class Shape
  protected attr x, y : integer;
  public abstract method boundingBox : Rectangle;
  public method area : integer
     return 0
  public method move (dx, dy : integer)
    begin
      x : = x + dx;
      y := y + dy
    end
  initialization(x, y : integer)
    begin
     self.x := x;
     self.y := y
  end
end

class Rectangle inherit Shape
  protected attr w, h : integer;
  public method boundingBox : Rectangle
     return new Rectangle(x, y, w, h)
  public method area : integer
     return w * h
  initialization(x, y, w, h : integer)
    begin
      super.initialization(x, y);
      self.w := w;
      self.h := h
    end
end

class Polygon extend Rectangle implement Shape
  protected attr i, n : integer;
  attr Xs : array of integer;
  attr Ys : array of integer;
  invariant n > 0
  initialization(Xs, Ys : array of integer, n: integer)
    begin
      self.n := n;
      x := Xs[0];
      y := Ys[0];
      w := $(MAX\ i \mid 0 \leq i < n \bullet Xs[i]) -$
                   $(MIN\ i \mid 0 \leq i < n \bullet Xs[i]);$
      h := $(MAX\ i \mid 0 \leq i < n \bullet Ys[i]) -$
                   $(MIN\ i \mid 0 \leq i < n \bullet Ys[i]);$
      i : = 0;
      while i < n-1 do
        self.Xs[i], self.Ys[i] := Xs[i+1], Ys[i+1];
    end
  public method move (dx, dy : integer)
    post $\forall\ i \mid 0 \leq i < n - 1 \bullet (dx =$
          old Xs[i] - Xs[i]) $\wedge$ (dy = old Ys[i] -Ys[i])
    begin
      super.move(dx, dy);
      i := 0;
      while i < n - 1 do
        Xs[i], Ys[i] := Xs[i] - dx, Ys[i] - dy
    end
  public method area : integer
    ...
  end
end
```

4. DOCUMENTATION GENERATION

Lime's support for automatic documentation generation was influenced by early work on literate programming and documentation system like *Javadoc* and *Doxygen*. Both Javadoc and Doxygen generate on-line interface documentation in HTML format. The design for LimeD is along those lines:

- LimeD generates documentation directly from the source code;
- LimeD provides a behavioral interface specification, not only a syntactic interface;
- LimeD shows the subclass and subtype hierarchies.

For a project, LimeD generates a summary page and a page for each individual class. For quickly accessing class documentation, a list with linked indices for all classes is generated and acts as a navigation menu. The documentation of each individual class starts with the class description extracted from the documentation comment in the source file. Documentation comments can contain embedded HTML code. The document may contain the following parts:

- **Class Invariant** with the invariant defined in the current class and the invariants inherited from all supertypes. The inherited invariants are conjoined to generate a single expression. All the information is extracted from the current class and from the Java class files of all supertypes.
- **Class Hierarchy** displayed graphically; Lime supports single subclassing.
- **Type Hierarchy** presented as an indented list; Lime supports multiple subclassing.
- **Attribute** with all non-private attributes defined in the current class.
- **Inherited Attribute** with all attributes inherited from superclasses and supertypes.
- **Method** contains all methods defined in the current class. It gives the method signature and the precondition and postcondition defined in the current class. If the method redefines or implements a method of a supertype, it also gives the precondition and postcondition defined in supertypes. These are extracted from the Java *class* files of all supertypes.
- **Inherited Method** contains all inherited methods. It gives the method signature, precondition and postcondition.

5. OUTLOOK

Currently the development is still in an experimental stage. An exception handling mechanism needs to be integrated and the specification language needs to be extended with abstract date types. Currently specifications can only use the data types of the programming language.

6. REFERENCES

[1] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with assertions. In K. Havelund and G. Rosu, editors, *Proceedings of the First Workshop on Runtime Verification, Electronic Notes in Theoretical Computer Science,* volume 55. Elsevier Science, July 2001.

[2] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03), Electronic Notes in Theoretical Computer Science,* volume 66, pages 1–17, Trondheim, Norway, June 5–7, 2003. Elsevier Science.

[3] Y. Cheon and G. T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In *International Conference on Software Engineering Research and Practice (SERP),* pages 322–328. Computer Science Research, Education, and Applications (CSREA) Press, Las Vegas, Nevada, USA, June 2002.

[4] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL '90)*, pages 125–135, San Francisco, January 1990. ACM Press. Addison-Wesley.

[5] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th international Conference on Software Engineering,*, pages 258–267, Berlin, Germany, March 1996. IEEE Computer Society Press.

[6] S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *ACM SIGPLAN Notices , Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 37, pages 231–245, Seattle, Washington, USA, November 2002.

[7] R. Kramer. iContract - the Java design by contract tool. *TOOLS 26: Technology of Object-Oriented Languages and Systems,* pages 295–307, 1998.

[8] B. B. Kristensen, O. L. Madsen, B. Moeller-Pedersen, and K. Nygaard. The BETA programming language. In B. D. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[9] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[10] B. Meyer. *Object-Oriented Software Construction 2nd edition*. Prentice-Hall, 1997.

[11] A. L. Powell, J. C. French, and J. C. Knight. A systematic approach to creating and maintaining software documentation. In *Proceedings of the 1996 ACM symposium on Applied Computing,* pages 201–208, Philadelphia, Pennsylvania, February 1996.

[12] E. Sekerinski. Concurrent object-oriented programs: From specification to code. In *First International Symposium on Formal Methods for Components and Objects, FMCO 02*, Lecture Notes in Computer Science 2852, pages 403–423, Leiden, The Netherlands, 2003. Springer-Verlag.

[13] J. M. Wing. Writing Larch interface language specification. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

Designing a Programming Language to Provide Automated Self-testing for Formally Specified Software Components

Roy Patrick Tan
Department of Computer Science
Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
rtan@vt.edu

Stephen H. Edwards
Department of Computer Science
Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
edwards@cs.vt.edu

1. INTRODUCTION

Writing software is an error-prone activity. Compilers help detect some of these errors: syntactic mistakes plus those semantic mistakes that can be detected through the type system. However, locating faults beyond those detectable by the compiler (and other static analysis tools) is often relegated to the programmer, who must write thorough tests to ensure confidence in the correctness of the software.

Although the specification and verification community has traditionally focused on decreasing software bugs by static verification, research has increasingly explored the dynamic analysis of the conformance of software components to its specifications. That is, researchers are investigating systems that can tell us whether a program's behavior is consistent with its specification while the program is being executed. While dynamic techniques do not offer the same degree of assurance as full static verification, they may provide useful pragmatic benefits without the human intervention needed by currrent generation verification tools. When interpreted as a testing technique, dynamic analysis offers us a glimpse of future testing tools that offer another line of automatic error detection that augments the compiler, and helps the programmer reduce the number of tests he has to write.

Modern unit testing tools such as JUnit allow some automation of the testing process. Specifically, they allow the automated execution of tests. The job of writing tests remains the responsibility of the programmer. In writing a test for a software component, the programmer must `a.` exercise a component such that a bug is likely to manifest; and `b.` write code to detect the bug.

