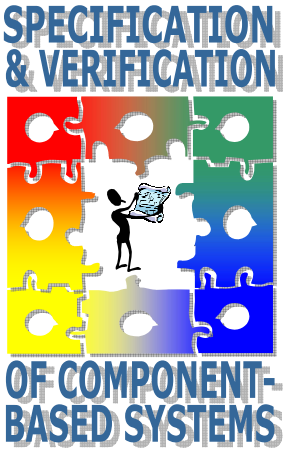
# SAVCBS 2005
# Specification and Verification of Component-Based Systems




*ESEC/FSE'05*
*5th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*
*Lisbon, Portugal*
*September 5-9, 2005*

Technical Report #05-19, Department of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, IA 50011-1041, USA

# SAVCBS 2005 PROCEEDINGS

## Specification and Verification of Component-Based Systems

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

September 5-6, 2005
Lisbon, Portugal

Workshop at ESEC/FSE'05
5th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering

# SAVCBS 2005
# TABLE OF CONTENTS

# SAVCBS 2005
# 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:**
Gary T. Leavens (Iowa State University), Program Committee Chair
Jonathan Aldrich (Carnegie Mellon University)
Mike Barnett (Microsoft Research)
Betty H. C. Cheng (Michigan State University)
Stephen H. Edwards (Virginia Tech)
Cormac Flanagan (University of California at Santa Cruz)
Dimitra Giannakopoulou (RIACS /NASA Ames Research Center)
Gerard J. Holzmann (NASA/JPL)
Joe Kiniry (University College Dublin)
K. Rustan M. Leino (Microsoft Research)
Jeff Magee (Imperial College, London)
Peter Müller (ETH Zürich)
Corina Pasareanu (NASA Ames Research Center)
Erik Poll (Raboud University Nijmegen)
Andreas Rausch (University of Kaiserslautern)
Robby (Kansas State University)
Wolfram Schulte (Microsoft Research)
Natalia Sharygina (Carnegie Mellon University, SEI)
Murali Sitaraman (Clemson University)

**Sponsors:**

# SAVCBS 2005
# WORKSHOP INTRODUCTION

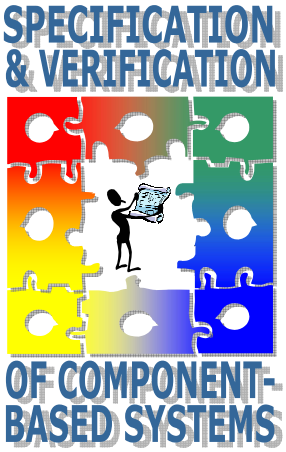
This workshop is concerned with how formal (i.e., mathematical) techniques can be or should be used to establish a suitable foundation for the specification and verification of component-based systems. Component-based systems are a growing concern for the software engineering community. Specification 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 2005
# INVITED PRESENTATION



# Thread-Modular Verification by Context Inference

Ranjit Jhala*
Computer Science Department
Jacobs School of Engineering
University of California, San Diego, USA
jhala@cs.ucsd.edu

## ABSTRACT

Multithreaded programs are notoriously difficult to verify: the interleaving of concurrent threads causes an exponential explosion of the control state, and if threads can be dynamically created, the number of control states is unbounded. A classical thread-modular approach to verification is to consider the system as composed of a "main" thread and a context which is an abstraction of all the other "environment" threads of the system, and to then verify that (a) that this composed system is safe ("assume"), and, (b) that the context is indeed a sound abstraction ("guarantee"). Once the appropriate context has been divined, the above checks can be discharged by existing methods. Previously, such a context had to be provided manually, and if the given context is imprecise, then either check may fail leaving us with no information about whether the system is safe or not.

We show how to automate the thread-modular approach by: (a) finding a model for the context that is simultaneously (i) abstract enough to permit efficient checking and (ii) precise enough to preclude false positives as well as yield real error traces when the checks fail, and (b) showing how to infer such a context automatically. We give a novel way to construct stateful contexts, by representing individual environment threads as abstract finite state machines, and tracking arbitrarily many threads by counting the number of threads at each abstract state. We infer stateful contexts by iteratively weakening the abstract reachability analysis used for sequential programs, until an appropriate context is obtained.

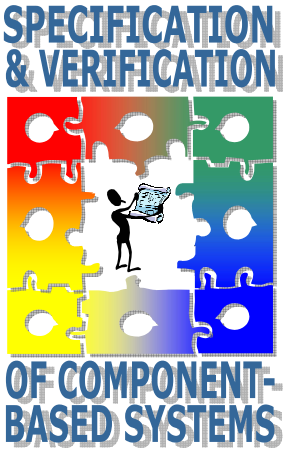
We have implemented this algorithm in our C model checker BLAST, and used it to look for race conditions on networked embedded systems applications written in NesC, which use non-trivial synchronization idioms, that cause previous, imprecise analyses to race false alarms. We were able to find potential races in some cases and prove the absence of races in others.

*Joint work with Thomas A. Henzinger(EPFL, Switzerland) and Rupak Majumdar (UCLA, USA).

## Biography

Ranjit Jhala is an Assistant Professor of Computer Science at the University of California, San Diego. Previously, he received a Ph.D. in Computer Science from the University of California, Berkeley, and before that, a B. Tech in Computer Science and Engineering from the Indian Institute of Technology, Delhi. He is interested in Programming Languages and Software Engineering, more specifically, in techniques for building reliable computer systems. The majority of his work has been on the BLAST software verification system which draws from, combines and contributes to techniques for Automated Deduction, Program Analysis and Model Checking.

# SAVCBS 2005 PAPERS



# Assume-Guarantee Testing

Colin Blundell
Dept. of Comp. and Inf. Science
University of Pennsylvania
Philadelphia, PA 19104, USA
blundell@cis.upenn.edu

Dimitra Giannakopoulou
RIACS/NASA Ames
NASA Ames Research Center
Moffett Field, CA 94035-1000, USA
dimitra@email.arc.nasa.gov

Corina S. Păsăreanu
QSS/NASA Ames
NASA Ames Research Center
Moffett Field, CA 94035-1000, USA
pcorina@email.arc.nasa.gov

## ABSTRACT

Verification techniques for component-based systems should ideally be able to predict properties of the assembled system through analysis of individual components before assembly. This work introduces such a modular technique in the context of testing. Assume-guarantee testing relies on the (automated) decomposition of key system-level requirements into local component requirements at design time. Developers can verify the local requirements by checking components in isolation; failed checks may indicate violations of system requirements, while valid traces from different components compose via the assume-guarantee proof rule to potentially provide system coverage. These local requirements also form the foundation of a technique for efficient predictive testing of assembled systems: given a correct system run, this technique can predict violations by alternative system runs without constructing those runs. We discuss the application of our approach to testing a multi-threaded NASA application, where we treat threads as components.

## Keywords

verification, testing, assume-guarantee reasoning, predictive analysis

## 1. INTRODUCTION

As software systems continue to grow in size and complexity, it is becoming common for developers to assemble them from new or reused components potentially developed by different parties. For these systems, it is important to have verification techniques that are modular as well, since verification is often the dominant software production cost. Developers could use such techniques to avoid expensive verification of assembled systems, instead performing verification primarily on individual components. Unfortunately, the task of extracting useful results from verification of components in isolation is often difficult: first, developing environments that will appropriately exercise individual components is challenging and time-consuming, and second, inferring system properties from results of local verification is typically nontrivial. The growing popularity of component-based systems makes it important for verification researchers to investigate these challenges.

Assume-guarantee reasoning is a technique that has long held promise for modular verification. This technique is a "divide-and-conquer" approach that infers global system properties by checking individual components in isolation [4, 13, 15, 17]. In its simplest form, it checks whether a component $M$ guarantees a property $P$ when it is part of a system that satisfies an assumption $A$, and checks that the remaining components in the system ($M$'s environment) satisfy $A$. Extensions that use an assumption for each component in the system also exist. Our previous work developed techniques that *automatically* generate assumptions for performing assume-guarantee model checking at the design level [2, 5, 9], ameliorating the often difficult challenge of finding an appropriate assumption.

While design verification is important, it is also necessary to verify that an implementation preserves the design's correctness. For this purpose, we have also previously developed a methodology that uses the assumptions created at the design level to model check source code in an assume-guarantee style [10]; with this methodology, it is possible to verify source code one component at a time. Hence, this technique has the potential to meet the challenges of component-based verification.

Unfortunately, despite the increased scalability that one can achieve by using assume-guarantee techniques in model checking, it remains a difficult task in the hands of experts to make the technique scale to the size of industrial systems. Furthermore, model checkers do not exist for many languages commonly used in industry. This work explores the benefits of assume-guarantee reasoning for testing, which is still the predominant industrial verification technique. We have developed *assume-guarantee testing*, which reuses properties, assumptions, and proof rules from design-level assume-guarantee verification to enhance both unit testing and whole-system testing. The contributions of assume-guarantee testing are as follows:

1. During unit testing, assume-guarantee testing has the potential to obtain *system coverage* and detect *system-level errors*. Our approach applies assume-guarantee reasoning to component test traces, using assumptions as environ-

ments to drive individual components. This process provides guarantees on trace *compositions* that are analogous to the guarantees obtained by assume-guarantee model checking. Hence, the technique can infer that a (potentially large) set of system traces satisfies a global property by checking traces of components in isolation against assume-guarantee pairs. Moreover, component test traces that fail their assume-guarantee premises may uncover *system-level* violations. Assumptions restrict the context of the components, thus reducing the number of false positives obtained by verification (*i.e.*, errors that will never exhibit themselves in the context of the particular system in which the component will be introduced). As a result, the likelihood that a failed local check corresponds to a system-level error is higher. Early error detection is desirable, as it is well established that errors discovered earlier in the development phase are easier and cheaper to fix.

2. During whole-system testing, assume-guarantee testing has the potential to efficiently detect bugs and provide coverage. In this context, our approach projects system traces onto individual components, and applies assume-guarantee reasoning to the projections. This technique is an efficient means of *predictive testing*. Predictive testing detects the existence of bad traces from good traces [19]. It exploits the insight that one can reorder independent events from a trace to obtain different legal traces. Typically, predictive testing techniques discover these alternative traces by composing independent events in different orders. Our technique uses assume-guarantee reasoning to obtain results about the alternative interleavings *without* explicitly exploring them, and thus is potentially more efficient.

We experimented with our assume-guarantee testing framework in the context of the Eagle runtime analysis tool [3], and applied our approach to a NASA software system also used in the demonstration of our design-level assume-guarantee reasoning techniques. In the analysis of a specific property ($P$) during these experiments, we found a discrepancy between one of the components and the design that it implements. This discrepancy does not cause the system to violate $P$; monolithic model checking would therefore not have detected it.

The remainder of the paper is organized as follows. We first provide some background in Section 2, followed by a discussion of our assume-guarantee testing approach and its advantages in Section 3. Section 4 describes the experience and results obtained by the application of our techniques to a NASA system. Finally, Section 5 presents related work and Section 6 concludes the paper.

## 2. BACKGROUND

**LTSs.** At design level, this work uses Labeled Transition Systems (LTSs) to model the behavior of communicating components. Let $\mathcal{A}ct$ be the universal set of observable actions and let $\tau$ denote a local action *unobservable* to a component's environment. An LTS $M$ is a quadruple $\langle Q, \alpha M, \delta, q0 \rangle$ where:

- $Q$ is a non-empty finite set of states
- $\alpha M \subseteq \mathcal{A}ct$ is a finite set of observable actions called the *alphabet* of $M$
- $\delta \subseteq Q \times \alpha M \cup \{\tau\} \times Q$ is a transition relation
- $q0 \in Q$ is the initial state

Let $M = \langle Q, \alpha M, \delta, q0 \rangle$ and $M' = \langle Q', \alpha M', \delta', q0' \rangle$. We say that $M$ *transits* into $M'$ with action $a$, denoted $M \xrightarrow{a} M'$, if and only if $(q0, a, q0') \in \delta$ and $\alpha M = \alpha M'$ and $\delta = \delta'$.

**Traces.** A *trace* $t$ of an LTS $M$ is a sequence of observable actions that $M$ can perform starting at its initial state. For $\Sigma \subseteq \mathcal{A}ct$, we use $t{\restriction}\Sigma$ to denote the trace obtained by removing from $t$ all occurrences of actions $a \notin \Sigma$. The set of all traces of $M$ is called the *language* of $M$, denoted $\mathcal{L}(M)$.

Let $t = \langle a_1, a_2, \ldots, a_n \rangle$ be a finite trace of some LTS $M$. We use $[t]$ to denote the LTS $M_t = \langle Q, \alpha M, \delta, q0 \rangle$ with $Q = \{q_0, q_1, \ldots, q_n\}$, and $\delta = \{(q_{i-1}, a_i, q_i)\}$, where $1 \leq i \leq n$.

**Parallel Composition.** The parallel composition operator $\parallel$ is commutative and associative. It combines the behavior of two components by synchronizing the actions common to their alphabets and interleaving the remaining actions. Formally, let $M_1 = \langle Q_1, \alpha M_1, \delta_1, q0_1 \rangle$ and $M_2 = \langle Q_2, \alpha M_2, \delta_2, q0_2 \rangle$ be two LTSs. Then $M_1 \parallel M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q0 \rangle$, where $Q = Q_1 \times Q_2$, $q0 = (q0_1, q0_2)$, $\alpha M = \alpha M_1 \cup \alpha M_2$, and $\delta$ is defined as follows, where $a$ is either an observable action or $\tau$ (note that commutativity implies the symmetric rules):

$$\frac{M_1 \xrightarrow{a} M_1',\ a \notin \alpha M_2}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2}$$

$$\frac{M_1 \xrightarrow{a} M_1',\ M_2 \xrightarrow{a} M_2',\ a \neq \tau}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2'}$$

**Properties and Satisfiability.** A property is specified as an LTS $P$, whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors over $\alpha P$. An LTS $M$ satisfies $P$, denoted as $M \models P$, if and only if $\forall t \in \mathcal{L}(M).t{\restriction}\alpha P \in \mathcal{L}(P)$.

**Assume-guarantee Triples.** In the assume-guarantee paradigm a formula is a triple $\langle A \rangle\ M\ \langle P \rangle$, where $M$ is a component, $P$ is a property and $A$ is an assumption about $M$'s environment [17]. The formula is true if whenever $M$ is part of a system satisfying $A$, then the system guarantees $P$. At design level in our framework, the user expresses all of $A, M, P$ as LTSs.

**Assume-guarantee Reasoning.** Consider for simplicity a system that is made up of components $M_1$ and $M_2$. The aim of assume-guarantee reasoning is to establish $M_1 \parallel M_2 \models P$ without composing $M_1$ with $M_2$. For this purpose, the simplest proof rule consists of showing that the following two premises hold: $\langle A \rangle\ M_1\ \langle P \rangle$ and $\langle true \rangle\ M_2\ \langle A \rangle$. From these, the rule infers that $\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle$ also holds. Note that for this rule to be useful, the assumption must be more abstract than $M_2$, but still reflect $M_2$'s behavior. Additionally, an appropriate assumption for the rule needs to be strong enough for $M_1$ to satisfy $P$. Unfortunately, it is often difficult to find such an assumption.

Our previous work developed frameworks that compute assumptions automatically for finite-state models and safety properties expressed as LTSs. More specifically, Giannakopoulou et al. [9] present an approach to synthesizing the assumption that a component needs to make about its environment for a given property to hold. The assumption produced is the *weakest*, that is, it restricts the environment no more and no less than is necessary for the component to satisfy the property. Barringer et al. [2] and Cobleigh et al. [5] use a learning algorithm to compute assumptions in an *incremental* fashion in the context of simple and symmetric assume-guarantee rules, respectively.

# 3. ASSUME-GUARANTEE TESTING

This section describes our methodology for using the artifacts of the design-level analysis, *i.e.* models, properties and generated assumptions, for testing the implementation of a software system. This work assumes a top-down software development process, where one creates and debugs design models and then uses these models to guide the development of source code, possibly by (semi-) automatic code synthesis.

Our approach is illustrated by Figure 1. Consider a system that consists of two (finite-state) design models $M_1$ and $M_2$, and a global safety property $P$. Assume that the property holds at the design level (if the property does not hold, developers can use the feedback provided by the verification framework to correct the models). The assume-guarantee framework that is used to check that the property holds will also generate an assumption $A$ that is strong enough for $M_1$ to satisfy $P$ but weak enough to be discharged by $M_2$ (*i.e.* $\langle A \rangle\ M_1\ \langle P \rangle$ and $\langle true \rangle\ M_2\ \langle A \rangle$ both hold), as described in Section 2.

Once design-level verification establishes the property, it is necessary to verify that the property holds at the implementation level, *i.e.* that $C_1 \parallel C_2 \models P$. This work assumes that each component implements one of the design models, *e.g.* components $C_1$ and $C_2$ implement $M_1$ and $M_2$, respectively, in Figure 1. We propose *assume-guarantee testing* as a way of checking $C_1 \parallel C_2 \models P$. This consists of producing test traces by each of the two components, and checking these traces against the respective assume-guarantee premises applied at the design level. If each of the checks succeeds, then the proof rule guarantees that the composition of the traces satisfies the property $P$.

We illustrate assume-guarantee testing through a simple example. Consider a communication channel that has two components, designs $M_1$ and $M_2$ and corresponding code $C_1$ and $C_2$ (see Figure 2). Property $P$ describes all legal executions of the channel in terms of events $\{in, out\}$; it essentially states that for a trace to be legal, *in* must occur in the trace before any occurrence of *out*. Figure 2 also shows the assumption $A$ that design-level analysis of $M_1 \parallel M_2$ generates (see Section 2). Note that although $M_1 \parallel M_2 \models P$, $C_1 \parallel C_2$ does not. Testing $C_1$ and $C_2$ in isolation may produce the traces $t_1$ and $t_2$ (respectively) that Figure 3 (left) shows. Checking $\langle true \rangle\ t_2\ \langle A \rangle$ during assume-guarantee testing will detect the fact that $t_2$ violates the assumption $A$ and will thus uncover the problem with the implementation. Assume now that the developers do not use assume-guarantee testing, but rather test the assembled

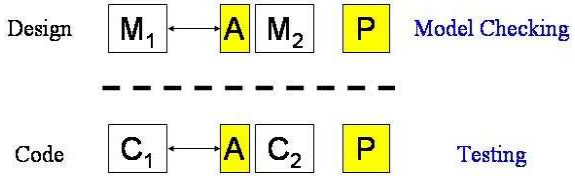



**Figure 1: Design and code level analysis**

system (we call the latter monolithic testing). The system might first produce the first two traces illustrated in Figure 3 (right). These traces satisfy the property, which could lead the developers to mistakenly believe that the system is correct. They may even achieve some coverage criterion without detecting the bug, as discussed later in this section.

In summary, assume-guarantee testing can obtain results on all interleavings of two individual component traces simply by checking each against the appropriate assume-guarantee premise. In the context of our example, checking $t_1$ and $t_2$ corresponds to checking all four traces illustrated in Figure 3 (right).

While our example illustrates the benefits of assume-guarantee reasoning for unit testing, similar benefits apply to testing of assembled systems. When the system is assembled, the testing framework uses assume-guarantee reasoning to conduct analysis that can efficiently predict, based on correct system runs, violations by alternative system runs. We discuss both flavors of assume-guarantee testing in more detail below.

## 3.1 Assume-Guarantee Component Testing

The first step in assume-guarantee component testing involves the implementation of 1) $U_A$ for $C_1$, where $U_A$ encodes $C_1$'s universal environment restricted by assumption $A$, and 2) the universal environment $U$ for $C_2$. The universal environment for a component may exercise any service that the component provides in any order, and may provide or refuse any service that the component requires. The next step is to execute $C_1$ in $U_A$ and $C_2$ in $U$ to produce sets of traces $T_1$ and $T_2$ respectively. The technique then performs assume-guarantee reasoning, checking each trace $t_1 \in T_1$ against $P$ and each trace $t_2 \in T_2$ against $A$. If either of these checks fail (as in Figure 3), this is an indication that there is an incompatibility between the models and the implementations, which the developers can then correct. If all these tests succeed, then the assume-guarantee rule implies that $[t_1]||[t_2] \models P$, for all $t_1 \in T_1$ and $t_2 \in T_2$.

Using this approach, one can check system correctness through local tests of components. It is possible to perform assume-guarantee testing as soon as each component becomes "code complete", and *before* assembling an executable system or even implementing other components. A secondary advantage of this approach is that it ameliorates the problem of choosing appropriate testing environments for components in isolation. This is a difficult problem in general, as finding an environment that is both general enough to fully exercise the component under testing and specific enough to avoid many false positives is usually a time-consuming iterative process. Here, this problem is reduced to that of correctly

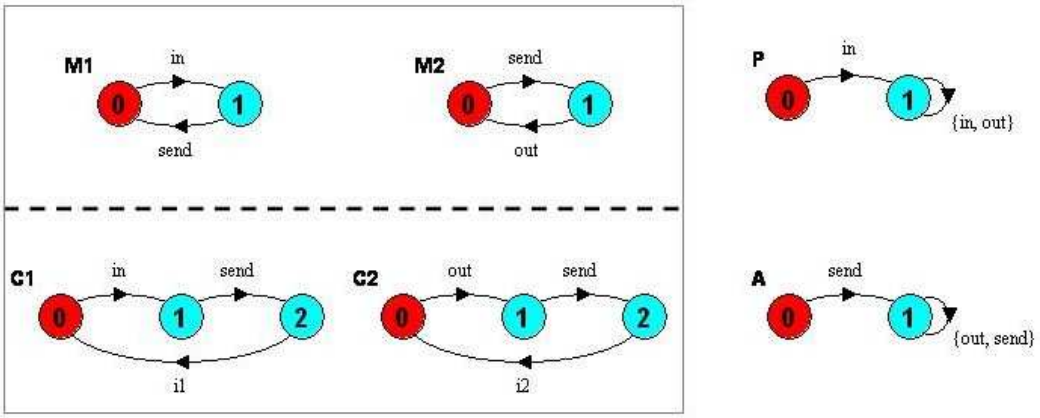



**Figure 2: Assume-guarantee testing**

implementing $U_A$ and $U$. Note that alternatively, one may wish to check preservation of properties by checking directly that each implemented component refines its model. In our experience, for well-designed systems, the interfaces between components are small, and the generated assumptions are much smaller than the component models. Therefore, it is more efficient to check the assumptions than to check refinement directly. Finally, note that, when checking components in isolation, one has more control over the component interface (since it is exercised directly rather than through some other component). As a result, it is both easier to reproduce problematic behavior, and to exercise more traces for constructing sets $T_1$ and $T_2$.

**Coverage.** Unlike model checking, testing is not an exhaustive verification technique. As a result, it is possible for defects to escape despite testing. For this reason, software quality assurance engineers and researchers on software testing have traditionally associated the notion of *coverage* with the technique. Coverage criteria dictate how much testing is "enough" testing. A typical coverage criterion that works on the structure of the code is "node" coverage, which requires that the tests performed cover all nodes in the control flow graph of a system's implementation. Assume that in our example our coverage criterion is node coverage for $C_1$ and $C_2$. Then $t_1$ and $t_2$ in Figure 3 (left) together achieve 100% coverage. Similarly, the first trace of the assembled system in Figure 3 (right) achieves 100% node coverage. It is therefore obvious that assume-guarantee testing has the potential of checking more behaviors of the system even when it achieves the same amount of coverage. This example also reflects the fact that traditional coverage criteria are often not appropriate for concurrent or component-based systems, which is an area of active research. One could also measure coverage by the number of behaviors or paths through the system that are exercised. The question of what benefits assume-guarantee reasoning can provide in such a context is open research.

**Discussion.** As stated above, our hope is that by checking individual traces of components, the technique covers multiple traces of the assembled system. Unfortunately, this is not always true, due to the problem of *incompatible traces*, which are traces that do not execute the same shared events

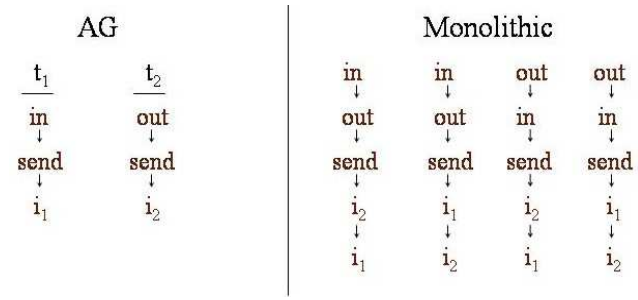



**Figure 3: Discovering bugs with fewer tests**

in the same order. These traces are from different execution paths, and thus give the empty trace on composition. For example, suppose that the first event in $t_1$ is a function call on the procedure `foo` in $C_1$, while the first event in $t_2$ is a function call on the procedure `bar` in $C_2$; these traces executed on different paths and are incompatible. Thus, assume-guarantee testing faces the question of producing compatible traces during component testing. One potential way to guarantee that $T_1$ and $T_2$ contain compatible traces is to use the component models as a coverage metric when generating traces in $T_1$ and $T_2$, and require that each set of traces cover certain sequences of shared events in the models.

## 3.2 Predictive Analysis on Assembled Systems

Assume-guarantee testing can also be a mechanism for predictive testing of component assemblies. Assume-guarantee testing for predictive analysis has the following steps:

- Obtain a system trace $t$ (by running $C_1||C_2$).
- Project the trace on the alphabets of each component; obtain $t_1 = t{\restriction}\alpha C_1$ and $t_2 = t{\restriction}\alpha C_2$.
- Use the design-level assumption to study the composition of the projections; *i.e.* check that $\langle A \rangle\ [t_1]\ \langle P \rangle$ and $\langle true \rangle\ [t_2]\ \langle A \rangle$ hold, using model checking.

The approach is illustrated in Figure 4: on the right, we show a trace $t$ of $C_1||C_2$ that does not violate the property.

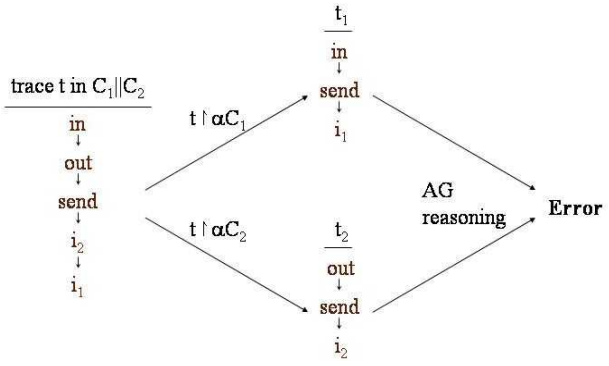



**Figure 4: Predictive analysis**

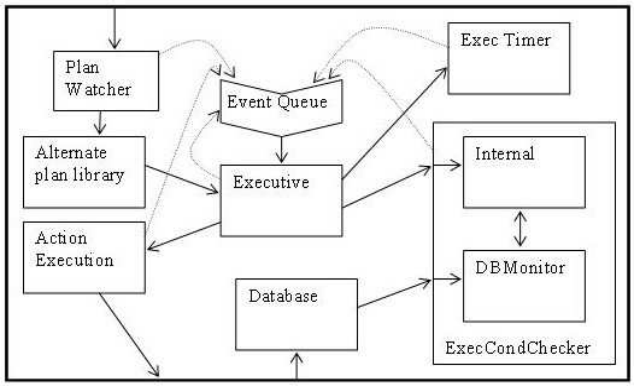



**Figure 5: The Executive of the K9 Mars Rover**

On the left, we show the projections $t_1$ and $t_2$. Note that $\langle true \rangle$ $[t_2]$ $\langle A \rangle$ does not hold, hence from a single "good" trace the methodology has been able to show that $C_1||C_2$ violates the property. Using the design-level assumption to analyze the projections is more efficient than composing the projections and checking that the composition satisfies the property (as is performed by other predictive analysis techniques) as long as the assumption is small; in our experience, this is often the case [9].

An alternative approach is to generate the assumption directly from the projected trace $t_1$, and then test that $t_2$ satisfies this assumption. This approach is a way to do assume-guarantee predictive testing in a system where there are no design-level models. However, it may not be practical to generate a new assumption for each trace; we plan to experiment with this approach in the future.

**Discussion.** It is desirable to use assume-guarantee predictive testing as a means of efficiently generating system coverage. This technique does not suffer from incompatible traces, as the two projected traces occur in the same system trace and are thus guaranteed to be compatible. However, to gain the full benefits of assume-guarantee testing in this context, trace generation should take into account the results of predictive analysis. For example, suppose that trace generation produces a trace $t$, projected onto $t_1$ and $t_2$. Assume-guarantee testing proves that $[t_1]||[t_2] \models P$. Further trace generation should avoid traces in $[t_1]||[t_2]$ since these are covered by the assume-guarantee checking of $t_1$ and $t_2$. Again, one possible way to ensure avoidance of such redundant traces is to use the design-level model as a coverage metric; two traces that have different sequences of shared events through the model will project onto different traces. Test input generation techniques could also be useful for this purpose. This topic is a subject of future work.

# 4. EXPERIENCE

Our case study is the planetary rover controller K9, and in particular its executive subsystem, developed at NASA Ames Research Center. We performed this study in the context of an ongoing collaboration with the developers of the rover, in which we have performed verification *during* development to increase the quality of the design and implementation of the system. Below we describe the rover executive, our design-level analysis, how we used the assumptions generated by this analysis to conduct assume-guarantee testing, and results of this testing.

## 4.1 K9 Rover Executive Subsystem

The executive sub-system commands the rover through the use of high-level *plans*, which it interprets and executes in the context of the execution environment. The executive monitors execution of plan actions and performs appropriate responses and cleanup when execution fails.

The executive implementation is a multi-threaded system (see Figure 5), made up of a main coordinating component named *Executive*, components for monitoring temporal conditions *ExecTimerChecker* and state conditions *ExecCondChecker*, and an *ActionExecution* thread that is responsible for issuing the commands (actions) to the rover. The communication between different components (threads) is made through an *EventQueue*. The implementation has 35K lines of C++ code and it uses the POSIX thread library.

## 4.2 Design-level Analysis

We previously developed detailed design models for the executive subsystem [9]. We then checked these models in an assume-guarantee manner for several properties specified by the developers. Model checking of the design models uncovered a number of synchronization problems such as deadlocks and data races, which we then fixed in collaboration with the developers. After finishing this process, for each property we had an assumption on one of the components stating what behavior was needed of it for the property to hold of the entire system.

## 4.3 Assume-guarantee Testing Framework

We have developed a framework that uses the assumptions and properties built during the design-level analysis for the assume-guarantee testing of the executive implementation. In order to apply assume-guarantee testing, we broke up the implementation into two components, with the *Executive* thread, the *EventQueue* and the *ActionExecution* thread on one side ($M_1$), and the *ExecCondChecker* thread and the other threads on the other side ($M_2$), as shown in Figure 5.

To test the components in isolation, we generated *environments* that encode the design-level assumptions (as described in Section 3). We implemented each environment as

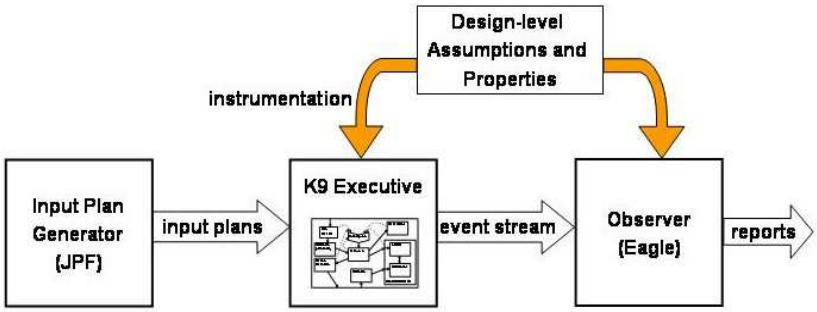



**Figure 6: Testing Environment**

a thread running a state machine (the respective design-level assumption) that executes in an infinite loop. In each iteration of the loop, the environment makes a random choice to perform an "active" event (such as calling a component function) that is enabled in the current state; the state machine then makes the appropriate transition. To make function calls on the component, we provided dummy values of irrelevant arguments (while ensuring that these dummy values did not cause any loss of relevant information). The environment implementations also provide stubs for the external functions that the component under testing calls; when called, these functions cause state machine transitions.

The methodology uses the Eagle run-time monitoring framework [3] to check that the components conform with the assume-guarantee pairs. Eagle is an advanced testing framework that provides means for constructing test oracles that examine the internal computational status of the analyzed system. For run-time monitoring, the user instruments the program to emit events that provide a trace of the running system. Eagle then checks to see whether the current trace conforms to formalized requirements, stated as temporal logic assertions or finite-state automata.

For our experiments, we instrumented (by hand) the code of the executive components to emit events that appear in the design-level assumptions and properties. We also (automatically) translated these assumptions and properties into Eagle monitors.

Note that in order to run the executive system (or its components), the user needs to provide an input plan and an environment simulating the actual rover hardware. For our assume-guarantee testing experiments, the hardware environment was stubbed out. For plan input generation, we built upon our previous work, which combines model checking and symbolic execution for specification-based test input generation [21]. To generate test input plans, we encoded the plan language grammar as a nondeterministic specification. Running model checking on this model generates hundreds of input plans in a few seconds.

We have integrated the above techniques to perform assume-guarantee testing on the executive (see Figure 6). We first instrument the code and generate Eagle monitors encoding design-level assumptions and properties. The framework generates a set of test input plans, a script runs the executive on each plan and it calls Eagle to monitor the generated run-time traces. The user can choose to perform a whole-program (monolithic) analysis or to perform assume-guarantee reasoning.

## 4.4 Results

We ran several experiments (according to different input plans). For one property, we found a discrepancy between the implementation and the models. The property ($P$) states that the *ExecCondChecker* should not push events onto the *EventQueue* unless the *Executive* has sent the *ExecCondChecker* conditions to check. The design-level assumption ($A$) on the *ExecCondChecker* states that the property will hold as long as the *ExecCondChecker* sets a flag variable to 1 before pushing events, since these assignments only happen in response to the *Executive* sending conditions.

To check this property, we generated an environment that drives component $C_1$ (which contains the *Executive*) according to assumption $A$. We instrumented $C_1$ to emit relevant events and we ran Eagle to check if the generated traces conform to property $P$.

We also generated a universal environment for component $C_2$ (which contains the *ExecCondChecker*); we instrumented $C_2$ to emit events and we used Eagle to check if the generated traces conform to $A$. In fact, component $C_2$ did not conform with the assumption. The obtained counterexample traces exhibit a scenario where the *ExecCondChecker* pushes events onto the *EventQueue* without first setting the flag variable to 1. This turned out to be due to the fact that an input plan can contain null conditions. Instead of putting these in the condition list for monitoring, the *ExecCondChecker* immediately pushes an event to the queue. This behavior exposed an inconsistency between the models and the implementation, which we corrected. Monolithic model checking of the property $P$ would not have uncovered this inconsistency.

# 5. RELATED WORK

Assume-guarantee reasoning leverages the observation that verification techniques can analyze the individual components of large systems in isolation to improve performance. Formal techniques and tools for support of component-based design and verification are gaining prominence; see for example [1, 6, 8]. All these approaches use some form of environment assumptions (either implicit or explicit), to reason about components in isolation.

Our previous work [10] presented a technique for using design-level assumptions for compositional analysis of source code. That work used model checking (Java PathFinder [20]), while the focus here is on testing. Dingel [7] also uses model checking (the VeriSoft state-less model checker [11]) for performing assume-guarantee verification for C/C++ components. However, the burden of generating assumptions is on the user.

Our work is also related to specification-based testing, a widely-researched topic. For example, Jagadeesan et al. [14] and Raymond et al. [18] use formal specifications for the generation of test inputs and oracles. These works generate test inputs from constraints (or assumptions) on the environment of a software component and test oracles from the guarantees of the component under test. The AsmLT Test Generator [12] translates Abstract State Machine Language (AsmL) specifications into finite state machines (FSMs) and different traversals of the FSMs are used to construct test inputs. We plan to investigate the use of different traversal techniques for test input generation from assumptions and properties (which are in essence FSMs). None of the above-described approaches address predictive analysis.

Sen et al. [19] have also explored predictive runtime analysis of multithreaded programs. Their work uses a partial ordering on events to extract alternative interleavings that are consistent with the observed interleaving; states from these interleavings form a lattice that is similar to our composition of projected traces. However, to verify that no bad state exists in this lattice, they construct the lattice level by level, while this work proposes using assume-guarantee reasoning to give similar guarantees without having to explore the composition of the projected traces.

Levy et al. [16] use assume-guarantee reasoning in the context of runtime monitoring. Unlike our work, which aims at improving testing, the goal of their work is to combine monitoring for diverse features, such as memory management, security and temporal properties, in a reliable way.

## 6. CONCLUSIONS AND FUTURE WORK

We have presented assume-guarantee testing, an approach that improves traditional testing of component-based systems by predicting violations of system-level requirements both during testing of individual components and during system-level testing. During unit testing, our approach uses design-level assumptions as environments for individual components and checks generated traces against premises of an assume-guarantee proof rule; the assumptions restrict the context in which the components operate, making it more likely that failed checks correspond to system-level errors. During testing of component assemblies, the technique uses assume-guarantee reasoning on component projections of a system trace, providing results on alternative system traces. We have experimented with our approach in the verification of a non-trivial NASA system and report promising results.

Although we have strong reasons to expect that this technique can significantly improve the state of the art in testing, quantifying its benefits is a difficult task. One reason is the lack of appropriate coverage criteria for concurrent and component-based systems. Our plans for future work include coming up with "component-based" testing coverage criteria, *i.e.* criteria which, given the decomposition of global system properties into component properties, determine when individual components have been tested enough to guarantee correctness of their assembly. One interesting avenue for future research in this area is the use of the models as a coverage metric.

## 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 *International Conference on Computer-Aided Verification*, June–July, 1998.

[2] H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu. Proof rules for automated compositional verification through learning. In *International Workshop on Specification and Verification of Component-Based Systems*, Sept. 2003.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *International Conference on Verification, Model Checking and Abstract Interpretation*, Jan. 2004.

[4] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Symposium on Logic in Computer Science*, June 1989.

[5] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Apr. 2003.

[6] L. de Alfaro and T. A. Henzinger. Interface automata. In *Symposium on the Foundations of Software Engineering*, Sept. 2001.

[7] J. Dingel. Computer assisted assume guarantee reasoning with VeriSoft. In *International Conference on Software Engineering*, May 2003.

[8] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *European Symposium on Programming*, Apr. 2002.

[9] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *International Conference on Automated Software Engineering*, Sept. 2002.

[10] D. Giannakopoulou, C. S. Păsăreanu, and J. M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *International Conference on Software Engineering*, May 2004.

[11] P. Godefroid. Model checking for programming languages using VeriSoft. In *Symposium on Principles of Programming Languages*, Jan. 1997.

[12] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *International Symposium on Software Testing and Analysis*, July 2002.

[13] O. Grumberg and D. E. Long. Model checking and modular verification. In *International Conference on Concurrency Theory*, Aug. 1991.

[14] L. J. Jagadeesan, A. A. Porter, C. Puchol, J. C. Ramming, and L. G. Votta. Specification-based testing of reactive software: Tools and experiments (experience report). In *International Conference on Software Engineering*, May 1997.

[15] C. B. Jones. Specification and design of (parallel) programs. In *IFIP World Computer Congress*, Sept. 1983.

[16] J. Levy, H. Saidi, and T. E. Uribe. Combining monitors for run-time system verification. *Electronic Notes in Theoretical Computer Science*, 70(4), Dec. 2002.

[17] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logic and Models of Concurrent Systems*, volume 13, New York, 1984. Springer-Verlag.

[18] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *Real-Time Systems Symposium*, Dec. 1998.

[19] K. Sen, G. Rosu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Conference on Formal Methods for Open Object-Based Distributed Systems*, 2005.

[20] W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *International Conference on Automated Software Engineering*, Sept. 2000.

[21] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis*, July 2004.

# Specification and Verification of Inter-Component Constraints in CTL

Nguyen Truong Thang Takuya Katayama
School of Information Science
Japan Advanced Institute of Science and Technology
email: {thang, katayama}@jaist.ac.jp

## ABSTRACT

The most challenging issue of component-based software is about component composition. Current component specification, in addition to the syntactic level, is very limited in dealing with semantic constraints. Even so, only static aspects of components are specified. This paper gives a formal approach to make component specification more comprehensive by including component semantic. Fundamentally, the component semantic is expressed via the powerful temporal logic CTL. There are two semantic aspects in the paper, component *dynamic behavior* and *consistency* - namely a component does not violate some property in another when composed. Based on the proposed semantic, components can be efficiently cross-checked for their consistency by an incremental verification method - OIMC, even for many future component extensions.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking*

## 1. INTRODUCTION

As an unanimity within the software engineering community, high quality software are structured from lowly coupled *components*. Within the component-based approach, composing components properly is very essential. Component-based software idealizes the *plug-and-play* concept. The current component technology generally supports component matching at the syntactic level. Components can be syntactically checked and hence *plugged*. However, they do not *play* as expected. A major issue of concern is the mismatches of the components in the context of an assembled system. A main source of this phenomenon is because a component violates some property inherent to another. In our opinion, the problem is two-fold: the underlying logic is not powerful enough to express component properties; and even if formally specified, it is difficult to verify the properties in an open way - future components are not known in advance. For instance, temporal inter-component constraints are difficult to formally specify, much harder to check among components with the current specification methods. In this paper, the introduction of the temporal logic CTL [4] into component semantic is exactly towards that goal. This paper addresses two points: how to explicitly specify such a component semantic; and given that kind of information in the component interface, how to efficiently analyze components and to decide whether they are safe to be composed together.