Current research suggests that the use of formal specifications, coupled with the right infrastructure, may alleviate much of the tedious process of writing the tests. For example, JML-JUnit [?] removes the need to write code that detects a component failure. It can act as a test-oracle by checking Java classes against their specifications as the test cases are being executed. While JML-Junit suggests an ideal strategy for combining test execution with specification-based oracles, an appropriate test case generation strategy is necessary for effective performance [?, Tan04]

Not surprisingly, formal specifications can also play a role in automatically generating the test-cases themselves. Korat [?], for example, can quickly generate linked data structures for inputs by using an invariant checker to filter out impossible structures. A method to automatically generate test-cases proposed by one of the authors also leverages the runtime-checking of specifications [?].

The possibility of combining these supporting techniques into a unified approach to dynamic verification leads to a new question: what form would such a consolidated testing tool take? Based on the concept that the best tool is one that is so transparent it is invisible, we envision infusing the necessary infrastructure directly within the programming language itself. A language that provides the necessary support for formal behavioral description could generate a compiled component that has the ability to execute and report the results of tests it has created for itself. This would amount to a built-in self-testing capability that comes for free, as a side effect of writing formal specifications.

2. A VISION FOR A NEW STYLE OF PROGRAMMING LANGUAGE

We envision that in the future, when a software component is ready for testing, it will have a formal specification written for it. Ideally, the specification would be complete, correct, and written well before the implementation. More probably, the specification might have missing parts, incorrect parts, and would have evolved as the implementation was written.

In any case, when the developer is ready to test his component, he runs his testing tool, and the tool will automatically generate the test-cases, run them, determine which tests passed or failed, and generate a report. This report will tell the programmer which parts of the component it found to be inconsistent with its specification. Depending on the report, the software engineer may fix some of his implementation code; he may refine his specifications; or he may write additional tests that the automated tool does not cover. The process iterates until the developer is confident

```
    public void testPush() {
        IntStack stack = new IntStack();
        stack.push(5);
        Assert.assertTrue(stack.size() == 1);
        Assert.assertTrue(stack.top() == 5);
    }
```

Figure 1: A JUnit test case method for an integer stack.

that the component works as specified.

In this scenario, the software developer writes much fewer test cases, though he is not completely rid of writing them. Just like most modern programmers do not normally need to bother with low-level details such as register allocation, programmers of the future will not have to write lower-level test cases. Instead, the programmer may concentrate on writing test cases for more subtle, hard-to-find bugs.

The programmer here never has to write any code to determine the correct behavior of the component under test. Instead, he has to write formal specifications. Making the programmer write specifications may be the most difficult part of transitioning from the current way of writing software. However, this may be mitigated in part by the fact that the techniques we are considering to bring us closer to this vision do not require complete or comprehensive specifications.

3. TECHNIQUES FOR AUTOMATED UNIT TESTING

Much of the ground work for our scenario for the future of unit testing has already been done. We believe it is possible to automate at least partially the two things a developer has to do manually in testing: exercise the component, and detect a fault if it occurs. The apporoaches we have been looking at have these two key aspects: software components have to be specified formally, and that there is a runtime environment that executes these specifications alongside the implementation. That is, specifications such as preconditions, postconditions, and class invariants must be checkable at runtime, whenever a method is called—a design-by-contract style of specification execution.

3.1 Using Specifications as a Test-Oracle

Figure 1 is an example of what a JUnit test case method for a stack may look like. Take note the two `assertTrue` calls, these assertions tells JUnit what must be true after the push statement. Every test of push has to have “assertions” similar to the one in Figure 1. It would be advantageous if we could write assertions in a single location that tells JUnit what must be true after every call to the push method.

Using the Java Modeling Language (JML) [?], to specify Java classes, you can do exactly this. The commented parts of Figure 2 is the postcondition of the `push` method. Since JML can execute the postcondition every time the push method is called, there is no more need to write assertions for the push test. Instead we can let JML detect the fault for us.

This is in fact what the JML-JUnit tool does; it uses JML's runtime checking of specifications as a test oracle. Thus, with formal specifications and the right runtime infrastructure, specifications can be used to check correct behavior in lieu of manual assertions inserted in test cases.

```
    //@ensures size() == \old(size()) + 1
    //@        && top() == x;
    public void push(int x) {
        //...
    }
```

Figure 2: A partial JML specification of the push method.

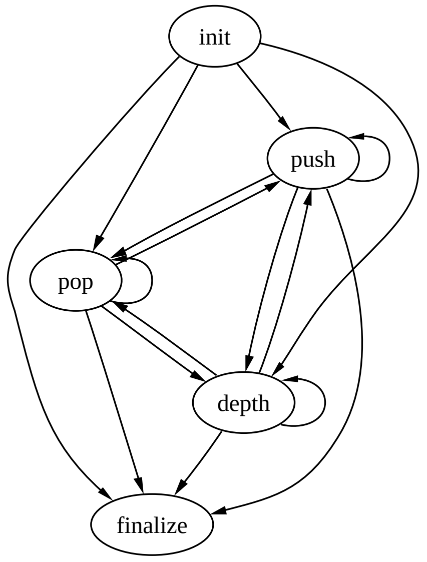


Figure 3: A flow graph for a stack component

3.2 Test cases generation

In [?], one of the authors (Edwards) presents a strategy of generating test cases using flow graphs which in turn is based on the methodology described by Zweben and Heym [?]. We present a brief explanation:

Given a specified component, we build a graph where any walk represents a possible object lifetime. We define a flowgraph as follows:

> A flowgraph is a directed graph where each vertex represents one operation provided by the component and a directed edge from vertex v_1 to v_2 indicates the possibility that control may flow from v_1 to v_2.[?]

In other words, when there is an edge from v_1 to v_2, it means that there exists an object state where v_2 can be legally called after v_1. A flowgraph for any component can be constructed in the following way: Represent every method as a node in a graph. Construct a complete, directed graph with self-loops from these vertices. And then, add two more nodes, *begin* and *end*; place a directed edge from *begin* to every node and from every node to *end*. Additionally, there is an edge going from *begin* to *end*. Figure 3, for example, is a flowgraph for a stack component.

Thus, a walk from the *begin* vertex to the *end* vertex represents a sequence of method calls from object initialization to object finalization, i.e. a possible object lifetime. It is easy to see that each feasible walk can be a test-case for the component.

There are two problems that come to mind, one is that some of the walks may be infeasible. For example, the sequence of method calls represented by $begin \rightarrow push \rightarrow pop \rightarrow pop$ for a stack component may be infeasible, because the last pop call violates the method's precondition. The other problem is that there are a pontentially infinite number of feasible walks through the graph.

The problem of infeasible walks can be solved by using the dynamic execution of specifications to detect them. An infeasible path is detected when a precondition failure occurs while executing a sequence of method calls represented by a walk on the flowgraph.

The second problem of choosing the right walks to use as test-cases is an open topic for research. There are several possible ways to achieve this:

- Random walk. Random walks are simple to implement but may not be best.

- Bounded exhaustive enumeration. For example, choose all walks going through 5 nodes or less.

- Various machine-learning algorithms. Tonella [?], for example, reports on an experiment that uses evolutionary algorithms to essentially generate these walks.

Work is ongoing to investigate the efficacy of each of these strategies.

4. SUPPORTING AUTOMATED TESTING

In the previous section, we see that a software developer who is willing to write formal specifications may be able to take advantage of a higher level of automated testing. Aside from the developer's willingness to write the formal specifications, however, the developer must possess tools that can take advantage of these specifications.

What are the necessary requirements to be able to build these tools? What language features must exist for our automated testing strategy to work? The basic necessities are that the programming language must have its components formally specifiable, and that there is a runtime system that can execute the specifications in a design-by-contract style.

We believe that any language with design-by contract style specifications (and the ability to check them at runtime) is amenable to the automated testing strategy we outline above. However, we have also listed a number of secondary characteristics that may be beneficial:

- Simple specification language—a simple language allows for easier programmer buy-in, part of this is to have the specification language be as close as possible to the implementation language, to make it easier to learn.

- Support for modular reasoning—modular reasoning means that each module (e.g. a class) is as encapsulated as possible; that it can be reasoned about in isolation of the rest of the program, and thus can be tested in isolation.

- Small programming language—a small language without too many features may make for simpler specification.

- Ability to measure other metrics—such as time for every method call, code coverage, etc.

Several programming languages have the necessary characteristics to implement tools that follow our testing strategy. The aforementioned JML-JUnit tool, for example, already uses runtime checkable specifications as a test oracle. Eiffel, which popularized design by contract, is certainly a candidate for this type of tool. Theoretically, design-by-contract extensions to popular scripting languages such as Python [?] and Ruby can also be used.

Each of these languages, however, also have characteristics that makes building automated testing tools for them difficult. For example, Java allows direct access of data members, breaking modularity of reasoning. The meta-programming features of Python and Ruby, might allow developers to circumvent specification checking. Eiffel breaks the Liskov substitution principle [?] All the languages considered are also fairly feature-rich; building a tool that covers all the features of one of these languages may be beyond the resources of academic researchers. The use of reference semantics in all these languages introduce aliasing, which also introduces all the difficulties associated with specifying them.

We have decided to take on the challenge of designing a new programming language and its runtime system. Tentatively called Sulu, we are designing it with the goal that every component written in this language can be tested automatically. By implementing a new language we will have the advantage of having complete control of the language, we can make it only as large as necessary, place only the features we require. It can also serve as a platform for future research.

5. DISCUSSION

We discuss many of the technical concepts of building the automated testing tools that we envision. But beyond building the tools, we must be able to measure their effectiveness. We must also measure other metrics like the number of test cases, the time it takes for the automated process to generate them, and the time it takes to execute the tests.

By deciding to implement a new programming language, we encounter a new set of challenges; what features should we put in the new language? What should be left out? We must strike a balance between making it small enough to be easy to implement, and big enough to show that our techniques are also applicable to mainstream languages.

The key elements of a specification language and the ability to check the specifications against the implementation at runtime will be included, of course, but what about other features? Sulu will be component-based. That is, it will have strong separation of an object's specification and its implementation. It will use value semantics to avoid the difficulties of specifying aliased variables. Performance concerns will be addressed somewhat by allowing a swap operator [?].

We are still actively evaluating whether to include other features, such as the object-oriented concepts of inheritance and polymorphism. These features may make results from future experiments more comparable to mainstream languages, but it may also mean a much more complicated implementation and specification language.

Another crucial question that may need to be addressed is the cost/benefit to the software developer. Will the promise of automated test-case generation convince practitioners to write formal specifications? How effective should the tools be to facilitate this change?

If the developer does write formal specifications, this artifact may be useful for other analysis tools. How can traditional verification tools be used in conjuction with the testing tools to help us build better, more reliable software?

6. CONCLUSION AND RELATED WORK

In this paper, we have outlined our vision for unit testing, that testing tools will come to the fore as another level of automatic error detection.

We have discussed the techniques we are implementing as we develop our testing platform, but there is a fair amount of other research on the automatic generation of test cases. ASTOOT [**?**] and DAISTS [**?**] approach the problem quite differently. They automatically generate test cases directly from algebraic specifications through term-rewriting.

Korat [**?**] is a system that automatically generates linked data structures that can be used as parameters for methods under test. Korat's use of a "representation invariant" to filter out infeasible data structures is similar and may be compatible with our technique.

Tonella [**?**] describes a system in Java that uses an evolutionary algorithm to generate method sequences essentially equivalent to walks in our flowgraph model. However, he does not consider the problem of infeasible method sequences.

The basic techniques to achieve this vision already exist, and discussed how we can support our techniques in a modern programming language.

Open Incremental Model Checking (Extended Abstract)

Nguyen Truong Thang Takuya Katayama
School of Information Science
Japan Advanced Institute of Science and Technology
email: {thang, katayama}@jaist.ac.jp

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking*

1. INTRODUCTION

Separation of concerns is the core of successful software [5]. One of the most prominent form of concerns are *hyperslices* [5] or features. A system is structured by composing several separate features. The terms feature, hyperslice and component are used interchangeably henceforth.

This paper focuses on the interaction between two components: base and extension. Specifically, the extension refines or modifies the base, i.e. the interferences of the base and extension execution paths occur. Unlike traditional modular model checking methods which treat systems as static, a new method of model checking, called *open incremental model checking* (OIMC) in our opinion, is proposed to address the changes to systems [1]. Given a base component, an extension component is attached such that the extension does not violate some property inherent to the base. A primitive model and a simple verification procedure are suggested to ensure the consistency between two components [1]. The model checking is executed in an *incremental* manner within the extension component only. This approach is also *open* for various kinds of changes. This paper is quite different the work of [1] in several key points such as proposing a generalized model with overriding capability (Section 3), an explicit consistency condition among components (Section 4). More importantly, we also examine key issues not addressed in [1] such as the soundness (Section 5.1) and scalability (Section 5.2). Discussion about the contribution of this paper, its future and related work are presented in Section 6.

2. BACKGROUND

CTL is a restricted subset of CTL* in which each temporal operator among **X** (“next”), **F** (“eventually”), **G** (“always”), **U** (“until”) and **R** (“release”) must be preceded by a quantifier of **A** (“for all paths”) and **E** (“for some path”). With respect to CTL, the incremental verification method has been attempted by [1]. Its key ideas are:

- Proposing a simple formal model whose interface is fixed with single *exit* and single *reentry*. Importantly, this model is additive, i.e. the extension is not allowed to override any behavior of the base.
- Presenting a verification algorithm to check whether a property continues to hold at those exit states. Fundamentally, the algorithm is based on the assumption about labels at all reentry states. From the assumption, the conclusion about the extension component with respect to the property is drawn. The unproven assumption is a weakness of [1] in terms of soundness.

In this paper, the formal interface is generalized to accommodate multiple exit and reentry points with overriding capability. The soundness of OIMC with respect to this generalized model is then tackled in two aspects: proving assumptions at reentry states instead of simply assuming them (1); and based on the proven facts at reentry states, proving the property preservation in the base component (2). Later, the scalability problem of OIMC is discussed with respect to the reality that many subsequent extension components will be incorporated into the newly evolved system. We investigate the complexity of OIMC under such a situation to preserve the property of the original base component.