Most current approaches for component interface definition deal with primarily syntactic issues among static interface elements such as *operations* and *attributes*, like those of the CORBA Interface Definition Language (IDL) [9]. Regarding a component's exact capability, essential semantic aspects of the component should also be described. In this paper, the *dynamic behavior* and *component consistency* are introduced, while the encapsulation principle is enforced. The dynamic behavior of a component is represented by a state transition model. Besides, associated with a component's behavior is a certain set of inherent properties. Certainly, another component, when interacting with that component, must preserve constraints at the interface of the former so that those inherent properties continue to hold. This characteristic is called *component consistency*. Moreover, as written in CTL, many complex semantic constraints of component consistency can be formally specified. The paper then presents an efficient algorithm to analyze consistency between components. Further, the algorithm is also scalable, not only the direct component extension but also many future compositions, as long as the consistency constraints at the interfaces are preserved.

In this paper, Section 3 introduces the formal dynamic behavior model of components. Section 4 is about component consistency and how to verify it. Later, Section 5 is concerned with specification of components and their composition.

## 2. BACKGROUND

The most common form of component deployment in practice, namely Commercial-Off-The-Shelf (COTS), is on very independent components. The computation paths of these components rarely interleave with each other. The relationship between COTS can be named *functional addition*. Besides COTS, there is another aspect of components involving in-house component development and integration. Components evolve through *functional refinement*. These

components are relatively coupled thus COTS is not a recommended option in this situation. There is a strong dependency from the refining component to the base component. Even though the discussion in this paper focuses on component refinement, the results can be well applied to COTS because analyzing COTS is obviously simpler. In practice, there is a trade-off between the simplicity of component specification and proper use of components. For COTS, component specification at the syntactical level may prove to be sufficient for component deployment in most of the cases. However, for component refinement where components are fairly coupled, semantic specification is vital.

Unlike the current component technology using UML (Unified Modeling Language) and OCL (Object Constraint Language) [16] to express semantic constraints, constraints in this paper are related to the temporal logic CTL. CTL$^*$ logic is formally expressed via two quantifiers **A** ("for all paths") and **E** ("for some path") together with five temporal operators **X** ("next"), **F** ("eventually"), **G** ("always"), **U** ("until") and **R** ("release") [4]. CTL (Computation Tree Logic) is a restricted subset of CTL$^*$ in which each temporal operator must be preceded by a quantifier. An incremental verification technique for CTL properties has been attempted by [6, 13]. It is named *open incremental model checking* (OIMC) for the *open* and *incremental* characteristics of the algorithm. Suppose that a base component is refined by another component. The approach consists of the following activities:

1. Deriving a set of *preservation constraints* at the interface states of the base such that if those constraints are preserved, the property inherent to the base under consideration is guaranteed.

2. The refining component does not violate the above property of the base if, during its execution, the above constraints are preserved.

## 3. DYNAMIC BEHAVIOR SPECIFICATION

There are two types of semantic mentioned in this paper: component dynamic behavior (this section) and component consistency (Section 4).

In the typical case of component refinement, there are two interacting components: *base* and *extension (or refinement)*. Between the base and its extension, on the base side, is an interface consisting of *exit* and *reentry* states [6, 13]. An exit state is the state where control is passed to the extension. A reentry state is the point at which the base regains control. Correspondingly, the extension interface contains *in*- and *out*-states at which the refinement component receives and returns system control. Let $AP$ be a set of atomic propositions. The dynamic behavior of a component is independently represented by a state transition model.

DEFINITION 1. *A state transition model $M$ is represented by a tuple $\langle S, \Sigma, s_0, R, L\rangle$ where $S$ is a set of states, $\Sigma$ is the set of input events, $s_0 \in S$ is the initial state, $R \subseteq S \times PL(\Sigma) \rightarrow S$ is the transition function (where $PL(\Sigma)$ denotes the set of guarded events in $\Sigma$ whose conditions are propositional logic expressions), and $L : S \rightarrow 2^{AP}$ labels each state with the set of atomic propositions true in that state.*

A base is expressed by a transition model $B = \langle S_B, \Sigma_B, s_{o_B}, R_B, L_B\rangle$ and an *interface* $I$. The interface is a tuple of two state sets $I = \langle exit, reentry\rangle$, where $exit, reentry \subseteq S_B$. An extension is similarly represented by a model $E = \langle S_E, \Sigma_E, \perp, R_E, L_E\rangle$. $\perp$ denotes no-care value. Its interface is $J = \langle in, out\rangle$.

$E$ can be semantically plugged with $B$ via *compatible* interface states. Logically, along the computation flow, when the system is in an exit state $ex \in I.exit$ of $B$ matched with an in-state $i \in J.in$ of $E$, denoted as $ex \leftrightarrow i$, it can enter $E$ if the conditions to accept extension events, namely the set of atomic propositions at $i$, are satisfied. That is, $\bigwedge L_B(ex) \Rightarrow \bigwedge L_E(i)$, where $\bigwedge$ is the inter-junction of atomic propositions. Similar arguments are made for the matching of a reentry state $re \in I.reentry$ and an out-state $o \in J.out$. The conditions resemble to pre- and post-conditions in *design by contract* [12].

DEFINITION 2. *Within interfaces $I$ and $J$ of $B$ and $E$, the pairs $\langle ex, i\rangle$ and $\langle re, o\rangle$ can be respectively mapped according to the following conditions.*

- $ex \leftrightarrow i$ *if* $\bigwedge L_B(ex) \Rightarrow \bigwedge L_E(i)$.
- $re \leftrightarrow o$ *if* $\bigwedge L_E(o) \Rightarrow \bigwedge L_B(re)$.

The actual mapping configuration is decided by the modeler at composition time. Subsequently, $ex$ and $re$ will be used in place of $i$ and $o$ respectively in this paper.

DEFINITION 3. *Composing the base $B$ with the extension $E$, through the interface $I$ produces a composition model $C = \langle S_C, \Sigma_C, s_{0_C}, R_C, L_C\rangle$ as follows:*

- $S_C = S_B \cup S_E$; $\Sigma_C = \Sigma_B \cup \Sigma_E$; $s_{0_C} = s_{0_B}$;
- *$R_C$ is defined from $R_B$ and $R_E$ in which $R_E$ takes precedent, namely any transition in $B$ is overridden by another transition in $E$ if they share the same starting state and input event;*
- $\forall s \in S_B, s \notin I.exit \cup I.reentry : L_C(s) = L_B(s)$;
- $\forall s \in S_E, s \notin J.in \cup J.out : L_C(s) = L_E(s)$;
- $\forall s \in I.exit \cup I.reentry : L_C(s) = L_B(s)$.

In this formal specification, the behavior of $B$ can be partially overridden by $E$ because $E$ takes precedent during composition.

DEFINITION 4. *The* closure *of a property $p$, $cl(p)$, is the set of all sub-formulae of $p$ including itself.*

- $p \in AP : cl(p) = \{p\}$
- *$p$ is one of* $\mathbf{AX}\ f, \mathbf{EX}\ f, \mathbf{AF}\ f, \mathbf{EF}\ f, \mathbf{AG}\ f, \mathbf{EG}\ f$ : $cl(p) = \{p\} \cup cl(f)$
- *$p$ is one of* $\mathbf{A}\ [f\ \mathbf{U}\ g], \mathbf{E}\ [f\ \mathbf{U}\ g], \mathbf{A}\ [f\ \mathbf{R}\ g], \mathbf{E}\ [f\ \mathbf{R}\ g]$ : $cl(p) = \{p\} \cup cl(f) \cup cl(g)$
- $p = \neg f : cl(p) = cl(f)$
- $p = f \vee g$ *or* $p = f \wedge g : cl(p) = cl(f) \cup cl(g)$

DEFINITION 5. *The truth values of a state $s$ with respect to a set of CTL properties $ps$ within a model $M = \langle S, \Sigma, s_0, R, L\rangle$, denoted as $\mathcal{V}_M(s, ps)$, is a function: $S \times 2^{CTL} \rightarrow 2^{CTL}$.*

- $\mathcal{V}_M(s, \emptyset) = \emptyset$

- $\mathcal{V}_M(s, \{p\} \cup ps) = \mathcal{V}_M(s, \{p\}) \cup \mathcal{V}_M(s, ps)$
- $\mathcal{V}_M(s, \{p\}) = \begin{cases} \{p\} & \text{if } M, s \models p \\ \{\neg p\} & \text{otherwise} \end{cases}$

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 $p$.

OIMC is rooted at *assumption model checking* [11]. This method is particularly useful for open systems - future extensions are not known in advance. Hence, OIMC is applicable to component-based software. The idea to the component refinement context is explained in the following. The composite model $C$ can be treated as the combination of two sequential components $B$ and $E$. In addition to existing execution paths defined in $B$, a typical execution path in $C$ consists of three parts: initially in $B$, next in $E$ and then back to $B$. Associated with each reentry state $re$ of $E$ is a computation tree rooted at the state and lying completely in $B$. This tree possesses a set of temporal properties. If these properties at $re$ are known, without loss of correctness, we can efficiently derive the properties at the upstream states in $E$ by ignoring model checking in $B$ to find the properties at $re$. Instead, we start from these reentry states with the associated properties; check the upstream of the extension component, and then the base component if needed [1]. The properties associated with a reentry state $re$ are assumed with truth values from $B$, $As(re) = \mathcal{V}_B(re, cl(p))$. $As$ is the assumption function of this assumption model checking. Of course, this method relies on whether $As(re)$ is proper.

The assumption $As$ at a reentry state $re$ is *proper* if the seeding values are exactly the properties associated with the tree at $re$ in $C$, i.e. $\mathcal{V}_B(re, cl(p)) = \mathcal{V}_C(re, cl(p))$. The assumption $As$ is definitely proper if $re$ is not affected by $E$. The problem arises if $ex$ is reachable from $re$ in $B$. On the other hand, $re$ is reachable from $ex$ in $E$. This situation creates a circular dependency between interface states $ex$ and $re$. Dealing with such a circular structure is indeed very important to the verification result of assumption model checking. In fact, this is the weak point of assumption model checking. In this paper, that topic is out of the scope. Subsequent discussions consider $As$ is proper.

# 4. INTER-COMPONENT CONSISTENCY

Given a structure $B = \langle S_B, \Sigma_B, s_{0_B}, R_B, L_B \rangle$ as in Definition 1, a property $p$ holding in $B$ is denoted by $B, s_{0_B} \models p$. $C$ is formed by composing $B$ and $E$, $C = B + E$. $B$ and $E$ are *consistent* with respect to $p$ if $C, s_{0_B} \models p$.

## 4.1 A Theorem on Component Consistency

Due to the inherently inside-out characteristic of model checking, after checking $p$ in $B$, at each state $s$, $\mathcal{V}_B(s, cl(p))$ are recorded.

DEFINITION 6. *$B$ and $E$ are in* conformance *at an exit state $ex$ (with respect to $cl(p)$) if $\mathcal{V}_B(ex, cl(p)) = \mathcal{V}_E(ex, cl(p))$.*

In this definition, $\mathcal{V}_E(ex, cl(p))$ are derived from the assumption model checking within $E$, and the seeded values at any reentry state $re$ are $As(re) = \mathcal{V}_B(re, cl(p))$.

[1]There is no need to model check the base again if the consistency constraints associated with the exit states of $B$ are preserved at the respective in-states of $E$.

THEOREM 7. *Given a base $B$ and a property $p$ holding on $B$, an extension $E$ is attached to $B$ at some interface states. $E$ does not violate property $p$ if $B$ and $E$ conform with each other at all exit states.*

The proof details are in [13]. Even though this paper focuses on component refinement, with regards to COTS, the above theorem also holds. A COTS component can be indeed regarded as a special case of refinement in which there is only a single exit state and no reentry state with the base. The computation tree of the COTS deviates from the base and never joins the base again. After being composed with a COTS, instead of an assumption model checking within the COTS, a standard model checking procedure can be executed entirely within the COTS to find the properties at the exit state. The conformance condition to ensure the consistency between the two components can be applied as usual. The only difference in Definition 6 lies in $\mathcal{V}_E(ex, cl(p))$ for each exit state $ex$. In component refinement, these truth values are derived from the assumption model checking within $E$ with the assumption values $\mathcal{V}_B(re, cl(p))$ at any reentry state $re$. On the contrary, in COTS, there is no assumption at all. Hence, the model checking procedure in $E$ is then exactly standard CTL model checking.

Figure 1 depicts the composition preserving the property $p = \mathbf{A}[f\ \mathbf{U}\ g]$ when $B$ and $E$ are in conformance. The composition is done via a single exit state $ex$. The reentry state $re$ is not shown but it does not affect the subsequent arguments [2]. $E$ overrides the transition $ex$-$s_3$ in $B$. $B'$ is the remainder of $B$ after removing the overridden transition. In the figure, within $B$, $p = \mathbf{A}[f\ \mathbf{U}\ g]$ holds at $s_1$, $s_2$ and $ex$. The figure only shows $\mathcal{V}_E(ex, p) = \mathcal{V}_B(ex, p) = \mathbf{A}[f\ \mathbf{U}\ g]$. In fact, $B$ and $E$ conform at $ex$ with regards to $cl(p)$. After removing the edge $ex$-$s_3$, the new paths in $E$ together with the remaining computation tree in $B'$ still preserve $p$ at $ex$ directly; and consequently $s_2$ and $s_1$ indirectly. As $p$ is preserved at the initial state $s_1$, $B$ and $E$ are consistent.

By Theorem 7, component semantic specification requires $\mathcal{V}_B(s, cl(p))$ for tuples of any potential interface state $s$ and any CTL property $p$ inherent to $B$. They serve as constraints for component consistency (Section 5.2).

## 4.2 Open Incremental Model Checking

Components can be verified to be consistent via OIMC. Initially, a CTL property $p$ is known to hold in $B$. We need to check that $E$ does not violate $p$. From Theorem 7, the incremental verification method only needs to verify the conformance at all exit states between $B$ and $E$. Corresponding to each exit state $ex$, within $E$, the algorithm to verify preservation constraints $\mathcal{V}_B(ex, cl(p))$ can be briefly described as follows:

1. Seeding $\mathcal{V}_B(re, cl(p))$ at any reentry state $re$. The assumption function $As$ is: $As(re) = \mathcal{V}_B(re, cl(p))$.
2. Executing a CTL assumption model checking procedure in $E$ to check $\phi$, $\forall \phi \in cl(p)$. In case of COTS, a standard CTL model checking is executed within $E$ instead.
3. Checking if $\mathcal{V}_E(ex, cl(p)) = \mathcal{V}_B(ex, cl(p))$.

[2]In fact, this figure is intended to represent both component refinement and COTS.

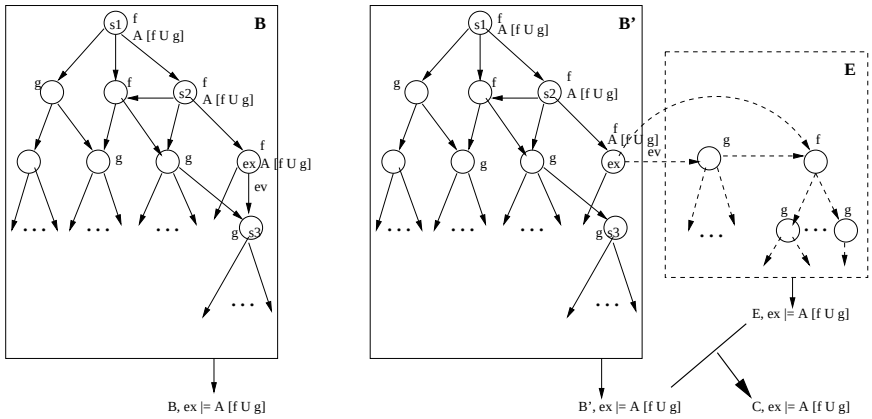



**Figure 1: An illustration of conformance $\mathcal{V}_E(ex, cl(p)) = \mathcal{V}_B(ex, cl(p))$ where $E$ overrides $B$. The property $p = \mathbf{A}\,[f\ \mathbf{U}\ g]$ is preserved in $B$ due to the conformance.**

At the end of the algorithm, if at all exit states, the truth values with respect to $cl(p)$ are matched respectively, $B$ and $E$ are consistent with respect to $p$.

## 4.3 Scalability of OIMC

This section addresses the scalability of the algorithm in Section 4.2. We consider the general case of the $n$-th version of the component ($C_n$) during software evolution as a structure of components $B$, $E_1$, $E_2$, ..., $E_n$ where $E_i$ is the refining component to the $(i-1)$-th evolved version ($C_{(i-1)}$), $i = \overline{1, n}$. The initial version is $C_0 = B$ and $C_i = C_{i-1} + E_i$. We check for any potential conflict between $B$ and $E_i$ regarding $p$ via OIMC. Theorem 8 claims that OIMC is scalable. The detailed proof is in [13].

THEOREM 8. *If all respective pairs of base ($C_{(i-1)}$) and refining ($E_i$) components conform, the complexity of OIMC to verify the consistency between $E_n$ and $B$ is independent from the n-th version $C_n$, i.e. it only executes within $E_n$.*

# 5. COMPONENT SPECIFICATION

This paper advocates the inclusion of two additional semantic aspects of component specification to facilitate proper component composition. Given a base component $B = \langle S_B, \Sigma_B, s_{o_B}, R_B, L_B \rangle$, the semantic aspects are: dynamic behavior (via state transition model in which only potential future interface states are visible to other components - Section 3) and their associated consistency constraints (via the truth values of $\mathcal{V}_B(s, cl(p))$ at such an interface state $s$, where $p$ is a CTL property holding in the base component - Section 4).

## 5.1 Interface Signature

Component signatures are the fundamental aspect to the component interface. As commonly recognized, the traditional interface signature of a component contains *attributes* and *operations*. First, through attributes [3], the current state of a software component may be externally observable. The component's clients can observe and even change the values of those attributes. Second, the environment interacts with the component through operations. The operations represent services or functions the component provides.

Unlike above two static aspects, the introduction of dynamic behavior of a component to the interface is recommended in this paper. Components in reality resemble classes in the object-oriented (OO) approach. This specification style hence follows the encapsulation principle of OO technology so that only essential information is exposed. Only the partial dynamic model of the component consisting of potential future interface states is visible to clients. The rest of the model can be hidden. Associated with a visible interface state $s$ is the set of atomic propositions $L(s)$ (Definition 1). These propositions are often expressed via logic expressions among attributes above.

## 5.2 Interface Constraints

The interface signature only shows the individual elements of the component for interaction with clients in syntactic terms. In addition to the constraints imposed by their associated types, the attributes and operations of a component interface may be subject to a number of further semantic constraints regarding their use. In general, there are two types of such constraints: internal to individual components and inter-component relationships. The first type is simple and has been thoroughly mentioned in many component-related works [9, 16]. The notable examples are the operation semantics according to pre-/post-conditions of operations; and range constraints on attributes. For the second type, current component technology such as CORBA IDL (Interface Definition Language), UML and OCL [16] etc. is limited to a very weak logic in terms of expressiveness. For example, different attributes in components may be interrelated by their value settings; or an operation of a component can only be invoked when a specific attribute value of another is in a given range etc [9]. The underlying logic only expresses the constraint at the moment an interface element is invoked, i.e. static view, regardless of execution history.

The paper introduces two inter-component semantic constraints. The first constraint is based on the *plugging* compatibility for a refining component to be plugged at a special state of the base. This situation resembles the extension of use-case scenarios. The base gives the basic interacting sce-

[3] Attribute is termed as property in [9] which is essentially the entities expressing states of components. To distinguish them from temporal properties inherent to components in Section 5.2, those entities are named as attributes.

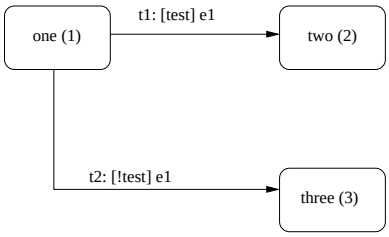



**Figure 2: The dynamic behavior model of the "black" component.**

narios of the component with clients. The refining component refines some of those scenarios further at a certain point from which the component deviates from the pre-defined course to enter new traces in the extension component. Such a point corresponds to an exit state in Definition 2.

On the other hand, the second semantic constraint emphasizes on how to make components *play* once they are plugged. Importantly, this constraint type is expressed in terms of CTL so its scope of expressiveness is enormous. In contrast to the logic above, CTL can describe whole execution paths of a component, i.e. dynamic view. Via OIMC in Section 4.2, a refining client $E$ to a base component $B$ can be efficiently verified on whether it preserves the property $p$ of $B$.

Once composed, the new component $C = B + E$ exposes its new interface signatures and constraints. Static aspects like attributes and operations are simply the sum of those in $B$ and $E$. The dynamic behavior of $C$ is exposed according to the composition of corresponding visible parts of $B$ and $E$. In terms of constraints, any potential interface state $s$ is exposed with the set of propositions $L_C(s) = L_B(s)$ according to Definition 3. On the other hand, the consistency constraint at $s$ is derived either from $\mathcal{V}_B(s, cl(p))$ (for any $s \in S_B$) or $\mathcal{V}_E(s, cl(p))$ which is resulted from the above execution of OIMC within $E$ (if $s \in S_E$). Subsequent refinements to $C$ follow the same manner as the case of $E$ to $B$ because of Theorem 8.

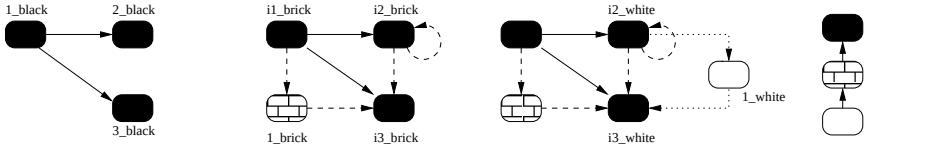
## 5.3 Component Specification and Composition

Component specification can be represented via interface signatures and constraints written in an illustrative specification language below. The major goal of this language is to minimize the "conceptual distance" between architectural abstractions and their implementation [1]. Encoding state diagrams directly into the interface; and refining existing component specifications in pure programming languages are difficult. Instead, a language similar to that of [1] for declaring and refining state machines in layering manner is used. Based on the exemplary specification, components are implemented as classes in typical object-oriented languages. Component composition is then done via class aggregation/merging. Component attributes and operations are declared in the object-oriented style like C++. The `virtual` keyword is used to only name an element without actual memory allocation. The element will be subsequently mapped to the actual declaration in another component. This mechanism resembles `mergeByName` in Hyper/J [15] in which component entities sharing the same label are merged into a single entity during component composition.

Figure 2 shows the dynamic model of a simple component, while below is the corresponding specification of the component. The interface signatures should declare: edges with name, start state, end state, transition guard and input event; as well as transition action. At the end are the semantic constraints of the component written in both types shown in Section 5.2, namely plugging compatible conditions and inherent temporal properties at potential interface states. For illustration purpose and due to space limitation, this producer-consumer example is very much simplified so that only some key transitions and states are shown. Because of this over-simplified model, the whole dynamic behavior of the component is visible to clients. In practice, regarding the encapsulation principle, only essential part of the model for future extension is visible. The rest of the model is hidden from clients. There are three components: "black" (the base $B$ of Figure 3a with solid transitions - item-producing function); "brick" (the first refinement $E$ of Figure 3b expressed via dashed transitions - variable-size buffer and item-consuming function); and "white" (the second refinement $E'$ of Figure 3c depicted in dotted transitions - optimizing data buffer).

```
Component B {
Signature:
  states 1_black, 2_black, 3_black;

  /* edge declarations */
  edge t1:  1_black -> 2_black
  condition test // OK if adding k items to buffer
  input event e1 // producing k items
  do { produce(k)...  }; /* t1 action */

  edge t2:  1_black -> 3_black;
  ...  /* similarly defined */

  // operations and attributes declaration
  boolean test;
  int cons, prod;// consumed, produced items
  int buffer[];// a bag of data items
  ...
  init(){ state = 1_black; ...};
  produce(n){ prod = prod + n;...};

Constraint:
  /* compatible plugging conditions - CC */
  1_black_cc:  cons = prod;// empty buffer
  2_black_cc:  test = true, cons < prod;
  3_black_cc:  test = false, cons ≤ prod;

  /* Inherent properties - IP */
  1_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  2_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  3_black_ip:  AG (cons ≤ prod), cons ≤ prod;
}
```

As components are composed with each other, they can be progressively refined/extended in layering manner. The process adds states, actions, edges to an existing component. The original component and each refinement are expressed as separate specifications that are encapsulated in distinct layers. Figure 3 shows this hierarchy: the root component is generated by the specification from Figure 2 or Figure 3a; its immediate refinements are in turn generated from component specifications according to the order in the Figures 3b




(a) Original diagram (b) First refinement (c) Second refinement (d) aggregation hierarchy

**Figure 3: Component refinements and component composition via class aggregation.**

and 3c.

```
Component E {/* for refining black */
Signature:
  states 1_brick, i1_brick, i2_brick, i3_brick;

  /* edges declaration */
  edge t3:  i2_brick -> i3_brick
  condition ...  // ready to consume
  input event ...  // consuming k items
  do { consume(k)...  }; /* t3 action */

  edge t4:  i1_brick -> 1_brick
  condition ...  // ready to change buffer size
  input event ...  // change the size
  do { changesize();...  }; /* t4 action */

  edge t5:  1_brick -> i3_brick;
  edge t6:  i2_brick -> i2_brick;
    // buffer inquiry only, consuming zero item
  ...  /* similarly defined */

  // operations and attributes declaration
  virtual int cons;// mapped with cons in B
  virtual int prod;// mapped with prod in B
  virtual int buffer[];// mapped with buffer in B
  consume(n){ cons = cons + n;...};
  changesize(){ buffer = malloc();...};

Constraint:
  1_brick_cc:  cons ≤ prod;
  i1_brick_cc:  cons ≤ prod;
  i2_brick_cc:  test = true, cons < prod;
  i3_brick_cc:  test = false, cons < prod;
}

Component E' {/* for refining black + brick */
Signature:
  states 1_white, i2_white, i3_white;

  /* edges declaration */
  edge t7:  i2_white -> 1_white
  condition ...  // ready to compact buffer
  input event ...// compact the data buffer
  do { resetbuffer();...  }; /* t7 action */

  edge t8:  1_white -> i3_white;
  ...  /* similarly defined */

  // operations and attributes declaration
  virtual int cons;// mapped with cons in B
  virtual int prod;// mapped with prod in B
  virtual int buffer[];// mapped with buffer in B
  resetbuffer(){ prod = prod - cons; cons = 0;...};

Constraint:
  1_white_cc:  cons ≤ prod, cons = 0;
  i2_white_cc:  test = true, cons ≤ prod;
  i3_white_cc:  test = false, cons ≤ prod;
}
```

Aggregation then plays a central role in this component implementation style. All the states and edges in Figure 3a are aggregated with the refinement of Figure 3b; and this figure is in turn united with the refinement of Figure 3c. The component to be executed is created by instantiating the bottom-most class of the refinement chain of Figure 3d.

The following explains the preservation of the constraint in $B$ by all subsequent two component refinements $E$ and $E'$. Informally, the property means that under any circumstance, the number of produced items by the component is always greater or equal to that of consumed items. In terms of CTL notation, $p = \mathbf{AG}\,(cons \leq prod)$. The closure set of $p$ is hence $cl(p) = \{p, a\}$, where $a = (cons \leq prod)$.

Initially, $B$ is composed with $E$. Interface plugging conditions are used to map compatible interface states among components. The base exposes three interface states $1\_black$, $2\_black$ and $3\_black$. On the other hand, the refinement component exposes four interface states, namely $1\_brick$, $i1\_brick$, $i2\_brick$ and $i3\_brick$. Based on the respective atomic proposition sets at those states, corresponding interface states are mapped accordingly.

For instance, first $\bigwedge L_B(1\_black) = (cons = prod) \Rightarrow \bigwedge L_E(i1\_brick) = (cons \leq prod)$. According to Definition 2, $i1\_brick \leftrightarrow 1\_black$. On the other hand, because $\bigwedge L_E(i2\_brick) = \bigwedge L_B(2\_black)$, $i2\_brick \leftrightarrow 2\_black$. Similarly, $\bigwedge L_E(i3\_brick) \Rightarrow \bigwedge L_B(3\_black)$, $i3\_brick \leftrightarrow 3\_black$. Here, $i1\_brick$ and $i2\_brick$ perform exit states of the base component, while $i2\_brick$ and $i3\_brick$ are reentry states.

The composite model of the two components $C_1 = B + E$ is shown in Figure 3b. After the designer decides on the mapping configuration between interface states, and properly resolves any mismatches at the syntactic level between $B$ and $E$, the semantic constraint of consistency between the two due to $p$ is in focus. The OIMC algorithm in Section 4.2 is applied as follows:

1. Copying $\mathcal{V}_B(s, cl(p))$ to the respectively mapped out-states $i2\_brick$ and $i3\_brick$ in $E$ for any reentry state $s$ such as $2\_black$ and $3\_black$.

2. Executing assumption model checking within $E$ to find $\mathcal{V}_E(i1\_brick, cl(p))$ and $\mathcal{V}_E(i2\_brick, cl(p))$. Note that, the model checking procedure is executed within the dashed part in Figure 3b. The solid transitions belong to the base component $B$ and are hence ignored.

3. Checking if $\mathcal{V}_E(i1\_brick, cl(p)) = \mathcal{V}_B(1\_black, cl(p))$ and

$\mathcal{V}_E(i2\_brick, cl(p)) = \mathcal{V}_B(2\_black, cl(p))$. If so, $B$ and $E$ conform.

The model checking is very simple and hence its details are skipped. At the end, $B$ and $E$ components conform at all exit states. According to Theorem 7, $p$ is preserved by the second component after evolving to $C_1 = B + E$.

$C_1$ is then extended with $E'$. Notably, the interface of the new component $C_1$ is derived from $B$ and $E$ as below:

```
Component C1 {
Signature:
  states 1_black, 2_black, 3_black, 1_brick;

  /* edge declarations */
  edge t1:  1_black -> 2_black;
  edge t2:  1_black -> 3_black;
  edge t3:  2_black -> 3_black;
  edge t4:  1_black -> 1_brick;
  edge t5:  1_brick -> 3_black;
  edge t6:  2_black -> 2_black;
  /* identical to each component's declaration */

  // operations and attributes declaration
  boolean test;
  int cons, prod;// consumed, produced items
  int buffer[];
  init(){ state = 1_black; ...}
  consume(n){ cons = cons + n;...};
  produce(n){ prod = prod + n;...};
  changesize(){ buffer = malloc();...};

Constraint:
  /* compatible plugging conditions - CC */
  1_black_cc:  cons = prod;
  2_black_cc:  test = true, cons < prod;
  3_black_cc:  test = false, cons ≤ prod;
  1_brick_cc:  cons ≤ prod;

  /* Inherent properties - IP */
  1_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  2_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  3_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  1_brick_ip:  AG (cons ≤ prod), cons ≤ prod;
}
```

The approach in composing $E'$ with $C_1$ is similar to the above, we have the following mapping configuration between interface states: $i2\_white \leftrightarrow 2\_black$, $i3\_white \leftrightarrow 3\_black$. The same result is achieved, $p$ is preserved by $E'$. More importantly, the verification method is executed within $E'$ only, i.e. the dotted part in Figure 3c. After composing $E'$, the component becomes $C_2 = C_1 + E'$ shown below:

```
Component C2 {
Signature:
  states 1_black, 2_black, 3_black, 1_brick, 1_white;

  /* edge declarations */
  edge t1:  1_black -> 2_black;
  edge t2:  1_black -> 3_black;
  edge t3:  2_black -> 3_black;
  edge t4:  1_black -> 1_brick;
  edge t5:  1_brick -> 3_black;
  edge t6:  2_black -> 2_black;
  edge t7:  2_black -> 1_white;
  edge t8:  1_white -> 3_black;
  /* identical to each component's declaration */

  // operations and attributes declaration
  boolean test;
  int cons, prod;// consumed, produced items
  int buffer[];
  init(){ state = 1_black; ...}
  consume(n){ cons = cons + n;...};
  produce(n){ prod = prod + n;...};
  changesize(){ buffer = malloc();...  };
  resetbuffer(){ prod = prod - cons; cons = 0;...};

Constraint:
  /* compatible plugging conditions - CC */
  1_black_cc:  cons = prod;
  2_black_cc:  test = true, cons < prod;
  3_black_cc:  test = false, cons ≤ prod;
  1_brick_cc:  cons ≤ prod;
  1_white_cc:  cons ≤ prod, cons = 0;

  /* Inherent properties - IP */
  1_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  2_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  3_black_ip:  AG (cons ≤ prod); cons ≤ prod;
  1_brick_ip:  AG (cons ≤ prod); cons ≤ prod;
  1_white_ip:  AG (cons ≤ prod); cons ≤ prod;
}
```

In brief, $p$ is preserved by both extensions $E$ and $E'$. In this example, the scalability of incremental model checking is maintained as it only runs on the refinements, independently from the bases $B$ and $C_1$ respectively.

## 6. RELATED WORK

Modular model checking is rooted at assume-guarantee model checking [10, 14]. However, unlike the counterpart in hardware verification [8, 10] focusing on parallel composition of modules, software modular verification [11] is restricted by its sequential execution nature. Incremental model checking inspires verification techniques further. There is a fundamental difference between those conventional modular verification works [8, 10, 14] and the proposed approach including this paper and [6]. Modular verification in the former works is rather closed. Even though it is based on component-based modular model checking, it is not prepared for change. If a component is added to the system, the whole system of many existing components and the new component are re-checked altogether. On the contrary, the OIMC approach in this paper and [6] is incrementally modular and hence more open. It only checks the new system's consistency within the new component. Certainly, this merit comes at the cost of "fixed" preservation constraints at exit states. These constraints can deliver a false negative for some cases of component conformance.

Regarding the assumption aspect in component verification, [7] presents a framework for generating assumption on environments in which the component satisfies its required property. This work differs OIMC in some key points. First, the constraints in OIMC are explicitly fixed at $\mathcal{V}_B(ex, cl(p))$ for any exit state $ex$, whereas based on a fully specified component model including error states, [7] generates assumption about operation calls by which the environment

does not lead the component to any error state. Second, the approach in [7] is viewed from a static perspective, i.e. the component and the external environment do not evolve. If the component changes after adapting some refinements, the assumption-generating approach is re-run on the whole component, i.e. the component model has to be re-constructed; and the assumption about the environment is then generated from that model.

The OIMC technique introduced in this paper is similar to [6]. However, our work proposes more precisely and explicitly the conformance condition between components. Further, to enable the plug-and-play idea in component-based software, the corresponding component specification are separately specified. Their composition is based on plugging condition among compatible interface states. In addition, the scalability of component consistency is not mentioned in [6]. Without Theorem 8, the approach is not applicable for future component composition.

Finally, like the proposal in Section 5 about encapsulating dynamic behavior model into component interface, i.e. state-full interface, two closely related works [3, 5] also advocate the use of *light-weight formalism* to capture temporal aspects of software component interfaces. More specifically, this paper simply relies on state transition model in the most general sense, while the approach in [3, 5] presents a finer realization of state-full model in which states are represented by control points in operations of components; and edges are actually operation calls. That approach focuses on the order of operation calls in a component [4]. By formalizing a component through a set of input, output and internal operations, the compatibility between component interfaces with regards to the structure of component operations is defined and checked. In addition, the two approaches target different aspects of consistency. This paper is concerned with component consistency in terms of CTL properties, whereas the approach in [3, 5] is involved with the correctness and completeness of operation declarations within components.

## 7. CONCLUSION

This paper focuses on the refinement aspect of components in which components are relatively coupled. However, the results of this paper can be equally applied to COTS. This paper advocates the inclusion of *dynamic behavior* and *component consistency* written in CTL to the component interface to better deal with component matching. Besides the traditional static elements such as operations and attributes, component interface should include potential interface states together with the associated plugging conditions and consistency constraints at those states. Next, based on the proposed specification structure, an efficient and scalable model checking method (OIMC) is utilized to verify whether components are consistent.

Current well-known model checkers do not support assumption model checking. A future work is to encapsulate the assumption feature into an open-source model checker such as NuSMV [2].

## 8. REFERENCES


[1] D. Batory, C. Johnson, B. MacDonald, and D. V. Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *Proc. International Conference on Software Reuse*, July 2000.

[2] R. Cavada, A. Cimatti, G. Keighren, et al. *NuSMV 2.2 Tutorial*. CMU and ITC-irst, nusmv@irst.itc.it, 2004.

[3] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, M. Jurdzinski, and F. Y. C. Mang. Interface compatibility checking for software modules. In *Proceedings of the Computer-Aided Verification - CAV*. LNCS Springer-Verlag, 2002.

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

[5] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Symposium on Foundations of Software Engineering*. ACM Press, 2001.

[6] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Proc. Symposium on the Foundations of Software Engineering*, September 2001.

[7] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of the International Conference on Automated Software Engineering*, 2002.

[8] 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.

[9] J. Han. An approach to software component specification. In *Proceedings of International Workshop on Component Based Software Engineering*, 1999.

[10] 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.

[11] K. Laster and O. Grumberg. Modular model checking of software. In *Conference on Tools and Algorithms for the Constructions and Analysis of Systems*, 1998.

[12] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1997.

[13] T. T. Nguyen and T. Katayama. Handling consistency of software evolution in an efficient way. In *Proc. IWPSE*, pages 121–130, 2004.

[14] C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes of Computer Science*. Springer-Verlag, 1999.

[15] P. Tarr and H. Ossher. *Hyper/J(TM) User and Installation Manual*. IBM Research, IBM Corp., 2000.

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


[4] In [3], operations are named as methods.

# A Specification Language for Coordinated Objects

Gabriel Ciobanu
Romanian Academy
Institute of Computer Science
Iaşi, Romania
gabriel@iit.tuiasi.ro

Dorel Lucanu
"A.I. Cuza" University
Faculty of Computer Science
Iaşi, Romania
dlucanu@info.uaic.ro

## ABSTRACT

The paper presents a specification language of autonomous objects supervised by a coordinating process. The coordination is defined by means of an interaction wrapper. The coordination semantics is described in the terms of bisimulation relations. The properties of the coordinated objects are expressed as temporal formulas, and verified by specific model-checking algorithms. We use the alternating bit protocol to exemplify our specification language and its semantics.

This approach allows a clear separation of concerns: the same coordinating process can be used with different concurrent objects, and the same objects can be used with a different coordinator. Thus our specification language allows easy modifications and customization. The method is effective in assembling increasingly complex systems from components. Moreover, composing different coordinating processes can be done without changing the code of the coordinated objects. In this way, the difficult task of implementing the mechanism of coordination becomes substantially easier.

## Keywords

coordination, process algebra, classes, objects, bisimulation, temporal logic.

## 1. INTRODUCTION

Coordination of concurrent activities is an important goal of object-oriented concurrent programming languages, as well for component-based software community. As far as there was no support for components abstraction and high-level coordination, it is difficult to ignore the mismatch between conceptual designs and the implementation. Object-oriented languages offer little support for synchronization of concurrent objects. While in theory the set of provided constructs is sufficient to solve the coordination problems, in practice only good programmers are able to handle non trivial tasks. Another difficulty was given by the fact that a low-level approach does not allow the composition of different coordination policies without changing the implementation of the coordinated entities. The main problems are represented by no separation of concerns (expressing coordination abstraction is difficult because the code of coordination is strongly tied to the implementation of the coordinated objects), the absence of abstraction (no declarative means to specify coordination), the lack of compositionality and flexibility, and the difficulties for a programmer to implement the desired coordination.

As a possible solution, this paper introduces and studies a specification language where the components are described as objects, coordination is defined as a process, and their integration is given by a wrapper. Semantic integration of the coordinating process and coordinated entities is based on bisimulation. Coordinating process and coordinated components are rather independent. The explicit description of collaboration between components defines interaction policies and rely on the methods of the objects they coordinate. We use a wrapper which, together with the processes associated to objects provides an interface of the underlying component. The three languages corresponding to objects, coordinating process and wrapper reflect the intuitions and practices of software engineers. In order to support formal manipulation and reasoning, our implementation provides a semantics based on the models of class specification in hidden algebra, labelled transition systems represented as coalgebras, and some theoretical results expressing the coordination in terms of bisimulations and coalgebra homomorphisms. These formal aspects are not visible to the user; the implementation hides them, but ensures both a good matching between intuitions and practices of users, and a sound executable system.

## 2. CLASSES AND OBJECTS

In this section we present the specification of classes and their objects. We propose a specification language with syntax closer to that of object-oriented programming language and with semantics described in hidden algebra [9]. A *class specification* consists of specification of *attributes* and specification of operations. An operation specification includes the signature of the operation and its behavioural specification expressed in the terms of its parameters and attributes values before and after its execution. The grammar supplying the syntax for class specification is given in Figure 1. The decoration of the attributes names with the prime symbol `'` in a method specification is similar to that used in Z, and a decorated attribute name refers the value of the attribute after the execution of the method. A given set of primitive data types including `Bool`, `Int`, ... is assumed.

EXAMPLE 1. *Alternating Bit Protocol*
*The alternating bit protocol (ABP) is a communication protocol consisting of four components (see fig. 2): a sender, a receiver, and two communication channels. Here is a brief description of each component.*

⟨*class_spec*⟩ ::= ⟨*header*⟩ {⟨*body*⟩}
⟨*header*⟩ ::= `class`⟨*class_name*⟩ |
`class`⟨*class_name*⟩ `extends` ⟨*class_list*⟩
⟨*class_list*⟩ ::= ⟨*class_name*⟩ | ⟨*class_name*⟩, ⟨*class_list*⟩
⟨*body*⟩ ::= ⟨*att_spec_list*⟩$_{\text{opt}}$ ⟨*opn_spec_list*⟩$_{\text{opt}}$
⟨*att_spec_list*⟩ ::= ⟨*att_spec*⟩ | ⟨*att_spec_list*⟩ ⟨*att_spec*⟩
⟨*att_spec*⟩ ::= ⟨*type*⟩ ⟨*att_name*;⟩
⟨*opn_spec_list*⟩ ::= ⟨*opn_spec*⟩ | ⟨*opn_spec_list*⟩ ⟨*opn_spec*⟩
⟨*opn_spec*⟩ ::= ⟨*type*⟩ ⟨*opn_name*⟩() {⟨*assert_list*$_{\text{opt}}$⟩} |
⟨*type*⟩ ⟨*opn_name*⟩(⟨*param_list*⟩) {⟨*assert_list*⟩$_{\text{opt}}$}
⟨*param_list*⟩ ::= ⟨*param*⟩ | ⟨*param*⟩,⟨*param_list*⟩
⟨*param*⟩ ::= ⟨*type*⟩ ⟨*param_name*⟩
⟨*assert_list*⟩ ::= ⟨*assert*⟩ | ⟨*assert*⟩;⟨*assert_list*⟩
⟨*assert*⟩ ::= boolean expression over attributes names,
parameters, and decorated attributes names
⟨*type*⟩ ::= ⟨*class_name*⟩ | `Bool` | `Int` | ···

**Figure 1: Class specification grammar**

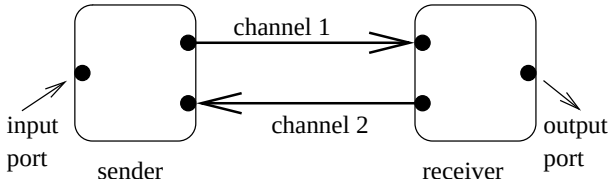



**Figure 2: Alternate Bit Protocol**

**Sender.** *The sender starts by reading a data element at the input port. Then a frame consisting of the read data element and a control bit (= a boolean value) is transmitted via channel 1 to the receiver, until a correct acknowledgment has arrived via channel 2. An acknowledgement is interpreted as being correct if the boolean value read via channel 2 is the same with the control bit. Each time a correct acknowledgement is arrived, the control bit change its value.*
**Channels.** *The channel 1 transports frames consisting a data element and a boolean value from the sender to the receiver and the channel 2 transports boolean values from the receiver to the sender. The problem with the two channels is that they are unreliable, that is the message could be damaged in transit. We assume that if something goes wrong with the message, the receiver/sender will detect this by computing a checksum. The channel are supposed to be fair in the sense that they will not produce an infinite consecutive sequence of erroneous outputs.*
**Receiver.** *The receiver starts by receiving a frame via channel 1. If the control bit of the frame is correct (i.e., different from the control bit of the receiver), then the data element is sent to the output port. Each time the frame received is not damaged (checksum is OK) and the control bit is correct, the receiver changes the value of its control bit.*

*We specify first an abstract class including only the common attributes of the sender and receiver:*

```
class AbsComp
{
  Bool bit;
  Data data;
  Bool ack;
}
```

*The class corresponding to the sender is derived from this abstract class, the class corresponding to the receiver being similar:*

```
class Sender extends AbsComp
{
  Bool chBit() {
    bit' = not bit;
    data' = data;
    ack' = ack;
  }
  void read() {
    bit' = bit;
    ack' = ack;
  }
  void sendFrame() {
    bit' = bit;
    data' = data;
    ack' = ack;
  }
  void recAck(Bool pack) {
    bit' = bit;
    data' = data;
    ack' = pack;
  }
}
```

*An assertion of the form "*`bit'` = `not bit;`*" in* `chBit()` *says that the value of the attribute* `bit` *is changed by the method. For each method, even if an attribute is not changed by its execution, this is explicitly specified. For instance, the method* `read` *is underspecified because we know nothing about the attribute* `data` *after its execution; it may have any* `Data` *value.*

Every object is an autonomous unit of execution which is either executing the sequential code of exactly one method, or passively maintaining its state. An *object instance* is a pair $(R \mid state)$, where $R$ is an object reference and $state$ is an ordered sequence of pairs $(attribute, value)$. It is not necessary to have all attributes included in a particular object instance. We consider that an object instance is a canonical element of an behavioural equivalence class, where the behavioural operations are those included in the description of the instance. The result of an execution of a method $R.m(\mathbf{d})$ over a state $st$ consists of a new state $st'$ whose attributes values are computed according to the behavioural specification of $m$; we write $st' = R.m(\mathbf{d})(st)$. For instance, we have

$$\texttt{S.chBit}()((\texttt{bit},\texttt{true}),(\texttt{ack},\texttt{false}),(\texttt{data},d)) = ((\texttt{bit},\texttt{false}),(\texttt{ack},\texttt{false}),(\texttt{data},d)).$$

We suppose that an object reference uniquely determines the class it belongs to. A *configuration* is a commutative sequence of object instances such that an object reference occurs at most once in the sequence. We also consider a special configuration *err* for signaling the occurrence of an exception.

We consider a simple set of commands:

⟨*cmd*⟩ ::= $R$ = `new` $C(\mathbf{d})$ | `delete` $R$ | $R.m(\mathbf{d})$ |
$R_1.m_1(\mathbf{d}_1) \| R_2.m_2(\mathbf{d}_2)$ | ⟨*cmd*⟩;⟨*cmd*⟩ |
`if` ⟨*bexpr*⟩ `then` ⟨*cmd*⟩ `else` ⟨*cmd*⟩ | `throw error`()

where $R, R_i$ range over object references, $m, m_i$ over methods, $\mathbf{d}, \mathbf{d}_i$ over data value sequences, and $C$ over class names. The metavariable ⟨*bexpr*⟩ denotes the boolean expressions. We omit here their

formal definition; intuitively, a boolean expression is a propositional formula written in the terms of data values, attributes, and relational operators. A boolean expression $e$ is *satisfied* by a configuration *cnfg*, written $cnfg \models e$, if and only if the evaluation of the boolean expression $e$ in the configuration *cnfg* returns true. We also assume that the sequential composition ; is associative.

The *operational semantics* is given by the labelled transition system defined by following rules:

$$cnfg \xrightarrow{R=\texttt{new}\,C(d_1,\ldots,d_n)} cnfg,(R|(att_1,d_1),\ldots,(att_n,d_n));$$

$$cnfg,(R|state) \xrightarrow{\texttt{delete}\,R} cnfg$$
(we recall that a configuration is a commutative sequence);

$cnfg \xrightarrow{R.m(\mathbf{d})} cnfg'$ if and only if $cnfg'$ is obtained from *cnfg* by replacing the object instance $(R \mid state)$ with $(R \mid state')$, where $state' = R.m(\mathbf{d})(state)$;

$cnfg \xrightarrow{R_1.m_1(\mathbf{d}_1)\|R_2.m_2(\mathbf{d}_2)} cnfg_2$ if and only if $R_1 \neq R_2$, $cnfg'$ is obtained from *cnfg* by replacing the object instance $(R_i \mid state_i)$ with $(R_i \mid state_i')$, where $state_i' = R_i.m_i(\mathbf{d}_i)(state_i)$, $i = 1,2$;

$cnfg \xrightarrow{cmd_1;cmd_2} cnfg''$ if and only if there is $cnfg'$ such that
$cnfg \xrightarrow{cmd_1} cnfg'$ and $cnfg' \xrightarrow{cmd_2} cnfg''$;

if $cnfg \xrightarrow{cmd_1} cnfg'$ and $cnfg \models e$, then
$cnfg \xrightarrow{\texttt{if}\ e\ \texttt{then}\ cmd_1\ \texttt{else}\ cmd_2} cnfg'$;

if $cnfg \xrightarrow{cmd_2} cnfg'$ and $cnfg \not\models e$, then
$cnfg \xrightarrow{\texttt{if}\ e\ \texttt{then}\ cmd_1\ \texttt{else}\ cmd_2} cnfg'$;

$$cnfg \xrightarrow{\texttt{throw error}()} err.$$

# 3. COORDINATION

We introduce a coordinating process providing a high-level description of the interaction between objects. Its syntax is inspired by process algebras as CCS and $\pi$-calculus [14]. Interaction with the environment is given by some global actions, and interaction between components is given by a nondeterministic matching between complementary local actions. Each process is described by a set of equations. In some sense, we can think such a description as an abstract system interface. The computational world of our coordinator contains processes and messages. Local actions represents interaction channels. In this way, the coordinating process models a network in which messages are sent from one object to another. This formalism is not suitable to describe state changes (the state changes are described by objects).

The process expressions $E$ are defined by guarded processes, nondeterministic choice $E_1 + E_2$, and parallel composition $E_1 \mid E_2$. We have also an empty process 0. Guarded processes are presented by either a global action followed by a process expression, an object guard followed by a process expression, or by a local action followed by a process expression. The first case describes a global action involving a state change execution of an object. The second case describes a link between the actions of an object over a certain guard, followed or not by a process expression depending on the truth evaluation of the guard. Finally, each local action *act* involves automatically the existence of its complementary local action denoted by $\tilde{}act$; these two complementary local actions establish a synchronization and a communication between objects.

A process is described as a sequence of declarations (global actions, local actions, processes and guards) followed by a set of equations. The syntax grammar for processes is:

```
proc ⟨proc_spec_name⟩
{
  global actions : ⟨lact_list⟩;
  local actions : ⟨gact_list⟩;
  processes : ⟨proc_id_list⟩;
  guards : ⟨guard_id_list⟩;
  equations :
    ⟨eqn_list⟩
}
```

where

$\langle lact\_list\rangle ::= \langle label\_list\rangle$
$\langle gact\_list\rangle ::= \langle label\_list\rangle$
$\langle label\_list\rangle ::= \langle label\rangle \mid \langle label\rangle,\ \langle label\_list\rangle$
$\langle label\rangle ::= \langle identifier\rangle \mid \tilde{}\langle identifier\rangle$
$\langle proc\_id\_list\rangle ::= \langle id\_list\rangle$
$\langle guard\_id\_list\rangle ::= \langle id\_list\rangle$
$\langle id\_list\rangle ::= \langle identifier\rangle \mid \langle identifier\rangle,\ \langle id\_list\rangle$
$\langle eqn\_list\rangle ::= \langle eqn\rangle \mid \langle eqn\rangle;\ \langle eqn\_list\rangle$
$\langle eqn\rangle ::= \langle proc\_id\rangle = \langle pexpr\rangle;$
$\langle pexpr\rangle ::= 0 \mid \langle label\rangle.\langle pexpr\rangle \mid [\langle guard\_id\rangle]\langle pexpr\rangle \mid$
$[\texttt{not}\ \langle guard\_id\rangle]\langle pexpr\rangle \mid \langle pexpr\rangle + \langle pexpr\rangle \mid$
$\langle pexpr\rangle|\langle pexpr\rangle$

The metavariable $\langle proc\_id\rangle$ denotes the identifiers occurring in processes list, and $\langle guard\_id\rangle$ denotes the identifiers occurring in guards list.

A coordinating process specification is finally given by equations of parametric process expressions. For example, the specification of ABP communication protocol as a coordination between a Sender and a Receiver can be described in the following way:

```
proc ABP
{
  global actions: in, out, alterS, alterR;
  local actions: ch1, ch2;
  processes: A, A', V, B, B', T;
  guards: sok, rok;
  equations:
    A  = in.A';
    A' = ~ch1.ch2.V;
    V  = [sok]alterS.A + [not sok]A';
    B  = ch1.T;
    T  = [rok]B' + [not rok]out.alterR.B;
    B' = ~ch2.B;
}
```

The structural operational semantics of a coordinating process specification is given by a labelled transition system. The semantic rules are presented in Figure 3. In these rules, $\gamma$ is a function mapping each guard_id into a boolean value, *gact* ranges over the labels occurring in the global actions list, *lact* ranges over the labels occurring in the local actions list, and *act* can be both *gact* or $\tau(lact)$. Based on these rules, the operational dynamics of the previous ABP

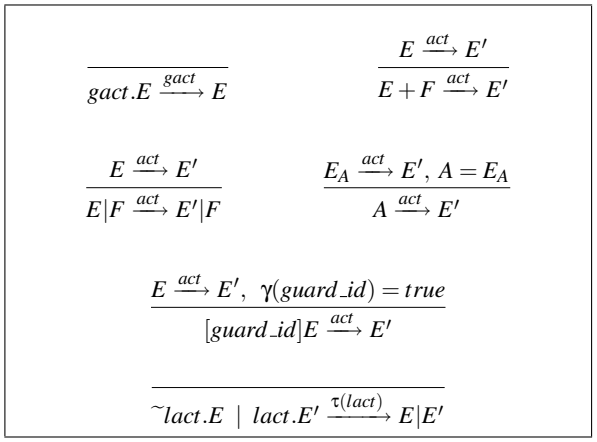


**Figure 3: The coordinating process operational rules**

process with reliable communication, modelled by $\gamma(sok) = true$ and $\gamma(rok) = false$:

$$A \mid B \xrightarrow{in} A' \mid B \xrightarrow{\tau(ch1)} ch2.V \mid T \xrightarrow{out}$$

$$ch2.V \mid alterR.B' \xrightarrow{alterR} ch2.V \mid B' \xrightarrow{\tau(ch2)}$$

$$V \mid B \xrightarrow{alterS} A \mid B.$$

We use the notation $\tau(lact)$ for the interaction between two processes prefixed by local actions $lact$ and $\tilde{}lact$, respectively. Interaction is therefore provided by pairs of actions $lact$ and $\tilde{}lact$ corresponding to some methods by means of an interaction wrapper.

# 4. INTERACTION WRAPPER

If we consider the coordinating process as an abstract interface of the system, then an interaction wrapper describes an implementation of this interface by means of a collection of objects. The coordinating process gives some directives, and the coordinated objects interpret these directives by using an interaction wrapper providing the appropriate link between the high level coordinating process and the lower level executing objects. This is the way we get a desirable separation of concerns, ensuring a suitable abstract level for designing large component-based systems without losing the details of low-level implementation of components. In some sense, our specification language has similarities with a symphonic orchestra, where independent players are synchronized by a conductor. The concert sheet followed by the conductor represent a high-level approach of the concert, and the instrumental sheet of the orchestra players are usually larger, containing more details. The link between the players and the coordinating conductor is given by certain entry moments and orchestral scores. The wrapper provides the players, and the entry scores implementing the desired resulting music at a certain moment. Therefore the wrapper provides the objects, and the necessary information for their executions in order to realize a coordinated interaction.

The syntax for the interaction wrappers is given by the grammar presented in Figure 4.

EXAMPLE 2. *The wrapper for previous described protocol ABP instruct a Sender* S *and a Receiver* R *in order to correctly follow the directives of the protocol:*

⟨*wrap_spec*⟩ ::= ⟨*wrap_name*⟩(⟨*wparam_list*⟩)
implementing ⟨*proc_spec_name*⟩
{⟨*amap_list*⟩ ⟨*gmap_list*⟩}
⟨*wparam_list*⟩ ::= ⟨*wparam*⟩ | ⟨*wparam_list*⟩; ⟨*wparam*⟩
⟨*wparam*⟩ ::= ⟨*class_name*⟩ ⟨*object_ref*⟩
⟨*amap_list*⟩ ::= ⟨*amap*⟩ | ⟨*amap_list*⟩ ⟨*amap*⟩
⟨*amap*⟩ ::= ⟨*action_name*⟩ -> ⟨*cmd*⟩;
⟨*gmap_list*⟩ ::= ⟨*gmap*⟩ | ⟨*gmap_list*⟩ ⟨*gmap*⟩
⟨*gmap*⟩ ::= ⟨*guard_name*⟩ -> ⟨*bexpr*⟩;

**Figure 4: Wrapper syntax grammar**

```
wrapper w(Sender S, Receiver R) implementing ABP
{
  in -> S.read();
  alterS -> S.chBit();
  alterR -> R.chAck();
  tau(ch1) ->
    R.recFrame(S.data, S.bit) ||
    S.sendFrame();
  tau(ch2) ->
    S.recAck(R.ack()) || R.sendAck();
  out -> R.write();
  sok -> S.bit == S.ack;
  rok -> R.bit =/= R.ack;
}
```

*A directive* in *received from the coordinating process is translated into an execution of method* read *by* S. *The directives* alterS *and* alterR *are translated into executions of methods* chBit *and* chAck *by* S *and* R*, respectively. Whenever a* $\tau(ch1)$ *directive is possible at the level of the coordinating process, it is translated into a synchronization of the methods* sendFrame *of* S *and* recFrame *of* R*. This synchronization of the autonomous objects is accompanied by a communication between them; this is given by the fact that the arguments of the receiver method* recFrame *are attributes of the sender. A similar translation is done for* $\tau(ch2)$*. Finally, the last two lines of the interaction wrapper for ABP emphasize a nice feature related to the concerns separation. Instead of using a matching or a mismatching process algebra to compare the sending bit and the received acknowledge, we clearly separate the computational and coordinating aspects by moving the comparisons at the object level, followed by a true/false result to the coordination process.*

If *act* is an action name, and **R** a sequence of object references, then $w(\mathbf{R})(act)$ denotes the command associated to *act* by the particular wrapper $w(\mathbf{R})$. The *operational semantics* of such a wrapper $w(\mathbf{R})$ is given by the labelled transition system of the objects configurations, namely

$$cnfg \xrightarrow{act} cnfg' \text{ iff } cnfg \xrightarrow{w(\mathbf{R})(act)} cnfg',$$

where *cnfg* and *cnfg'* are configurations including the instances of the objects referred by **R** and related by the command corresponding to the action name *act* via the interaction wrapper *w*.

The definition of the interaction wrapper (and its strong relationship to the dynamics of the involved objects) allows us to define an *integrated semantics* in a nice and advanced way. Taking in consideration that each configuration is supervised by a coordinating process, the whole coordination activity is described as follows:

from the current configuration $cnfg$ supervised by the process $P$, the transition $cnfg \xrightarrow{w(\mathbf{R})(act)} cnfg'$ is valid if and only if there is a process $P'$ such that $P \xrightarrow{act} P'$ and $P'$ supervises $cnfg'$.

In other words, the supervision relation is a bisimulation between the labelled transition system defined by the wrapper, and the labelled transition system defined by the coordinating process specification. This is formally defined in Section 6.

# 5. IMPLEMENTATION

We use hidden algebra [8] and Maude [5] for obtaining executable specifications for classes and their objects. In hidden algebra, an object is specified by:

1. a set $V$ of *visible sorts* with a standard interpretation; the meaning of a visible sort $v$ is a given set $D_v$ of data values;
2. a hidden sort $\mathsf{St}$ called *state sort*;
3. a set $\Sigma$ of operations including:
   (a) constants $init \in \mathsf{St}$, or generalized constants $init : v_1 \cdots v_n \rightarrow \mathsf{St}$ indicating initial states,
   (b) *methods* $g : \mathsf{St}\, v_1 \cdots v_n \rightarrow \mathsf{St}$ with $v_1, \cdots, v_n \in V$,
   (c) *attributes* $q : \mathsf{St}\, v_1 \cdots v_n \rightarrow v$ with $v, v_1, \cdots, v_n \in V$.

The properties of methods and attributes are described by equations. The main feature of hidden algebra is given by *behavioural equivalence* which abstractly characterizes an object state. Two states are behavioural equivalent if and only if they cannot be distinguished by experiments. An experiment is represented by a $\Gamma$-term with the result sort visible and with a place-holder for the state, where $\Gamma$ is a subsignature of $\Sigma$ including the *behavioural operations*. In other words, the result of an experiment is given by an attribute value after the execution of a sequence of methods over the current configuration; all the methods and attributes are in $\Gamma$. If the behavioural equivalence can be decided using only attributes, then an abstract state is characterized by its attributes values.

Each class is implemented in Maude by a module defining a hidden sort for the state of the objects, together with attributes and methods of the class. Here is the module for the sender:

```
fmod SENDER is
  sort Sender .
  inc ABP-DATA .
  op bit : Sender -> Bit .
  op data : Sender -> Data .
  op ack : Sender -> Bit .
  op send : Sender -> Sender .
  op chBit : Sender -> Sender .
  op read : Sender -> Sender .
  op recAck : Sender Bit -> Sender .
  *** equations defining properties
  *** of the methods
endfm
```

The sort `Sender` is hidden and it is used to describe the instances of the class. The sorts `Bit` and `Data` are visible and they models the data values.

A configuration of objects is a set of pairs (object reference, object state). We use the sort `ObjectReference` for object references, and the sort `ObjectState` for object states. For each class we add a distinguished subsort of `ObjectReference` and a distinguished subsort of `ObjectState`.

```
fmod CONFIG is
  sorts ObjectReference ObjectState EmptyConfig Config .
  subsort EmptyConfig < Config .
  op empty : -> EmptyConfig .
  op `(_|_`) : ObjectRef ObjectState -> Config .
  op _`,_ : Config Config -> Config
          [assoc comm id: empty] .
  op _.read$`(`)_ : ObjectRef Config -> ObjectState .
  op _.update$`(_`)_ :
     ObjectRef ObjectState Config -> Config .
  *** equations defining properties of read$()_
  *** and update$(_)_
endfm
```

The composition `_,_` of two configurations is specified as being commutative, associative, and having the identity `empty`.

For ABP we have a configuration containing objects of classes Sender, Receiver and Channel. The hidden sorts defined in `SENDER`, `RECEIVER`, and `CHANNEL` are the state sorts and we make them subsorts of the ObjectState sort. We also add three distinguished sorts for the references to these objects. The ABP configurations expose the methods and attributes of three classes using attributes and methods with similar names and arguments. We show here only the `bit` attribute and the `recFrame` method.

```
fmod ABP-CONFIG is
  inc CONFIG + SENDER + RECEIVER + CHANNEL .
  subsort Sender < ObjectState .
  subsort Receiver < ObjectState .
  subsort Channel < ObjectState .
  sort ObjectRef<SENDER> .
  subsort ObjectRef<SENDER> < ObjectRef .
  sort ObjectRef<RECEIVER> .
  subsort ObjectRef<RECEIVER> < ObjectRef .
  sort ObjectRef<CHANNEL> .
  subsort ObjectRef<CHANNEL> < ObjectRef .

  op _.bit`(`)_ : ObjectRef<SENDER> Config -> Bool .
  eq S .bit() C = bit(S .read$() C) .

  op _.recFrame`(_`,_`)_ :
    ObjectRef<RECEIVER> Data Bit Config -> Config .
  eq S .recFrame(D, B) C = recFrame(S .read$() C, D, B) .

  *** other methods and attributes,
  *** and their equations
endfm
```

The initial ABP configuration contains a `Sender` object referred by the reference `S`, a `Receiver` object referred by the reference `R`, a data channel referred by the reference `CHD`, and an acknowledge channel referred by the reference `CHA`. We have the constants `initS`, `initR`, `initCHD` and `initCHA` as the initial states for each object, and `init` as the initial configuration.

```
fmod ABP is
  inc CONFIG<SENDER+RECEIVER> .
  op S : -> ObjectRef<SENDER> .
  op R : -> ObjectRef<RECEIVER> .
  ops CHD CHA : -> ObjectRef<CHANNEL> .
  op initS : -> Sender .
  op initR : -> Receiver .
  ops initCHD initCHA : -> Channel .

  eq bit(initS) = b1 .
  eq ack(initS) = b0 .
  eq data(initS) = d1 .
  *** other equations for initR, initCHD, initCHA

  op init : -> Config .

  eq init = < S | initS >, < R | initR >,
    < CHD | initCHD >, < CHA | initCHA > .
endfm
```

Since Maude implements rewriting logic, it is capable to specify a process algebra [18]. The following Maude module defines the syntax of our process algebra:

```
mod PROC is
  sorts Action Process Guard
        ActionProcess ActionProcessSeq .
  subsort ActionProcess < ActionProcessSeq .
  op ~_ : Action -> Action .
  op tau_ : Action -> Action .
  op _._ : Action Process -> Process  [strat(0) frozen] .
  op _|_ : Process Process -> Process [frozen assoc comm] .
  op _+_ : Process Process -> Process [frozen assoc comm] .
  op [_]_ : Guard Process -> Process  [frozen] .
  op _\_ : Process Action -> Process  [frozen] .
  op next : Process Config -> ActionProcessSeq .
  *** equations
endm
```

The operations are declared as being "frozen", i.e., we forbid the use of the rewriting rules in the evaluation of the arguments. An action is represented by a constant of sort `Action`, or by a term of sort `Action` as it is $\sim$`a` which identifies the complemented action of `a`. We denote by `tau(a)` the special action $\tau$ generated by a pair `(a, ~a)`. The distinction between local and global actions is given by the restriction operator $P \setminus L$, where $L$ is (a subset of) the set of local actions. This syntax is closer to that used for CCS [18].

Given a process, we want to have the list of actions which can be executed. For this we introduce `next` operation; `next` builds a list of pairs (action, process) for a specific process and configuration. For concurrent processes, the list of pairs (action, process) contains the possible interleavings of actions which can be executed by each process, together with the possible communications between each pair of processes.

```
mod ABP-PROC is
  inc PROC .
  ops ABP S S1 V CHD CHA R T R1 : -> Process .
  ops in alterS alterR out schd
    scha rchd rcha chdt chat : -> Action .
  ops rOK sOK : -> Guard .

  rl S => in . S1 .
  rl S1 => ~ schd . scha  . V .
  rl V => ([sOK](alterS . S)) + ([not sOK]S1) .
  rl CHD => schd . chdt . ~ rchd . CHD .
  rl CHA => rcha . chat . ~ scha . CHA .
  rl R => rchd . T .
  rl T => ([not rOK]R1) + ([rOK](~ out . alterR . R1)) .
  rl R1 => ~ rcha . R  .
  rl ABP => ( S | CHD | CHA | R )
    \ scha \ schd \ rcha \ rchd .
endm
```

The interaction wrapper is described by the following module:

```
mod ABP-COORDINATED is
  inc ABP + ABP-PROC .
  op w : Action Config -> Config .
  var C : Config .
  eq w(in, C) = S .read() C .
  eq w(tau schd, C) =
     CHD .read( S .bit() C,
                S .data() C ) S .send() C .
  eq w(chdt, C) = CHD .transfer() C .
  eq w(tau rchd, C) =
     R .recFrame( CHD .data() C,
                  CHD .bit() C ) CHD .send() C .
  eq w(~ out, C) = R .write() C .
  eq w(alterR, C) = R .chAck() C .
  eq w(tau rcha, C) =
     CHA .read( R .ack() C , null ) R .sendAck() C .
  eq w(chat, C) = CHA .transfer() C .
  eq w(tau scha, C) =
     S .recAck( CHA .bit() C ) CHA .send() C .
  eq w(alterS, C) = S .chBit() C .

  eq eval(sOK, C) = (S .bit() C == S .ack() C) and
                    (CHA .error() C == b0) .
  eq eval(rOK, C) = (CHD .error() C == b0) and
                    (R .bit() C =/= R .ack() C) .
endm
```

# 6. FORMAL SEMANTICS OF THE IMPLEMENTATION

When designing a software system it is important to clarify its intended purpose. In [17], it is expressed that if the main purpose is to support reasoning and formal approach, then the system should strive for minimality; on the other hand if the main purpose is to be intuitive for the practitioners, then it should reflect the expected intuitions and practices. We want to put these purposes together, offering both intuitive languages for components, coordinator and wrapper, and supporting formal approaches and reasoning. We achieve this goal by using the formal models of object specification in hidden algebra, and a coalgebraic treatment of the labelled transition systems for the coordinator and wrapper.

A *model* for a class specification in hidden algebra is a $\Sigma$-algebra $M$ such that $M_v = D_v$ for each visible sort $v \in V$. The $M$-interpretation of a $\Gamma$-context $c$ is a function which maps a variable assignment $\vartheta : X \to D$ and a state $st$ to the value $[\![\mathfrak{c}]\!]_M(\vartheta)(st)$ obtained by replacing the occurrences of $\_$ in $\mathfrak{c}$ by $st$, the occurrences of $x \in X$ by $\vartheta(x)$, and evaluating the operations in $\mathfrak{c}$ according to their interpretation in $M$. The *behavioural equivalence* relation $\equiv$ over the set of states $M_{\mathsf{St}}$ is defined as follows: $st \equiv st'$ iff $[\![\mathfrak{c}]\!]_M(\vartheta)(st) = [\![\mathfrak{c}]\!]_M(\vartheta)(st')$ for each $\Gamma$-experiment $c$.

If $\mathcal{B}$ is a specification of concurrent objects, then a $\mathcal{B}$-model consists of a model for each class together with a model for their instances and configurations. This can be expressed in the terms of models for structured specifications [6].

The operational semantics of a process algebra is given by a labelled transition system. We prefer to represent a labelled transition system as a coalgebra. We denote by $\mathsf{Set}$ the category of sets. Let $A$ be a given set of *action names*. Let $\mathsf{T_{LTS}} : \mathsf{Set} \to \mathsf{Set}$ the functor given by

$$\mathsf{T_{LTS}}(X) = \{Y \subseteq A \times X \mid Y \text{ finite}\}$$

for each set $X$, and $\mathsf{T_{LTS}}(f)$ is the function $\mathsf{T_{LTS}}(f) : \mathsf{T_{LTS}}(X) \to \mathsf{T_{LTS}}(X')$ given by

$$\mathsf{T_{LTS}}(f)(Y) = \{(a, f(x)) \in A \times X' \mid (a,x) \in Y\}$$

for each function $f : X \to X'$. A labelled transition system associated to a process algebra is a coalgebra $\pi : P \to \mathsf{T_{LTS}}(P)$, where $P$ is a set of *processes*. We have $p \xrightarrow{act} q$ iff $(act, q) \in \pi(p)$.

Considering a wrapper $w(\mathbf{R})$, we define a coalgebra $w(\mathbf{R})_M : M_{\mathsf{Config}} \to \mathsf{T_{LTS}}(M_{\mathsf{Config}})$ by $(act, cnfg') \in w(\mathbf{R})_M(cnfg)$ iff one of the following two conditions holds:

1. if the result sort of $w(\mathbf{R})(act)$ is $\mathsf{Config}$, then $cnfg' = [\![w(\mathbf{R})(act)]\!]_M(cnfg)$, and
2. if $w(\mathbf{R})(act) = O.q(X_1, \ldots, X_n)(Y)$ with $q$ an attribute, then $cnfg' = [\![Y]\!]_M(cnfg)$.

Moreover, we suppose that there is a coalgebra

$$w(\mathbf{R})_M/_{\equiv} : M_{\mathsf{Config}}/_{\equiv} \to \mathsf{T_{LTS}}(M_{\mathsf{Config}}/_{\equiv})$$

which commutes the following diagram:

$$\begin{array}{ccc} M_{\mathsf{Config}} & \xrightarrow{can} & M_{\mathsf{Config}}/_{\equiv} \\ {\scriptstyle w(\mathbf{R})_M}\downarrow & & \downarrow{\scriptstyle w(\mathbf{R})_M/_{\equiv}} \\ \mathsf{T_{LTS}}(M_{\mathsf{Config}}) & \xrightarrow[\mathsf{T_{LTS}}(can)]{} & \mathsf{T_{LTS}}(M_{\mathsf{Config}}/_{\equiv}) \end{array}$$

where *can* denotes the canonical onto morphism. The behavioural equivalence must be preserved by the transitions defined by $w(\mathbf{R})_M$. Therefore the action terms include only behavioural congruent operations, the transitions

are extended to the quotient model, and we require the commutativity of the above diagram. The coalgebra $w(\mathbf{R})_M$ represents the labelled transition system $cnfg \xrightarrow{act} cnfg'$ defined in Section 4.

A $(\pi, w(\mathbf{R}))$-*coordinated* $\mathcal{B}$-*model* is a triple $(M, \gamma, proc)$, where $M$ is a $\mathcal{B}$-model, $proc : M_{\mathsf{Config}}/_{\equiv} \to P$ is a partial colouring operation (supervising operation), and $\gamma : dom(proc) \to \mathsf{T}_{\mathsf{LTS}}(M_{\mathsf{Config}})$ is a coalgebra commuting the following diagram:

$$\begin{array}{ccccc} M_{\mathsf{Config}}/_{\equiv} & \xleftarrow{\text{id}} & dom(proc) & \xrightarrow{proc} & P \\ {\scriptstyle w(\mathbf{R})_M}\downarrow & & \downarrow{\scriptstyle \gamma} & & \downarrow{\scriptstyle \pi} \\ \mathsf{T}_{\mathsf{LTS}}(M_{\mathsf{Config}}/_{\equiv}) & \xleftarrow[\mathsf{T}_{\mathsf{LTS}}(\text{id})]{} & \mathsf{T}_{\mathsf{LTS}}(dom(proc)) & \xrightarrow[\mathsf{T}_{\mathsf{LTS}}(proc)]{} & \mathsf{T}_{\mathsf{LTS}}(P) \end{array}$$

The rectangle from the right-hand side of the diagram says that *proc* is a homomorphism of coalgebras. We take the advantage of using hidden algebra, and we define the configuration supervisor as an attribute. By defining *proc* as a partial function, we restrict the coordinator to be aware of the coordinated configurations, and nothing more.

Each $(\pi, w(\mathbf{R}))$-coordinated $\mathcal{B}$-model defines a bisimulation:

PROPOSITION 1. *Let* $\gamma^{\sim} : graph(proc) \to \mathsf{T}_{\mathsf{LTS}}(graph(proc))$ *be the coalgebra given by* $(a, \langle cnfg', p' \rangle) \in \gamma^{\sim}(\langle cnfg, p \rangle)$ *iff* $proc(cnfg) = p$, $proc(cnfg') = p'$, *and* $(act, cnfg') \in \gamma(cnfg)$. *Then* $\gamma^{\sim}$ *is a bisimulation between* $w(\mathbf{R})_M$ *and* $\pi$.

A *homomorphism* of $(\pi, w(\mathbf{R}))$-coordinated $\mathcal{B}$-models from $(M, \gamma, proc)$ to $(M', \gamma', proc')$ consists of a $\Sigma$-homomorphism $h : M \to M'$ such that $proc'(h_{\mathsf{Config}}(cnfg)) = proc(cnfg)$ for all $cnfg \in dom(proc)$.
Let $\mathsf{Mod}(\mathcal{B}, \pi, w(\mathbf{R}))$ denote the category of the $(\pi, w(\mathbf{R}))$-coordinated $\mathcal{B}$-models. The following result shows that a homomorphism preserves the coalgebraic structure:

PROPOSITION 2. *Let* $h : M \to M'$ *be a homomorphism of* $(\pi, w(\mathbf{R}))$-*coordinated* $\mathcal{B}$-*models defined as above.*
*Then* $h_{\mathsf{Config}} : dom(proc) \to dom(proc')$ *is a coalgebra homomorphism from* $\gamma$ *to* $\gamma'$.

# 7. TEMPORAL PROPERTIES OF THE COORDINATED OBJECTS

Since the semantics of the coordinated objects is given by labelled transitional systems, we are able to use the temporal formulas for describing their properties. We use *Computational Tree Logic* (CTL) [4]. CTL is a branching time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be the "actual" path that is desired. The CTL *formulas* are inductively defined as follows:

$$p ::= R.att()(\_) = d \mid R.att(d_1, \ldots, d_n)(\_) = d$$
$$\phi ::= \mathtt{tt} \mid \mathtt{ff} \mid p \mid (\neg\phi) \mid (\phi_1 \wedge \phi_2) \mid \mathtt{EX}\phi \mid \mathtt{EG}\phi \mid \mathtt{E}[\phi_1 \,\mathtt{U}\, \phi_2].$$

The intuitive meaning of an atomic proposition of the form $R.att()(\_) = d$ is "the value of the attribute *att* for the object *R* in the current configuration is *d*". The meaning for each propositional connective is the usual one. The set of propositional connectives is extended in the standard way. The temporal connectives are pairs of symbols. The first is E, meaning "there Exists one path". The second one is X, G, or U, meaning "neXt state", "all future state (Globally)", and "Until", respectively. We can express other five operators, where A means "for All paths":

$$\mathtt{EF}\phi = \mathtt{E}[\mathtt{tt}\,\mathtt{U}\,\phi] \qquad \mathtt{AX}\phi = \neg\mathtt{EX}(\neg\phi)$$
$$\mathtt{AG}\phi = \neg\mathtt{EF}(\neg\phi) \qquad \mathtt{AF}\phi = \neg\mathtt{EG}(\neg\phi)$$
$$\mathtt{A}[\phi_1 \,\mathtt{U}\, \phi_2] = \neg\mathtt{E}[\neg\phi_2 \,\mathtt{U}\, (\neg\phi_1 \wedge \neg\phi_2)] \wedge \neg\mathtt{EG}\neg\phi_2$$

The satisfaction relation $cnfg \models p$, expressing that a configuration *cnfg satisfies* a CTL atomic proposition $p$, is defined as follows:

1. $cnfg \models R.att()(\_) = d$ if and only if $cnfg = cnfg_1, (R|state)$ and $R.att()(state) = d$.
2. $cnfg \models R.att(d_1, \ldots, d_n)(\_) = d$ if and only if $cnfg = cnfg_1, (R|state)$ and $R.att(d_1, \ldots, d_n)(state) = d$.

Then the satisfaction relation $cnfg \models \phi$ is extended to arbitrary CTL formulas $\phi$ in the standard way [4].

For the ABP example, the following CTL formula:

$$\mathtt{AG}((\mathtt{S.bit}()(\_) = \mathtt{true} \wedge \mathtt{R.ack}()(\_) = \mathtt{false} \wedge \mathtt{S.data}()(\_) = \mathtt{d}) \rightarrow$$
$$\mathtt{AF}(\mathtt{S.bit}()(\_) = \mathtt{false} \wedge \mathtt{R.ack}()(\_) = \mathtt{true} \wedge \mathtt{R.data}()(\_) = \mathtt{d}))$$

expresses the fact that if in the current configuration the sender `S` has the `bit` equal to `true` and the sending `data` equal to $d$, the receiver `R` has the `ack` equal to `false`, then always there is in the future a configuration where `S` has the `bit` equal to `falsa`, `R` has the `ack` equal to `false` and the received `data` equal to $d$.

The temporal formulas are verified using a model-checking algorithm [4]. Our approach consists in extracting an SMV description [13] of the labelled transition system in a similar way to that described in [11]. The advantage of this method is that it allows the use of underspecified methods and of CTL formulas.

The algorithm building the Kripke structure is implemented by a Maude "built-in" operation called `writeSmv`. This operation yields a SMV module describing the Kripke structure [13]. The signature of this operation is given as follows:

```
mod SMV-WRITER is
  inc CTL .

  op writeSmv : Config Process FormulaSeq FormulaSeq
    String -> Nat [special (...)] .
endm
```

The first two arguments are the starting configuration and process needed to build the Kripke structure. The third argument is a sequence of CTL formula that represents the fairness constraints. The fourth argument is a sequence of CTL formula that represents the temporal properties to be verified. The last argument is the name of the output file. The result of this operation is the number of states generated.