3. FEATURE SPECIFICATION MODEL

Each component is separately modeled via a state transition model. Let AP be a set of atomic propositions.

Definition 1. *A state transition model M is a tuple $\langle S, \Sigma, s_0, R, L\rangle$ where S is a set of states, Σ is the set of input events, $s_0 \in S$ is the initial state, $R \subseteq S \times PL(\Sigma) \to S$ is the transition function (where $PL(\Sigma)$ denotes the set of propositional logic expressions over Σ), and $L : S \to 2^{AP}$ labels each state with the set of atomic propositions true in that state.*

Typically, there are two components to consider: a base and an extension. Between the base and its extension is an interface consisting of *exit* and *reentry* states. An exit state is the state where control is passed to the extension. On the other hand, a reentry state is the point at which the base regains control. A *base* is expressed by a transition model B and an *interface* I, where $B = \langle S_B, \Sigma_B, s_{o_B}, R_B, L_B\rangle$. An interface is a tuple of two state sets $I = \langle exit, reentry\rangle$, where $exit, reentry \subseteq S_B$ and $exit, reentry \neq \emptyset$. On the other hand, an *extension* is represented by a model $E = \langle S_E, \Sigma_E, \smile, R_E, L_E\rangle$, if considered separately from the base B. $\smile$ denotes no-care value. The interface of E is $J = \langle in, out\rangle$.

E can be inserted to B via the *compatible* interface states according to the following.

- An exit state $ex \in exit$ of B can be matched to an in-state $i \in in$ if $L_E(i) \subseteq L_B(ex)$.
- A reentry state $re \in reentry$ of B can be matched to an out-state $o \in out$ if $L_B(re) \subseteq L_E(o)$.

Subsequently, states ex and re will be used in place of i and o whenever interface states are referred.

Definition 2. *Composing the base B with the extension E, through the interfaces I and J produces a composition model $C = \langle S_C, \Sigma_C, s_{0_C}, R_C, L_C\rangle$. C is defined from $B = \langle S_B, \Sigma_B, s_{0_B}, R_B, L_B\rangle$ and $E = \langle S_E, \Sigma_E, \smile, R_E, L_E\rangle$.*

- $S_C = S_B \cup S_E$; $\Sigma_C = \Sigma_B \cup \Sigma_E$; $s_{0_C} = s_{0_B}$;
- *R_C is defined from R_B and R_E. For each $s \in S_C$, let $\vee_s^E = \bigvee pl_i$ where $(s, pl_i) \in Dom(R_E)$,*
  - $\forall (s, pl_i) \in Dom(R_E)$: $R_C(s, pl_i) = R_E(s, pl_i)$
  - $\forall (s, pl_B) \in Dom(R_B)$: $R_C(s, pl_B \wedge \neg\vee_s^E) = R_B(s, pl_B \wedge \neg\vee_s^E)$
- $\forall s \in S_B, s \notin I.exit \cup I.reentry : L_C(s) = L_B(s)$;
- $\forall s \in S_E, s \notin J.in \cup J.out : L_C(s) = L_E(s)$;
- $\forall s \in I.exit \cup I.reentry : L_C(s) = L_B(s) \cup L_E(s)$;

The propositional logic expressions between different transitions from the same state are disjoint. $\vee_s^E$ represents the union of all events directing to the extension from a state s. In the composition definition above, a transition (s, pl_B, s') in B can be partially or completely overridden by E. In case of being overridden, s is certainly an exit state because overriding only occurs at exit states. The transition is completely removed from C if $pl_B \wedge \neg\vee_s^E = false$. Otherwise, it is partially overridden. This is another key difference between our model and the former [1] in which E is not allowed to override any transition in B. The former is called *additive-only* composition, while ours is *limited overriding.*

Definition 3. *The* closure *of a property p, $cl(p)$, is the set of all sub-formulae of p, including itself.*

Definition 4. *The truth values of state s with respect to a set of CTL properties ps within a model $M = \langle S, \Sigma, s_0, R, L\rangle$, denoted $\mathcal{V}_M(s, ps)$, is a function: $S \times 2^{CTL} \to 2^{CTL}$ defined according to the following:*

- $\mathcal{V}_M(s, \emptyset) = \emptyset$
- $\mathcal{V}_M(s, \{p\} \cup ps) = \mathcal{V}_M(s, \{p\}) \cup \mathcal{V}_M(s, ps)$
- $\mathcal{V}_M(s, \{p\}) = \begin{cases} \{p\} & \text{if } M, s \models p \\ \{\neg p\} & \text{otherwise} \end{cases}$

CTL denotes the set of all CTL properties. Hereafter, $\mathcal{V}_M(s, \{p\}) = \{p\}$ (or $\{\neg p\}$) is written in the shorthand form as $\mathcal{V}_M(s, p) = p$ (or $\neg p$) for individual property p.

In the subsequent discussion, incremental model checking is represented by an *assumption model checking* [4] in E only rather than in C. The *assumption function* for that model checking is a function $As : I.reentry \to 2^{CTL}$. In such a situation, the reentry states re in E are assumed with truth values seeded from B, $\mathcal{V}_B(re, cl(p))$, namely $As(re) = \mathcal{V}_B(re, cl(p))$.

Definition 5. *The assumption function As is* proper *at a reentry state re if the assumed truth values are exactly those resulted at re from the standard model checking in C, i.e. $\mathcal{V}_B(re, cl(p)) = \mathcal{V}_C(re, cl(p))$.*

4. PROPERTIES PRESERVATION AT BASE STATES

A property p is adhered to the base $B = \langle S_B, \Sigma_B, s_{0_B}, R_B, L_B\rangle$ if it holds for every state in B, i.e. $\forall s \in S_B : B, s \models p$. An extension E is *composable* with B with respect to p if $\forall s \in S_B : C, s \models p$ where C is the composition of B and E.

The key problem this paper tries to deal with is: Given B and p, what are the conditions for E so that B and E are composable with respect to p?

With respect to the generalized model, the soundness issue is very important. It will be discussed in Section 5.1. Another question relates to the scalability of OIMC (Section 5.2) with respect not only to E but also to many future extensions to the composition C.

4.1 A Theorem on Component Consistency

Due to the inside-out characteristic of model checking, during verifying p in B, $\mathcal{V}_B(s, cl(p))$ are recorded at each state s. The truth values $\mathcal{V}_B(ex, cl(p))$ at any exit state ex serve as the conformance for the composition between B and E.