The use of the `writeSmv` operation for the ABP correctness formula is as follows:

```
red writeSmv(
    init, ABP, none,
    (AG(
        AP(S .bit() C == b1)
        & AP(R .ack() C == b0)
        & AP(S .data() C == d1)
       ->
            AP(S .bit() C == b0)
            & AP(R .ack() C == b1)
            & AP(R .data() C == d1))
        )
    ),
    "abp.smv") .
```

It is worth to note that we use a simplified form of the correctness formula. The execution of the SMV model checker over the input file "abp.smv" provides the following result:

```
-- specification
    !E(1 U (S_bit__ = b1 & R_ack__ = b0 & ... is false
```

SMV says that the correctness property does not hold, and provides a counterexample. The counterexample shows an infinite path where a channel always generates errors. We recall that ABP works properly under the assumption that the channels are fair, i.e., they do not produce such infinite sequences of errors. This assumption can be easily specified by adding a fairness constraint for each channel.

```
red writeSmv(
    init, ABP,
    (AP(CHD .error() C == b0)
        & AP(CHD .procName() C == transfer>),
     AP(CHA .error() C == b0)
        & AP(CHA .procName() C == transfer>)),
    (AG(
        AP(S .bit() C == b1)
        & AP(R .ack() C == b0)
        & AP(S .data() C == d1)
       ->
            AF(AP(S .bit() C == b0)
            & AP(R .ack() C == b1)
            & AP(R .data() C == d1))
        )
    ),
    "abp.smv") .
```

The two fairness constraints are specified by the lines 3-6. Using these constraints, SMV succeeds to prove the correctness of ABP under the fairness assumption:

```
-- specification
    !E(1 U (S_bit__ = b1 & R_ack__ = b0 & ... is true
resources used:
processor time: 0.062 s,
BDD nodes allocated: 10057
Bytes allocated: 1695620
BDD nodes representing transition relation: 1191 + 1
```

# 8. CONCLUSION

Modelling complex systems out of components and building corresponding applications is currently a challenge for software community. This task assumes the description of the whole system from different points of view: data, concurrency, synchronization, communication, coordination. Since each specific aspect related to data, concurrency, and coordinated component-based systems can be described using a specific formalism, the final description of the system could be a multiformalism specification. HiddenCCS formalism introduced in [2, 3] is a formal specification framework based on hidden algebra and CCS. This specification extends the object specification with synchronization and communication elements associated with methods and attributes of the objects, and use a CCS description of the interaction patterns. The operational semantics of hiddenCCS specifications is based on labelled transition systems. Another related multiformalism can also be found in [16]. Frølund and Agha [7] introduce independent support constructs for coordination of concurrent objects using synchronizers. We extend their synchronizers in providing a more general and elegant approach based on process algebra and related notions.

In [1] the authors focus on a formal basis for one aspect of software architecture design, namely the interactions between components. They define architectural connectors as explicit semantic entities characterizing the participants roles in an interaction. A system is described by its components and connectors. A variant of CSP is used to define the roles, ports, and glue specifications. An important motivation for the authors is represented by the potential for automating the analysis of architectural description by using tools to determine whether connectors are well formed, and ports are compatible with their roles. In [12], the authors describe a distributed a software architecture in terms of its components and their interactions, associating behavioural specifications with the components and then checking whether the system satisfies certain properties. The approach is based on labelled transition systems to specify the behaviour, and compositional reachability analysis to check composite system models. Our approach add an important feature to these architecture description languages, namely a useful separation of concerns. Moreover, we use an efficient model checker for verifying temporal properties of a component-based system.

More precisely, in this paper we design a specification language for coordinated objects with a syntax closer to OOP languages, and a semantics of coordination given by an integrating bisimulation. The interaction between the coordinating process and autonomous objects is given via interaction wrapper. Operational semantics of the coordinated objects is given by labelled transition systems, and we express their temporal properties in CTL. These temporal properties can be verified automatically by using adapted algorithms and new specific tools.

# 9. REFERENCES


[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, vol.6, pp.213-249, 1997.

[2] G. Ciobanu and D. Lucanu. Specification and verification of synchronizing concurrent objects. In J.Derrick E.Boiten and G.Smith (Eds.): *Integrated Formal Methods 2004*, Lecture Notes in Computer Science, vol.2999, pp.307-327, Springer, 2004.

[3] G. Ciobanu and D. Lucanu. Communicating Concurrent Objects in HiddenCCS. *Electronic Notes in Theoretical Computer Science*, vol.117, pp.353-373, Elsevier, 2004.

[4] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada Maude: Specification and Programming in Rewriting Logic *Theoretical Computer Science*, vol. 285, pp.187-243, 2002.

[6] F. Durán and J. Meseguer. Structured Theories and Institutions. *Theoretical Computer Science*, vol.309, pp.357-380, 2003.

[7] S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proc. ECOOP'93*, Lecture Notes in Computer Science, vol. 707, pp.346-360, Springer, 1993.

[8] J. Goguen, and G. Malcolm. A hidden agenda. *Theoretical Computer Science* vol.245, pp.55-101, 2000.

[9] Gh. Grigoraş and D. Lucanu. On Hidden Algebra Semantics of Object Oriented Languages. *Sci. Ann. of the "A.I.Cuza" Univ. of Iaşi*, 14(Computer Science), pp.51-68, 2004.

[10] B. Jacobs. Coalgebraic Reasoning about Classes in Object-Oriented Languages. In Electronic Notes in Theoretical Computer Science, vol.11, pp.231-242, Elsevier, 1998.

[11] D. Lucanu, and G. Ciobanu. Model Checking for Object Specifications in Hidden Algebra. In B.Steffen, G.Levi (Eds.) *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science vol.2937, Springer, pp.97-109, 2004.

[12] J. Magee, J. Krammer, and D. Giannnakopoulou. Analysing the Behaviour of Distributed Software Architecture: a Case Study. In *Proc. of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, Tunis, pp.240-245, 1997.

[13] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[14] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[15] J. Rutten. Universal Coalgebra: A Theory of Systems. *Theoretical Computer Science* vol.249, pp.3-80, 2000.

[16] G. Salaün, M. Allemand, and C. Attiogbé. A Formalism Combining CCS and CASL. Research Report 00.14, IRIN, Univ. Nantes, 2001.

[17] M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. In J. van Leeuwen (Ed.): *Computer Science Today: Recent Trends and Developments*, Lecture Notes in Computer Science, vol.1000, Springer, 1995.

[18] A. Verdejo, and N. Martí-Oliet. Implementing CCS in Maude 2. In *4th WRLA*, *Electronic Notes in Theoretical Computer Science*, vol.71, pp.239-257, Elsevier, 2002.

# Component-Interaction Automata as a Verification-Oriented Component-Based System Specification

Luboš Brim[*] , Ivana Černá[†] , Pavlína Vařeková[‡] , Barbora Zimmerova[‡]
Faculty of Informatics
Masaryk University, Brno
602 00 Brno, Czech Republic
{brim,cerna,xvareko1,zimmerova}@fi.muni.cz

## ABSTRACT

In the paper, we present a new approach to component interaction specification and verification process which combines the advantages of both architecture description languages (ADLs) at the beginning of the process, and a general formal verification-oriented model connected to verification tools at the end. After examining current general formal models with respect to their suitability for description of component-based systems, we propose a new verification-oriented model, *Component-Interaction automata*, and discuss its features. The model is designed to preserve all the interaction properties to provide a rich base for further verification, and allows the system behaviour to be configurable according to the architecture description (bindings among components) and other specifics (type of communication used in the synchronization of components).

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.11 [**Software Engineering**]: Software Architecture

## General Terms

Component-based specification languages

## Keywords

ADLs, I/O automata, Interface automata, Team automata, Component-Interaction automata, component interaction, verification.

[*]The author has been partially supported by the grant GACR 201/03/0509

[†]The author has been partially supported by the grant No. 1ET408050503

[‡]The authors have been supported by the grant No. 1ET400300504

## 1. INTRODUCTION

Verification of interaction properties in component-based software systems is highly dependent on the chosen specification language. There are many possibilities how to specify component behaviour and interaction in component-based software systems. The specification languages typically fall into two classes with diverse pros and cons.

The first set of specification languages is called Architecture Description Languages (ADLs). Architecture description languages, like Wright [3], Darwin/Tracta [15, 16], Rapide [11], and SOFA [17, 2], are very suitable for specification of hierarchical component architecture with defined interconnection among components and behaviour constraints put on component communication and interaction. Moreover they are very comprehensible for software engineers and often provide a tool support. The essential drawback of the ADLs is that their specification power is limited by the underlying model which is often not general enough to preserve all the interaction properties which might arise through the component composition. Additionally, the verification within an ADL framework usually supports a verification of only a small fixed set of properties often unique for the language.

The second set consists of general formal models usually based on the automata theory (I/O automata [14, 12], Interface automata [8], Team automata [5]). These automata-based models (as opposite to ADLs) are highly formal and general, and usually supported by automated verification tools (model-checkers in particular). However, these models are designed for modelling of component interaction only and therefore are unable to describe the interconnection structure of hierarchical component architecture which also influences the behaviour. That is one of the reasons why these models are often considered to be unusable in software engineering practice.

The point we want to address in our research is to combine these two approaches to gain the benefits of both of them. In particular, we would like to develop a general automata-based formalism which allows for the specification of component interactions according to the interconnection structure described in particular ADL. The transition set of the model should be therefore configurable according to the architecture description (bindings among components) and other specifics (type of communication used in the synchronization of components). In addition, the formalism should allow for an easy application of available automated verifica-

tion techniques. The idea is to support the specification and verification process automatically or semi-automatically so that it is accessible also for users with no special theoretical knowledge of the underlying model. The specification and verification process will constitute of the following phases.

1. The user selects an appropriate ADL and specifies the system architecture and component behaviour using an ADL tool.
2. Component behaviour description is transformed into the general formal model automatically using the model framework.
3. The hierarchical component composition is build within the framework with respect to the architecture description and synchronization type.
4. The result is verified directly within the model framework or transformed to a format accepted by verification tools.

In this paper we want to address the first step towards this goal. We propose an appropriate general verification-oriented specification formalism which covers all important features of component interaction in component-based systems, including hierarchical interconnection interaction, and enables its adjustment to the ADL specification. At the same time, the model is defined in such a way that a direct application of model checking techniques is possible.

The paper is organised as follows. After discussing related work in Section 2, Section 3 focuses on the automata-based models appropriate for component interaction specification, and gives the reasons for introduction of a new model in Section 4. Section 5 concludes by discussion the most important features of the proposed model, and presents the plans for future work.

## 2. RELATED WORK

Our approach to support the specification and verification process by combining ADL specification and a general formal model verification has not been considered yet. The reason is that the architecture description languages usually support some kind of verification of the interaction properties and therefore there is no visible need for use of a new general verification-oriented model.

Some of the architecture description languages addressing the issue of formal verification of behaviour properties of the system composed from components are *Wright* [3], *Darwin/Tracta* [15, 16], *Rapide* [11] and *SOFA* [17, 2]. *Wright* uses consistency and completeness checks defined in terms of its underlying model in CSP. Verification of component behaviour in *Darwin* is supported by the *Tracta* approach which defines component interactions using labelled transition systems (LTS) and employs model checking [10] to verify some of its properties (reachability analysis, safety and liveness properties). *Rapide* generates a system execution in a form of partially ordered set of events and allows its check against properties. *SOFA* uses *behavior protocols* to specify behaviour of a component frame (black-box specification view) and architecture (grey-box implementation view) and employs a compliance checking to verify their conformance.

As we emphasised in the introduction, these languages typically support verification of a limited set of interaction properties. Even if some ADLs attempt to support an exhaustive verification [10], they are limited by the underlying behaviour model which is usually designed for one particular type of communication and notion of erroneous behaviour and thus does not cover some important interaction properties. For example the parallel composition operator $\|$ used in Tracta does not assure, that we will be later able to detect all states where one of the (sub)components is ready to synchronize on a shared action but the others are not because in Tracta's notion of communication it is not an interesting behaviour to capture.

Another approach is a support of the specification and verification process using the formal general language only (I/O automata [14], Interface automata [8], Team automata [5]). The essential drawback of this approach is that the models specify just the interaction behaviour without describing the underlying architectural framework. That is restrictive especially when the interconnection structure of a system differs from the complete interconnection space determined by the actions shared among components.

## 3. AUTOMATA-BASED LANGUAGES

As already mentioned in the introduction, automata-based models are typically supported by automated verification tools. In this section we primarily concentrate on their applicability for capturing of behaviours of component-based systems, especially the interaction among components of the system. The best known models used in this context are I/O automata, Interface automata, and Team automata. For each of the models we give a brief definition and review its main features interesting in a light of modelling component-based systems.

*Notation:* Let $\mathcal{I} \subseteq \mathbb{N}$ be a finite set with cardinality $k$, and let for each $i \in \mathcal{I}$, $S_i$ be a set. Then $\Pi_{i\in\mathcal{I}} S_i$ denotes the set $\{(x_{i_1}, x_{i_2}, \ldots, x_{i_k}) \mid (\forall j \in \{1, \ldots, k\} : x_{i_j} \in S_{i_j}) \wedge \{i_1, i_2, \ldots, i_k\} = \mathcal{I} \wedge (\forall j_1, j_2 \in \{1, \ldots, k\} : j_1 < j_2 \Rightarrow i_{j_1} < i_{j_2})\}$. If $\mathcal{I} = \emptyset$ then $\Pi_{i\in\mathcal{I}} S_i = \emptyset$. For $j \in \mathcal{I}$, $proj_j$ denotes the function $proj_j : \Pi_{i\in\mathcal{I}} S_i \to S_j$ for which $proj_j((q_i)_{i\in\mathcal{I}}) = q_j$.

### 3.1 I/O automata

The *Input/Output automata* model (*I/O automata* for short) was defined by Nancy A. Lynch and Mark R. Tuttle in [18, 13] as a labelled transition system model based on nondeterministic automata. The I/O automata model is suitable for modelling distributed and concurrent systems with different input, output and internal actions. I/O automata can be composed to form a higher-level I/O automaton and thus form a hierarchy of components of the system.

**Definition:** A *(safe) I/O automaton* is a tuple $\mathcal{A} = (Q, \Sigma_{inp}, \Sigma_{out}, \Sigma_{int}, \delta, I)$, where

- $Q$ is a set of states.
- $\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}$ are pairwise disjoint sets of *input, output* and *internal* actions, respectively. Let $\Sigma = \Sigma_{inp} \cup \Sigma_{out} \cup \Sigma_{int}$ be called a *set of actions*.
- $\delta \subseteq Q \times \Sigma \times Q$ is a set of *labelled transitions* such that for each $a \in \Sigma_{inp}$ and $q \in Q$ there is a transition $(q, a, q') \in \delta$ (*input enableness*).
- $I \subseteq Q$ is a nonempty set of *initial states*.

Important feature to mention is that I/O automata are input enabled in all states, they can never block the input. It

means that in I/O automata we are unable to directly reason about properties capturing that a component $A$ is ready to send output action $a$ to a component $B$ which is not ready to receive it (e.g. needs to finish some computation first). Other feature to notice is that the sets of input, output and internal actions of I/O automaton have to be pairwise disjoint. It can limit us in modelling some practical systems. For example when we want to model a system in Figure 1, consisting of $n$ component instances of the same type that enable event delegation (the component $C_i$ can delegate the method call which received from the component $C_{i-1}$ to the component $C_{i+1}$), we cannot do it directly. We have to use appropriate relabelling.

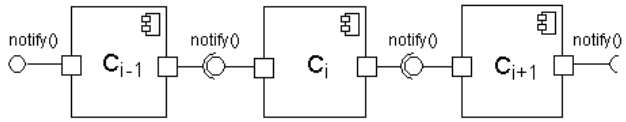



**Figure 1: Delegation of a method call, UML 2.0**

A set of I/O automata is *strongly compatible* if the sets of output actions of component automata are pairwise disjoint and the set of internal actions of every component automaton is disjoint with the action sets of all other component automata. Therefore a set of automata where two or more automata have the same output action is not strongly compatible and cannot be composed according to the next definition. At the same time this property is quite often in practical component-based systems, for example when two components are using the same service of another component. This problem could be solved by relabelling of the transitions. Note that simple including the identity of the component in the action name would not suffice because I/O automata are able to synchronize only on actions with the same name.

**Definition:** Let $\mathcal{S} = \{(Q_i, \Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}, \delta_i, I_i)\}_{i\in\mathcal{I}}$, where $\mathcal{I} \subseteq \mathbb{N}$ is finite, be a strongly compatible set of I/O automata. An I/O automaton $(\Pi_{i\in\mathcal{I}}Q_i, \Sigma_{inp}, \Sigma_{out}, \Sigma_{int}, \delta, \Pi_{i\in\mathcal{I}}I_i)$ is a *composition* of $\mathcal{S}$ iff

- $\Sigma_{inp} = \left(\bigcup_{i\in\mathcal{I}} \Sigma_{i,inp}\right) \setminus \left(\bigcup_{i\in\mathcal{I}} \Sigma_{i,out}\right)$,
- $\Sigma_{out} = \bigcup_{i\in\mathcal{I}} \Sigma_{i,out}$,
- $\Sigma_{int} = \bigcup_{i\in\mathcal{I}} \Sigma_{i,int}$ and
- for each $q, q' \in \Pi_{i\in\mathcal{I}}Q_i$ and $a \in \Sigma$, $(q, a, q') \in \delta$ iff for all $i \in \mathcal{I}$ if $a \in \Sigma_i$ then $(proj_i(q), a, proj_i(q')) \in \delta_i$ and if $a \notin \Sigma_i$ then $proj_i(q) = proj_i(q')$.

In the composition of strongly compatible I/O automata each input action $a$, for which an appropriate output action $a$ exists, is removed to preserve the condition of disjoint input and output action sets. The input actions then cannot be delegated out of the composed component to be linked in a higher level of composition. Another property of I/O automata to mention is that they do not allow to specify which outputs and inputs should be bound and which should stay unbound (according to the architecture description or type of synchronization).

## 3.2 Interface automata

The *Interface automata* model [8] was introduced in [7] by Luca de Alfaro and Thomas A. Henzinger. The model is designed for documentation and validation of systems made of components communicating throw their interfaces. Interface automata, as distinct from I/O automata, are not input enabled in all states and allow composition of two automata only. Moreover, composition is based on synchronization of one output and one input action (with the same name) which becomes hidden after the composition. That is natural for practical component-based systems.

An *interface automaton* is defined in the same way as I/O automaton with the only difference that interface automaton need not to be input enabled. The sets of input, output, and internal (called hidden in this case) actions again have to be pairwise disjoint.

**Definition:** Let $\mathcal{A}_i = (Q_i, \Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}, \delta_i, I_i)$, $i = 1, 2$, be interface automata. Then the set $\Sigma_1 \cap \Sigma_2$ is called $shared(\mathcal{A}_1, \mathcal{A}_2)$. Automata $\mathcal{A}_1, \mathcal{A}_2$ are *composable* iff

$$shared(\mathcal{A}_1, \mathcal{A}_2) = (\Sigma_{1,inp} \cap \Sigma_{2,out}) \cup (\Sigma_{2,inp} \cap \Sigma_{1,out}).$$

It means that, except of actions which are input of the first and output of the second automaton or vice versa, the sets of actions of two composable automata have to be disjoint.

**Definition:** Let $\mathcal{A}_i = (Q_i, \Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}, \delta_i, I_i)$, $i = 1, 2$, be composable interface automata. Then $(Q_1 \times Q_2, \Sigma_{inp}, \Sigma_{out}, \Sigma_{int}, \delta, I_1 \times I_2)$ is a *product of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ iff

- $\Sigma_{inp} = (\Sigma_{1,inp} \cup \Sigma_{2,inp}) \setminus shared(\mathcal{A}_1, \mathcal{A}_2)$,
- $\Sigma_{out} = (\Sigma_{1,out} \cup \Sigma_{2,out}) \setminus shared(\mathcal{A}_1, \mathcal{A}_2)$,
- $\Sigma_{int} = (\Sigma_{1,int} \cup \Sigma_{2,int}) \cup shared(\mathcal{A}_1, \mathcal{A}_2)$ and
- $((q_1, q_2), a, (q_1', q_2')) \in \delta$ iff
  $a \notin shared(\mathcal{A}_1, \mathcal{A}_2) \ \wedge \ (q_1, a, q_1') \in \delta_1 \ \wedge \ q_2 = q_2'$
  $a \notin shared(\mathcal{A}_1, \mathcal{A}_2) \ \wedge \ q_1 = q_1' \ \wedge \ (q_2, a, q_2') \in \delta_2$
  $a \in shared(\mathcal{A}_1, \mathcal{A}_2) \wedge (q_1, a, q_1') \in \delta_1 \wedge (q_2, a, q_2') \in \delta_2$.

The definition implies that the linking of input and output action, from the set of shared actions, is compulsory as in the I/O model. Additionally, the model does not permit multiple binding on the interfaces directly without renaming (e.g. two components using the same service provided by other component). Each input (output) action after linking to an appropriate output (input) action becomes internal action and therefore is not allowable for other linking.

The transition set of the product of two interface automata contains all syntactically correct transitions. *Composition* of two interface automata is a restriction of the product automaton. The restriction is defined with the help of *error* and *compatible states*, and *compatibility* of two automata (for formal definitions see [8]). The product state $(q_1, q_2)$ of two composable interface automata is an error state if it corresponds to a state where one of the automata is able to send an output action $a$ and the other one is not able to receive the action $a$ ($a$ is a shared action). A state $q$ is compatible if no error state is reachable from $q$ performing only output and internal actions. Two interface automata with initial states $q_1$, $q_2$ are compatible, if they are composable and the initial state $(q_1, q_2)$ of their product is a compatible state.

**Definition:** Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be compatible interface automata and $(Q, \Sigma_{inp}, \Sigma_{out}, \Sigma_{int}, \delta, I)$ be their product. Interface automaton $(Q, \Sigma_{inp}, \Sigma_{out}, \Sigma_{int}, \delta', I)$ is a *composition* of $\mathcal{A}_1$ and $\mathcal{A}_2$ iff $\delta' = \delta \backslash \{(q, a, q') \mid q$ is compatible, $a \in \Sigma_{inp}$, and $q'$ is not compatible$\}$.

The composition of interface automata is defined in two steps. In the first step the product automaton is built and the set of error and compatible states are formed. In the second step the transition function of the product automaton is restricted to disable transitions to incompatible states. It follows the *optimistic* assumption that two automata can be composed if there exists some environment that can make them work together properly. Then the composition of the automata consists of the transitions available in such environment.

The shortcoming of this approach is the explicit indication of erroneous behaviour (error states) that limits this approach to modelling solely the component-based systems with equivalent notion of what is and is not considered as an error (respecting one type of synchronization).

## 3.3 Team automata

The *Team automata* model [5] was first introduced in [9] by Clarence A. Ellis. This complex model is primary designed for modelling groupware systems with communicating teams but can be also used for modelling component-based systems. It is inspired by I/O automata. Team automata, as a main distinct from the previous models, allow freedom of choosing the transition set of the automaton obtained when composing a set of automata, and thus are not limited to one synchronization only.

A *team automaton* is defined in the same way as I/O automaton with the only difference that team automaton need not to be input enabled. A set of component automata is *composable* if the set of internal actions of every component automaton is disjoint with the action sets of all other component automata. The composition of team automata is defined over a *complete transition space.*

**Definition:** Let $\mathcal{S} = \{(Q_i, \Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}, \delta_i, I_i)\}_{i \in \mathcal{I}}$, where $\mathcal{I} \subseteq \mathbb{N}$ is finite, be a composable system of component automata and $a \in \bigcup_{i \in \mathcal{I}} \Sigma_i$. Then a *complete transition space* of $a$ in $\mathcal{S}$ is denoted $\Delta_a(\mathcal{S})$ and defined as

$\Delta_a(\mathcal{S}) = \{(q, a, q') \mid q, q' \in \Pi_{i \in \mathcal{I}} Q_i \wedge$
$\exists j \in \mathcal{I} : (proj_j(q), a, proj_j(q')) \in \delta_j \wedge$
$\forall i \in \mathcal{I} : ((proj_i(q), a, proj_i(q')) \in \delta_i \vee proj_i(q) = proj_i(q'))\}$.

$\mathcal{T} = (\Pi_{i \in \mathcal{I}} Q_i, \Sigma_{inp}, \Sigma_{out}, \Sigma_{int}, \delta, \Pi_{i \in \mathcal{I}} I_i)$ is a *team automaton over* $\mathcal{S}$ iff

- $\Sigma_{inp} = \left(\bigcup_{i \in \mathcal{I}} \Sigma_{i,inp}\right) \setminus \left(\bigcup_{i \in \mathcal{I}} \Sigma_{i,out}\right)$,
- $\Sigma_{out} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,out}$,
- $\Sigma_{int} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,int}$ and
- $\delta \subseteq \Pi_{i \in \mathcal{I}} Q_i \times \Sigma \times \Pi_{i \in \mathcal{I}} Q_i$, such that for all $a \in (\Sigma_{inp} \cup \Sigma_{out})$, $\delta$ restricted to $a$ is a subset of $\Delta_a(\mathcal{S})$, and for all $a \in \Sigma_{int}$, $\delta$ restricted to $a$ is equal to $\Delta_a(\mathcal{S})$.

The important fact to mention is that the composition hides every input action which is an output action of some other automaton in the composition. Therefore the input action cannot be used on a higher level of compositional hierarchy later on. Another important feature is that, when composing automata, we can loose some information about the behaviour of the system. For example, let us consider the component automata $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ from Figure 2 where $\mathcal{A}_1$ has one transition over *output* action $a$ and both $\mathcal{A}_2$ and $\mathcal{A}_3$ have one transition over *input* action $a$. After composing these three automata to the automaton $\mathcal{A}$ over $\{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3\}$ (with one transition over *output* action $a$), we cannot differentiate between synchronization of the input of $\mathcal{A}_2$ with the output of $\mathcal{A}_1$ and synchronization of $\mathcal{A}_3$ with $\mathcal{A}_1$. This can be quite restrictive for verification of properties capturing which components participated in the computation. Moreover, if we would need to express that an automaton $\mathcal{A}_1$ in a state $q_0$ can synchronize with $\mathcal{A}_2$ only, we cannot include this information in the composition withou renaming.

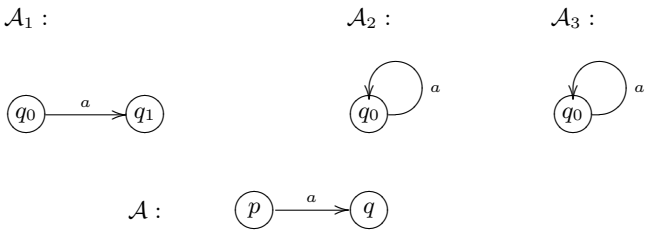



**Figure 2: Composition of automata ($p$ states for $(q_0, q_0, q_0)$, $q$ states for $(q_1, q_0, q_0)$)**

## 3.4 Summary

The characteristics of the current models described in this section make their applicability for the full description of interactions in component-based systems difficult. It is natural because studied models were often designed for a slightly different purpose (I/O automata, Team automata) and usually are limited to one strict type of synchronization (I/O automata, Interface automata) which we do not want to limit to. In some cases, relabelling and transformation of the component automata before each composition would be sufficient to express desired properties. But the price we would have to pay for it is in considerable state expanding, untransparency and uncomfortable use of the model. Moreover there are features (like strict synchronization at Interface automata or input enableness at I/O automata) which would be nontrivial to overcome.

# 4. COMPONENT-INTERACTION AUTOMATA

The issues mentioned in the previous section has motivated us to evolve a new verification-oriented automata-based formal model, *Component-Interaction automata*, designed for specification of interacting components in component-based systems with respect to several aspects of the systems (ADL interconnection structure, way of communication among components). The Component-Interaction automata make it possible to model all interesting aspects of component interaction in hierarchical component-based software systems without loosing any behaviours, and verify interaction behaviour of the systems as well. The model respects current ADLs to enable direct transformation of the ADL description to Component-Interaction automata, and current verification tools as Component-Interaction automata can be translated into their specification languages.

The Component-Interaction automata model is inspired by the Team automata model, mainly in freedom of choosing the transition set of the composed automaton what enables it to be architecture and synchronization configurable. However, Component-Interaction automata differ from Team automata in many aspects to be more comfortable in use for component-based systems, and to preserve important information about the interaction among synchronized components and hierarchical structure of the composed system.

Component-interaction automaton over a set of components $S$ is a nondeterministic automaton where every transition is labelled as an input, output, or internal. Sets of input, output and internal actions need not to be pairwise disjoint. Input (output) action is associated with the name of a component which receives (sends) the action. Internal action is associated with a tuple of components which synchronize on the action. In composition of component-interaction automata, only two components can synchronize and the information about their communication is preserved. If a component has an input (output) action $a$, the composition also can have $a$ as its input (output) action even after the linking of the action.

*Notation:* Let $\mathcal{I} \subseteq \mathbb{N}$ be a finite nonempty set with cardinality $k$, and let $\{S_i\}_{i\in\mathcal{I}}$ be a set. Then $(S_i)_{i\in\mathcal{I}}$ denotes the tuple $(S_{i_1}, S_{i_2}, \ldots, S_{i_k})$, where $\{i_1, i_2, \ldots, i_k\} = \mathcal{I}$ and for all $j_1, j_2 \in \{1, 2, \ldots, k\}$ if $j_1 < j_2$ then $i_{j_1} < i_{j_2}$.

## 4.1 Definition

**Definition:** A *component-interaction automaton* is a tuple $\mathcal{C} = (Q, Act, \delta, I, S)$ where

- $Q$ is a finite set of *states*,
- $Act$ is a finite set of *actions*, $\Sigma = ((X \cup \{-\}) \times Act \times (X \cup \{-\})) \setminus (\{-\} \times Act \times \{-\})$ where $X = \{n \mid n \in \mathbb{N},\ n \text{ occurs in } S\}$, is a set of *symbols* called an *alphabet*,
- $\delta \subseteq Q \times \Sigma \times Q$ is a finite set of *labelled transitions*,
- $I \subseteq Q$ is a nonempty set of *initial states* and
- $S$ is a tuple corresponding to a hierarchy of component names (from $\mathbb{N}$) whose composition $\mathcal{C}$ represents.

Symbols $(-, a, B)$, $(A, a, -)$, $(A, a, B) \in \Sigma$ are called *input*, *output* and *internal symbols* of the alphabet $\Sigma$, respectively. Accordingly, transitions are called input, output, and internal.

- The input symbol $(-, a, B)$ represents that the component $B$ receives an action $a$ as an input.
- The output symbol $(A, a, -)$ represents that the component $A$ sends an action $a$ as an output.
- The internal symbol $(A, a, B)$ represents that the component $A$ sends an action $a$ as an output, and synchronously the component $B$ receives the action $a$ as an input.

Remark, that component-interaction automaton need not have disjoint sets of input actions (those involved in input transitions), output actions (involved in output transitions), and internal actions (involved in internal transitions).

As it can be seen from the structure of symbols, only two components can synchronize on the same action. It is a natural way of component communication according to a client–server principle. If we would like to address multi-way synchronization, the model could be naturally extended to *Multi Component-Interaction automata*, where the symbols would be represented as tuples $(\mathsf{A}, a, \mathsf{B})$ where $\mathsf{A}$ stands for a set of sending components and $\mathsf{B}$ for a set of receiving components.

*Example 4.1.:* Let us consider the system from Figure 3 (modelled in UML 2.0). The component $C_1$ sends an action $a$ through the interface $I_2$ and sends an action $b$ through the interface $I_1$. The component $C_2$ receives an action $a$ through the interface $I_3$. $C_3$ sends $a$ through $I_6$, $C_4$ receives $a$ through $I_4$ and sends $b$ through $I_5$. Finally, $C_5$ sends $b$ through $I_7$.

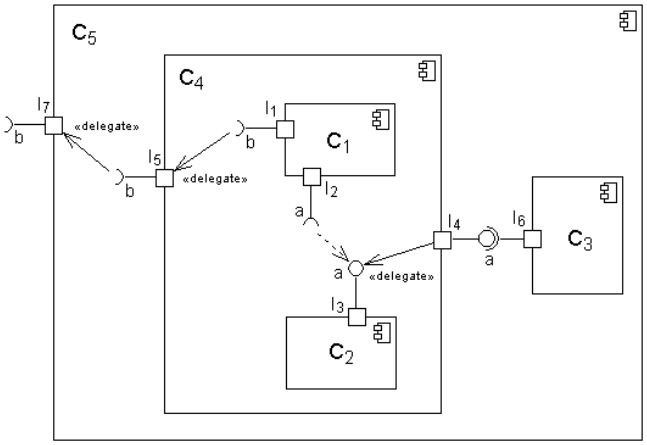



**Figure 3: Component model of a simple system**

Component-interaction automata $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$, modelling components $C_1$, $C_2$, and $C_3$ from Figure 3, respectively, follows (their graphical representation is in Figure 4).

$\mathcal{A}_1 = (\{q_0, q_1\}, \{a, b\}, \{(q_0, (1, a, -), q_1),$
$(q_1, (1, b, -), q_1)\}, \{q_0\}, (1))$

$\mathcal{A}_2 = (\{q_0\}, \{a\}, \{(q_0, (-, a, 2), q_0)\}, \{q_0\}, (2))$

$\mathcal{A}_3 = (\{q_0\}, \{a\}, \{(q_0, (3, a, -), q_0)\}, \{q_0\}, (3))$

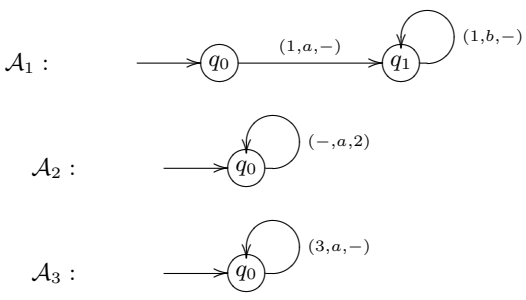



**Figure 4: Automata $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ modelling components $C_1$, $C_2$, and $C_3$.**

Component-interaction automata can be composed and form a hierarchical structure which is preserved as visible from the next definition.

**Definition:** Let $\mathcal{S} = \{(Q_i, Act_i, \delta_i, I_i, S_i)\}_{i\in\mathcal{I}}$, where $\mathcal{I} \subseteq \mathbb{N}$ is finite, be a system of component-interaction automata such that sets of components represented by the automata are pairwise disjoint. Then $\mathcal{C} = (\Pi_{i\in\mathcal{I}} Q_i, \cup_{i\in\mathcal{I}} Act_i, \delta, \Pi_{i\in\mathcal{I}} I_i, (S_i)_{i\in\mathcal{I}})$ is a component-interaction automaton *over* $\mathcal{S}$ iff
$\delta = \Delta_{OldInternal} \cup \delta_{NewInternal} \cup \delta_{Input} \cup \delta_{Output}$ where

$\Delta_{OldInternal} = \{(q, (A, a, B), q') \mid \exists i \in \mathcal{I} :$
$(proj_i(q), (A, a, B), proj_i(q')) \in \delta_i, \ \forall j \in \mathcal{I}, \ j \neq i :$
$(proj_j(q)) = proj_j(q')\}$

$\Delta_{NewInternal} = \{(q, (A, a, B), q') \mid \exists i_1, i_2 \in \mathcal{I}, i_1 \neq i_2 :$
$(proj_{i_1}(q), (A, a, -), proj_{i_1}(q')) \in \delta_{i_1} \ \wedge$
$\wedge \ (proj_{i_2}(q), (-, a, B), proj_{i_2}(q')) \in \delta_{i_2} \ \wedge$
$\wedge \ \forall j \in \mathcal{I} : i_1 \neq j \neq i_2 \ proj_j(q) = proj_j(q')\}$

$\delta_{NewInternal} \subseteq \Delta_{NewInternal}$

$\Delta_{Input} = \{(q, (-, a, B), q') \mid \exists i_1 \in \mathcal{I} :$
$(proj_{i_1}(q), (-, a, B), proj_{i_1}(q')) \in \delta_{i_1} \ \wedge \ \forall j \in \mathcal{I} : i_1 \neq j :$
$(proj_j(q) = proj_j(q'))\}$

$\delta_{Input} \subseteq \Delta_{Input}$

$\Delta_{Output} = \{(q, (A, a, -), q') \mid \exists i_2 \in \mathcal{I} :$
$(proj_{i_2}(q), (A, a, -), proj_{i_2}(q')) \in \delta_{i_2} \ \wedge \ \forall j \in \mathcal{I} : j \neq i_2 :$
$(proj_j(q) = proj_j(q'))\}$

$\delta_{Output} \subseteq \Delta_{Output}$

Transitions in $\Delta_{OldInternal}$ are internal transitions of the component automata. Transitions in $\delta_{NewInternal}$ arise from synchronization of two components. $\delta_{Input}$ and $\delta_{Output}$ are input and output transitions of the composed automaton provided by components of the composed automaton.

In the definition, we use auxiliary sets $\Delta_s$, $s \in \{NewInternal, Input, Output\}$. Each of these sets represents all possible transitions (complete transition space) over a specific set of symbols determined by the index $s$. The architecture of the modelled component-based system and other advanced characteristics determine which transitions from the complete transition space are included in the composed automaton. The idea is that the final transition set $\delta$ is formed automatically according to the rules specifying the complete transition space, interconnection rules generated from the ADL description and other specified characteristics.

*Example 4.2.:* Let us illustrate the composition on automata from Example 4.1. Automaton $\mathcal{A}_4$ (modelling the component $C_4$ from Figure 3) is a component-interaction automaton over $\mathcal{A}_1$ and $\mathcal{A}_2$. Automaton $\mathcal{A}_5$ (modelling the component $C_5$ from Figure 3) is a component-interaction automaton over $\mathcal{A}_3$ and $\mathcal{A}_4$. The architecture of the composition is determined by the UML 2.0 description of the system in Figure 3.

$\mathcal{A}_4 = (\{s_0, s_1\}, \{a, b\}, \{(s_0, (-, a, 2), s_0), (s_0, (1, a, 2)s_1),$
$(s_1, (1, b, -), s_1), (s_1, (-, a, 2), s_1)\}, \{s_0\}, ((1), (2)))$
$\Delta_{OldInternal} = \emptyset$
$\Delta_{NewInternal} = \{(s_0, (1, a, 2), s_1)\}$
$\delta_{NewInternal} = \{(s_0, (1, a, 2), s_1)\}$
$\Delta_{Input} = \{(s_0, (-, a, 2), s_0), (s_1, (-, a, 2), s_1)\}$
$\delta_{Input} = \{(s_0, (-, a, 2), s_0), (s_1, (-, a, 2), s_1)\}$
$\Delta_{Output} = \{(s_0, (1, a, -), s_1), (s_1, (1, b, -), s_1)\}$
$\delta_{Output} = \{(s_1, (1, b, -), s_1)\}$
Here $s_0$ and $s_1$ represent the states $(q_0, q_0)$ and $(q_1, q_0)$, respectively. For the graphical representation of $\mathcal{A}_4$ see Figure 5.

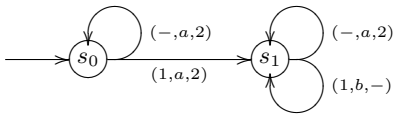



**Figure 5: Automaton $\mathcal{A}_4$ modelling component $C_4$**

$\mathcal{A}_5 = (\{p_0, p_1\}, \{a, b\}, \{(p_0, (3, a, 2), p_0), (p_0, (1, a, 2), p_1),$
$(p_1, (3, a, 2), p_1), (p_1, (1, b, -), p_1)\}, \{p_0\}, (((1), (2)), (3)))$
$\Delta_{OldInternal} = \{(p_0, (1, a, 2), p_1)\}$
$\Delta_{NewInternal} = \{(p_0, (3, a, 2), p_0), (p_1, (3, a, 2), p_1)\}$
$\delta_{NewInternal} = \{(p_0, (3, a, 2), p_0), (p_1, (3, a, 2), p_1)\}$
$\Delta_{Input} = \{(p_0, (-, a, 2), p_0), (p_1, (-, a, 2), p_1)\}$
$\delta_{Input} = \emptyset$
$\Delta_{Output} = \{(p_0, (3, a, -), p_0), (p_1, (1, b, -), p_1), (p_1, (3, a, -), p_1)\}$
$\delta_{Output} = \{(p_1, (1, b, -), p_1)\}$
Here $p_0$ and $p_1$ represent the states $((q_0, q_0), q_0)$ and $((q_1, q_0), q_0)$, respectively. For the graphical representation of $\mathcal{A}_5$ see Figure 6.

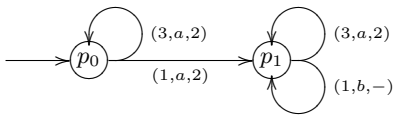



**Figure 6: Automaton $\mathcal{A}_5$ modelling component $C_5$**

The operation of composition let us model the hierarchical structure of component-based systems. The base of the composition are *primitive* component-interaction automata. A component-interaction automaton is primitive if it represents one individual component only. Automata $\mathcal{A}_1, \mathcal{A}_2$, and $\mathcal{A}_3$ from Example 4.1 are primitive. In the modelling and verification process we often need to consider only input and output transitions of a component-interaction automaton, which corresponds to the notion of primitiveness. Therefore we define a relation *primitive to* which enables us to transform any component-interaction automaton to a primitive one if we want to make the system less complex for further verification.

**Definition:** Let $\mathcal{C} = (Q, Act, \delta, I, S)$ be a component-interaction automaton. Then component-interaction automaton $\mathcal{C}' = (Q, Act, \delta', I, (n))$ is *primitive to* the component-interaction automaton $\mathcal{C}$ iff

- $n \in \mathbb{N}$ does not occur in $S$,
- $(q, (n, a, n), q') \in \delta'$ iff $\exists n_1, n_2 \in \mathbb{N} : (q, (n_1, a, n_2), q') \in \delta$,
- $(q, (-, a, n), q') \in \delta'$ iff $\exists n_2 \in \mathbb{N} : (q, (-, a, n_2), q') \in \delta$,
- $(q, (n, a, -), q') \in \delta'$ iff $\exists n_1 \in \mathbb{N} : (q, (n_1, a, -), q') \in \delta$.