Definition 6. *B and E are in* conformance *at the exit state ex (with respect to $cl(p)$) if $\mathcal{V}_B(ex, cl(p)) = \mathcal{V}_E(ex, cl(p))$.*

Theorem 7. *Given a base B and a property p, an extension E is attached to B at some interface states. Further, suppose that the assumption function As defined during model checking E is proper. If B and E conform with each other at all exit states, $\forall s \in S_B : \mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p))$.*

The theorem holds regardless of composition type, either additive or overriding. Due to space limitation, the proof of the theorem is skipped. Figure 1 depicts the composition

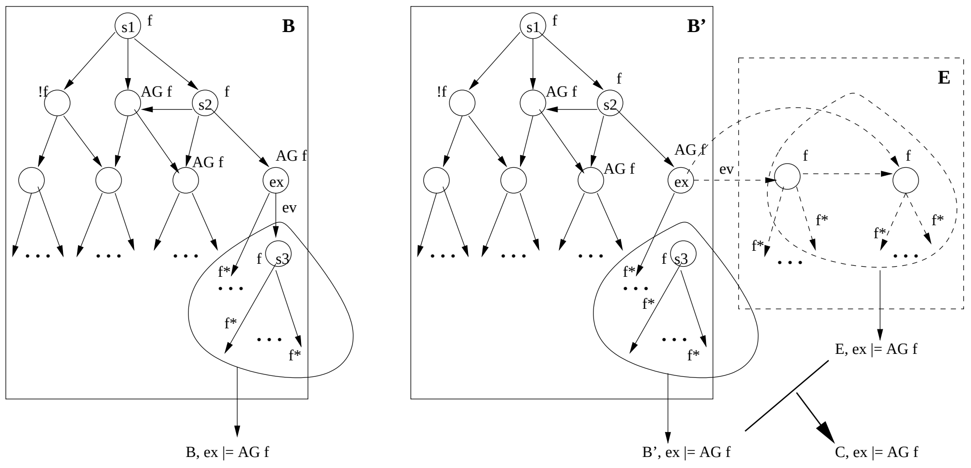


Figure 1: An illustration of base-overriding composition conformance. The truth value with respect to the property $p = \mathbf{AG}\, f$ is preserved at ex as well as all states in B.

preserving the property $p = \mathbf{AG} f$ when B and E are in conformance. The composition is done via a single exit state ex. Further, E overrides the transition ex-s_3 whose input event is ev in B. B' is the remainder of B after removing all overridden transitions. f^* denotes that f holds at all intermediate states along the computation path. In the figure, within B, $p = \mathbf{AG}\ f$ holds at s_2, ex and s_3 but not at s_1. As $\mathcal{V}_E(ex,p) = \mathcal{V}_{B'}(ex,p) = \mathcal{V}_B(ex,p) = \mathbf{AG}\ f$, B and E conform at ex. While the edge ex-s_3 is removed, the new paths in E together with the remaining computation tree in B' still preserve p at ex directly; and consequently s_2 indirectly. For s_1, its truth value $\mathcal{V}_C(s_1,p) = \neg p$ is preserved as well. On the other hand, s_3 is not affected by E. In this figure, we do not care about the descendant states in E. Thus, E is intentionally left open-end so that the reentry state re is not explicitly displayed. In this part, what E can deliver at ex is important regardless of ex's descendants. The arguments are still valid when the downstream of E converges to the reentry state re.

From Theorem 7, if there is a conformance at all exit states, all truth values with respect to $cl(p)$, surely including p, at base states are preserved. The following corollary is the answer to the problem earlier prescribed in this section - the key of this paper.

COROLLARY 8. *Given a model B and a CTL property p adhered to it, an extension E is attached to B at some interface states. Further, suppose that the assumption function As is proper. E does not violate p inherent to B if B and E conform with each other at all exit states.*

The properness of As is a major part for the soundness of the incremental verification which is mentioned with in Section 5.1. Instead of assuming As's properness, we need to prove it.

4.2 Open Incremental Model Checking

From Corollary 8, the preservation constraints are required at exit states only. Corresponding to an exit state ex, the algorithm to verify a preservation constraint in E can be briefly described as follows:

1. Seeding all reentry states re with $\mathcal{V}_B(re, cl(p))$.
2. Executing the standard CTL model checking procedure in E from re states backward to ex. The formula to check are $\forall \phi \in cl(p)$.
3. At the end of the the model checking procedure, checking if $\mathcal{V}_E(ex, cl(p)) = \mathcal{V}_B(ex, cl(p))$.
4. Repeating the procedure for other exit states.

At the end of the process, if at all exit states, the truth values with respect to $cl(p)$ are matched respectively. B and E are composable.

5. SOUNDNESS AND SCALABILITY ISSUES

5.1 Soundness Issue

In Section 4, the assumption function As is constructed by copying the truth values at reentry states re in B directly. The copying step implicitly assumes that As is proper at all reentry states. For the soundness of OIMC, this section is mainly concerned with proving As's properness, i.e. checking whether Theorem 7 remains valid if the assumption on the properness of As is dropped. Thus, the soundness problem in essence consists of two parts:

1. Proving that As is proper at all reentry states (This is to make sure that the label seeding steps at reentry states are correct). (Soundness Problem 1)
2. Based on the above As's properness, proving that the truth values with respect to $cl(p)$ are preserved at all exit states and hence at all base states. (Soundness Problem 2)

In OIMC, we are only concerned with the interface states between B and E because at these states, the associated

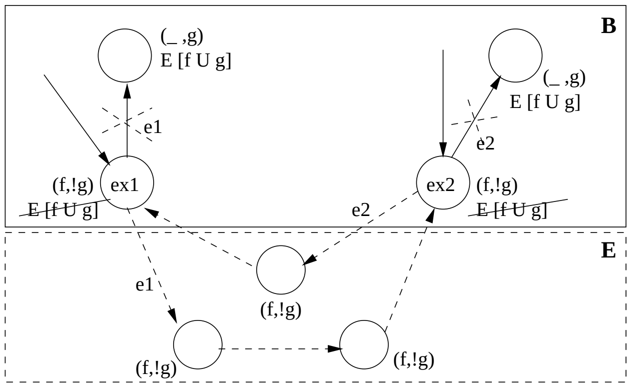


Figure 2: A composition failing to preserve $p = \mathbf{E}\,[f\,\mathbf{U}\,g]$ in case of extension-only cyclic dependency.

computation trees are first to change, if any. Certainly, the property changes at these states then propagate to ascendant states in B. By an observation, if the truth values with respect to $cl(p)$ are preserved at all interface states, the same thing happens at all base states.

Between these interface states are dependency relations due to CTL model checking, i.e. from a state s to any descendant state d of s. If $\mathcal{V}_C(d, cl(p)) \neq \mathcal{V}_B(d, cl(p))$ then it is likely that $\mathcal{V}_C(s, cl(p)) \neq \mathcal{V}_B(s, cl(p))$. These interface states together define a *dependency structure*. The soundness of Theorem 7 after dropping the assumption on As's properness is examined. The results are as follows:

- The theorem is sound if the dependency structure is acyclic, regardless of composition type (additive or overriding).
- The theorem is sound if the composition is additive, regardless of the dependency structure.
- It may fail in some extreme cases of overriding composition with cyclic dependency.