*Example 4.3.:* Automaton $\mathcal{B}_4$ (see Figure 7) is primitive to the automaton $\mathcal{A}_4$ from Example 4.2. ($s_0$ and $s_1$ represent the states $(q_0, q_0)$ and $(q_1, q_0)$, respectively).

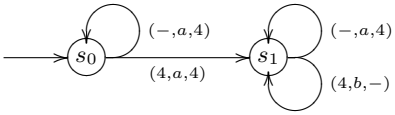



**Figure 7: Automaton $\mathcal{B}_4$**

## 4.2 Verification

For a given component-interaction automaton its behaviour can be defined through its *executions* and *traces*.

**Definition:** An *execution fragment* of a component-interaction automaton $\mathcal{C} = (Q, Act, \delta, I, S)$ is an infinite alternating sequence $q_0, x_0, q_1, x_1, \ldots$ of states and symbols of the alphabet $\Sigma$ such that $(q_i, x_i, q_{i+1}) \in \delta$ for all $0 \leq i$. An *execution* of $\mathcal{C}$ is an execution fragment $q_0, x_0, q_1, x_1, \ldots$ such that $q_0 \in I$. An execution fragment is *closed* if all its symbols are internal. A *trace* of $\mathcal{C}$ is a sequence $x_0, x_1, \ldots$ of symbols for which there is an execution $q_0, x_0, q_1, x_1, \ldots$.

In real component-based systems one needs to verify various properties of system behaviour. If the system is modelled as a component-interaction automaton the behaviour capturing the interaction among components and architectural levels are the traces. Both linear and branching time temporal logics have proved to be useful for specifying properties of traces. There are several formal methods for checking that a model of the design satisfies a given specification. Among them those based on automata [6] are especially convenient for our model of Component-Interaction automata.

We have experimentally verified several specifications of a component-based systems modelled as a component-interaction automata with the help of DiVinE [1, 4]. DiVinE (Distributed Verification Environment) is a model checking tool that supports distributed verification of systems. The DiVinE native input language si based on finite automata and the component-interaction automata can be translated into the DiVinE language. The tool supports verification of LTL properties.

# 5. CONCLUSIONS AND FUTURE WORK

The paper presents a new formal verification-oriented component-based specification language named Component-Interaction automata. This model is defined with the aim to support specification and verification of component interactions according to the interconnection architecture and other aspects of modelled system. On the one hand, Component-Interaction automata are close to architecture description languages which can be (semi)automatically transformed into Component-Interaction automata without loosing important behavioural characteristics. On the other hand, the proposed model is close to Büchi automata model and this admits automata-based verification of temporal properties of component interactions.

The Component-Interaction automata model aims to provide a direct and desirable way of modelling component-based systems which is meant to be more transparent and understandable thanks to the primary purpose oriented to component-based systems and their specifics. The model is inspired by some features of previously discussed models and differs in many others. It allows the freedom of choosing the transition set what allows its configurability according to the architecture description (inspired by Team automata) and is based on synchronization on one input and one output action with the same name which becomes internal later on (inspired by Interface automata). The model is designed to preserve all important interaction properties to provide a rich base for further verification. As a distinct from the models discussed in Section 3, it naturally preserves information about the components which participated in the synchronization and about the hierarchical structure, directly without renaming that would make the model less readable and understandable. Even if some component-based systems could be modeled by previously discussed models (I/O automata, Interface automata, Team automata) with appropriate relabelling, it would be for a price of considerable state expanding, untransparency and uncomfortable use of the model.

Nowadays we are developing an automatic transformation from SOFA ADL specification to Component-Interaction automata and from Component-Interaction automata to DiVinE model checking tool native input language. In the future, we intent to study Component-Interaction automata model in a more detailed way, considering mathematical (expressiveness), verification (properties and algorithms) and software engineering (reusability and compositionality) point of view.

# 6. REFERENCES

[1] Divine – Distributed Verification Environment. http://anna.fi.muni.cz/divine.

[2] J. Adamek and F. Plasil. Behavior protocols capturing errors and updates. In *Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE 2003), ETAPS*, pages 17–25, Warsaw, Poland, April 2003. University of Warsaw, Poland.

[3] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, School of Computer Science, May 1997.

[4] J. Barnat, L. Brim, I. Černá, and P. Šimeček. Divine – The Distributed Verification Environment. In *Proceedings of the Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05)*, July 2005.

[5] M. Beek, C. Ellis, J. Kleijn, and G. Rozenberg. Synchronizations in Team Automata for Groupware Systems. *Computer Supported Cooperative Work—The Journal of Collaborative Computing*, 12(1):21–69, 2003.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, January 2000.

[7] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[8] L. de Alfaro and T. A. Henzinger. Interface-based design. In *Proceedings of the 2004 Marktoberdorf Summer School*. Kluwer, 2004.

[9] C. Ellis. Team Automata for Groupware Systems. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge (GROUP'97)*, pages 415–424. ACM Press, New York, 1997.

[10] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, University of London, Imperial College of Science, Technology and Medicine, January 1999.

[11] D. C. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. In *Proceedings of DIMACS Partial Order Methods Workshop IV*, July 1996.

[12] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[13] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of PODC*, pages 137–151, April 1987.

[14] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[15] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of 5th European Software Engineering Conference (ESEC'95)*, September 1995.

[16] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA1)*, February 1999.

[17] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, November 2002.

[18] M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, April 1987.

# Performance Modeling and Prediction of Enterprise JavaBeans with Layered Queuing Network Templates

Jing Xu[1], Alexandre Oufimtsev[2], Murray Woodside[1], Liam Murphy[2]

[1] *Dept. of Systems and Computer Engineering, Carleton University, Ottawa K1S 5B6, Canada*

*Dept. of Computer Science, University College Dublin, Belfield, D4, Ireland*

xujing@sce.carleton.ca, alexu@ucd.ie, cmw@sce.carleton.ca, Liam.Murphy@ucd.ie

## Abstract

Component technologies, such as Enterprise Java Beans (EJB) and .NET, are used in enterprise servers with requirements for high performance and scalability. This work considers performance prediction from the design of an EJB system, based on the modular structure of an application server and the application components. It uses layered queueing models, which are naturally structured around the software components. This paper describes a framework for constructing such models, based on layered queue templates for EJBs, and for their inclusion in the server. The resulting model is calibrated and validated by comparison with an actual system.

## Keywords

Layered Queueing Network, template, Enterprise Java Bean, performance modeling, model calibration, software profiling.

## 1. Introduction and motivation

The approach to designing application servers based on component technologies such as Enterprise Java Beans and the J2EE standards [1] [5] [6] provides rapid development and the promise of scalability and good performance. J2EE and other approaches such as .NET do this by providing many services which applications require (such as support for concurrency, security, and transaction control) within the platform. As a result however the server platforms also have substantial overhead costs, and performance is a significant concern. Predictive models of a design can provide insight into potential problems and guidance for solutions. The use of predictive modeling to analyze software designs has been described extensively by Smith and Williams (e.g. [10]) and others (see for example [2][17]).

To build predictive models efficiently, the description of the platform should be separated from the components that implement the business logic of the application, the web interface, and the database. The infrastructure parts such as a J2EE platform can be modeled in advance and reused, with embedded parameters to describe possible deployments. When a specific application is designed, its elements are modeled and plugged into the platform sub-model. This provides a rapid model-building capability, in parallel with the rapid development process.

The process of defining component-based performance models, and of building models from components, was described in [4][16].

This work is based on a layered queueing network (LQN) formalism, which is defined in [9][13][14], and the introductory tutorial [15]. Layered queueing is a strategic choice. Compared to other formalisms surveyed in [2], it extends queueing networks to include software resources, and it avoids the state explosion of Markov models based on Petri Nets. Each software component is a distinct model entity, and contention for logical resources such as threads (which define the concurrency in the server platform) is captured.

The central contribution of this work is to demonstrate a sound procedure for modeling EJB applications using a template-based framework, calibrating the model using profiling data and validating it against measurement. Key issues include the relationship between the EJB application components and the platform (which are captured in submodel templates), the feasibility of measuring the model parameters from execution traces, and the accuracy with which the model predicts performance of an implementation under load. The paper [18] describes the design and use of the templates in greater detail.

## 2. LQN Evaluation

To demonstrate that the LQN model can be applied to this class of system with reasonable accuracy, a simulation model of a system with entity beans due to Llado and Harrison [7] was compared to its LQN equivalent. Figure 1 shows this LQN equivalent model.

The LQN notation is taken from [13][14][15]. Software entities are modeled as *tasks*, offering services called *entries*. In the Figure a task is a rectangle with a rectangular field at the right-hand end for the task name and parameters, and a field for each entry. The entries have parameters in the form "[demand]" for the host (CPU) demand in ms, and have blocking requests to other entries indicated by arrows with labels for the number of requests. The parameters for the tasks are multiplicities, which limit the number of concurrent instances. There are $N clients, $M bean threads, 1 thread for each of $I identical bean instances, and no limit (shown as *inf*) for the Container.

The part of the model within the large rectangle represents one bean Container with its services and beans, and forms a component that can be generated for each bean. The component has input and output interfaces consistent with a component-based model-building framework called CBML [16], which allows complex interacting systems to be built up from modules like this.

The solid arrowheads on the request arcs indicate that the requesting task is blocked (and its task-related resource is held) while the serving task executes. The model captures resource holding patterns, such as (1) the *Container* operation "invokeMethod" requires a bean instance, and (2) startup operations by the *prepareBean* entry of the critical section

psuedo-task *ConServ* (Container Services, with multiplicity 1), represent mutual exclusion.

The bean instances (both active and passive) are represented by a set of $I replicas of the task *Instance* (the shaded boxes in the Figure). Each replica is a single threaded task, showing that requests to the same instance must wait. They are shown as being requested with equal probability (1/$I for each instance).

The probability that a requested entity bean instance must be activated is represented by the probability (1-$p) on calls from the *prepareBean* entry to the call back functions (*activate, passivate, load* and *store*) on the active bean.

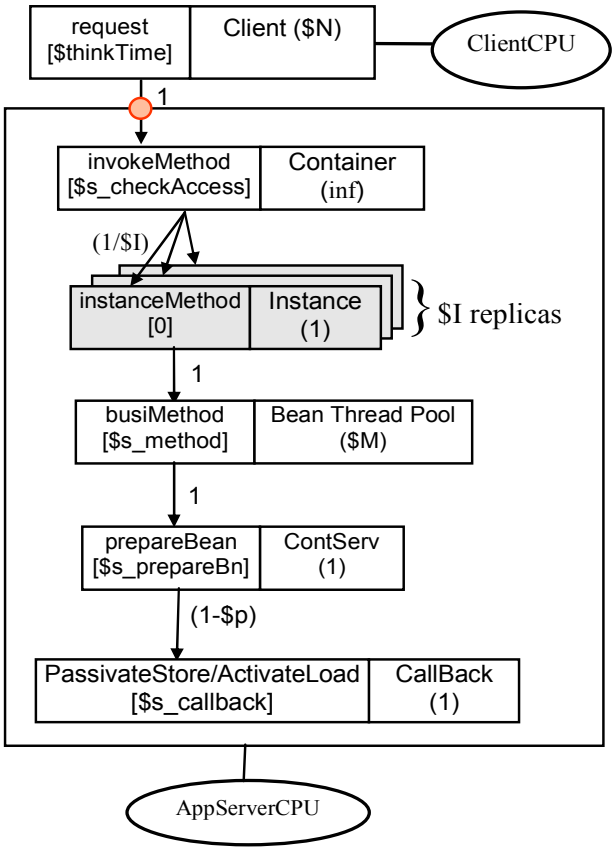



**Figure 1 LQN model for the system with Entity Beans in [8]**

With the same parameter values as were used in [8], the LQN model was solved with 20 instances ($I=20), pool size of 6 ($M=6), negligible execution demand for *invokeMethod*, and *prepareBean* ($s_checkAccess = 0.001, $s_prepareBn = 0.001) and business method (*busiMethod*) time of 4.1 ($s_method = 4.1). All the call back functions were aggregated to a single entry with a total demand of 0.4 (i.e. $s_callback = 0.4).

Figure 2 compares the simulation results from [8] with the LQN model. The results show approximately 6% differences between the two models with the LQN being a little pessimistic.

Llado and Harrison describe an extended queueing model for this system in [7], using decomposition and a custom-built solution strategy, which provides an even closer match to the simulation results. However the effort of creating such a model must be repeated for every configuration, and becomes more complex with multiple interacting beans. The current approach tries to overcome this by the use of a standardized model framework and a systematic model-building process based on LQN templates for the different kinds of beans.

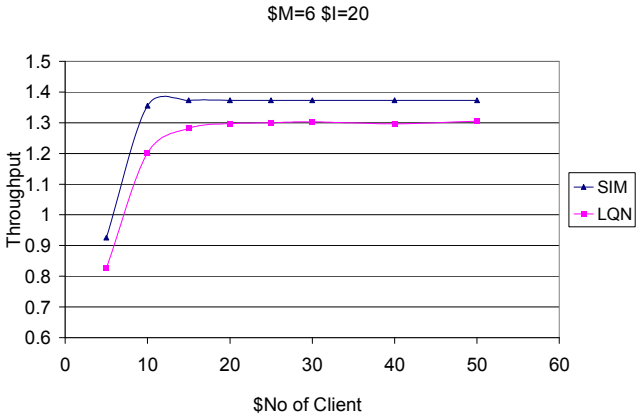



**Figure 2 LQN model predictions compared with Simulation Results [8]**

Figure 3 shows another set of results, which compares the throughput for different numbers of bean instances $I, with the same pool size $M = 6. It can be noted that the number of bean instances makes little difference since the system is saturated at the small sized thread pool. This behavior corresponds to the results obtained from [8].

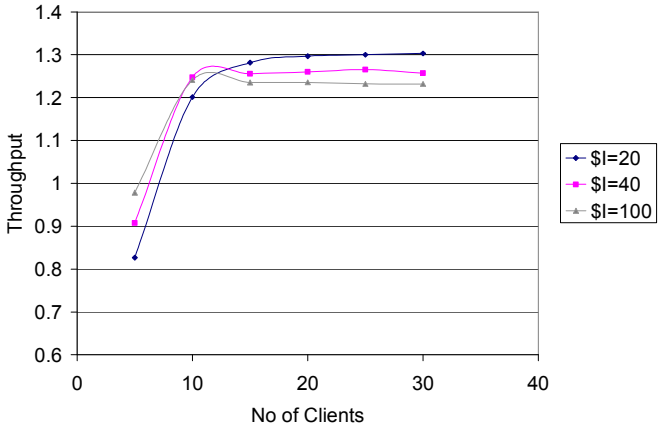



**Figure 3 Results for Different Numbers of Instances**

## 3. LQN sub-model templates

This section describes a LQN model template for each type of EJB. These templates follow the LQN component concept described in [16]. They can be instantiated according to specific function requirement in each scenario for system usage, and then be assembled into a complete LQN model for the whole scenario. An example on how to use these templates to build a complete LQN model for a business scenario will be shown in section 4.

### 3.1 LQN template for an Entity Bean

Figure 4 shows a LQN template for an Entity Bean. The entry *busiMethod* of the *activebean* is a placeholder for one or more methods of the Bean, and the entries *InvokeMethod*,

*InstanceMethod* and *getThread* are placeholders for resource requests on the calling path. All of these entries must be instantiated for each business method of the Bean, with calls between them as shown, connected to the interfaces.

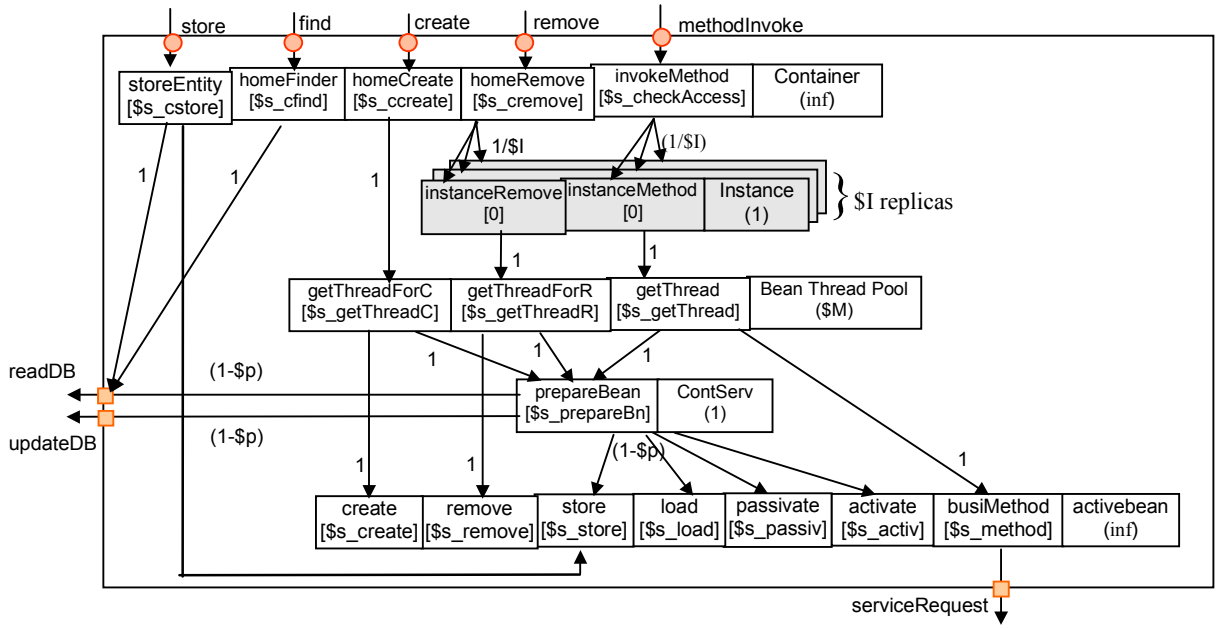



**Figure 4 Template for an Entity Bean**

The upper interfaces define provided services, with the placeholder *methodInvoke* for the Bean business methods. S*tore*, *find*, *create* and *remove* represent the Container Home Interfaces of the Bean. The *store* interface is used when a request to update the Entity state into the database is issued by another EJB component, for instance during a transaction-committing step of a Session Bean.

The lower interfaces define required services. S*erviceRequest* is a placeholder for function requests issued by the entity bean during its operation. The *readDB* and *updateDB* interfaces represent database operations during service and bean-instance context swapping.

When the template is instantiated, placeholders are instantiated as required for different services.

### 3.2 LQN template for a Stateless Session Bean

Figure 5 shows a LQN template for Stateless Session Bean. As it does not retain any state for a given client, each request can be directed to any available bean thread. After a service is finished, the bean thread is put back to the bean thread pool and ready for serving next request immediately. There can be outgoing requests.

For Stateless Session Beans the creation and removing of bean threads are controlled by the container. Clients do not create or remove bean threads.

### 3.3 LQN Template for a Stateful Session Bean

Figure 6 shows the LQN template for a Stateful Session Bean, which resembles that for an Entity Bean. However the passivation and activation operations use local storage rather than a database. The passivation and activation operations are aggregated and shown as callback functions from container to bean, represented by a pseudo-task *CallBack* whose workload is actually performed by the bean instance that is executing in the *ContServ* critical section. They inform the bean that the container is about to passivate or activate the bean instance, so that the bean instance can release or acquire corresponding resources such as sockets, database connection, etc.

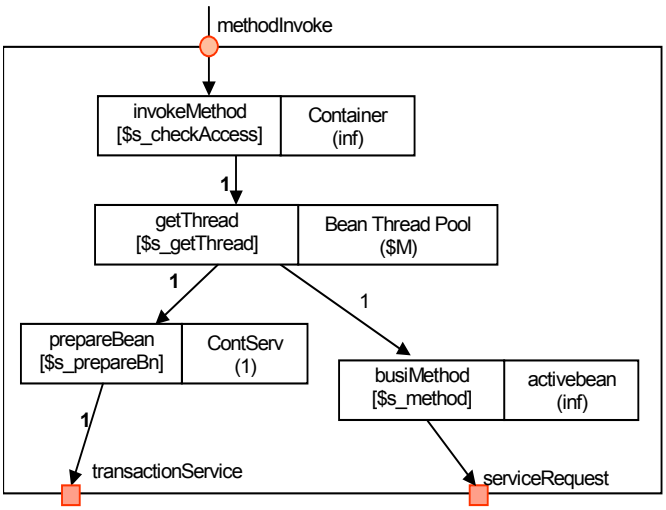



**Figure 5 Template for a Stateless Session Bean**

Assuming that each client has its own session bean there is no contention for a single bean instance, so the replica tasks for the instances are not modeled here. The thread pool is modeled as

executing the bean methods directly, with instances activated as necessary.

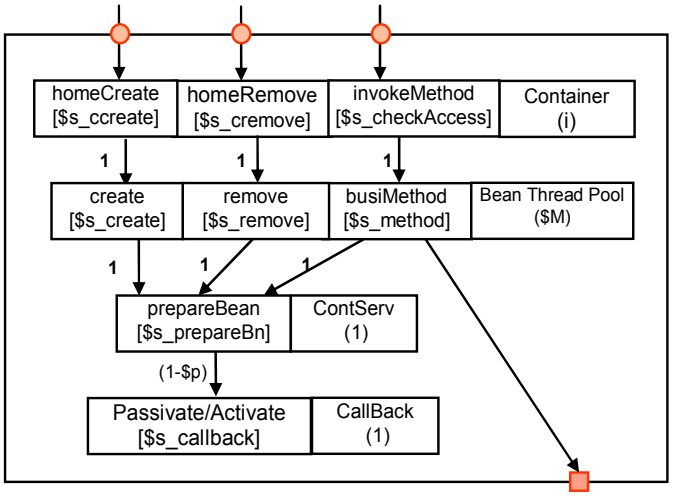



**Figure 6 Template for a Stateful Session Bean**

### 3.4 LQN Template for a Message Driven Bean

A message driven bean is similar to a stateless session bean, but its incoming calls are asynchronous messages (denoted by an arc with open arrowhead as shown in Figure 7).

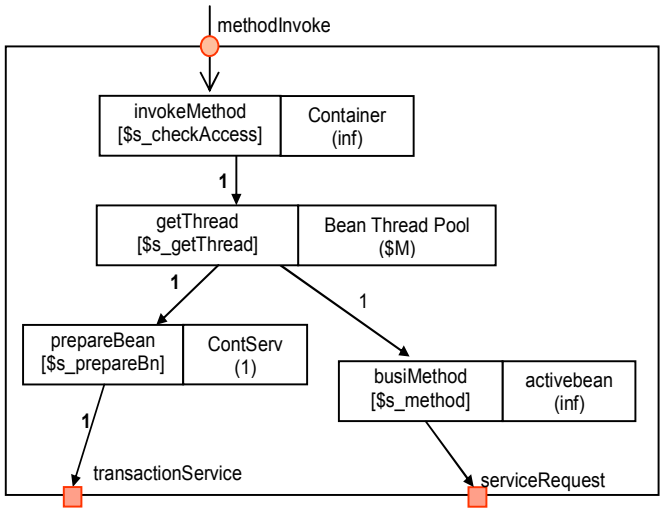



**Figure 7 Template for a Message Driven Bean**

## 4. Model Construction

A system is modeled by first modeling the beans as tasks with estimated parameters, then instantiating the template to wrap each class of beans in a container, and finally adding the execution environment including the database. Calls between beans, and calls to the database, are part of the final assembly. The model may be calibrated from running data, or by combining

- knowledge of the operations of each bean
- pre-calibrated workload parameters for container and database operations.

To illustrate this procedure, a simple system with a stateless session bean class and an entity bean class is used; this model was calibrated in the tests described in the next section.

The system chosen was based on the well-known Duke's Bank Application which is shipped with the J2EE documentation provided by Sun Microsystems [3]. The "Update Customer Information" Use Case was specialized to update e-mail information for customers. Figure 8 shows the scenario. It follows the EJB session façade pattern in which a Stateless Session Bean *CustomerControllerBean* delegates service requests from clients to an Entity Bean *CustomerBean*. The Session Bean first finds the required Entity Bean instance by Primary Key (PK) and then updates its email information.

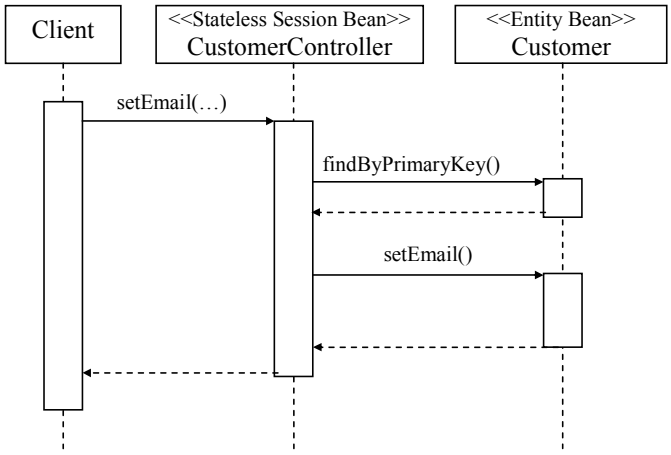



**Figure 8 Update Customer Information Scenario**

Figure 9 shows the completed LQN model for this scenario. The *CustomerControllerBean* sub-model instantiates the Stateless Session Bean template shown in Figure 5. The *CustomerBean* sub-model instantiates the Entity Bean template shown in Figure 4. The entries related to business methods are instantiated by the entries named *InvokeSetEmail*, *InstanceSetEmail* and *SetEmail*, with their parameters such as execution demands. The *CustomerControllerBean* requires two external services during the *SetEmail* operation, to find the customer by its primary key and update the email of the customer, giving two instances of the *serviceRequest* outgoing interface which are connected to the incoming interfaces of the *CustomerBean*.

This application uses Container Managed Persistence (CMP) strategy in which transactions are managed by containers. A transaction is started at the beginning of an invocation on the session bean *CustomerController* and is committed and ended right before the operation on session bean is completed. Any change on entity data is updated into database during the transaction committing stage. Therefore, the entity store operation is actually invoked by the session bean during its critical section for bean context swapping (represented by *prepareBean* in the model).

Underlying services from execution environment including *DataBase* and processing resource ApplicationCPU (*AppCPU*) are finally added to complete the model structure. Unused services such as bean creation/removal entries are omitted in the model.

The size for Bean Thread Pool $M can be obtained from container configuration during system deployment. The replica parameter $I represents the number of data records in the

database. Assuming that bean instances are accessed with equal probability, the hit rate $p = $M/$I. Alternatively, $p can be observed by measurement or benchmark. The experimental data shows that the hit rate quickly approaches its limit even at initial warmup phases when the number of replica tasks in the model is substantially greater than $M. Therefore, its value was limited to $I' = 3*$M for simplicity of the model.

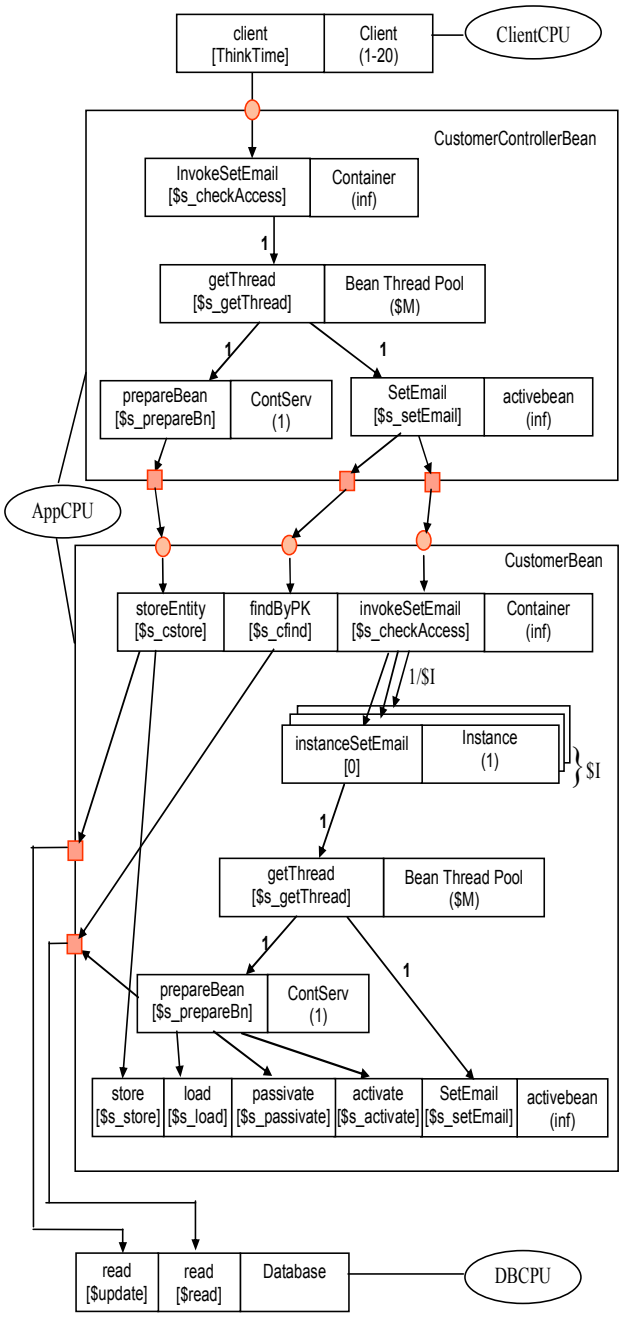



**Figure 9 Completed LQN model for Update Customer Information Scenario**

# 5. Application Profiling and Measurement

The Duke's Bank application [3] was modified in two ways: the Entity Beans were modified to use Container-Managed Persistence (CMP), and support for multiple concurrent clients was added.

## 5.1 Hardware platform

The testing environment includes three x86 machines described in Table 1: one for the EJB server, one for the database server, and one for client request generation. All the machines are connected to a dedicated switched 100 Mbps network. The client machine is more powerful than the servers to make sure that it does not become a bottleneck when generating the test load.

Due to a limited number of database entries in the application, the database server is lightly loaded.

**Table 1. Testing Environment Specification**

| Machine Type | Processor | Memory | HDD I/O System | OS |
|---|---|---|---|---|
| App Server | PIII-866 Mhz | 512 Mb | 20 Gb IDE | Debian Gnu/Linux 3.1 (Sarge) |
| Database Server | PIII-800 Mhz | 512 Mb | 20 Gb IDE | Debian Gnu/Linux 3.1 (Sarge) |
| Client | PIV-2.2 Ghz | 512 Mb | 80 Gb IDE | Debian Gnu/Linux 3.1 (Sarge) |

## 5.2 The software environment

The following software was used for testing purposes:

- operating system: Debian GNU/Linux 3.1 "sarge", kernel v 2.6.8-2
- database server: MySQL v. 4.0.23-7
- application server: JBoss v. 4.01sp1, connected to the database with Mysql Connector/J v 3.0.16.
- JVM: Java2SDK 1.4.2_03 for the application server, Java2SDK v5.0 for the client.
- client: a multithreaded testing suite developed in-house, that calls EJBs in the application server via RMI.

The newer version of SDK introduced some problems to the testbed, so SDK 5.0 was not used for the server setup.

Measurements on the container and program execution were obtained by running JProbe 5.2.1 Freeware profiler for Linux. Additional data, such as pool instance numbers and cache instance numbers were obtained using XMBean examples from the JBoss advanced tutorials. The UNIX sar utility was used to obtain various data available on the performance of the operating systems, including CPU, disk, and network usage.

The following options were used for JVM startup:

- the initial Java heap size was 384 MB to prevent performance loss in variations of the java heap size, with option -Xms384m,
- parameter -XX:+PrintCompilation was set to monitor the runtime behavior of the JVM. Generally, compilation messages suggest JVM is still adapting to certain type of workload.

- parameter -XX:CompileThreshhold was used to monitor the system's behavior with no JVM runtime optimizations

### 5.3 Testing Scenarios

The following changes were made to the application to emphasize contention for resources and to determine their overhead costs:

- container-managed persistence (CMP) entity beans are used instead of bean-managed persistence (BMP);
- multiple users are supported. The original Duke's bank only supports a single user;
- stateful session beans were converted to stateless session beans;
- the deployment descriptors were modified to limit the size of caches and pools for the beans to 10 to enforce activation/passivation, and artificial congestion at the pool/cache level.
- timeouts for bean passivation and removal was also significantly decreased to 5 seconds to allow shorter waiting time before next batch of client requests.

Two patterns of access to the data were followed by the simulated users. In **sequential access** all the beans are accessed in ascending order. In **random access**, the bean IDs are selected randomly. For profiler measurements a single request was entered and followed, once for each pattern.

For overall performance measurements the load was gradually increased from 1 to 20 users, with step size 1. 20 users for 10 entity beans were necessary to create an artificial situation of activation/passivation for a limited workload. To ensure that the bottleneck remains at the application server, the number of records in the database was kept to a low number of 300.

Every client performed the following loggable sequence in between warm-up and cool-down periods:

1. Update each customer record in ascending order (300 records in total);
2. Wait for other clients to finish.
3. Update a random customer record 300 times;
4. Wait for other clients to finish.

Average response time for each client thread was logged separately for random and sequential calls. Each set of measurements was carried out at least ten times after the warmup to check the results consistency. The thread pool and the cache of the entity beans were also monitored to ensure JBoss's compliance with imposed configuration restrictions. The database host has been 'warmed up' to achieve a constant response time to queries. The application server's operating system was warmed up to minimize its effect on the measurements. JBoss was restarted before the system, so JVM runtime adaptations could be observed.

## 6. Model Calibration and Comparison with Measurement Results

The measurement results are shown in Figures 10 and 11. Figure 10 shows an average throughput of the system. It can be seen that with 4 clients the system reaches the saturation point. The results produced are very consistent, with standard deviation (stdev) of 1.25 and 95% confidence intervals (CI95) of ±0.77. Average response time is shown in Figure 11. Both random and sequential access patterns are represented. For the random pattern, CI95 = ±1.94, and for the sequential pattern CI95 = ±2.01.

Clearly the test system is bottlenecked for as few as two customers, because the synthetic clients had zero "think time" between requests. Thus the throughput levels out and the response time becomes steadily greater as clients are added, due to waiting for the saturated processor.

There is a slight decrease in response time in the neighborhood of 4 users. This is due to the JVM adaptation (such as real-time compilation) that takes place during the initial part of the test. By the time the responses are recorded for 5 - 7 users the system has stabilized and thereafter the response time grows linearly with the number of users.

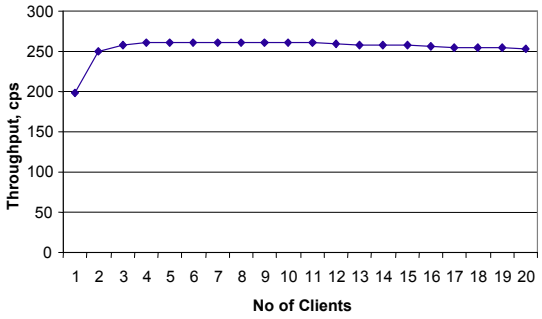



**Figure 10 Observed Throughput vs Number of Clients**

In Figure 11 the random access times are lower, probably because in sequential access every bean request requires an activation, while for random access there is a probability that the bean is already active.

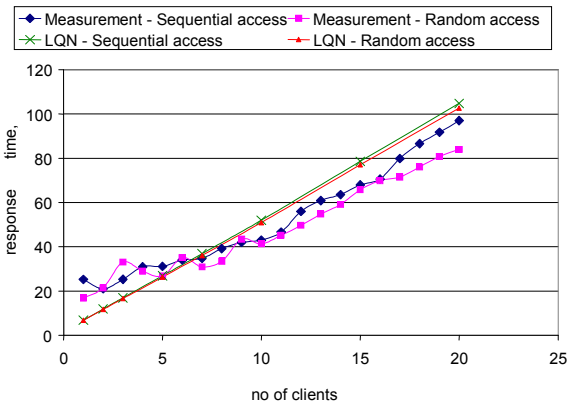



**Figure 11 Comparison of LQN results with measurement results**

### 6.1 Model Calibration

The model constructed in section 4 was calibrated from the profiling data under a single-client workload. Two factors are used to adjust the parameter values when doing the calibration.

***First,*** because the JProbe profiling tool itself introduced significant overhead on the execution, the execution demand values extracted from profiling data are adjusted to remove the contribution of overhead. This was done by using a *Profiling Ratio Factor* (PFC) based on the assumption that the profiling

overhead is proportionally distributed across the operations within some section of the scenario. The factor was obtained for each section by measuring the service time with and without profiling and taking the ratio. For Session Bean container services, PFC was 2.18. For Entity Bean container services, two PFC values were found. For finder operations, PFC was 3.78; for business method related operations, PFC was 7.81. The reason for the difference is that many more of the underlying methods are profiled than business methods. For general container services, such as security check operations, PFC was 2.98. For low-level operations, in which no underlying services are profiled, PFC=1.

***Second,*** during the measurements, the system warm up was observed when the same operations are repeated a number of times. JVM runtime optimizations are described in [11],[12]. For the server type JVM used here these include, but are not limited to, dead code elimination, loop invariant hoisting, common sub-expression elimination, constant propagation, null and range check eliminations, as well as full inlining and native code compilation.

The effect of JVM optimizations on the response time for the customer information update scenario used in this study is shown in Figure 12a. After initial volatility the response time stabilizes as shown in Figure 12b. The sporadic delays of about 150 ms are due to Garbage Collection. Setting the "CompileThreshold" parameter of JVM sufficiently high effectively turns the optimizations off.

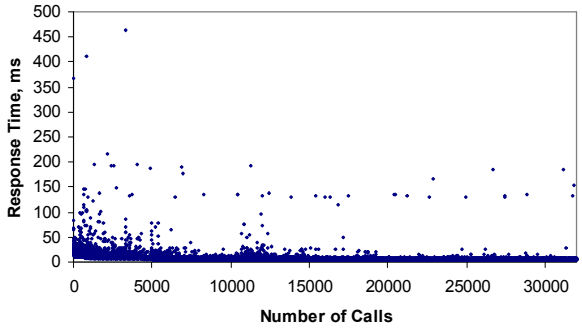



**Figure 12a Response time Variation**

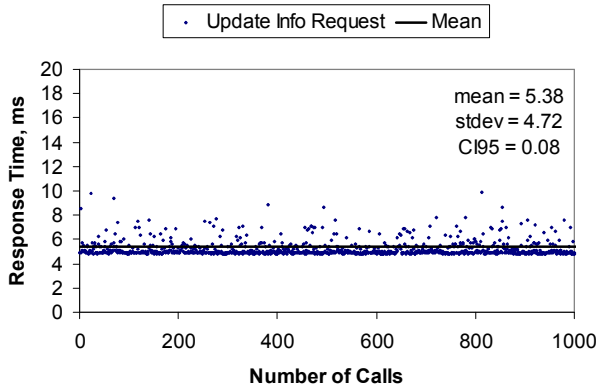



**Figure 12b Response time with an optimized JVM**

An unoptimized JVM becomes stable after just a couple dozen calls with a mean of 15 ms and sporadic Garbage Collection delays of approximately 220 ms. This initial delay is mainly caused by lazy loading of classes at JVM. Additional possible factors include OS processes, such as Virtual Memory Manager (VMM) moving swapped out pages back to RAM, the process and IO scheduler optimizing for the load, and connection establishment to the machine that runs the MySQL database.

Since the performance results were obtained by repeatedly invoking the same scenario, they are mostly warm system results. However, the profiling was done for a "cold" state with a single client in the system. A *Warm System Factor* (WSF) was introduced to adjust the service time values extracted from the profiling results, defined as the ratio of cold system state times over warm system state times. Based on the way performance measurements varied as the system warmed up, a factor of 3 is used for WSF.

After applying these factors, the following parameters were used in the model calibration:

- For CustomerControllerBean:
  $M = 10 $c_CheckAccess = 0.817ms
  $s_getThread = 0.002ms $s_prepareBn = 0.280ms
  $s_setEmail = 0.010ms

- For CustomerBean:
  $M = 10 $I' = 30 replicas
  $s_cstore = 0.303 $s_cfind = 1.740ms
  $s_checkAccess = 0.513ms $s_getThread = 0.003ms
  $s_prepareBn = 0.257ms $s_store = 0.003ms
  $s_load = 0.003ms $s_setEmail = 0.120ms
  $s_passivate = 0.001ms $s_activate = 0.001ms

- For Database
  $update = 2ms $read = 0.4ms

For sequential access, the hit rate $p should be 0 since the required bean instance is always in passive mode and needs context swapping every time. For random access, $p should 0.033, which is $M / $I = 10/300.

### 6.2 Model Predictions and Accuracy

Response time prediction results obtained from solving the model calibrated with above parameters are also shown in Figure 11. LQN predictions are low for small $N and high for large $N, which is due to progressive adaptation of the JVM during the experiment. For large $N the differences are between 6.2% to 23.9% for the sequential access case and 2.1% to 24.5% for the random access case.

The constant WSF for the impact of system warm-up is a compromise between the colder states at the left and the warmer states at the right. The greatest WSF that was observed in measurement is about 5, but it was not stable.

Software running in JVM continuously goes through optimizations by the VM. It is therefore proposed to use a range of values for CPU demand prediction, corresponding to warm system factor (WSF) values between 1 and 5. The value of 3 was used as a compromise in the middle of the range.

The LQN model also predicts resource utilizations which can be used to diagnose bottleneck. Model results show that with parameters taken in a cold state, the application processor is the bottleneck with 80%-95% utilization. With parameters that adjusted by WSF = 3, the bottleneck moves to the Bean Thread Pool of the Session Bean. Both these predictions are verified by measurement results. For a few clients the thread pool is not

saturated, but for many clients it is. The saturated thread pool does not slow down the system (or reduce the processor utilization) since the database was artificially configured to be very fast through a decrease of records in corresponding tables. This thread pool just limits the number of beans which are actively being processed at one time. However with a slower database, it could limit performance severely.

## 7. Conclusions

A template-based framework has been described for rapidly building predictive models of applications based on J2EE middleware. Most of the model, representing the J2EE platform, can be pre-calibrated, and the application description (in terms of its use of services) can be dropped in. The paper shows a complete procedure of constructing, calibrating, solving and analysis of the model for a real system.

The approach here has been customized to Enterprise Java Beans in a J2EE application server, but a similar approach can be applied to other technologies such as .NET. The framework uses Layered Queueing Network models, which can represent the various types of resources.

The LQN correctly predicts resource saturation of processor and thread resources. Predictions are affected by JVM adaptation, which must be taken into account when calibrating a model. CPU demand parameters measured on a cold system are up to 5 times of those on a warm system.

## Acknowledgement

The authors are grateful for the support of the Informatics Research Initiative of Enterprise Ireland (Mr. Oufimtsev and Dr. Murphy), and of Communications and Information Technology Ontario (Ms. Xu and Prof. Woodside).

## References

[1] E. Armstrong, J. Ball, S. Bodoff, D. Carson, I. Evans, D. Green, K. Haase, E. Jendrock, *The J2EE 1.4 Tutorial,* on-line document at java.sun.com/j2ee/1.4/docs/tutorial/doc, Sun MicroSystems, Dec. 16, 2004.

[2] S. Balsamo, A. DiMarco, P. Inverardi, and M. Simeoni, "Model-based Performance Prediction in Software Development," *IEEE Trans. on Software Eng*., vol. 30, no. 5 pp. 295-310, May 2004.

[3] S. Bodoff, D. Green, E. Jendrock, M. Pawlan, *The Dukes Bank Application*, on-line document at java.sun.com/j2ee/tutorial/1_3-fcs/doc/E-bank.html, Sun MicroSystems.

[4] V. Grassi, R. Mirandola, "Towards Automatic Compositional Analysis of Component Based Systems", *Proc Fourth Int.Workshop on Software and Performance*, Redwood Shores, CA, Jan. 2004, 00 59-63.

[5] Java Community Process, "*J2EE 1.4 Specification*", on-line document at http://java.sun.com/j2ee/1.4/download.html#platformspec, Nov. 24, 2003

[6] R. Johnson, *J2EE Design and Development*, Wiley Publishing Inc., Indianapolis.

[7] C.M. Llado, P.G. Harrison, "Performance Evaluation of an Enterprise Java Bean Server Implementation", *Proc second Int. Workshop on Software and Performance* (WOSP 2000), Ottawa, September 2000, pp 180-188.

[8] C.M. Llado, PhD thesis, Imperial College, London.

[9] J. A. Rolia and K. C. Sevcik, "The Method of Layers," *IEEE Trans. on Software Engineering*, vol. 21, no. 8 pp. 689-700, August 1995

[10] C. U. Smith and L. G. Williams, *Performance Solutions*. Addison-Wesley, 2002.

[11] Toshio Suganuma, Takeshi Ogasawara, Kiyokuni Kawachiya, Mikio Takeuchi, Kazuaki Ishizaki, Akira Koseki, Tatsushi Inagaki, Toshiaki Yasue, Motohiro Kawahito, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani, "Evolution of a Java just-in-time compiler for IA-32 platforms," IBM Journal of Research and Development, IBM Research in Asia Issue, Vol. 48, No. 5/6, pp. 767-795, 2004.

[12] Sun Microsystems, "*The Java Hotspot Virtual Machine, v.1.4.1, d2"*, online white paper at http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/JHS_141_WP_d2a.pdf, Sep 2002

[13] C.M. Woodside, E. Neron, E.D.S. Ho, and B. Mondoux, "An ``Active-Server" Model for the Performance of Parallel Programs Written Using Rendezvous," *J. Systems and Software,* pp. 125-131, 1986

[14] C.M. Woodside, J.E. Neilson, D.C. Petriu and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software", *IEEE Transactions on Computers*, Vol. 44, No. 1, January 1995, pp. 20-34

[15] M. Woodside, "*Tutorial Introduction to Layered Modeling of Software Performance*", Edition 3.0, May 2002 (Accessible from http://www.sce.carleton.ca/rads/ lqn/lqn-documentation/tutorialg.pdf)

[16] X.P. Wu and M. Woodside, "Performance Modeling from Software Components," in *Proc. 4th Int. Workshop on Software and Performance* (WOSP 04), Redwood Shores, Calif., Jan 2004, pp. 290-301.

[17] J. Xu, M. Woodside, and D.C. Petriu, "Performance Analysis of a Software Design using the UML Profile for Schedulability, Performance and Time," in *Proc. 13th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (TOOLS 03), Urbana, USA, Sept. 2003.

[18] J. Xu, M. Woodside, "Template-Driven Performance Modeling of Enterprise Java Beans", to appear in Proc. Workshop on Middleware for Web Services, Enschede, Netherlands, Sept. 2005.

# Classboxes – An Experiment in Modeling Compositional Abstractions using Explicit Contexts

Markus Lumpe
Iowa State University
Department of Computer Science
113 Atanasoff Hall
Ames, USA
lumpe@cs.iastate.edu

Jean-Guy Schneider
Faculty of Information & Communication Technologies
Swinburne University of Technology
P.O. Box 218
Hawthorn, VIC 3122, AUSTRALIA
jschneider@swin.edu.au

## ABSTRACT

The development of flexible and reusable abstractions for software composition has suffered from the inherent problem that reusability and extensibility are hampered by the dependence on position and arity of parameters. In order to address this issue, we have defined $\lambda\mathcal{F}$, a substitution-free variant of the $\lambda$-calculus where names are replaced with first-class namespaces and parameter passing is modeled using explicit contexts. We have used $\lambda\mathcal{F}$ to define a model for classboxes, a dynamically typed module system for object-oriented languages that provides support for controlling both the visibility and composition of class extensions. This model not only illustrates the expressive power and flexibility of $\lambda\mathcal{F}$ as a suitable formal foundation for compositional abstractions, but also assists us in validating and extending the concept of classboxes in a language-neutral way.

## 1. INTRODUCTION

In recent years, component-oriented software technology has become increasingly popular to develop modern, large-scale software systems [17]. The primary objective of component-based software is to take elements from a collection of reusable software components (i.e., *components-off-the-shelf*), apply some required domain-specific incremental modifications, and build applications by simply plugging them together. Moreover, with each reuse, it is expected that a component's quality improves, as potential defects are discovered and eliminated [15].

However, in order to be successful, the component-based software development approach needs to provide abstractions to represent different component models and composition techniques [1]. Furthermore, we need a canonical set of preferably language-neutral composition mechanisms that allows for building applications as compositions of reusable software components [14]. However, precise semantics is essential if we are to deal with multiple component models within such a common, unifying framework. As a consequence, we argue, any abstractions suitable for component-based software development need to be based on an appropriate formal foundation [8].

We have previously been studying a substitution-free variant of the $\lambda$-calculus, called $\lambda\mathcal{F}$, where names are replaced by *forms* and parameter passing is modeled using *explicit contexts* [8]. Forms are first-class namespaces that, in combination with a small set of purely asymmetric operators, provide a core language to define extensible, flexible, and robust software abstractions [9]. Explicit contexts, on the other hand, mimic $\lambda$-calculus substitutions, which are used to record named parameter bindings. For example, the $\lambda\mathcal{F}$-term `a[b]` denotes an expression `a`, the meaning of which is refined by the context `[b]`. That is, all occurrences of free variables in `a` are resolved using form `b`. Thus, the context `[b]` expresses the requirements posed by the free variables of `a` on its environment [13].

But is the $\lambda\mathcal{F}$-calculus a suitable formal foundation for defining compositional abstractions? As a proof of concept, we have used the $\lambda\mathcal{F}$-calculus to define a model for class extensions based on *classboxes* [2, 3]. Classboxes constitute a kind of *module system* for object-oriented languages that defines a packaging and scoping mechanism for controlling the visibility of isolated extensions to portions of complex, class-based systems. More precisely, classboxes define *explicitly named scopes* within which classes, methods, and variables are defined. In addition, besides the "traditional" operation of *subclassing*, classboxes support also the *local refinement* of imported classes by adding or modifying their features without affecting the originating classbox. Thus, classboxes allow one to import a class and apply some extensions to it without breaking the protocol defined between clients of that class in other classboxes. Consequently, classboxes and their associated operations provide a better control over changes, as they strictly limit the impact of changes to clients of the extending classbox.

The rest of this paper is organized as follows: in Section 2, we give a brief introduction into the $\lambda\mathcal{F}$-calculus, followed by a summary of the main characteristics of classboxes in

Section 3. In Section 4, we use $\lambda\mathcal{F}$ to define a general model of classes and classboxes. In Section 5, we present our $\lambda\mathcal{F}$ encodings of classbox operations. We conclude this paper in Section 6 with a summary of the main observations from our classbox modelings and outline future work in this area.

## 2. THE $\lambda\mathcal{F}$ CALCULUS

The design of the $\lambda\mathcal{F}$-calculus is motivated by our previous observations that the definition of general purpose compositional abstractions is hampered by the dependence on position and arity of parameters [7, 16]. Requiring parameters to occur in a specific order, to have a specific arity, or both, imposes a specification format in which programming abstractions are characterized not by the parameters they effectively use, but by the parameters they *declare*. This, however, limits our ability to combine them seamlessly with other, possibly unknown or weakly specified programming abstractions, because any form a parameter mismatch has to be resolved explicitly and, in general, manually.

To address this issue, we champion the concept of *dynamic binding* that allows for a software development approach in which new functionality can be added to an existing piece of code without affecting its previous behaviour [5]. Consequently, the $\lambda\mathcal{F}$-calculus is an attempt to combine the expressive power of the $\lambda$-calculus with the flexibility of position-independent data communication as well as late binding in expressions. In the following, we will briefly illustrate the main abstractions of $\lambda\mathcal{F}$; the interested reader is referred to [8] for further details.

The syntax of the $\lambda\mathcal{F}$-calculus is given in Figure 1. We presuppose a countably infinite set, $\mathcal{L}$, of *labels*, and let $l, m, n$ range over labels. We also presuppose a countably infinite set, $\mathcal{V}$, of *abstract values*, and let $a, b, c$ range over abstract values. We think of an abstract value as a representation of any programming value like integers, objects, types, and even forms themselves. However, we do not require any particular property except that equality and inequality be defined for elements of $\mathcal{V}$. We use $F, G, H$ to range over the set of forms and $M, N$ to range over the set of $\lambda\mathcal{F}$-terms.

Every form is derived from the empty form $\langle\rangle$, a form that does not define any bindings. A form $F$ can be extended by adding a binding for a label $l$ with a value $V$, written $F\langle l = V\rangle$. With projections we recover variable references of the $\lambda$-calculus. We require, however, that the subject of a projection denote a form. For example, the meaning of `F.l` is the value bound by label `l` in form `F`. A projection `a.l`, where `a` is not a form yields $\mathcal{E}$, which means "no value."

The expressive power of forms is achieved by the two asymmetric operators *form extension* and *form restriction*, written `F` $\oplus$ `G` and `F\G`, respectively. Form extension allows one to add or redefine a set of bindings simultaneously, whereas form restriction can be seen as a dual operation that denotes a form, which is restricted to all bindings of `F` that do not occur in `G`. In combination, these operators provide the main building block in a fundamental concept for defining adaptable, extensible, and more robust software abstractions [10].

Forms can also occur as values in binding extensions, denoted as *nested forms*. As in the case of binding extensions,

| | | | |
|---|---|---|---|
| $F, G, H$ | ::= | $\langle\rangle$ | *empty form* |
| | \| | $X$ | *form variable* |
| | \| | $F\langle l = V\rangle$ | *binding extension* |
| | \| | $F \oplus G$ | *form extension* |
| | \| | $F\backslash G$ | *form restriction* |
| | \| | $F \rightarrow l$ | *form dereference* |
| | \| | $F[G]$ | *form context* |
| $V$ | ::= | $\mathcal{E}$ | *empty value* |
| | \| | $a$ | *abstract value* |
| | \| | $M$ | $\lambda\mathcal{F}-$ *value* |
| $M, N$ | ::= | $F$ | *form* |
| | \| | $M.l$ | *projection* |
| | \| | $\lambda(X)\ M$ | *abstraction* |
| | \| | $M\ N$ | *application* |
| | \| | $M[F]$ | $\lambda\mathcal{F}-$ *context* |

**Figure 1: Syntax of the $\lambda\mathcal{F}$-Calculus.**

nested forms are bound by labels. However, rather than using a projection `F.l` to extract the nested form bound by label `l`, we use `F` $\rightarrow$ `l`, called *form dereference*. The reason for this is that we want to explicitly distinguish between components, which are encoded as forms, and plain component services, which are denoted by some values other than forms. If the binding involving label `l` does not actually map a nested form, then the result of `F` $\rightarrow$ `l` is $\langle\rangle$.

A *form context* `F[G]` denotes a closed form expression that is derived from `F` by using `G` as an environment to look up what would otherwise be free variables in `F`. We use form dereference to perform the lookup operation and a free variable is reinterpreted as a label. For example, if `X` is a free variable in `F` and `[G]` is a context, then the meaning of `X` in `F` is determined by the result of evaluating `G` $\rightarrow$ `X`. In the case that `G` does not define a binding for `X`, the result is $\langle\rangle$, which effectively removes the set of bindings associated with `X` from `F`. This allows for an approach in which a sender and a receiver can communicate open form expressions. The receiver of an open form expression can use its local context to close (or *configure*) the received form expression according to a site-specific protocol.

*Forms* and *projections* replace variables in $\lambda\mathcal{F}$. A form can be viewed as an *explicit namespace*, which can comprise an arbitrary number of bindings. The form itself can contain free variables, which will be resolved in the deployment environment or evaluation context, allowing for a computational model with *late binding*.

Both *abstraction* and *application* correspond to the notions used in the $\lambda$-calculus, that is, `X` in $\lambda$(`X`) `a` stands for the parameter in an abstraction. But unlike the $\lambda$-calculus, we do not use substitution to replace free occurrences of this name in the body of an abstraction – parameter passing is modeled by *explicit contexts*.

A $\lambda\mathcal{F}$-context is the counterpart of a form-context. A $\lambda\mathcal{F}$-*context* denotes a lookup environment for free variables in a $\lambda\mathcal{F}$-term. Moreover, $\lambda\mathcal{F}$-contexts provide a convenient mechanism to retain the bindings of free variables in the body of a function. For example, let $\lambda$(`X`) `a` be a function

and [F] be a creation context for it. Then we can use [F] to build a *closure* of λ(X) a. A closure is a package mechanism to record the bindings of free variables of a function at the time it was created. That is, the closure of λ(X) a is λ(X) (a[F]).

Denotational semantics is used to formalize the interpretation of forms and λ$\mathcal{F}$-terms. The underlying semantic model of forms is that of *interacting systems* [11]. Informally, the interpretation of forms (i.e., their observable behavior) is defined by an evaluation function $[\![]\!]^F$, which guarantees that feature access is performed from right-to-left [8]. In contrast to standard records, however, a given binding may not be observable in a form and, therefore, may not be used to redefine or hide an existing one. A binding is not observable if it cannot be distinguished from $\mathcal{E}$ or ⟨⟩. For example, the forms ⟨⟩⟨m = $\mathcal{E}$⟩, ⟨⟩⟨m = ⟨⟩⟩, and ⟨⟩ are all considered to be equivalent. Furthermore, the meaning of a λ$\mathcal{F}$-term depends on its *deployment context*. We write $[\![a]\!]^{LF}[H]$ to evaluate the λ$\mathcal{F}$-expression a in a deployment context $H$. Consider, for example, the following deployment context B that provides a Church encoding of Booleans. This context defines three bindings: True, False, and Not:

```
B = ⟨⟩⟨True = λ(X) X.true⟩
      ⟨False = λ(X) X.false⟩
      ⟨Not = λ(B) λ(V) B V⟨true = V.false⟩⟨false = V.true⟩⟩
```

Now, assume we want to determine the value denoted by the λ$\mathcal{F}$-expression (Not True). We can use B as a lookup environment for the free occurrences of the names Not and True, respectively. Thus, we have to evaluate

```
  [[(Not True)]]^LF[B]
= (λ(B) λ(V) B V⟨true = V.false⟩⟨false = V.true⟩) λ(X) X.true
= λ(V) B V⟨true = V.false⟩⟨false = V.true⟩ [⟨⟩⟨B = λ(X) X.true⟩]
= λ(V) (λ(X) X.true) V⟨true = V.false⟩⟨false = V.true⟩
```

which is a function that is equivalent to False. Due to lack of space, we omit the details of the definition of $[\![]\!]^F$; the interested reader is referred to [8].

## 3. CLASSBOXES IN A NUTSHELL

In order to address some of the problems of object-oriented programming languages with regard to incrementally changing the behaviour of existing classes, Bergel et al. have proposed the concept of *classboxes* [2, 3]. In their approach, a classbox can be considered as a kind of *module* that defines a controllable context in which incremental changes are visible. Besides the "traditional" operation of *subclassing*, classboxes support the operations of class *import* and class *extension*, respectively. In essence, a classbox exhibits the following main characteristics [3]:

1. It is a unit of scoping where classes (and their associated methods) are defined. A class belongs to the classbox it is first *defined*, but it can be made visible to other classboxes by either *importing* or *extending* it. When a classbox imports a class from another classbox (i.e., the *originating* classbox), the class behaves as if it was directly defined in this classbox. In order to resolve any dependencies, all ancestors of this class are *implicitly* imported also. A class extension can be viewed as an encapsulated import (i.e., self calls are bound early), combined with adding and/or overriding any of the original methods or instance variables.

2. Any extensions to a class are only visible to the classbox in which they are defined and any classboxes that either explicitly or implicitly import the extended class. Hence, overriding a particular method of a class in a given classbox will have no effect in the originating classbox.

3. Although class extensions are only locally visible, their effect extends to all collaborating classes within a given classbox, in particular to any subclasses that are either explicitly imported, extended, or implicitly imported.

In order to illustrate the concept of classboxes, consider the three classboxes *OriginalCB*, *LinearCB*, and *ColorCB*, respectively, given in Figure 2. The class Point defined in *OriginalCB* contains two protected instance variables x and y, a method move which moves a point by a given offset (dx,dy), and a method double that doubles the values of the x and y coordinates. The reader should note that the method move is invoked by double (using a self call). The class BoundedPoint is a direct specialization of Point. It ensures that the y coordinate of an instance never exceeds a given upper bound by. This bound is a constant in BoundedPoint, although this behaviour can be altered (as shown below).

The classbox *LinearCB imports* the class BoundedPoint from *OriginalCB*. As a consequence, Point is also *implicitly* imported, making it visible as the direct superclass of BoundedPoint. In order to define a non-constant bound, the class LinearBoundedPoint specializes BoundedPoint in *LinearCB* by overriding the method bound in an appropriate way (i.e., move checks if the y coordinate is smaller than the x coordinate).

The classbox *ColorCB extends* the class Point from *OriginalCB* by adding a protected instance variable c and a corresponding accessor method getColor. As a consequence, all instances of Point in *ColorCB* as well as the instances of any of its subclasses posses this additional behaviour. Therefore, the class LinearBoundedPoint *imported* from *LinearCB* also possesses the color property.

*ColorCB* also contains a *new* class BoundedPoint as a specialization of the extended class *Point* (restricting the x coordinate of its instances). Although it has the same name as BoundedPoint defined in *OriginalCB*, the two classes are *not* related. Hence, BoundedPoint defined in *ColorCB* is not the direct superclass of LinearBoundedPoint – the direct superclass of LinearBoundedPoint in *ColorCB* remains BoundedPoint defined in *OriginalCB*. This class is *implicitly* imported by LinearBoundedPoint and *co-exists* with BoundedPoint defined in *ColorCB*.

The semantics of extension operator deserves some further analysis. In [3, p. 118], it is stated that importing a class into a classbox is the same as extending this class with an empty set of methods. Furthermore, to guarantee the *locality of changes*, class extensions are purely local to the classbox within which they occur. Hence, it should be possible to use the operator extend to add a tracing mechanism

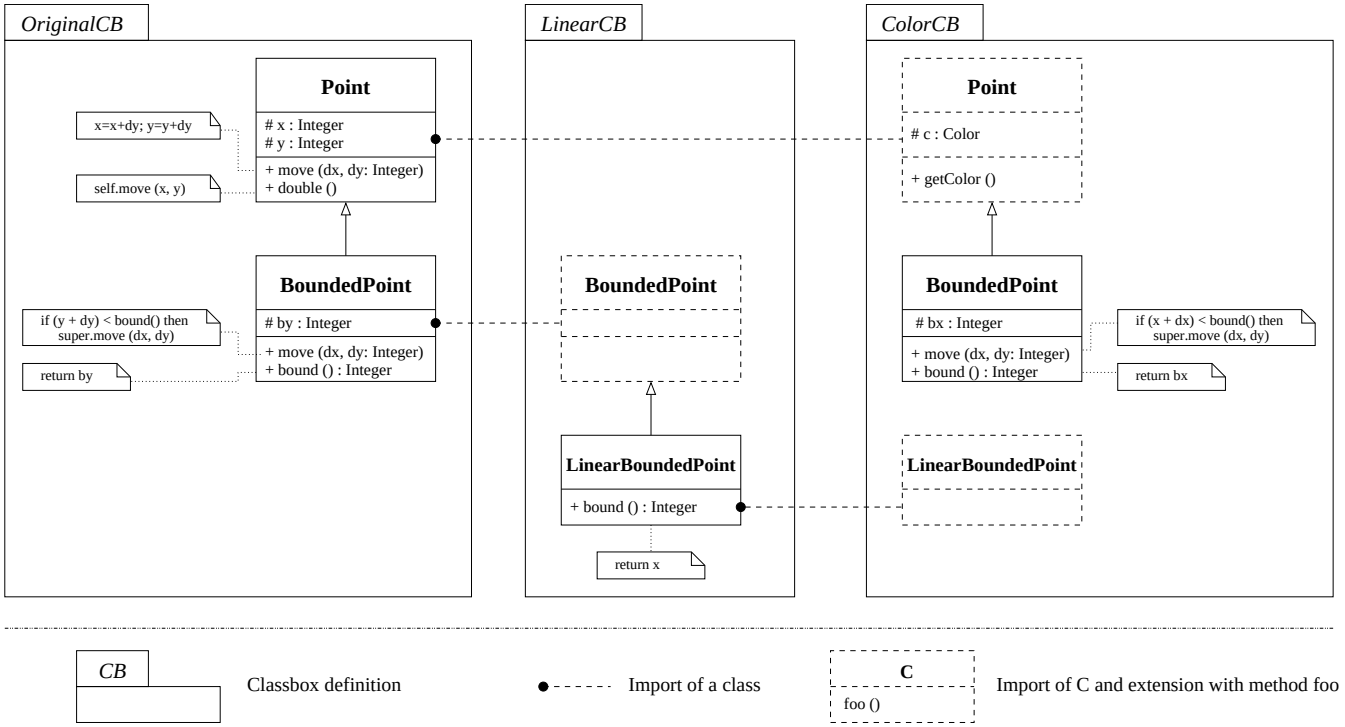



**Figure 2: Sample classboxes.**

to the class Point that logs all invocations of move in a new classbox, say *TraceCB*.

However, considering the way extend is formally defined in [3], we come to the conclusion that the semantics is slightly different and that extend should behave like C#'s new specifier [12], which can be used to hide any superclass method by declaring a new method with the same signature in a subclass. The effect is similar to binding self calls in superclass methods *early* (i.e., confine self calls occurring within superclass methods to the superclass). As a consequence, any instance derived from *TraceCB*'s Point class will not invoke the most recent definition of method move when required, but rather the original move defined in *OriginalCB*.

Therefore, to clarify the semantics of the classbox operators, we propose a revised notion of extending classes, which incorporates two separate extension operations: (i) *extension* of classes in which self calls are encapsulated to the context in which they occur, and (ii) *inclusion* of new behaviour by means of late binding of self calls. Such a separation will also allow us to seamlessly integrate the concept of *accessing the original method* (i.e., accessing the original implementation of method being redefined) presented in [2].

## 4. THE MODEL

As we have shown in earlier work [14], object- and component-oriented abstractions can most easily be modeled if classes are represented as first-class entities. This approach can be further generalized using a form-based framework (see Figure 3), which defines a hierarchy of meta-level abstractions to model *meta-classes*, *classes*, and *objects* [10]. The core of this meta-level framework is MetaModel, an abstraction that provides support for the instantiation of an object-oriented programming infrastructure. The underlying semantics of a specific programming infrastructure is captured by so-called *model generators*, *model wrappers*, and *model composers*, denoted by $G_m$, $W_m$, and $C_m$, respectively. The model abstractions $G_m$, $W_m$, and $C_m$ define the rules by which a concrete object-oriented programming system (e.g., the Java programming model) is governed. For example, to construct a Java-like programming infrastructure, we need to specify a generator $G_m^{Java}$, which defines the mechanism required for dynamic method lookup, and the single inheritance abstractions $W_m^{Class}$ and $C_m^{Class}$. To instantiate the Java-like infrastructure, we apply these three abstractions to MetaModel. The result is an infrastructure meta-object that can be used to create classes that adhere to Java semantics.

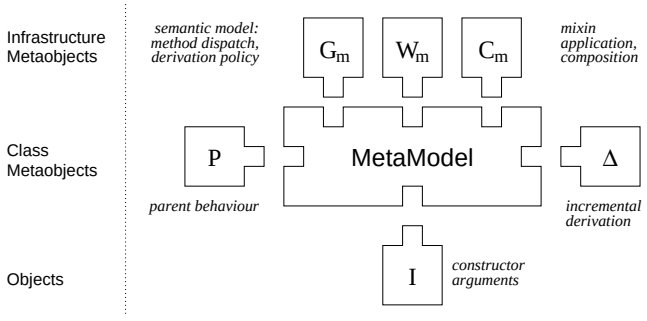



**Figure 3: A form-based meta-level framework.**

The behaviour of a class, on the other hand, is captured by an *incremental modification*, denoted by $\Delta$, and by a (possibly empty) *parent behaviour*, denoted by $P$, that cap-

tures the behaviour of its superclass(es). For example, to create a new class $C$, one has to define a new class generator $G_C$[1] that combines $C$'s incremental modification $\Delta_C$ with $C$'s parent behaviour $P$. To instantiate objects of the class $C$, one has to apply $G_C$ to some suitable *constructor arguments*, denoted by $I$. The result is a prototype instance that has to be passed to a model-specific wrapper, in order to establish the desired binding of self references within the newly created object.

In order to define classboxes in $\lambda\mathcal{F}$, we shall adapt an approach that is as close as possible to the original definition of classesboxes defined by Bergel et al. [3] with the exception that we shall define two separate extension operators. Furthermore, the reader should note that our classbox model does not require the composer abstraction which is mainly used for mixin application and composition [4, 19]. Hence, we shall only use incremental modifications, generators, and wrappers, respectively, to define a $\lambda\mathcal{F}$ representation of classes and classboxes.

We use the Greek letters $\alpha$, $\beta$, and $\gamma$ to denote classboxes, and $A, B, C$ to range over class names. A class $C$ in classbox $\alpha$ is represented by a named form $C_\alpha = \langle G, W \rangle$, where

- $C_\alpha$ is a so-called *decorated class name* for class $C$ [3] in which $\alpha$ identifies the originating classbox;
- $G$ is the generator for class $C$ combining $C$'s incremental modification $\Delta_C$ with $C$'s parent behaviour $P$; and
- $W$ is a wrapper yielding instances of class $C$ when applied to suitable constructor arguments.

Classboxes are actually *open class namespaces*. As a consequence, $G$ and $W$ are also open with respect to the environment being used to invoke them. Therefore, both $G$ and $W$ are parameterized over an *activation classbox*. An activation classbox is the fixed-point of classbox in which the corresponding class $C_\alpha = \langle G, W \rangle$ occurs explicitly by means of either import, subclassing, or extension. Thus, passing the activation classbox to $G$ and $W$, respectively, closes both abstractions and provides them with an appropriate lookup environment. This technique enables a late binding of $G$, which is the key mechanism for extending classes. The general structure of a class definition follows a format as shown below:

$$
\begin{array}{l}
C_\alpha = \\
\quad \textbf{let} \\
\quad\quad \Delta_C \;=\; \lambda(\mathsf{State})\ (\ \boxed{\mathit{Methods}_C}\ )\ [\mathsf{State}] \\
\quad\quad G_C \;=\; \lambda(\gamma)\ \lambda(I\,)\ P_\beta \oplus \Delta_C \langle\ I \oplus (\ \boxed{\mathit{State}_C}\ )\ \rangle \\
\quad\quad W_C \;=\; \lambda(\gamma)\ \mu_{\mathsf{self}} \langle ((\gamma \rightarrow C_\alpha).G\ (\beta \oplus \gamma))\ [\mathsf{self}] \rangle \\
\quad \textbf{in} \\
\quad\quad \langle G = G_C, W = W_C \rangle
\end{array}
$$

[1]The reader should note that a class generator $G_C$ defines the protocol between $\Delta_C$ and $P$, whereas a model generator $G_m$ defines the protocol between classes.

A class is characterized by the three abstractions $\Delta_C$, $G_C$, and $W_C$, respectively, all defined within the scope of $C_\alpha$. We use the syntactic form "**let** $v_1 = M_1$ ... $v_n = M_n$ **in** $N$" to define a $\lambda\mathcal{F}$-context containing the required private definitions of $\Delta_C$, $G_C$, and $W_C$, respectively, to capture the behaviour of $C_\alpha$.

The incremental modification $\Delta_C$ captures the behaviour defined by class $C$. In order to represent $C$'s behaviour, we use an approach based on the way *traits* are defined in the language Self [18]. However, to maintain a strict encapsulation of state, we do not blend state and methods. Instead, we model state as an explicit context, written [State], that provides an environment for each method to resolve the occurrences of private instance variables and the self reference.

The generator $G_C$ builds a prototype instance of class $C$. $G_C$ takes an activation classbox to provide a correct lookup environment to the parent behaviour $P_\beta$ originating from classbox $\beta$. Upon receiving the constructor arguments, denoted by $I$, the generator $G_C$ extends $P_\beta$ with the result of applying $C$'s incremental modification $\Delta_C$ to the combination of the constructor arguments and $C$'s state template.

The wrapper $W_C$ yields an object of class $C$ by building the fixed-point (denoted by $\mu_{\mathsf{self}}$) of the prototype instance being created within $W_C$. In order to create a prototype instance for class $C$, the wrapper uses the activation classbox $\gamma$ to a look up $C's$ generator. The expression $(\gamma \rightarrow C_\alpha).G$ denotes the fact that we look up the most recent definition of the named form $C_\alpha = \langle G, W \rangle$ in classbox $\gamma$ to invoke $C$'s generator. We apply this generator to $(\beta \oplus \gamma)$ that combines the classbox $\beta$ containing $C$'s parent behaviour $P_\beta$ with the activation classbox $\gamma$. The resulting classbox will contain not only the most recent definitions that class $C$ depends upon, but also the ones that have been implicitly imported. More specifically, $(\beta \oplus \gamma)$ denotes a lookup environment that is a transitive closure of $C$'s dependency graph.

A classbox is also represented as a form. The general structure of a classbox follows a format as shown in the example below:

$$
\begin{array}{l}
\mathit{ColorCB} = \\
\quad \textbf{let} \\
\quad\quad \mathsf{Point} = \langle G_{\mathsf{Point}}, W_{\mathsf{Point}} \rangle \\
\quad\quad \mathsf{BoundedPoint} = \langle G_{\mathsf{BoundedPoint}}, W_{\mathsf{BoundedPoint}} \rangle \\
\quad\quad \mathsf{LinearBoundedPoint} = \\
\quad\quad\quad\quad \langle G_{\mathsf{LinearBoundedPoint}}, W_{\mathsf{LinearBoundedPoint}} \rangle \\
\quad \textbf{in} \\
\quad\quad \langle \mathsf{Point}_{\mathsf{OriginalCB}} = \mathsf{Point}, \\
\quad\quad\ \mathsf{BoundedPoint}_{\mathsf{ColorCB}} = \mathsf{BoundedPoint}, \\
\quad\quad\ \mathsf{LinearBoundedPoint}_{\mathsf{LinearCB}} = \mathsf{LinearBoundedPoint} \rangle
\end{array}
$$

A classbox is a form that contains mappings from decorated class names to class definitions. Each decorated class name uniquely identifies the originating classbox of a class, that is, the classbox in which a class was first defined. For example, the classbox *ColorCB*, as shown in Figure 2, contains three classes Point, BoundedPoint, and LinearBoundedPoint, respectively. However, each class has a different originating classbox. Only class BoundedPoint originates in *ColorCB*. The classes Point and LinearBoundedPoint originate in *Orig-*

*inalCB* and *LinearCB*, respectively. In fact, both Point and LinearBoundedPoint occur as *extended classes* in *ColorCB*.

Decorated class names are the key ingredient in an approach that provides support for class extensions [3]. However, as labels are not first-class values in the $\lambda\mathcal{F}$-calculus, we cannot directly express decorated class names. But, $\lambda\mathcal{F}$ provides another abstraction that can be used instead: it is possible to denote operations involving decorated class names by so-called *abstract applications*. An abstract application (a M) is an $\lambda\mathcal{F}$-expression in which the function a is abstract, that is, a is defined *outside* $\lambda\mathcal{F}$. The intuition here is that a $\lambda\mathcal{F}$-term (i.e., (a M)) has to be embedded into a concrete target system that provides an interpretation of the abstract function. When applied to some argument, an abstract function has to yield a value that is again in $\lambda\mathcal{F}$.

To handle decorated class names in our $\lambda\mathcal{F}$-based model for classboxes, we need four abstract functions:

$$\mathsf{buildDecoratedName}\langle \mathsf{C}, \alpha\rangle = C_\alpha$$

$$\mathsf{lookupDecoratedName}\langle \mathsf{C}, \alpha\rangle = \begin{cases} C_\beta, \text{ if } \exists!\beta, (\alpha \rightarrow C_\beta) \neq \langle\rangle \\ \bot, \quad \text{otherwise} \end{cases}$$

$$\mathsf{lookupClass}\langle \mathsf{C}, \alpha\rangle = \alpha \rightarrow \mathsf{lookupDecoratedName}\langle \mathsf{C}, \alpha\rangle$$

$$\mathsf{buildClass}\langle \mathsf{C} = \langle G, W\rangle, \alpha\rangle = \langle \mathsf{lookupDecoratedName}\langle \mathsf{C}, \alpha\rangle = \langle G, W\rangle\rangle$$

The function buildDecoratedName takes a class name C and a classbox name $\alpha$ and returns a decorated class name $C_\alpha$ that is a valid $\lambda\mathcal{F}$-label. The function lookupDecoratedName takes a class name C and a classbox name $\alpha$ and returns a $\lambda\mathcal{F}$-label that denotes a valid decorated class name $C_\beta$, if such a name exists in the classbox $\alpha$. The function lookupClass takes a class name C and a classbox name $\alpha$ and returns a form that represents class $C$, as defined in classbox $\alpha$. Finally, the function buildClass takes a class $\mathsf{C} = \langle G, W\rangle$ and a classbox name $\alpha$ and yields a binding in which the label denotes a valid decorated class name for C in $\alpha$.

# 5. MODELING CLASSBOX OPERATIONS

In this section, we present our $\lambda\mathcal{F}$ encodings of classbox operations. More precisely, we show the encoding of *import* of classes, introduction of *subclasses*, *extension* of classes, and *inclusion* of new behaviour. The latter two operations are deduced from the original extend operator [3] by the refining process outlined in Section 3.

## 5.1 Import of classes

The import of a class C from classbox $\beta$ into classbox $\alpha$ is defined as shown below.

$$\begin{array}{l} C_\alpha = \\ \quad \textbf{let} \\ \quad\quad W_C = \lambda(\gamma)\ (\mathsf{lookupClass}\langle \mathsf{C}, \beta\rangle).W\ (\beta \oplus \gamma) \\ \quad \textbf{in} \\ \quad\quad (\mathsf{lookupClass}\langle \mathsf{C}, \beta\rangle)\langle W = W_C\rangle \end{array}$$

To import class C, we acquire its definition from classbox $\beta$ using the expression ($\mathsf{lookupClass}\langle \mathsf{C}, \beta\rangle$). However, the class C may depend on some behaviour for which an extended definition is given in classbox $\alpha$ (or the activation classbox $\gamma$ containing the extensions specified by classbox $\alpha$). Therefore, an imported class requires a new wrapper that combines the definitions of a class' originating classbox and the actual activation classbox. The result (i.e., $(\beta \oplus \gamma)$) is passed to the class' original wrapper that will use it to incorporate pertinent definitions into the class' behaviour. Finally, if not stated otherwise, the decorated class name $C_\alpha$ in this and all following encodings is the result of applying lookupDecoratedName to the class name C and the originating classbox name $\beta$ (i.e., $C_\alpha = \mathsf{lookupDecoratedName}\langle \mathsf{C}, \beta\rangle$).

## 5.2 Subclassing

We can define a class C as a subclass of class B originating from classbox $\beta$ using the following specification:

$$\begin{array}{l} C_\alpha = \\ \quad \textbf{let} \\ \quad\quad \Delta_C = \lambda(\mathsf{State})\ (\ \boxed{Methods_C}\ )\ [\mathsf{State}] \\ \\ \quad\quad G_C = \lambda(\gamma)\ \lambda(I) \\ \quad\quad\quad\quad \textbf{let} \\ \quad\quad\quad\quad\quad \mathsf{P} = ((\mathsf{lookupClass}\langle \mathsf{B}, \gamma\rangle).G\ \gamma)\ I \\ \quad\quad\quad\quad \textbf{in} \\ \quad\quad\quad\quad\quad \mathsf{P} \oplus \Delta_C\langle I \oplus \left(\begin{array}{|c|}\hline State_C \\ \hline \langle \mathsf{super} = \mathsf{P}\rangle \\ \hline \end{array}\right)\rangle \\ \\ \quad\quad W_C = \lambda(\gamma)\ \mu_{\mathsf{self}}\langle((\mathsf{lookupClass}\langle \mathsf{C}, \gamma\rangle).G\ (\beta \oplus \gamma))\ [\mathsf{self}]\rangle \\ \quad \textbf{in} \\ \quad\quad \langle G = G_C, W = W_C\rangle \end{array}$$

To construct class C, we acquire its superclass behaviour P using the expression $((\mathsf{lookupClass}\langle \mathsf{B}, \gamma\rangle).G\ \gamma)\ I$ and combine it with C's incremental modification $\Delta_C$. To acquire C's superclass behaviour, we dynamically look up B's generator with respect to the activation classbox $\gamma$, which guarantees that any relevant extensions to B's behaviour are also incorporated in class C. Methods in $\Delta_C$ may override methods in the superclass B. Overridden methods are accessible by means of the additional binding $\langle \mathsf{super} = \mathsf{P}\rangle$ passed to $\Delta_C$.

Since class B may also depend on some behaviour visible only to classbox $\beta$, we need to provide a reference of $\beta$ to C's wrapper $W_C$. This approach not only guarantees that references to class B can be resolved, but also that references to any superclasses of B can be resolved in classbox $\alpha$ without adding them to the visible scope of classbox $\alpha$. For example, the class LinearBoundedPoint in classbox *LinearCB*, as shown in Figure 2, implicitly depends on class Point defined in classbox *OriginalCB*. This dependency is resolved by providing the originating classbox of BoundedPoint to the wrapper of LinearBoundedPoint.

Finally, the classbox model as defined by Bergel et al. [3] requires that when defining a subclass, its class name must occur *fresh* in the defining classbox. Therefore, the decorated class name $C_\alpha$ is not derived from classbox $\beta$, but constructed with respect to the defining classbox $\alpha$ (i.e., $C_\alpha = \mathsf{buildDecoratedName}\langle \mathsf{C}, \alpha\rangle$).