The failing case is illustrated in Figure 2. Two exit states are mutually reentry states in E, namely cyclic dependency between ex_1 and ex_2. Further, E overrides critical paths rooted at ex_1 and ex_2, whose associated input events are e_1 and e_2, with respect to $p = \mathbf{E}\,[f\,\mathbf{U}\,g]$. Initially, $p = \mathbf{E}\,[f\,\mathbf{U}\,g]$ holds at both ex_1 and ex_2. However, after the overriding composition, at these states, the property no longer holds (being crossed out). The assumption function As is not proper at both states. Thus, the result of OIMC within E is not correct due to incorrect label seeding.

5.2 Scalability Issue

So far, we have investigated only one-step extension in which E is attached to B. We are concerned with the application of OIMC for subsequent extensions to C. We consider the n-th version ($C_n = C_{(n-1)} + E_n$) during software evolution as a structure of components B, E_1, E_2, ..., E_n. Here, E_i is the extension component to the $(i-1)$-th evolved version ($C_{(i-1)}$). The initial version is $C_0 = B$. The complexity of verification does not change after adding feature E_n according to Theorem 9 below whose proof is skipped. OIMC maintains its scalability - the incremental characteristic.

THEOREM 9. *Suppose that any pair of base and extension components $C_{(i-1)}$ and E_i respectively conform at all exit states, $i = \overline{1, (n-1)}$. The complexity of the incremental verification for confirming E_n not violating the property p in B only depends on the size of E_n (proportional to the number of states and transitions in E_n).*

6. CONCLUSION

Compared with the earlier work [1], this paper differs in several significant points. They include: a precise and generalized formal model of feature-based software with overriding possibility (1); the soundness problem of OIMC, especially in case of cyclic dependency between interface states, via two sub-problems: the properness of As and properties preservation at exit states (2); an unified condition, $\mathcal{V}_E(ex, cl(p)) = \mathcal{V}_B(ex, cl(p))$, for any legal composition of B and E (3); and the scalability of OIMC (4).

Comparing to the modular verification work [2, 3, 4], there is a fundamental difference in characteristic between those and the work of both [1] and ours. Modular verification in those work are rather closed. Even though it is based on component-based modular model checking, it is not prepared for changes. If a component is added, the whole system of many existing components and the new component is re-checked altogether. On the contrary, the approach in [1] and this paper is incrementally modular. It is also open for future changes. We only check the new system partially within the new component and its interface with the rest of the system. Certainly, this merit comes at the cost of "fixed" conditions at exit states. This "fixed" constraint can cause false negatives to some legal extensions. One of the future work is to relax the conformance condition based on matching truth values with respect to $cl(p)$.

7. REFERENCES

[1] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Proc. Symposium on the Foundations of Software Engineering*, September 2001.

[2] O. Grumberg and D. E. Long. Model checking and modular verification. In *International Conference on Concurrency Theory*, volume 527 of *Lecture Notes of Computer Science*. Springer-Verlag, 1991.

[3] O. Kupferman and M. Y. Vardi. Modular model checking. In *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[4] K. Laster and O. Grumberg. Modular model checking of software. In *Conference on Tools and Algorithms for the Constructions and Analysis of Systems*, 1998.

[5] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N-degrees of separation: Multi-dimensional separation of concerns. In *Proc. ICSE*, pages 109 – 117, 1999.

Toward Structural and Behavioral Analysis For Component Models

Hanh-Missi TRAN
LIFL[*]
missi@lifl.fr

Philippe BEDU
EDF - R&D[†]
philippe.bedu@edf.fr

Laurence DUCHIEN
LIFL[*]
laurence.duchien@lifl.fr

Hai-Quan NGUYEN
EDF - R&D[†]
quan.nguyenhai@edf.fr

Jean PERRIN
EDF - R&D[†]
jean.perrin@edf.fr

ABSTRACT

Component use is becoming more and more prevalent every day. Indeed advantages such as greater productivity represent interesting qualities for the creation of industrial applications. Important efforts are made to help engineers through the improvement of the design and the description of components and through the specification of contracts. However most of the approaches that associate components and contracts propose only run-time checking. In software architecture design, it would be useful to consider contracts when we check the validity of the architecture. Our work takes place in the context of the RM-ODP(Reference Model for Open Distributed Processing) and more precisely the DASIBAO methodology. This paper presents a component-based model associated with several contracts and it describes some verifications that can be performed on them.

Keywords

RM-ODP, structural and behavioral analysis,component-based architecture, ADL, assembly

1. INTRODUCTION

Software architecture is used as a main part of the specification of component-based systems. Reasoning about software architectures improves design, program understanding, and formal analysis. Nowadays most of the software architects tend to agree that the design of sophisticated and software-intensive distributed applications has to be performed according to different viewpoints. As proposed in UML's "4+1" viewpoint model [9], IEEE1471 [3] or ISO RM-ODP (Reference Model for Open Distributed Processing)[2], the separation of concerns during architecture specification helps the designers to manage the complexity of the development process. Viewpoints give some guidance on the models to be produced during a design process as well as the objectives of these models. EDF R&D (Electricity of France Group) has opted for a methodology of architecture design based on RM-ODP, which recommends the separation of stakeholders concerns and proposes five viewpoints. On top of this reference model, EDF is implementing an incremental specification method called DASIBAO [8]. This method defines the different transformations between the viewpoints and particularly between the models carried by each viewpoint. This approach takes all its dimension within the framework of the OMG-MDA (Object Management group Model-Driven Architecture) [1] where designers are expected to produce collections of models from different viewpoints.

However the various models built with this specification method have to guarantee an acceptable level of quality for the system to be created. Our work focuses on the fourth viewpoint which specifies the abstract structure of a model and its deployment in a distributed environment. This work is quite original because it introduces formal analysis abilities in the global architecture specification process based on RM-ODP. This paper presents our approach to model architectures in ODP's systems and a set of tools integrated in the CASBA (Component Assembly Structural & Behavioral Analyzer) system that has been developed jointly by LIFL and EDF R&D. Section 2 introduces our composition model. Then, Section 3 presents some structural elements that can be checked such as the meta-model conformance, the operation signature compatibility and the pre- and post-conditions. Section 4 proposes some behavioral contracts for handling behavioral composition. We propose a description language to specify the behavioral contracts and we check some liveness and safety properties. To avoid an explosion of the number of states, we propose some cuts to reduce behavioral composition. Then we present the results of our verification tool applied to a real application used by EDF. Finally, we conclude and give some perspectives.

[*]Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille
59655 - Villeneuve d'Ascq Cedex, France

[†]Electricité de France - Research Division
1, Avenue du General de Gaulle
92141 - Clamart Cedex, France

2. COMPONENT TYPES MODEL

Our architecture is specified with components. The concept of component used in this architecture is based on the following definition from Szyperski[14]: "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*" The elements needed to describe our component model and the relations between them are taken into account in the metamodel represented in Figure 1.

2.1 Components

Our approach only handles component as component type. Components may have attributes which represent their state. Moreover they are described by provided and required interfaces. A component communicates with other components through its interfaces. In our model, an interface is represented by a port which is associated with a single service. A service is specified by its signature which is composed of a name and of ingoing and outgoing parameters.