## 5.3 Extending imported classes

To specify the semantics of the *refined* extend operation, we need to define an information hiding protocol that, when

applied to a concrete class, renders the features of the extensions invisible to the class' behaviour. Hence, the extend operation yields a membrane for a class that permits super calls originating from extensions, but prevents the class' behaviour to see the extensions. This protocol is established by confining self calls to the context within which they occur (i.e., the original class or the extensions). The extension of class C with some behaviour B can be defined as follows:

$$
\begin{array}{l}
B^E_\beta = \\
\quad \textbf{let} \\
\quad\quad \Delta_B = \lambda(\mathsf{State})\ (\ \boxed{\mathit{Methods}_B}\ )\ [\mathsf{State}] \\
\\
\quad\quad G_B = \lambda(\mathsf{Class}) \\
\quad\quad\quad\quad \lambda(\gamma)\ \lambda(I\,) \\
\quad\quad\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad\quad\quad \mathsf{P} = \mu_{\mathsf{self}}\langle\ (\mathsf{Class}.G\ \gamma)\ [\mathsf{self}]\ \rangle\ I \\
\quad\quad\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad\quad\quad \mathsf{P} \oplus \Delta_B\langle\ I \oplus \left(\ \begin{array}{|c|}\hline \mathit{State}_B \\ \hline \langle \mathsf{super} = \mathsf{P}\rangle \\ \hline \end{array}\ \right)\rangle \\
\\
\quad \textbf{in} \\
\quad\quad \langle G = G_C\rangle
\end{array}
$$

$$C_\alpha = (\mathsf{lookupClass}\langle \mathsf{C}, \alpha'\rangle)\langle G = B^E_\beta.G\ (\mathsf{lookupClass}\langle \mathsf{C}, \alpha'\rangle)\rangle$$

The abstraction $B^E_\beta$ captures the behaviour of the extension B being used to modify class C. The extend operator requires that we *encapsulate* C's behaviour in order to protect it from any changes defined by B. That is, we have to combine the fixed-point of C's prototype instance with the incremental modification $\Delta_B$ defined by extension B.

The structure of $B^E_\beta$ is similar to the one required to define subclassing. However, an extension cannot be instantiated independently. Therefore, no wrapper is needed. When combined with a concrete class, the class' wrapper is responsible for providing a suitable environment to create objects of that class. Moreover, the definition of the revised extend operator guarantees that the extensions are local to the classbox in which they occur and that they do not affect the class' original behaviour, as it is shielded from the extensions by binding self calls in the class' methods early.

The purely functional $\lambda\mathcal{F}$-based encoding of the extend operator is, however, a source for a serious problem. Functional update of state yields a new object. The new object is created by passing the new state values to the wrapper of the object's class. However, the wrapper of an extended class (i.e., $\mu_{\mathsf{self}}\langle(\mathsf{Class}.G\ \gamma)[\mathsf{self}]\rangle$) does not include the extensions. Therefore, functional update yields an instance of the original class, not one of the extended class.

## 5.4 Include behaviour into imported classes

Inclusion is a new operator that enables *down calls* to class extensions. The inclusion operator is like the extension operator, excepted that we do not encapsulate the class' behaviour. This approach roughly corresponds to the concept of *mixin application* [19]. That is, if we apply an extension B to class C, then the class C stands for an *abstract subclass*, which is instantiated with the superclass B. The inclusion of extension B into class C is defined as follows:

$$
\begin{array}{l}
B^I_\beta = \\
\quad \textbf{let} \\
\quad\quad \Delta_B = \lambda(\mathsf{State})\ (\ \boxed{\mathit{Methods}_B}\ )\ [\mathsf{State}] \\
\\
\quad\quad G_B = \lambda(\mathsf{Class}) \\
\quad\quad\quad\quad \lambda(\gamma)\ \lambda(I\,) \\
\quad\quad\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad\quad\quad \mathsf{P} = (\mathsf{Class}.G\ \gamma)\ I \\
\quad\quad\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad\quad\quad \mathsf{P} \oplus \Delta_B\langle\ I \oplus \left(\ \begin{array}{|c|}\hline \mathit{State}_B \\ \hline \langle \mathsf{original} = \mathsf{P}\rangle \\ \hline \end{array}\ \right)\rangle \\
\\
\quad \textbf{in} \\
\quad\quad \langle G = G_B\rangle
\end{array}
$$

$$
\begin{array}{l}
C_\alpha = \\
\quad \textbf{let} \\
\quad\quad G_C = \lambda(\gamma)\ (B^I_\beta.G\ \mathsf{lookupClass}\langle \mathsf{C}, \alpha'\rangle)\ \gamma \\
\quad\quad W_C = \lambda(\gamma)\ (\mathsf{lookupClass}\langle \mathsf{C}, \alpha'\rangle).W\ (\beta \oplus \gamma) \\
\quad \textbf{in} \\
\quad\quad \langle G = G_C, W = W_C\rangle
\end{array}
$$

Inclusion extension is an operation that combines extend and import. However, unlike extension, we do not build the fixed-point of the parent behaviour P in $B^I_\beta$ to enable down calls to the extensions. In addition, methods in $\Delta_B$ may override methods in class C. The overridden methods of C are, however, accessible by means of the additional binding $\langle \mathsf{original} = \mathsf{P}\rangle$ passed to $\Delta_B$. This approach was recently proposed by Bergel et al. [2] in their Classbox/J model. Finally, the functions $G_C$ and $W_C$ define the protocol required to properly link class C and the inclusion extension B.

# 6. CONCLUSIONS, FUTURE WORK

In this paper, we have used the $\lambda\mathcal{F}$-calculus to define a model for *classboxes*, a dynamically typed module system for object-oriented languages that provides support for controlling both the visibility and composition of class extensions, and validated our model using a prototype implementation of the $\lambda\mathcal{F}$-calculus.

This work has shown that $\lambda\mathcal{F}$ is a powerful tool to model compositional abstractions such as classes, classboxes as well as their associated operations. Replacing $\lambda$-calculus names by first-class namespaces and parameter passing by explicit contexts, we argue, are the key concepts in obtaining the resulting flexibility and extensibility. Both asymmetric form extension as well as the late binding of free variables in form expressions due to explicit form contexts are essential features to express the model in such an elegant way. It has also shown that the meta-level framework we defined in previous work [10] where object-oriented abstractions were modeled as compositions of appropriately parameterized generator, wrapper, and composer abstractions offers enough flexibility to incorporate classboxes.

As we have discussed in Section 3, the formal definition of class extension presented in [3] does not fully match its informal description. As a consequence, extend has limited applicability when *hook methods* [6] are to be extended by independent behavioural properties such as, for example, extending the class Point with a tracing mechanism on the

method move. Therefore, we have proposed a revised notion of class extension, incorporating two separate extension operations: (i) *extension* of classes in which self calls are encapsulated to the context in which they occur, and (ii) *inclusion* of new behaviour by means of late binding of self calls. Such a separation has allowed us to clarify the semantics of the classbox operators and seamlessly integrate the concept of *accessing the original method* as defined in the Classbox/J model [2]. In this context we have also illustrated that the early binding of self calls is the source of a serious problem when object-oriented abstractions are modeled in a purely functional setting such as $\lambda\mathcal{F}$.

However, this work has also shown that not all abstractions needed to define a model for classboxes can be expressed within $\lambda\mathcal{F}$, that is, we were forced to use *abstract applications* to model the decoration of class names. It is however not yet fully understood whether this is a limitation of $\lambda\mathcal{F}$ or a result of how classboxes have been formalized both in [3] and in our model. Therefore, we will investigate whether the expressiveness of classboxes can also be achieved without using explicitly decorated classnames, allowing us to model classboxes entirely in $\lambda\mathcal{F}$.

Furthermore, the concept of classboxes does not allow for an *explicit* co-existence of both the original of a class as well as an extension thereof. For example, it could become necessary for the class Point defined in *OriginalCB* and its extension with the color property to co-exist within the classbox *ColorCB*. Currently, both classes may *implicitly* co-exist within *ColorCB* and it is possible to create instances of the extended version of Point, but not of Point defined in *OriginalCB*. It is, however, not yet clear how to extend classboxes with this feature and whether this extension is of real practical value.

## Acknowledgements

We would like to the members of the Centre for Component Software and Enterprise Systems at Swinburne for inspiring discussions on these topics as well as Alexandre Bergel and the anonymous reviewers for commenting on earlier drafts.

## 7. REFERENCES

[1] U. Aßmann. *Invasive Software Composition*. Springer, 2003.

[2] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proceedings OOPSLA '05*, San Diego, USA, Oct. 2005. ACM Press. To appear.

[3] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling Visibility of Class Extensions. *Journal of Computer Languages, Systems & Structures*, 31(3–4):107–126, May 2005.

[4] G. Bracha and W. Cook. Mixin-based Inheritance. In N. Meyrowitz, editor, *Proceedings OOPSLA/ECOOP '90*, volume 25 of *ACM SIGPLAN Notices*, pages 303–311, Oct. 1990.

[5] L. Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Computer Science*, 192:201–231, Feb. 1998.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[7] M. Lumpe. *A $\pi$-Calculus Based Approach for Software Composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, Jan. 1999.

[8] M. Lumpe. A Lambda Calculus With Forms. In T. Gschwind, U. Aßmann, and O. Nierstrasz, editors, *Proceedings of the Fourth International Workshop on Software Composition*, LNCS 3628, pages 83–98, Edinburgh, Scotland, Apr. 2005. Springer.

[9] M. Lumpe and J.-G. Schneider. Form-based Software Composition. In M. Barnett, S. Edwards, D. Giannakopoulou, and G. T. Leavens, editors, *Proceedings of ESEC '03 Workshop on Specification and Verification of Component-Based Systems (SAVCBS '03)*, pages 58–65, Helsinki, Finland, Sept. 2003.

[10] M. Lumpe and J.-G. Schneider. A Form-based Metamodel for Software Composition. *Science of Computer Programming*, 56:59–78, Apr. 2005.

[11] R. Milner. *Communicating and Mobile Systems: the $\pi$-Calculus*. Cambridge University Press, 1999.

[12] H. Mössenböck. *C# to the Point*. Addison-Wesley, 2005.

[13] O. Nierstrasz and F. Achermann. A Calculus for Modeling Software Components. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Proceedings of First International Symposium on Formal Methods for Components and Objects (FMCO 2002)*, LNCS 2852, pages 339–360, Leiden, The Netherlands, 2003. Springer.

[14] O. Nierstrasz, J.-G. Schneider, and M. Lumpe. Formalizing Composable Software Systems – A Research Agenda. In *Proceedings the 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 271–282. Chapmann & Hall, 1996.

[15] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, sixth edition, 2005.

[16] J.-G. Schneider and M. Lumpe. Synchronizing Concurrent Objects in the Pi-Calculus. In R. Ducournau and S. Garlatti, editors, *Proceedings of Langages et Modèles à Objets '97*, pages 61–76, Roscoff, Oct. 1997. Hermes.

[17] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, Second edition, 2002.

[18] D. Ungar and R. B. Smith. SELF: The Power of Simplicity. In *Proceedings OOPSLA '87*, volume 22 of *ACM SIGPLAN Notices*, pages 227–242, Dec. 1987.

[19] M. Van Limberghen and T. Mens. Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems. *Object-Oriented Systems*, 3(1):1–30, Mar. 1996.

# A Specification-Based Approach to Reasoning About Pointers

Gregory Kulczycki
Virginia Tech
Falls Church, VA
gregwk@vt. edu

Murali Sitaraman
Clemson University
Clemson, SC
murali@cs.clemson.edu

Bruce W. Weide
Atanas Rountev
The Ohio State University
Columbus, OH
weide@cse.ohio-state.edu
rountev@cse.ohio-state.edu

## ABSTRACT

This paper explains how a uniform, specification-based approach to reasoning about component-based programs can be used to reason about programs that manipulate pointers. No special axioms, language semantics, global heap model, or proof rules for pointers are necessary. We show how this is possible by capturing pointers and operations that manipulate them in the specification of a software component. The proposed approach is mechanizable as long as programmers are able to understand mathematical specifications and write assertions, such as loop invariants. While some of the previous efforts in reasoning do not require such mathematical sophistication on the part of programmers, they are limited in the kinds of properties they can prove about programs that use pointers. We illustrate the idea using a "Splice" operation for linked lists, which has been used previously to explain other analysis techniques. Not only can the proposed approach be used to establish shape properties given lightweight specifications, but also it can be used to establish total correctness given more complete specifications.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *correctness proofs, formal methods.* D.3.3 [**Programming Languages**]: Language Constructs and Features – *data types and structures.*

## General Terms

Languages, Verification.

## Keywords

Pointer specification, reasoning, heap memory management.

## 1. INTRODUCTION

Reasoning about program code involving pointers or references is notoriously difficult [29]. Various logics have been developed for object-oriented languages such as Java and C# in which references are implicit [1][18]. Reasoning about programs in these languages is complicated due to the possibility of aliasing. Various static analysis techniques also have been applied to languages with and without explicit deallocation (e.g., [6][22][30]). These techniques are fast and flexible, but they are also limited in what they can prove about a program's run-time behavior. Both object-oriented logics and general program analysis techniques tend to rely on global reasoning about entire heap abstractions. Frame properties [3] make reasoning about heap locations somewhat less demanding in object-oriented logics, and a separation logic [20] has been suggested as a way to further localize reasoning to portions of the heap structure. Nonetheless, a single heap abstraction is still assumed. Building on previous work in shape analysis [30], Hackett and Rugina [6] describe an approach that avoids global heap abstractions. Instead, it uses local reasoning about individual heap locations to find potential errors.

In this paper, we consider from a language design perspective the problem of pointers and reasoning about programs that use them. We describe a way to implement and reason about programs involving pointers by using a formally specified generic component that encapsulates pointer-like behavior and that is especially well suited for the implementation of linked data structures. Furthermore, no global heap abstraction is used in reasoning about pointers. Instead, shared conceptual (or specification-only) variables whose scope is at the component level are defined in the specification. These variables record the state of the pointer structure, keeping track of such information as which locations are mapped to which objects and how the locations are linked to one another. The component permits explicit deallocation and thereby allows users to reason about memory errors that do not arise with garbage collection, so it can accommodate situations and languages where no automatic garbage collection is assumed. Section 2 briefly describes the specification of the pointer component abstraction and its operations.

A key contribution of the specification-based approach to reasoning about programs with pointers is that programmers can use and reason about pointers using the same techniques that they use to reason about all other components in a program. This does not preclude a language designer from inventing special syntax for pointers as long as the meaning of that syntax can be described in terms of the operations specified in the component. In particular, for the component to exhibit the run-time performance of language-supplied pointers, a compiler for a language with component-provided pointers need not implement these component operations as typical calls, but may consider them to be built-in constructs. For example, even though a programmer using our component will reason about the pointer assignment statement "*p* = *q*" as a call *Relocate(p, q)*—a procedure call that must conform to the contract specified by its precondition and postcondition—the compiler may implement this statement as a single machine instruction that overwrites *p*'s value with the address stored in *q*.

A second contribution of the specification-based approach is that it facilitates more powerful reasoning about properties of pointer-based programs than previous static analyses relying on special rules to handle language-supplied pointers. This is the topic of Section 3. In that section, we illustrate the issues using an exam-

ple "Splice" operation for a linked list. The example is taken from a recent paper on a general approach to shape analysis [6]. In that paper, Hackett and Rugina introduce and use a region-based shape analysis algorithm to establish the "shape property" that the Splice code does not introduce cycles into lists. They describe a non-trivial algorithm that partitions memory into regions, keeping track of the relationships between regions using a unification-based points-to analysis [27] that they augment with context sensitivity. Individual "configurations" are used to track the state of individual heap locations. These configurations can be analyzed independently of each other, eliminating the need to keep track of how an entire heap abstraction changes over the course of a program's execution.

Shape analysis is fully automatic and, unlike our specification-based approach, does not require programmer-supplied assertions. However, the authors note, for example, that their analysis would not apply if the Splice code were written slightly differently. More importantly, shape analysis techniques—and other static analysis techniques—are limited in the kinds of properties they can be used to prove. We illustrate these issues using both lightweight and heavyweight specifications for the "Splice" operation. Whereas the lightweight specification is sufficient to prove the assertion that an implementation of Splice does not introduce cycles, a more complete specification of the operation shows the potential of the pointer specification approach to analyze the full behavior of the operation and its implementation.

## 2. SPECIFYING POINTER BEHAVIOR

A formal specification of a component to capture pointer behavior is given in the technical report [12], where the design rationale for the specification and performance ramifications are discussed. The specification is general and it allows reasoning about any pointer-based data structures, including lists and trees. A skeleton of this specification is discussed in this subsection as a prelude to specification-based reasoning about pointers.

### 2.1 Mathematical Modeling

Without loss of generality, the specification in Figure 1 is given in the RESOLVE notation [23][25]. The specification *defines* Location as a mathematical set. The exact set of addresses that correspond to locations is an internal implementation detail, and it is suppressed in the specification. For the purposes of reasoning, the client programmer need only know that Location is a set and Void is a specific location element from that set. At any given program state, some locations are *free*, or available for allocation; and some locations are *taken*, or already allocated. A key aspect of the specification is to formalize how locations become linked to each other following various pointer manipulation operations. Hence the name *Location_Linking_Template* for the concept.

The concept is parameterized by the type of information associated with each location and the number of links from each location. If the nodes of a list contain GIF pictures, for example, then Info is a type representing GIF pictures. Similarly, the number of links depends on the application. For example, a singly linked list requires one link from each location, whereas a *k*-ary tree requires *k* links from each location.

To capture the behavior of a system of linked locations, the concept defines and uses three global, conceptual variables: *Contents*(*q*) is the information at a given location *q*, *Target*(*q*, *i*) is the location targeted by the *i*-th link of *q*, and *Is_Taken*(*q*) is true if and only if a given location *q* is allocated, and therefore, taken. These variables are not programming variables; they are used solely for specification and reasoning. They are similar to specification-only variables used in other formalisms for object-oriented programs [4][14].

```
Concept Location_Linking_Template (type Info;
                                   evaluates k: Integer);
    Defines Location: Set;
    Defines Void: Location;

    Var Target: Location × [1..k] → Location;
    Var Contents: Location → Info;
    Var Is_Taken: Location → B;

    Initialization ensures ∀q: Location, ¬Is Taken(q);
    Constraints ¬Is Taken(Void) and (∀q: Location,
        if ¬Is Taken(q) then Info.Is_Initial(Contents(q)) and
        ∀j: [1..k], Target(q, j) = Void) and …

    Type Family Position is modeled by Location;
        exemplar p;
        Initialization ensures p = Void;

    Operation Take_New_Location(updates p: Position);
        …
    Operation Abandon_Location(clears p: Position);
        …
    Operation Relocate(updates p: Position;
                       preserves q: Position);
        ensures p = q;

    Operation Follow_Link(updates p: Position;
                          evaluates i: Integer);
        requires Is Taken(p) and 1 ≤ i ≤ k;
        ensures p = Target(#p, i);

    Operation Redirect_Link(preserves p: Position;
                  evaluates i: Integer; preserves q: Position);
        updates Target;
        requires 1 ≤ i ≤ k and Is_Taken(p);
        ensures ∀r: Location, ∀j: [1..k],
                Target(r, j) = { q              if r = p and j = i
                               { # Target(r, j) otherwise          ;

    Operation Check_Colocation(preserves p, q: Position;
                               replaces are_colocated: Boolean);
                                   …
    Operation Swap_Locations(preserves p: Position;
            evaluates i: Integer; updates new_target: Position);
        …
    Operation Swap_Contents(preserves p: Position;
                            updates I: Info);
        …
    Operation Is_At_Void(preserves p: Position): Boolean;
        …
    Operation Location_Size(): Integer;
        …
end Location_Linking_Template;
```

**Figure 1. A Skeleton of Pointer Behavior Specification.**

The concept is constrained to behave as specified in the invariant constraints clause: the Void location can never be taken or allocated, i.e., it is always free; and all locations that are freely available are in an initialized state, i.e., their contents have default information and their links point to Void. Finally, the number of locations available is related to the total available memory capacity. Hence, locations are limited, and allocations without corresponding deallocations will eventually deplete the pool. Initially, all locations are assumed to be free and, therefore, initialized, as specified in the **constraints** clause. This does not mean that an implementation of the pointer component must initialize every location at the beginning; of course, it would not. It just means that any newly allocated location is guaranteed to be initialized, and this objective can be achieved as and when it is needed.

Given this model, a programming variable of type Position is simply viewed mathematically as a location. When a programmer declares a new pointer variable, it is initially just the Void location. Only after allocation does the variable become a location that can contain information.

A system of linked locations is established when a client instantiates the pointer component with the type of information that each location holds and the number of links coming from each location. Figure 2 gives an informal view of an example system of linked locations where type Info is assumed to be Greek letters and the number of links from each location is assumed to be one. Here, the circles represent locations that contain information (Greek letters) and a fixed number of links (just one in the example) to other locations.

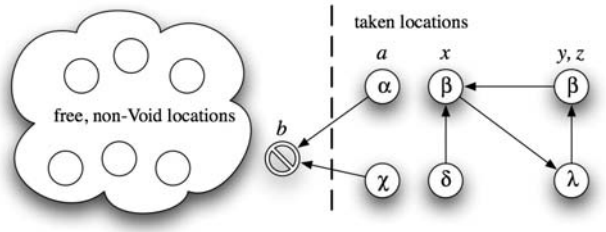



**Figure 2. A system of linked locations.**

The figure shows that some of the locations are taken and others are free. All the pointer variables (*a*, *b*, *x*, *y*, *z*) except for *b* are at allocated locations. For example, the information content at the location of variable *x* is β. The pointer variables *y* and *z* are colocated, i.e., they are aliased, and the link from that location points to the location of *x*. The pointer variable *b* resides at the special Void location, which is perpetually free. Due to poor programming or possible reliance on a garbage collector, some of the allocated locations have become inaccessible, such as the locations containing information χ and δ. To manipulate pointer variables to reach a state like the one in Figure 2, a programmer has to declare pointer variables and call suitable operations, as explained in the next subsection.

## 2.2 Discussion of Pointer Operations

The parameters in the specifications of operations in Figure 1 use various modes to help the programmer understand rough effects of a call to the arguments before reading the subsequent formal specification. The **updates** mode indicates that the operation modifies this argument; the **clears** mode ensures that the argument will return from the procedure with an initial value of its type; the **preserves** mode prohibits any changes to the argument's value; the **replaces** mode indicates that the incoming argument value will be ignored but replaced; and the **evaluates** mode indicates that the operation expects an expression in this position—it is typically used with types that are often returned from functions, such as integers.

The *Take_New_Location* operation allows a programmer to associate information with a specified pointer variable. Every call to this operation leads to a new location being taken. Internally, this operation allocates memory for a new object of type Info and makes *p* point to it. A taken location remains taken until the client abandons it, which she can do using the *Abandon_Location* operation. When a location is abandoned, memory for the information it contained is reclaimed and the pointer variable that is being explicitly abandoned is repositioned to the Void location. The *Location_Size* operation helps a programmer determine if there is sufficient memory for a new allocation, i.e., if there are any more free locations available for taking. A careful programmer may need this operation to check availability before calling the *Take_New_Location* operation.

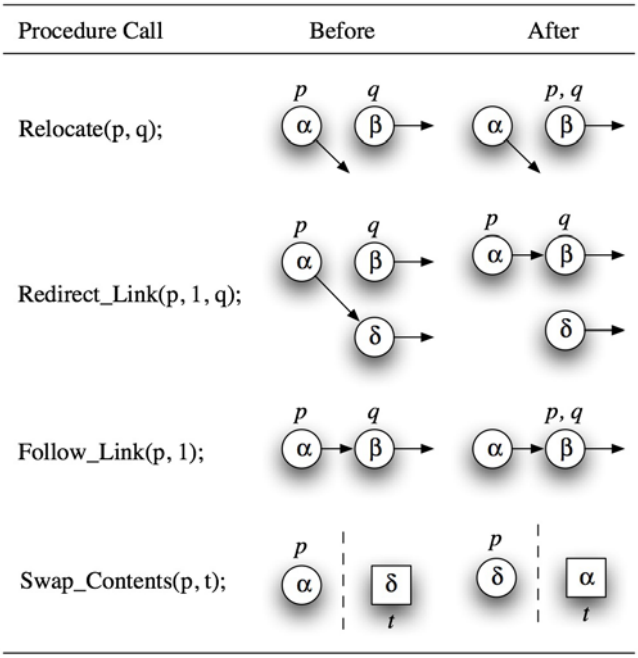



**Figure 3. The effects of selected calls on a system.**

Now we consider the formal specification of operation *Redirect_Link*. This operation redirects the *i*-th link at position *p*'s location to *q*'s location. The values of both *p* and *q* are preserved, since both occupy the same locations they did before the operation is invoked. The **updates** clause after the formal parameters lists any component-level conceptual variables that are affected by the operation. In this case, the *Target* variable will be modified, but the *Contents* and the *Is_Taken* variables will remain unchanged. The RESOLVE specification language adheres to an implicit frame property in that operation invocations may only affect explicit parameters to the operation or component conceptual variables listed in the **updates** clause. In this case, the operation specifies that the values of both position parameters are preserved and that any integer parameter is treated as an expression,

so the only variable that is modified is *Target*. The postcondition describes how the *Target* variable is modified: the *Target* function does not change except that it now maps the tuple (*p*, *i*) to *q*. Note that the hash (#) symbol denotes the incoming value of a variable. The specifications for *Relocate* and *Follow_Link* are straightforward.

The remaining operations allow a client to manipulate links and information at occupied locations. They also allow position variables to be associated with different locations. Figure 3 gives before and after views of a system for invocations of selected operations. Note that the box in the *Swap_Contents* operation is not a location. It simply indicates the value of Info variable *t*. In this case, *t*'s value is α before the operation and δ after the operation. In contrast, *p*'s value (which is a location) remains the same before and after the operation, but the *Target* variable will have been updated. All the operations shown here have formal specifications associated with them [12]. We have shown only a few of these in Figure 1, owing to space constraints.

Because the pointer component supports explicit deallocation, memory errors can arise within the context of a system. A location is considered accessible if there is a path to that location from a location occupied by some position variable. In Figure 2, for example, two taken locations are not accessible: the location containing δ and the location containing χ. The location containing λ is accessible even though it is not occupied by a position variable because there is a path to it from the location occupied by *x*. A location that is taken but not accessible is a memory leak; a location (other than Void) that is free but accessible is a dangling reference. The latter situation is not, perforce, a memory error; but it becomes one if that pointer variable is then used before being updated.

## 3. EXAMPLE

This section illustrates how the pointer component can be used for both lightweight and heavyweight specifications and subsequent reasoning. The Splice operation takes as input two singly-linked lists of locations: one that begins with a location occupied by position *p* and another that begins with a location occupied by position *q*. The length of *q*'s list must be less than or equal to the length of *p*'s list. The operation modifies the first list so that it is a perfect shuffle of the locations in the original lists. A shuffled list contains all the elements of both lists with their original orderings preserved, similar to what happens when you shuffle a deck of cards. If location *x* appears before location *y* in one of the original lists, then *x* appears before *y* in the shuffled list. A perfect shuffle interleaves elements from each of the lists.

### 3.1 Simple Splice Specification

Figure 5 gives a lightweight specification and code for Splice (a minor syntactic variation of the version in [6]). The specification is sufficient to meet the goal of a typical shape analysis for the operation, namely to "statically verify that, if the input lists […] are disjoint and acyclic, then the [output list] is acyclic" [6] (page 3). Note that we use a syntactic shortcut throughout this section for ease of reading (and writing) the specifications. Since all locations in these examples have exactly one link, we leave out the link number where it is typically required. For example, we use *Target*(*p*) instead of *Target*(*p*, 1).

The specification defines two mathematical functions used in the specification. *Is_Reachable_in*(*n*, *p*, *q*) is true if and only if location *q* is reachable from location *p* in *n* hops. That is, if *x* is a variable at location *p*, this function is true if and only if *x* will arrive at location *q* by following his first link *n* times, but no fewer. The term $Target^k(p)$ means *k* iterations of the function *Target* starting with *p*. The requirement that "**if** $Target^k(p) = q$ **then** k ≥ hops" for all k, guarantees, for example, that *Is_Reachable_in*(10, *p*, *q*) is false if *q* links back to itself and it only takes 5 hops for *p* to reach *q*. *Is_Reachable*(*p*, *q*) is true iff *q* is reachable from *p* in any number of hops. When the *Var* keyword follows *Definition*, it indicates that the value of the function may vary for the same input values in different program states. For example, *Is_Reachable*(*p*, *q*) may be true in one state and false in the next if one of the links between them was redirected. The *Distance* between *p* and *q* is the number of hops it takes to get from *p* to *q* if *q* is reachable from *p*; otherwise, the distance is zero.

**Definition Var** Is_Reachable_in(hops: **N**; p, q: Location;): **B** =
  $Target^{hops}(p) = q$ **and** ∀k: **N**, **if** $Target^k(p) = q$ **then** k ≥ hops;

**Definition Var** Is_Reachable(p, q: Location): **B** =
  ∃k: **N** ϶ Is_Reachable_in(k, p, q);

**Definition Var** Distance(p, q: Location): **N**

$$= \begin{cases} k & \text{if Is\_Reachable\_in}(k, p, q) \\ 0 & \text{otherwise} \end{cases};$$

**Operation** Splice(**preserves** p: Position; **clears** q: Position);
  **updates** Target;
  **requires** ( $\exists k_1, k_2$: **N** ϶ Is_Reachable_in($k_1$, p, Void) **and**
    Is_Reachable_in($k_2$, q, Void) **and** $k_2 \le k_1$ ) **and**
    ( ∀r: Location, **if** Is_Reachable(p, r) **and**
      Is_Reachable(q, r) **then** r = Void );
  **ensures** Is_Reachable(p, Void);
**Procedure**
  **Var** r: Position;
  **Var** s: Position;
  Relocate(r, p);
  **While** (**not** At_Void(q))
    **decreasing** Distance(q, Void);
    **maintaining** Is_Reachable(p, Void);
  **do**
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
  **end**;
**end** Splice;

**Figure 5. A lightweight specification for Splice.**

The Splice operation preserves *p* and clears *q*. In other words, *p* is unchanged and *q* is Void after the operation. The **updates** clause indicates that Target is the only conceptual variable that is modified.

The **requires** clause is fulfilled only if the linked lists beginning at *p* and *q* are acyclic and disjoint. The only way that a location can reach Void is if there are no cycles in the linked structure

beginning with that location. The Void location always links to itself. Provided that lists are free of cycles, $k_1$ and $k_2$ represent the lengths of lists *p* and *q*, respectively, and $k_1$ must be greater than or equal to $k_2$. Finally, if any location other than Void is reachable by both *p* and *q*, then the lists are not disjoint. The simple **ensures** clause is true when the output list *p* is acyclic.

Figure 6 illustrates the execution of the splice operation when *p* is a linked list with four locations and *q* is a linked list with two locations. Part (a) represents the state of the system at the beginning of the first loop, part (b) represents the system at the beginning of the second loop iteration, and part (c) represents the state of the system when the loop terminates.

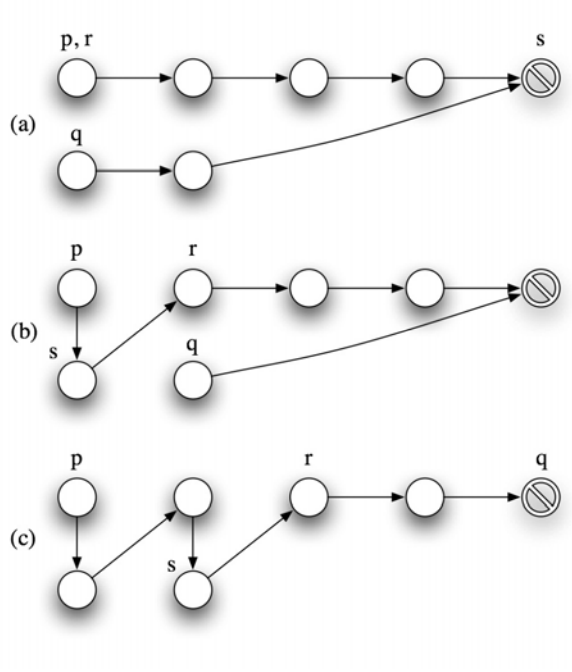



**Figure 6. An animation of the implementation for Splice.**

The correctness of the implementation can be proved using the formal proof system detailed in [7][10] and summarized in [25]. The proof process takes the programmer-supplied invariant for the loop, establishes that it is invariant, and employs it in completing the proof. For the present example, the loop invariant asserts that it is always possible to reach the Void location from *p*. It is obviously true at the beginning of the first iteration, since *p* does not change and we know from the precondition that Void is reachable from *p*. All that is left to prove is that the invariant holds from one iteration to the next. This follows from the fact that the following lemmas hold in each state in the loop.

**Lemma** #1: Is_Reachable(q, Void);
**Lemma** #2: Is_Reachable(r, Void);
**Lemma** #3: Is_Reachable(p, r) **or** Is_Reachable(p, q);

The proof of correctness of Splice follows from the invariant and the negation of the loop condition. For proving termination, the process uses a progress metric given in the **decreasing** clause. The progress metric states that the "distance" from *q* to Void decreases with each iteration of the loop. This argument establishes the same result as the intricate shape analysis proposed in [6]. A limitation of the Splice operation as specified in this section is that it cannot be used as a meaningful guide to anyone who implements the operation. In fact, even an implementation that does nothing at all will guarantee the postcondition because it follows directly from the precondition. The next section provides a more detailed specification for the Splice operation.

## 3.2 Full Splice Specification

The full specification of the Splice operation is given in Figure 5.

**Definition Var** Is_Reachable_in(hops: **N**; p, q: Location;): **B** =
Target$^{hops}$(p) = q **and** ∀k: **N**, **if** Target$^{k}$(p) = q **then** k ≥ hops;

**Definition Var** Is_Reachable(p, q: Location): **B** =
∃k: **N** ϶ Is_Reachable_in(k, p, q);

**Definition Var** Distance(p, q: Location): **N**

$$= \begin{cases} k & \text{if Is\_Reachable\_in}(k, p, q) \\ 0 & \text{otherwise} \end{cases};$$

**Definition Var** Is_Info_Str(p, q: Location; α: Str(Info)): **B** =
∃n: N ϶ Is_Reachable_in(n, p, q) **and**
$\alpha = \prod_{k=1}^{n} \langle \text{Contents}(\text{Target}^{k}(p)) \rangle$;

**Operation** Splice(**preserves** p: Position; **clears** q: Position);
**updates** Target;
**requires** ( ∃$k_1$, $k_2$: **N** ϶ Is_Reachable_in($k_1$, p, Void) **and**
Is_Reachable_in($k_2$, q, Void) **and** $k_2 \le k_1$ ) **and**
( ∀r: Location, **if** Is_Reachable(p, r) **and**
Is_Reachable(q, r) **then** r = Void );
**ensures** ( ∀t: Location, **if not** Is_Reachable(#p, t) **and**
**not** Is_Reachable(#q, t) **then** Target(t) = #Target(t) ) **and**
( ∀α, β, γ: Str(Info), **if** Is_Info_Str(p, Void, α) **and**
Is_Info_Str(#p, Void, β) **and** Is_Info_Str(#q, Void, γ)
**then** α ≤!≥ (β, γ) );

**Procedure**
**Var** r: Position;
**Var** s: Position;
Relocate(r, p);
**While** (**not** At_Void(q))
**decreasing** Distance(q, Void);
**maintaining** ( ∀t: Location, **if not** Is_Reachable(#p, t) **and**
**not** Is_Reachable (#q, t) **then** Target(t) = #Target(t) ) **and**
( ∀χ, δ, ε. β, γ, ρ: Str(Info), **if** Is_Info_Str(p, r, χ) **and**
Is_Info_Str(r, Void, δ) **and** Is_Info_Str(q, Void, ε) **and**
Is_Info_Str(#p, Void, β) **and** Is_Info_Str(#q, Void, γ) **and**
ρ ≤!≥ (δ, ε) **then** χ o ρ ≤!≥ (β, γ) );
**do**
Relocate(s, r);
Follow_Link(r);
Redirect_Link(s, q);
Follow_Link(s);
Follow_Link(q);
Redirect_Link(s, r);
**end**;
**end** Splice;

**Figure 7. A full specification for Splice.**

We assume the definitions given above and introduce another one: *Is_Info_Str*. The function *Is_Info_Str*(*p*, *q*, α) is true if and only if *q* is reachable from *p* and α is the string of all the objects of type Info contained in the locations between *p* and *q*. The

string includes the object in *p* but not the object in *q*. For example, for the system of linked locations in Figure 2, *Is_Info_Str*(*x*, *y*, ⟨β, λ⟩) is true.

The **requires** clause in this specification has not changed from the last one. However, the **ensures** clause is more detailed. It has two main conjuncts. The first conjunct indicates which portions of the *Target* variable do not change. It asserts that the links of locations do not change for locations that are not part of either input list. Note that we know that the contents and the taken status of all locations in the system are not affected by this operation because the variables *Contents* and *Is_Taken* are not included in the **updates** clause. The second main conjunct in the **ensures** clause describes how the lists are modified. Essentially it says that α is the string of Info objects derived from the output list, β and γ are the strings of Info objects derived from the two input lists, and α is a perfect shuffle (or "interleaving") of β and γ, which we denote by $\alpha \leq!\geq (\beta, \gamma)$. Recall that a perfect shuffle of two strings is a shuffle that interleaves the first *n* elements of each string, where *n* is the length of the shorter string. A perfect shuffle always starts with the first element of the first string. For example, ⟨a, e, b, f, c, d⟩ is a perfect shuffle of the strings ⟨a, b, c, d⟩ and ⟨e, f⟩.

Since the postcondition has been strengthened, the loop invariant also needs to be stronger. Like the **ensures** clause, the loop invariant is divided into two main conjuncts. The first conjunct simply mirrors the first part of the **ensures** clause. The second conjunct asserts that if ρ is a perfect shuffle of the linked lists beginning with *r* and *q*, then when the string beginning with *p* and ending with *r* is concatenated with ρ, the resulting list is a perfect shuffle of the input lists. For convenience, we have chosen strings β and γ to designate the same strings in the invariant as they do in the **ensures** clause. The invariant includes a few extra strings: χ, which is the string of Info objects between *q* and Void; δ, which is the string of objects between *r* and Void; and ε, which is the string of objects between *q* and Void.

At the beginning of the first iteration of the loop, χ is the empty string, while δ = β and ε = γ. So, when $\rho \leq!\geq (\delta, \varepsilon)$, we also know that $\chi \circ \rho \leq!\geq (\beta, \gamma)$. When the loop terminates, *q* = Void, so ε represents the empty string and therefore ρ = δ and $\chi \circ \rho = \chi \circ \delta$. But $\chi \circ \delta$ is the same as α in the ensures clause, so that $\alpha \leq!\geq (\beta, \gamma)$ follows directly from $\chi \circ \rho \leq!\geq (\beta, \gamma)$.

Finally, the proof of correctness for the Splice operation must show that $\chi \circ \rho \leq!\geq (\beta, \gamma)$ is maintained in the invariant. If we assume that $\chi \circ \rho \leq!\geq (\beta, \gamma)$ at the beginning of some arbitrary iteration, we must then show that $\chi' \circ \rho' \leq!\geq (\beta, \gamma)$ at the beginning of the next iteration, where χ′ and ρ′ are the new values of χ and ρ. (Note that β and γ are based on #*p* and #*q*, so they do not change from one iteration to the next.) Since *r* and *q* both advance exactly one location, we know that $\delta = \langle x \rangle \circ \delta'$ and $\varepsilon = \langle y \rangle \circ \varepsilon'$ for some Info objects *x* and *y*. A perfect shuffle of δ′ and ε′ will be the same as a perfect shuffle of δ and ε except that it will no longer hold *x* and *y*. In other words, $\rho' = \langle x \rangle \circ \langle y \rangle \circ \rho$. While ρ loses these objects, the Info string χ picks them up. In the code, position *s* moves to the location containing *x*, redirects the link there to the location containing *y*, follows the link, and then redirects the link at that location toward *r*'s new location. As a result of this traversal, $\chi' = \chi \circ \langle x \rangle \circ \langle y \rangle$. When χ′ and ρ′ are concatenated we get $\chi' \circ \rho' = \chi \circ \rho$, so that $\chi \circ \rho \leq!\geq (\beta, \gamma)$ implies $\chi' \circ \rho' \leq!\geq (\beta, \gamma)$.

Of course, a formal proof of the invariant would be much more intricate, but this should give the reader an idea of how to proceed.

# 4. DISCUSSION

Using programmer-supplied loop invariants (similar to our approach for handling loops), Jenson et al. have discussed in [9] how to prove heap-related properties and find counterexamples to claimed properties. Their implementation has been shown to be 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 Møller and Schwartzbach have extended the results to all data structures that can be expressed as "graph types" [17]. The Alloy approach "targets properties of the heap" [28] in a quest to root out erroneous implementations of linked data structures and null dereferences. The ESC/Java tool [15] has the ability to statically detect heap-related errors in Java. Though we have focused only on Hackett and Rugina's work in this paper, there is significant other work in shape analysis, including work on parametric shape analysis that allows more questions to be answered concerning heaps [22]. None of these efforts is based on a general, formal specification of pointer behavior.