2.2 Components assembly

The model does not include explicit connectors between components. Nonetheless, if an architect needs one, he can model it in a component. Communications between components or more precisely between their ports are specified by an assembly link. The semantics of a link corresponds to a synchronous call from the required port to the provided port. The choice regarding a port structure involves that a required port can only be bound to a provided port whereas a provided port can be bound to several required ports.

2.3 Components composition

In order to build a complex architecture, we use **composite components**. They are differentiated from **primitive components** because they contain subcomponents which may be primitive components and also composite components hence a recursive definition of a component. The ports of a composite component are called **delegated ports**. Indeed a call to a provided port of a composite component is forwarded to a provided port of one of its subcomponents. Moreover a call from a required port of a composite component results from the forwarding of a call from a required port from one of its subcomponents.

2.4 Functional contracts

The conditions of validity of a component assembly are improved by associating an **assembly contract** composed of a **pre-condition** and a **post-condition** to each port. These conditions focus on the attributes of the component and on the parameters of the signature. Thus in addition to the verification of the signature compatibility between two linked ports, there is an analysis that checks respectively the compatibility of the precondition and postcondition of a port with the precondition and postcondition of the linked port.

Furthermore **behavioral contracts** are added to the components. These contracts describe the expected behavior of a component and are used to generate the behavior of the components assembly. An appropriate tool has been developed to check some properties on it.

3. STRUCTURAL VERIFICATION

Our tool provides basic verification features common to several ADLs (Architecture Description Languages). It offers a syntactic verification by checking if the components model is in accordance with the metamodel. The metamodel is translated into an XML schema to use the mechanism of validation of XML documents against XML schema.

Another analysis focuses on the assembly links. The previous mechanism verifies that a required port is bound to only one provided port. Moreover there is an analysis on the compatibility of the signatures of bound ports that is based only on their parameters. Indeed we consider that the name of the signature can only be used to identify the port and that it does not give the semantics of the service of the port. The compatibility of a port signature with another uses the notions of covariance and contravariance. These concepts are bound to the paradigm of object-oriented programming. They are distinct mechanisms: *"The so-called contravariance rule correctly captures the subtyping relation. A covariant rule, instead, characterizes the specialization of code"*[6]. The compatibility of port signatures is characterized by three levels. There is a **strong compatibility** of the signature of the required port with the signature of the provided port when there is a contravariance of the ingoing parameters and a covariance of the outgoing parameters. If there is a covariance instead of a contravariance or vice versa, there is only a **weak compatibility**. There is **no compatibility** when there is neither a covariance nor a contravariance between the parameters.

The compatibility of the assembly contracts associated with a required port and a provided port is checked on top of these verifications. Three levels characterize this compatibility. In our model, pre-conditions and post-conditions specify conjunctions and disjunctions of linear inequations. Given *P1* and *P2* two logical formulas and x_1,...,x_n the values in these logical formulas, the strong compatibility can be checked by:
$\forall\,(x_1,...,x_n) \in \{(x_1,...,x_n)/\mathrm{P1}(x_1,...,x_n)=\mathrm{true}\}$, $\mathrm{P1}(x_1,...,x_n) \Rightarrow \mathrm{P2}(x_1,...,x_n)$

Our tool uses CiaoProlog[4], an implementation of Prolog that offers a constraint solver on real values. Because CiaoProlog finds values that solve the constraints, it checks in fact:
$\neg\,(\exists\,(x_1,...,x_n) \in \{(x_1,...,x_n)/\mathrm{P1}(x_1,...,x_n)=\mathrm{true}\}, \neg(\mathrm{P1}(x_1,...,x_n) \Rightarrow \mathrm{P2}(x_1,...,x_n)))$

Given *P1* and *P2* two logical formulas and x_1,...,x_n the values in these logical formulas, the weak compatibility can be checked by:
$\exists\,(x_1,...,x_n) \in \{(x_1,...,x_n)/\mathrm{P1}(x_1,...,x_n)=\mathrm{true}\}$, $\mathrm{P1}(x_1,...,x_n) \cap \mathrm{P2}(x_1,...,x_n) \neq \emptyset$

Our pre-conditions and post-conditions are written in a language very close to Java Modeling Language (JML)[10] which can be used as a design by contract language for Java. For example, instead of using the keyword *result*, the post-condition is specified with the name of the outgoing parameter. More complex pre-conditions and post-conditions could be expressed by the use of boolean expressions on top of the arithmetical ones. The research of solutions could be made by associating the use of a constraint solver and a SAT solver.

The previous verifications correspond to a structural approach. The analysis of the behavioral contracts performs verifications in a dynamic approach.

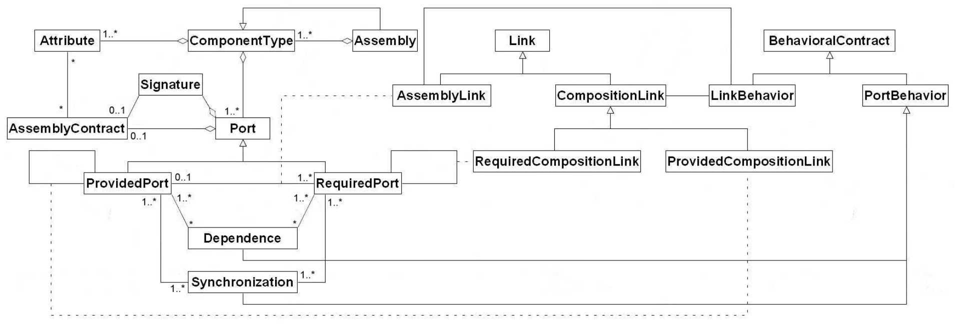


Figure 1: Metamodel of components composition

4. BEHAVIORAL VERIFICATION

In order to check if the system runs as required, the behavior of the component assembly is analyzed. Several parts of the behavior of the component assembly need to be described: they are called **behavioral contracts**. This section first presents the language to specify the behavioral contracts and then the verifications that are performed.

4.1 Component behavior

A component can be viewed as either a black box or a white box. Thus its external behavior can be distinguished from its internal one. The external and internal behaviors are the same in the case of a primitive component. Its behavior is composed of the ways its ports are called. These communications are described in behavioral contracts. We distinguish **dependences** from **synchronizations** as shown in the metamodel (figure 1). A dependence represents a behavioral contract which specifies the internal communications of a component. It consists of the specification of the required ports that are needed by a provided port and the way they are called by the provided port. A synchronization deals with the concurrency issues.

In order to get the internal behavior of a composite component, we add the behavior specified by the composition links, the assembly links between its subcomponents to the behavior of its subcomponents. A communication through either an assembly link or a composition link represents a call from a port to another port.

4.2 Description language

The execution of a service is represented by a sequence of events. Given a sequence S, its execution is translated into S.call $\rightarrow$ S.begin $\rightarrow$ S.end $\rightarrow$ S.return. This means that S is called, then begins, ends and finally returns a value. The operator $\rightarrow$ symbolizes a partial order because the relation is not reflexive but symmetric and transitive. The operators of the description language are based on this operator.

The sequence and alternative operators are both used in the specification of dependences and synchronizations. Let A et B be two services. *A;B* means that A is executed and then B is executed. It is translated into $A.call \rightarrow A.begin \rightarrow A.end \rightarrow A.return \rightarrow B.call \rightarrow B.begin \rightarrow B.end \rightarrow B.return$. $A|B$ means that either A or B is executed. Thus possible traces are either $A.call \rightarrow A.begin \rightarrow A.end \rightarrow A.return$ or $B.call \rightarrow B.begin \rightarrow B.end \rightarrow B.return$.

The other two operators are only used in dependences. The call operator represents the communication from a provided port to the required ports and the parallel operator represents parallel composition of services. Let A, B and C be three services. $A\{B\}$ means that the execution of A is composed of the execution of B. It is translated into $A.call \rightarrow A.begin \rightarrow B.call \rightarrow B.begin \rightarrow B.end \rightarrow B.return \rightarrow A.end \rightarrow A.return$. $A\{B||C\}$ means that the execution of A is composed of the interleaving execution of B and C.

The last operator ***** is used to specify that there can be an undetermined number of executions of a service or that this service is not executed. For example, if we specify $A*|B$, it means that either the service A is executed several times or not at all or the service B is executed. This operation may be used to describe loops.

The null sequence symbolized by $\emptyset$ comes in addition to these operators. It indicates that no service is executed.

4.3 Behavior verification

In order to analyze on the behavior of a component model, we transform the behavioral elements into FSP (Finite State Process)[11] processes. We operate this translation first by generating a behavior formed of the behaviors of the composition and assembly links and the dependences and then by making a composition of it with the synchronizations. The analysis uses a verification tool for concurrent systems, named LTSA (Labelled Transition System Analyser)[12], which supports FSP and a LTL (Linear Temporal Logic) checker to check safety and liveness properties such as deadlocks or absence of reachability.

This approach works well when applied on small architectures. However large architectures are represented by complex hierarchical component structure and the analysis of the behavior of such architectures may lead to state explosion problems. Thus the behavior of composite components has to be minimized. This approach is close to the TRACTA approach[7]. Both are based on FSP but because the behavioral contracts are described with our own language, the produced FSP specifications do not use all the features of this language.

4.4 Behavior minimization

The first way to obtain the external behavior from the internal behavior of a composite component is to use the FSP minimization operator. However the FSP processes describe a composite component internal behavior. Moreover each time an analysis uses its external behavior, the internal behavior of each subcomponent is minimized again. To address this problem, we have decided to perform this minimization with our description language. This operation

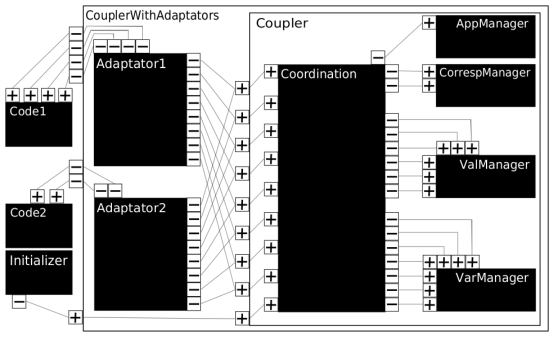


Figure 2: Architecture of the CALCIUM coupler

aims at producing the behavior of a composite component as if it were a primitive component. Thus this behavior is composed of dependences and synchronizations.

The transitivity of the operator $\rightarrow$ is the basis of the reduction of an internal behavior into an external one. Indeed the calling operator is based on the operator $\rightarrow$ and the behaviors of the dependences and the assembly and delegation links use the calling operator. The beginning of the minimization consists in transforming the ports that do not call any other ports into the null sequence. Then the transitivity of the calling operator allows the behavior to be reduced.

The minimization of synchronizations may lead to the loss of information on the behavior. Because our language is based on services and not on events, it is currently not tractable enough to realize a minimization on it. Our verification tool uses the minimization based on our behavior language but only minor changes would be needed in order to use the minimization feature in FSP.

5. RESULTS

The most significant example verified by our verification tool is an existing application from EDF. The figure 2 gives an idea of the complexity of the architecture. Required and provided ports are symbolized respectively by - and +. The example represents the use of a generic coupler of scientific code named CALCIUM[5] which first version was developped in 1994. This coupler is used to study the interactions between codes of different domains in physics. It manages the exchange of values between the codes.

Several assembly and behavioral contracts are added to the architecture shown in the figure 2. The structural verification takes some time to be performed, due to the number of compatibilities of assembly contracts to be checked. The behavioral analysis can not be done because of the explosion of the number of states. For example, the potential state space for the behavior of the component *Coordination* is wide of 2^{170} states and an usual desktop computer does not have enough memory to handle it.

6. CONCLUSION AND FUTURE WORK

Our component model is used to specify functional architectures in the computational viewpoint. We integrate contracts into the model to carry out strong verifications on the components model. Assembly contracts add conditions to the validity of components assembly. Moreover behavioral contracts specify the communications within a component. Furthermore the use of behavior minimization associates hierarchical composition with behavioral composition in our architecture. The verifications we describe are implemented by tools in CASBA. These tools can be called from a graphical interface integrated in the modelling tool ArgoUML [13] which allows the design and the analysis of component model based on our metamodel.

The structural and behavioral verifications of our component model represent only a part in our approach of architecture building. Indeed we now need to specify how to go from a computational viewpoint, which is the fourth viewpoint in RM-ODP, to an engineering viewpoint which is its last one. Thus our goal is to integrate non functional requirements in functional architectures in order to produce technical architectures. This leads us to propose a new concept called **architectural figure** inspired by previous works on the reuse of architectural systems such as the architectural patterns and styles. This architectural figure represents a component model slightly different from our previous model which allows us to transform the functional architecture into a technical one. Thus our future work will focus on providing tools to describe figures associated with quality attributes, to realize the transformation of functional architectures into technical ones with architectural figures and to analyse quality aspects of the produced architectures.

7. REFERENCES

[1] www.omg.org/mda.

[2] Iso/iec, open distributed processing reference model - parts 1, 2, 3, 4. ISO 10746 or ITU-T X.901, 1995.

[3] Recommended practice for architectural description. IEEE Standard P1471, 2000.

[4] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. Lopez, and G. Puebla. The ciao prolog system: A next generation logic programming environment. Technical Report 3/97.1, CLIP, April 2004.

[5] C. Caremoli and J.-Y. Berthou. *CALCIUM V2: Guide d'utilisation.*

[6] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[7] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures.* PhD thesis, Imperial College of Science Technology and Medecine, University of London, March 1999.

[8] A. W. Group. Dasibao: Methodology for architecturing odp systems. Technical report, EDF R&D, 2002.

[9] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(5):42–50, November 1995.

[10] G. Leavens and Y. Cheon. Design by contract with jml. Draft paper, March 2004.

[11] J. Maggee and J. Kramer. *Concurrency - State Models and Java Program.* John Wiley & Sons, 1999.

[12] J. Maggee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA1)*, 1999.

[13] J. E. Robbins. *Cognitive Support Features for Software Development Tools.* PhD thesis, University of California, Irvine, 1999.

[14] C. Szyperski. *Component Software - Beyond Object Programming.* 1998.