The idea of capturing pointer behavior in the form of a component is not new. Safe pointers [16] and checked pointers [21] are generic C++ classes designed to alleviate memory errors in C++ by implementing all or part of the memory management code inside a pointer-like data structure. In contrast, our pointer specification supports manual memory management and the memory errors that can occur as the result of it. Though we have focused only on proving properties and correctness through reasoning, the results can be combined with previous work [26] to identify errors through analysis. In addition, these errors are statically predictable in the context of a formal specification and verification system that does not treat reasoning about pointers different from reasoning about any other component.

Despite the fact that many object-oriented languages avoid most memory errors by using automatic garbage collection, implicit pointers (references) remain a serious problem for both formalists and practitioners. This is due primarily to aliasing [8]. Aliasing—in the absence of a complete model of pointers and their referents—breaks encapsulation [19] and hence thwarts modular reasoning [5]. When pointers are appropriately modeled, formal specification and verification is complicated because the model must cope with soundness [29]. Therefore, various proposals have been introduced to control object aliasing, such as [5][19]. Reasoning about programs that incorporate these techniques is typically done in the context of object-oriented logics that use a global heap abstraction.

A complete formal specification of the pointer component described here was omitted due to space considerations, but it can be found in [12]. Future research includes exploring how the specification can be adapted for languages with automatic garbage collection, and how we can develop both lightweight and

heavyweight performance specifications [11][24] towards analyzing the performance of pointer-based programs.

## 5. ACKNOWLEDGMENTS

We would like to acknowledge Bill Ogden for his insights into the design of the pointer specification. We would also like to thank Kunal Chopra and Jason Mashburn for intricate discussions of the topics in this paper. This work is funded in part by the National Science Foundation grant CCR-0113181.

## 6. REFERENCES

[1] Abadi, M. and Leino, K. R. M. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th Internation Joint Conference*, pages 682–696. Springer-Verlag, New York, 1997.

[2] Barnett, M., Leino, K. R. M., and Schulte, W. The Spec# programming system: An overview. In *CASSIS 2004*, LNCS vol. 3362, Springer, 2004.

[3] Borgida, A., Mylopoulos, J., and Reiter, R. … and nothing else changes?: The frame problem in procedure specifications. In *Proceedings of the 15th International Conference on Software Engineering*. IEEE Computer Society Press, pages 303–314, 1993.

[4] Cheon, Y., Leavens, G. T., Sitaraman, M., and Edwards, S. Model variables: Cleanly supporting abstraction in design by contract. *Software, Practice, and Experience*, 35 (6), pages 583–599, 2005.

[5] Clarke, D. G., Potter, J. M., and Noble, J. Ownership types for flexible alias protection. In *Proceedings OOPSLA '98*, pages 48–64, 1998.

[6] Hackett, B. and Rugina, R. Region-based shape analysis with tracked locations. In *Proceedings POPL '05*, January 2005.

[7] Heym, W. *Computer Program Verification: Improvements for Human Reasoning*. Ph.D. thesis, The Ohio State University, 1995.

[8] Hogg, J., Lea, D., Wills, A., deChampeaux, D., and Holt, R. *The Geneva Convention on the treatment of object aliasing*. OOPS Messenger, 3(2):11–16, 1992.

[9] Jensen, J. L., Jorgensen, M. E., Klarlund, N., and Schwartzbach, M. I. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings SIGPLAN Conference on Programming Language Design and Implementation, 1997*.

[10] Krone, J. *The Role of Verification in Software Reusability*. Ph.D. thesis, The Ohio State University, 1988.

[11] Krone, J., Ogden, W. F., and Sitaraman, M. Modular verification of performance correctness. *In OOPSLA 2001 SAVCBS Workshop Proceedings*, 2001. http://www.cs.iastate.edu/~leavens/SAVCBS/papers-2001/index.html.

[12] Kulczycki, G., Sitaraman, M., Ogden, W. F., and Hollingsworth, J. E. *Component Technology for Pointers: Why and How*, Technical Report RSRG-03-03, Clemson University, Clemson, SC. 2003. http://www.cs.clemson.edu/~resolve/reports/RSRG-03-03.pdf

[13] Leavens, G. T., Cheon, Y., Clifton, C., Ruby, C., and Cok. D. R. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, vol. 55, pages 185-205, Elsevier, 2005.

[14] Leino, K. R. M. Data groups: specifying the modification of extended state. In *Proceedings OOPSLA '98*, pages 144–153, 1998.

[15] Leino, K. R. M., Nelson, G., and Saxe, J. B. *ESC/Java User's Manual*, Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[16] Meyers, S. *More Effective C++*. Addison-Wesley, 1995.

[17] Møller, A. and Schwartzbach, M. I. The pointer assertion logic engine. In *ACM SIGPLAN Notices*, 36(5), pages 221–231, May 2001.

[18] Müller, P. and Poetzsch-Heffter, A. Modular specification and verification techniques for object-oriented software components. In *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, editors, Cambridge University Press, Cambridge, United Kingdom, 2000.

[19] Noble, J., Vitek, J., and Potter, J. Flexible alias protection. *ECOOP '98*. Lecture Notes in Computer Science, vol. 1445, pp. 158–185, 1998.

[20] O'Hearn, P., Reynolds, J., and Yang, H. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, 2142:1–19, 2001.

[21] Pike, S. M., Weide, B. W., and Hollingsworth, J. E. Checkmate: concerning C++ dynamic memory errors with checked pointers. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*. ACM Press, March 2000.

[22] Sagiv, M., Reps, T., and Wilhelm, R. Parametric shape analysis via 3-valued logic. *In ACM Transactions on Programming Languages and Systems*, 24(3), pp. 217–298, 2002.

[23] Sitaraman, M. and Weide, B.W. Component-based software using RESOLVE. *ACM Software Engineering Notes*, 19(4), pp. 21–67, 1994.

[24] Sitaraman, M. Impact of performance considerations on formal specification design. *Formal Aspects of Computing*, 8(6):716–736, 1996.

[25] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W. Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E. Reasoning about software-component behavior. In *Proceedings of the 6th International Conference on Software Reuse*, pages 266–283. Springer-Verlag, 2000.

[26] Sitaraman, M., Gandi, D. P., Kuechlin, W., Sinz, C., and Weide, B. W. DEET for Component-Based Software, *In Proceedings FSE Workshop on Specification and Verification of Component-Based Systems*, October 2004.

[27] Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, January 1996.

[28] Vaziri, M. and Jackson, D. Checking heap-manipulating procedures with a constraint solver. *TACAS '03*, Warsaw, Poland, 2003.

[29] Weide, B.W., and Heym, W.D. Specification and verification with references. In *Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems*, October 2001.

[30] Wilhelm, R., Sagiv, M., and Reps, T. Shape analysis. *In Proceedings of the 2000 International Conference on Compiler Construction*, Berlin, Germany, April 2000.

[31] Wing, J. M. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.

# Dream Types

## A Domain Specific Type System for Component-Based Message-Oriented Middleware

Philippe Bidinger, Matthieu Leclercq, Vivien Quéma, Alan Schmitt, Jean-Bernard Stefani

Projet Sardes, INRIA Rhône-Alpes

## ABSTRACT

We present a type system for the Dream component-based message-oriented middleware. This type system aims at preventing the erroneous use of messages, such as the access of missing content. To this end, we adapt to our setting a type system developed for extensible records.

## 1. INTRODUCTION

Component-based frameworks have emerged in the past two decades. They are commonly used to build various software systems, including Web applications (EJB [1], CCM [14]), middleware (dynamicTAO [11], OpenORB [4]), or even operating systems (OSKit [10], THINK [9]).

A typical example of such frameworks is Dream [12]. Dream allows the construction of message-oriented middleware and builds upon the Fractal component model [5] and its Java implementation. It provides a library of components that that can be assembled using the Fractal architecture description language (ADL) and that can be used to implement various communication paradigms, such as message queues, event/reaction, publish/subscribe, etc.

A system built out of Dream components typically comprises several components which may exchange *messages*, which may modify them (*e.g.*, setting a time stamp), and which may behave differently according to their contents (*e.g.*, routing a message). In the current Java implementation of the Dream framework, every message has type `Message`, independently of its contents. As a consequence, certain assemblages of Dream components type-check and compile correctly in Java but lead to run-time failures, typically when a component processes a message that does not have the proper expected structure.

Catching such configurations errors early on, when writing the architecture description of a Dream assemblage, would be of tremendous benefits to programmers using the Dream framework. In other words, what would be required would be a type-safe ADL that would allow the typing of component structures and reject ill-typed component configurations.

As a first step towards this goal, we propose in this paper a type system for Dream components, concentrating on message types that accurately describe the internal structure of a message. To this end, we adapt existing work on type systems for *extensible records* [16, 17] and describe how components and component assemblages may be typed. The resulting type system captures a number of errors that can be made when writing ADL descriptions of Dream configurations.

The paper is structured as follows: Section 2 describes the Dream framework, and typical configuration errors the type system is intended to capture. Section 3 introduces types for messages and for components manipulating messages. Section 4 describes related work, and Section 5 concludes the paper.



## 2. THE DREAM FRAMEWORK

### 2.1 The Fractal component model

Dream is based on the Fractal component model [5], a component model for Java. Fractal distinguishes between two kinds of components: *primitive* components and *composite* components. The latter provide a means to deal with a group of components as a whole.

A component has one or more ports that correspond to access points supporting a finite set of methods. Ports can be of two kinds: server ports, which correspond to access points accepting incoming method calls, and client ports, which correspond to access points supporting outgoing method calls. The signatures of both kinds of ports is described by a standard Java interface declaration, with an additional role indication (server or client).

A component is made of two parts: the *content part* is either a standard Java class (in the case of a primitive component), or a set of sub-components (in the case of a composite component); the *controller part* comprises interceptors and controllers. Examples of controllers are the *binding controller* that allows binding and unbinding the component's client ports to server ports of other components, and the *lifecycle controller* that allows starting/stopping components.

Figure 1 illustrates the different constructs in a typical Fractal component architecture. Thick gray boxes denote the controller part of a component, while the interior of these boxes correspond to the content part of a component. Arrows correspond to bindings, and tee-like structures protruding from gray boxes are ports.

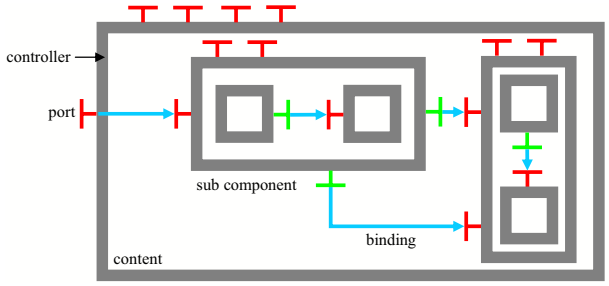



**Figure 1: Architecture of a Fractal component**

## 2.2 The Dream framework

Dream components are standard Fractal components with one characteristic feature: the presence of input/output interfaces that allow Dream components to exchange *messages*. Messages are Java objects that encapsulate named *chunks*. Each chunk implements an interface that defines its type. As an example, messages that need to be causally ordered have a chunk that implements the `Causal` interface. This interface defines methods to set and get a matrix clock.

Messages are always sent from outputs to inputs (Figure 2 (a)). There are two kinds of output and input interfaces, corresponding to the two kinds of connections: *push* and *pull*. The push connection corresponds to message exchanges initiated by the output port (Figure 2 (b)). The pull interaction corresponds to message exchanges initiated by the input port (Figure 2 (c)).

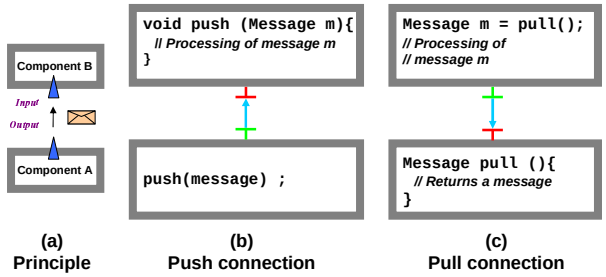



**Figure 2: Input/output interfaces connection**

Dream provides a library of components encapsulating functions and behaviors commonly found in message-oriented middlewares. These components can be assembled to implement various asynchronous communication paradigms: message passing, publish/subscribe, event/reaction, etc. Here are a few examples of Dream components:

- *message queues* are used to store messages. Several kinds exist, differing by the way messages are sorted: FIFO, LIFO, causal order, etc.
- *transformers* have one input to receive messages and one output to deliver transformed messages. Typical transformers include stampers.
- *routers* have one input and several outputs (also called "routes"), and route messages received on their input to one or several routes.
- *multiplexers* have several inputs and one output; for every message received on an input a multiplexer adds a chunk that identifies the input on which the message arrived; the multiplexer then forwards the message to the output.
- *duplicators* have one input and several outputs, and copy messages they receive on their input to all their outputs.
- *channels* allow message exchanges between different address spaces. Channels are distributed composite components that encapsulate, at least, two components: a *ChannelOut*—whose role is to send messages to another address space—, and a *ChannelIn*—which can receive messages sent by the ChannelOut.

## 2.3 Configuration errors

The main data structures manipulated by Dream components are *messages*. A message is a finite set of named *chunk*. A chunk can be any Java object. Basic operations over messages allow to read, remove, add, or update a chunk of a given name. They can potentially lead to three kinds of run-time errors.

- A chunk is absent when it should be present (*e.g.*, for a read, remove, or update).
- A chunk is present when it should be absent (*e.g.*, for an add).
- A chunk does not have the expected type (*e.g.*, for a read).

Experience with the dream framework has shown that many such errors are consequences of an erroneous architecture definition of the system. For instance, in figure 3, the architecture definition is obviously incorrect: component `readTS` expects messages with a `TS` chunk, whereas component `addTS` expects messages without `TS` chunk. Since both components receive exactly the same messages (duplicated by the `duplicate` component), one of them will fail.

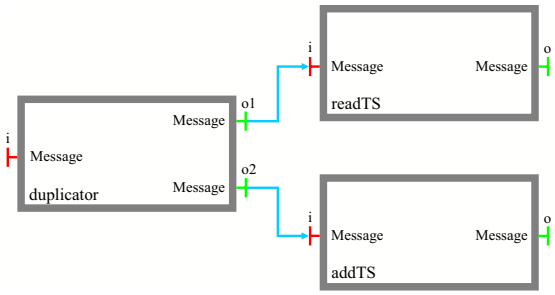



**Figure 3: Example**

One can tell that the architecture definition of 3 is incorrect because the behavior of the components is clear from their name. However, the typing annotations are clearly insufficient to allow the previous analysis.

In the current component model of Dream, connections between components are constrained by the host language (*e.g.* Java) type system. Ports are associated to Java interface types, and two ports can be connected if and only if their corresponding Java types coincide. This scheme suffers from limitations of the Java type system, in particular, the absence of polymorphism and rich record types.

We propose to define a polymorphic type system for the composition of components in order to overcome those limitations. It allows the specification of the more common behaviors of Dream components, seen as messages transformers. It provides the guarantees that, if components conform individually to their type, the composed system will not fail with any of the run-time errors identified above.

# 3. DREAM TYPES

## 3.1 Presentation

A record is a finite set of associations, called *fields*, between labels and values. Many languages, such as Ocaml, use records as primitive values. In [16, 17] Rémy describes an extension of ML where all common operations on records are supported. In particular, the addition or removal of fields and the concatenation of records. He then defines a static type system that guarantee that the resulting programs will not produce run-time errors, such accessing a missing field.

Dream messages can be seen as records, where each chunk correspond to a field of the record, and Dream components can be seen as polymorphic functions. Polymorphism is important for at least two reasons. First, the same component can be used in different contexts with different types. Second, polymorphism allows to relate the types of the client and server interfaces, and thus allows to specify more precisely the behavior of a component. We can almost directly use the results of [16, 17] in order to type Dream components. Note however that we work on a different level of abstraction: we give types to components and check that the way we connect them is coherent. In particular, we do not type-check the code of the components.

In the following, we first give the main ideas behind messages types and component types, and present the main formal results in the next subsection.

We type messages as extensible records [16]. Informally, The type of a message consists of a list of pairwise distinct labels together with the type of the corresponding value, or a special tag if the message does not contain a given label. Moreover, a final information specifies the content of the (infinitely many) remaining labels. In addition, we use a convenient type constructor `ser`: if $\tau$ is an arbitrary type, $\texttt{ser}(\tau)$ is the type of values of type $\tau$ in a serialized form.

Figure 4 defines several examples of message types.

$$\mu_1 = \{a : \texttt{pre}(\texttt{A}); b : \texttt{pre}(\texttt{B}); \texttt{abs}\}$$
$$\mu_2 = \{a : \texttt{pre}(\texttt{A}); b : \texttt{pre}(\texttt{B}); c : \texttt{abs}; \texttt{abs}\}$$
$$\mu_3 = \{a : \texttt{pre}(X); \texttt{abs}\}$$
$$\mu_4 = \{a : Y; \texttt{abs}\}$$
$$\mu_5 = \{a : \texttt{pre}(\texttt{A}); Z\}$$
$$\mu_6 = \{a : \texttt{pre}(\texttt{A}); b : Z'; Z''\}$$
$$\mu_7 = \{a : \texttt{pre}(\texttt{A}); a : \texttt{pre}(\texttt{B}); \texttt{abs}\}$$
$$\mu_8 = \{a : X; b : \texttt{abs}; X\}$$

**Figure 4: Examples of message types**

A message $m$ of type $\mu_1$ contains exactly two labels $a$ and $b$, associated to values of type `A` and `B` respectively (the importance of the `pre` constructor will be made clear later). It does not contain any other label, as specified by the `abs` tag. We can note that $m$ can equivalently be seen as a value of type $\mu_2$. Indeed, $\mu_1$ and $\mu_2$ represent the same sets of values, which we write $\mu_1 = \mu_2$. Richer types can be constructed using *type variables*. In type $\mu_3$, $X$ represents an arbitrary type. Informally, a message of type $\mu_3$ must contain a label $a$, but the type of the associated value is not specified: the `pre` constructor allows us to impose the presence of a given field, even if its type is unspecified. Similarly, in $\mu_4$, $Y$ is a *field* variable. It can be either `abs`, $\texttt{pre}(A)$ for any type $A$, or $\texttt{pre}(X)$ for any type variable $X$. Finally, in $\mu_5$, $Z$ is a *row* variable that represent either `abs` or any list of fields. Note that we have $\mu_5 = \mu_6$. Remark also that some syntactically correct types, such as $\mu_7$ and $\mu_8$, can be meaningless: in particular labels must not occur twice, and a variable cannot have two different sorts (here $X$ is used both as a field variable with label $a$, and as a row variable).

A component has a set of *server ports* and *client ports*. Each port is characterized by its name, and the type of the values it can carry. The type of a component is essentially a polymorphic function type. Figure 5 gives examples of components and component types. *id* has a polymorphic type. Its client and server ports can be used with any type $X$. *dup* duplicates its arguments. $add_a$ adds a new field with label $a$ to the messages it receives on client port $i$. Note that these messages must not contain label $a$. $remove_a$ removes the field named $a$, that may or may not be present. *reset* reset the value associated to label $a$ to some initial value. *serialize* gets an arbitrary message $\{X\}$ and returns a new message with one field which is the serialized form of $\{X\}$. *deserialize* is the converse operation.

$$id : \forall X.\{i : \{X\}\} \to \{o : \{X\}\}$$
$$dup : \forall X.\{i : \{X\}\} \to \{o_1 : \{X\}; o_2 : \{X\}\}$$
$$add_a : \forall X.\{i : \{a : \texttt{abs}; X\}\} \to \{o : \{a : \texttt{pre}(\texttt{A}); X\}\}$$
$$remove_a : \forall X, Y.\{i : \{a : Y; X\}\} \to \{o : \{a : \texttt{abs}; X\}\}$$
$$reset : \forall X.\{i : \{a : \texttt{pre}(\texttt{A}); X\}\} \to \{o : \{a : \texttt{pre}(\texttt{A}); X\}\}$$
$$serialize : \forall X.\{i : \{X\}\} \to \{o : \{s : \texttt{ser}(\{X\}); \texttt{abs}\}\}$$
$$deserialize : \forall X.\{i : \{s : \texttt{ser}(\{X\}); \texttt{abs}\}\} \to \{o : \{X\}\}$$

**Figure 5: Examples of component types**

As for message types, some component types are meaningless. Consider the following type:

$$\forall X.\{i : \{a : \texttt{pre}(X); \texttt{abs}\}\} \to \{o : \{a : X\}\}$$

The two occurrences of $X$ are used with a different meaning. The first one is a type variable whereas the second one is a field variable. In a more subtle way, the following type is incorrect:

$$\forall X.\{i : \{X\}\} \to \{o : \{a : \texttt{pre}(\texttt{A}); X\}\}$$

Both occurrences of $X$ are row variable. However, the first one includes rows that may contain a field with label $a$, whereas the second does not.

Figure 6 depicts the same architecture definition as in figure 3, using these more precise types. The definition will be well-typed if and only if we can solve the equations:

$$\{X\} = \{ts : \texttt{pre}(\texttt{A}); Y\}$$
$$\{X\} = \{ts : \texttt{abs}; Z\}$$

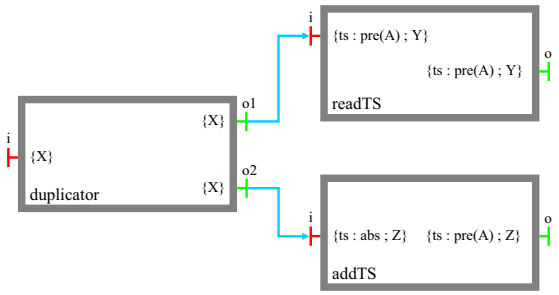



**Figure 6: Example 1 revisited**

Remark that we have chosen different type variables for each component. The equations do not have any solution, and thus the system is not well typed.

## 3.2 Formally

### 3.2.1 Syntax

We first introduce the syntax of messages types, which are very similar to record types.

$$\begin{array}{rcll} \tau & ::= & \mu \mid \mathtt{ser}(\tau) \mid \sigma_B \mid \alpha & \text{types} \\ \mu & ::= & \{\rho^{\emptyset}\} & \text{message types} \\ \rho^L & ::= & \xi^L \mid \mathtt{abs}^L \mid a : \phi; \rho^{L \uplus \{a\}} & \text{row} \\ \phi & ::= & \theta \mid \mathtt{abs} \mid \mathtt{pre}(\tau) & \text{fields} \\ \sigma_B & ::= & \mathtt{A} \mid \mathtt{B} \mid \ldots & \text{base types} \end{array}$$

A type $\tau$ may either be a message type $\mu$, a serialized type $\mathtt{ser}(\tau)$, a base type $\sigma_B$, or a type variable $\alpha$. A message type $\mu$ is a record type, described by a row $\rho^{\emptyset}$. We suppose that $a, b, c, \ldots$ range over a denumerable set of message labels $\mathsf{L}_m$ and $L$ over finite subsets of $\mathsf{L}_m$. Intuitively, a row $\rho^L$ must not contain any field whose label is in $\mathsf{L}_m$. In the case of a message type, there is no restriction as to which labels may occurs, hence the $\emptyset$ superscript.

A row $\rho$ may either be a row variable $\xi$, the empty row $\mathtt{abs}$, or the concatenation of a field $a : \phi$ with a row where label $a$ does not occur. This restriction is enforced by the use of the $L$ superscript. For instance $\{a : \theta; (a : \theta'; \xi^L)\}$ is not syntactically correct. The $\uplus$ operator denotes disjoint union, it is only defined for disjoint sets.

The presence information $\phi$ is either a field variable $\theta$, the indication that the field is absent $\mathtt{abs}$, or that it is present and carries a value of type $\tau$, denoted $\mathtt{pre}(\tau)$.

Finally, $\sigma_B$ range over base types, corresponding to Java types in Dream. We often write $\{a : \phi; b : \phi'; \xi^L\}$ for $\{a : \phi; (b : \phi'; \xi^L)\}$.

We next give the syntax of component types.

$$\begin{array}{rcll} C & ::= & \forall \widetilde{\alpha} \widetilde{\theta} \widetilde{\xi^L}. \{I^{\emptyset}\} \to \{I^{\emptyset}\} & \text{Component} \\ I & ::= & i : \mu; I \mid \emptyset & \text{Interface Set} \end{array}$$

We define $\mathsf{L}_p$ as a denumerable set of *ports*, or interface names, ranged over by $i$,$o$ and their decorated variants. A component type is composed of a set of *input* interfaces and a set of *output* interfaces. An interface consists of a port and the type of values exchanged on this port. We suppose ports to be distinct in a given interface set (input or output). We write $\widetilde{x}$ for a finite set of variables, and require that every (type, field, or row) variable be bound in the $\forall$ prefix of the component type.

In the previous subsection, we used the same syntactic category for type, row, and field variables and we omitted the superscripts on rows. The reason is that the sorts of variables and the superscripts can be automatically inferred. For instance, the type $\forall X.\{i : \{X\}\} \to \{o : \{a : \mathtt{pre}(\mathtt{A}); X\}\}$ is incorrect because it cannot be rewritten as $\forall \xi^L.\{i : \{\xi^L\}\} \to \{o : \{a : \mathtt{pre}(int); \xi^L\}\}$: $L$ should be $\emptyset$ in the first occurrence of $\xi$ and $\{a\}$ in the second one.

An *architecture definition* $D$ is given by a list of component names and their type, and a list of connections between ports. For both syntactic categories, we let $\epsilon$ denote an empty list (of components or connections). We let $c$ and its decorated variants range over $\mathsf{L}_c$, a denumerable set of *component names*.

$$\begin{array}{rcll} D & ::= & (Cp, Co) & \text{Architecture Definition} \\ Cp & ::= & \epsilon \mid c : C, Cp & \text{Components} \\ Co & ::= & \epsilon \mid c.o = c'.i, Co & \text{Connections} \end{array}$$

An architecture definition $(Cp, Co)$ is well-formed if

- Component names in $Cp$ are pairwise distinct.
- For every connection $c.o = c'.i$ in $Co$, $c : C$ and $c' : C'$ are in $Cp$ for some $C$, $C'$. Moreover, $o$ is a client port (i.e., it is a port of the output set of interfaces) of $C$ and $i$ a server port of $C'$.
- Any port is connected at most once.

### 3.2.2 Typing

We write $\mathsf{T}$ the set of rows $\rho^L$ for all $L$. We define an equational theory $E$ on $\mathsf{T}$ with the following axioms and rule.

$$\begin{array}{c} a : \phi; (a' : \phi'; \rho^L) = a' : \phi'; (a : \phi; \rho^L) \\ a : \phi; \mathtt{abs}^L = a : \phi; (b : \mathtt{abs}; \mathtt{abs}^{L \uplus \{b\}}) \\ a : \phi; \xi^L = a : \phi; (b : \theta; \xi'^{L \uplus \{b\}}) \\ \rho^L = \rho'^L \implies a : \phi; \rho^{L \uplus \{a\}} = a : \phi; \rho'^{L \uplus \{a\}} \end{array}$$

The first axiom states that the order in the definition of the fields does not matter. The second states that the $\mathtt{abs}$ row denotes rows containing only absent fields. The third axiom states that a row variable denotes rows with fields whose presence information is not specified.

We know from [16] that the problem of unification in $\mathsf{T}$ modulo $E$ is decidable and syntactic: every solvable unification problem has a most general unifier.

From an architecture definition $D = (Cp, Co)$, we can generate a set of equations $E(D)$. First we get a list $Cp'$ by suppressing all quantifiers in $Cp$, assuming variables are first renamed such that no variable appear in two distinct types. We write $Cp'(c)$ for the type of component $c$ in $Cp'$. For a component type $C$, we note $T_C(C.o)$ for the type associated with client interface $o$. We define similarly $T_S(C.i)$.Using these definitions, we can define $E$ as follows:

$$E(Cp, Co) = \{T_C(Cp'(c).o) = T_S(Cp'(c).i) \mid c.o = c.i \in Co\}$$

An architecture definition $D$ is typable if and only if $E$ admits an unifier.

### 3.3 Example

Figure 7 (a) depicts a stack of dream components. The component *producer* at the top of the left stack generates messages consisting of a unique chunk of type `TestChunk` and name *tc*.

$$producer : \{\} \rightarrow \{o : \{tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}\}\}$$

The component *serializer* returns messages with a unique chunk *sc* that is the serialized form of the messages received on input port *i*.

$$serializer : \forall X.\{i : \{X\}\} \rightarrow \{o : \{sc : \mathtt{ser}(\{X\}); \mathtt{abs}\}\}$$

Component *addIP* adds a chunk of type `IPChunk` and name *ipc* to a message that does not contain an *ipc* chunk.

$$addIP : \forall X.\{i : \{ipc : \mathtt{abs}; X\}\} \rightarrow \{o : \{ipc : \mathtt{pre}(\mathtt{IPChunk}); X\}\}$$

*channelOut* dispatches messages on the network, and requires them to define at least an *ipc* chunk of type `IPChunk`.

$$channelOut : \forall X.\{i : \{ipc : \mathtt{pre}(\mathtt{IPChunk}); X\}\} \rightarrow \{o : \{ipc : \mathtt{pre}(\mathtt{IPChunk}); X\}\}$$

The right stack performs the symmetric actions. Figures 7 (b) and (c) show two incorrect architectures. In (b), the deserializer component is missing and in (c) the deserializer and addIP components are inverted. Architecture (a) is well-typed but (b) and (c) are not. Consider architecture (b), we deduce the following equations from the linking (note that bound variables have been renamed).

$$\{tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}\} = \{U\} \quad (1)$$
$$\{sc : \mathtt{pre}(\mathtt{ser}(U)); \mathtt{abs}\} = \{ipc : \mathtt{abs}; Z\} \quad (2)$$
$$\{ipc : \mathtt{pre}(\mathtt{IPChunk}); T\} = \{ipc : \mathtt{pre}(\mathtt{IPChunk}); Z\} \quad (3)$$
$$\{ipc : \mathtt{pre}(\mathtt{IPChunk}); Z\} = \{Y\} \quad (4)$$
$$\{Y\} = \{ipc : \mathtt{pre}(\mathtt{IPChunk}); X\} \quad (5)$$
$$\{ipc : \mathtt{abs}; X\} = \{tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}\} \quad (6)$$

From 6, we deduce that

$$X = \{tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}\}$$

Then from 5, we have

$$Y = \{ipc : \mathtt{pre}(\mathtt{IPChunk}); tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}\}$$

It follows from 4 and 3 that

$$T = Z = \{tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}\}$$

Besides, we deduce from 2 that

$$Z = \{sc : \mathtt{pre}(\mathtt{ser}(U)); \mathtt{abs}\}$$

The terms $tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}$ and $sc : \mathtt{pre}(\mathtt{ser}(U)); \mathtt{abs}$ are obviously not unifiable and thus the system is not typable.

We implemented this type system in Ocaml. It takes as input an architecture definition, checks that it is well-sorted, generates a system of equations and try to solve it. We used this tool to check the validity of several assemblages. Figure 8 corresponds to the input file for architecture (c).

Our prototype fails to solve the equations corresponding to this architecture. The output corresponds to a set of equations equivalent to the initial system, when the unification algorithm encounters a contradictory equation (*e.g.*`abs` = `IPChunk`).

```
producer:
  {}->{o:{tc:pre(TestChunk);abs}}
consumer:
  {i:{tc:pre(TestChunk);abs}}->{}
serializer:
  {i:{'x}}->{o:{s:pre(ser({'x}));abs}}
deserializer:
  {i:{s:pre(ser({'x}));abs}}->{o:{'x}}
addIP:
  {i:{ipc:abs;'x}}->{o:{ipc:pre(IPChunk);'x}}
removeIP:
  {i:{ipc:pre(IPChunk);'x}}->{o:{ipc:abs;'x}}
channelOut:
  {i:{ipc:pre(IPChunk);'x}}->{o:{ipc:pre(IPChunk);'x}}
channelIn:
  {i:{'x}}->{o:{'x}}
composite c is
  {}->{}
with
  producer.o = serializer.i
  serializer.o = addIP.i
  addIP.o = channelOut.i
  channelOut.o = channelIn.i
  channelIn.o = deserializer.i
  deserializer.o = removeIP.i
  removeIP.o = consumer.i
end
```

**Figure 8: file `archC.d`**

```
% dtype archC.d
No solution
-----------
abs = IPChunk
abs = abs
{tc:TestChunk;abs} = {ipc:IPChunk;'removeIP_x}
{ipc:abs;'removeIP_x} = {tc:TestChunk;abs}
```

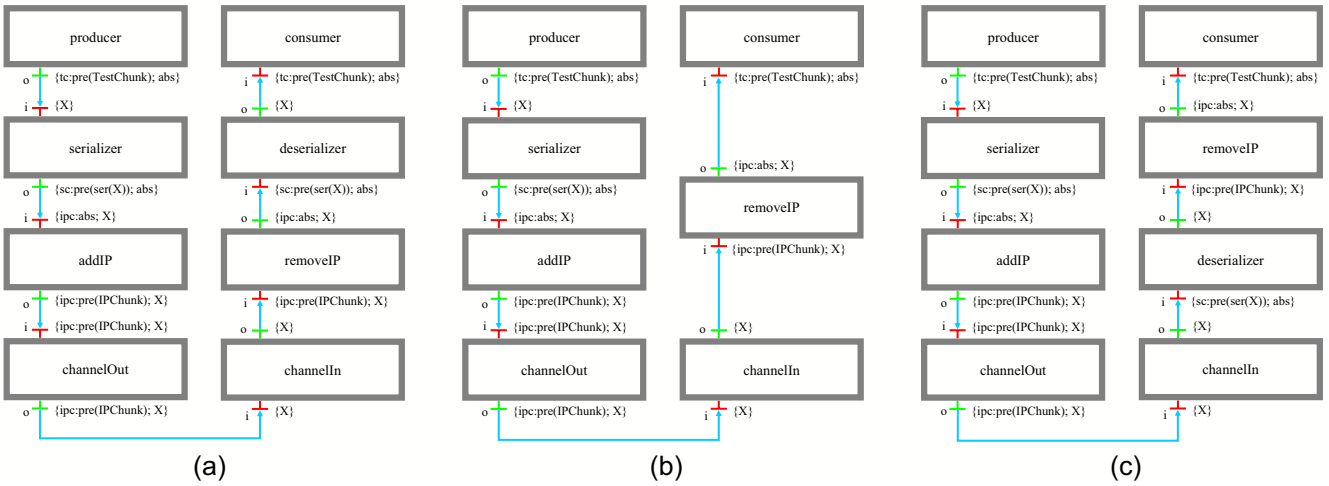



**Figure 7: Example: a stack of components**

## 3.4 Discussion and limitations

The main limitation is that this typing discipline is too restrictive to type certain Dream components. Typically, they can exhibit different behavior depending on the presence of a given label in a message (*e.g.*, routers). Consider for instance a component `route` that routes messages it gets on its client port on different server port depending on the presence of a given label. We would like its type to be something like:

$$\texttt{route} : \forall X.\ \{i : \{a : \texttt{pre(A)}; X\}\} \rightarrow \{o_1 : \{X\}; o_2 : \{\texttt{abs}\}\}$$
$$\wedge \quad \{i : \{a : \texttt{abs}; X\}\} \rightarrow \{o_1 : \{\texttt{abs}\}; o_2 : \{X\}\}$$

Similarly, some components may output messages of different types.

$$\texttt{produce} : \{\} \rightarrow \{o : \{a : \texttt{abs}; b : \texttt{B}; \texttt{abs}\}\}$$
$$\wedge\, \{\} \rightarrow \{o : \{a : \texttt{A}; b : \texttt{abs}; \texttt{abs}\}\}$$

In both cases, we can find approximating types that allow us to type a definition involving these components. For instance:

$$\texttt{route} : \forall XYZ.\quad \{i : \{X\}\} \rightarrow \{o_1 : \{Y\}; o_2 : \{Z\}\}$$
$$\texttt{produce} : \forall XY.\qquad\quad \{\} \rightarrow \{o : \{a : X; b : Y; \texttt{abs}\}\}$$

In doing so, we lose any guarantee about the correctness of the architecture definition, since obviously, the code of the components does not conform to these types.

# 4. RELATED WORK

The type system presented in this paper constitutes an example of a domain specific type system, tailored to checking architectural constraints in the component-based Dream environment. Type systems that capture various properties of programs have of course been intensively studied for various languages, including ML, Java, as well as in more abstract settings such as process algebras and the $\pi$-calculus [18]. Exploiting type systems for checking architectural constraints has received less attention, but has nevertheless been the subject of various works in the past decade. We can mention for instance work on the Wright language [3], which supports the verification of behavioral compatibility constraints in a software architecture, matching a component with a role; recent work on ArchJava [2], which uses ownership types to enforce communication integrity in a Java-based component model; and more recent work on behavioral contracts for component assembly [6]. The type systems (or compatibility relations) used in these works, however, do not capture the architectural constraints that are dealt with in this paper. Both the Wright system and the behavioral contract system would need to be extended to deal with the record types that characterize Dream messages, and the ArchJava type system is tailored to enforce communication integrity, *i.e.* , to prevent aliasing that may destroy a component integrity. The work which is closest to ours is probably the recent work on the type system for the Ptolemy II system [13], which combines a rich set of data types, including structured types such as (immutable) arrays and records, and a behavioral type system which extends the work on interface automata [7, 8] for capturing temporal aspects of component interfaces. The Ptolemy II type system would not be directly applicable to our Dream constraints, though, for it features only immutable record types. However, a combination of extensive record types as in this paper and behavioral types of the Ptolemy II system is definitely worth investigating.

# 5. CONCLUSION AND FUTURE WORK

We have presented a domain specific type system for messages and components that manage messages in the Dream framework. This type system is based on existing work on extensible records, and is rich enough to address component assemblages such as protocol stacks, as illustrated in Section 3.3.

An obvious shortcoming of our approach is that we do not formally state the guarantees provided by the type system, namely that there will be no run-time error due to the access of an absent message chunk, the addition of a chunk whose name is already present in the message, or the use of a chunk's contents at a wrong type. We have taken the more pragmatic approach of first implementing and testing the expressivity of the type system. We plan on formalizing the behavior of Dream components and state the guarantees

of our type system as continuation of this work.

Experimenting with our type system has shown that it is not precise enough when the behavior of a component depends on the structure of a message, as described in Section 3.4. To address this issue, we plan on adapting existing works on intersection types, such as [15], to our setting.

Finally, we are studying the integration of our type checking phase in the Dream ADL processing workflow.

## 6. REFERENCES


[1] Enterprise JavaBeansTM Specification, Version 2.1, August 2002. Sun Microsystems, `http://java.sun.com/products/ejb/`.

[2] J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In *Proceedings ECOOP 2002, LNCS 2548*. Springer, 2002.

[3] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. In *ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 3*, pages 213–249, July 1997.

[4] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, , and K. Saikoski. The Design and Implementation of Open ORB v2. In *IEEE Distributed Systems Online Journal, vol. 2 no. 6*, November 2001.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An Open Component Model and its Support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2004)*, Edinburgh, Scotland, 2004.

[6] C. Carrez, A. Fantechi, and E. Najm. Behavioural contracts for a sound assembly of components. In *FORTE*, volume 2767 of *Lecture Notes in Computer Science*. Springer, 2003.

[7] L. de Alfaro and T. Henzinger. Interface Automata. In *Proceedings of the joint 8th European software engineering conference and 9th ACM SIGSOFT international symposium on the foundations of software engineering (ESEC/FSE 01)*, 2001.

[8] L. de Alfaro and T. Henzinger. Interface Thoeries for Component-Based Design. In *Proceedings of EMSOFT '01*, volume 2211 of *Lecture Notes in Computer Science*. Springer, 2001.

[9] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A Software Framework for Component-based Operating System Kernels. In *USENIX Annual Technical Conference*, 2002.

[10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *SOSP'97*, 1997.

[11] F. Kon, T. Yamane, K. Hess, R. H. Campbell, and M. D. Mickunas. Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio, USA, January 2001.

[12] M. Leclercq, V. Quéma, and J.-B. Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Reconfigurable MOMs. In *Proceedings of the 3rd Workshop on Reflective and Adaptive Middleware (RM'2004)*, Toronto, Canada, October 2004.

[13] E. Lee and Y. Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 16(3), 2004.

[14] P. Merle, editor. *CORBA 3.0 New Components Chapters*. OMG TC Document ptc/2001-11-03, November 2001.

[15] B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

[16] D. Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[17] D. Rémy. Typing record concatenation for free. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[18] D. Sangiorgi and D. Walker. *The $\pi$-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

# Non-null References by Default in the Java Modeling Language

Patrice Chalin, Frédéric Rioux

Dept. of Computer Science and Software Engineering,
Dependable Software Research Group, Concordia University
1455 de Maisonneuve Blvd. West, Montréal
Québec, Canada, H3G 1M8
{chalin,f_rioux}@cse.concordia.ca

## ABSTRACT

Based on our experiences and those of our peers, we hypothesized that in Java code, the majority of declarations that are of reference types are meant to be non-null. Unfortunately, the Java Modeling Language (JML), like most interface specification and object-oriented programming languages, assumes that such declarations are possibly-null by default. As a consequence, developers need to write specifications that are more verbose than necessary in order to accurately document their module interfaces. In practice, this results in module interfaces being left incompletely and inaccurately specified. In this paper we present the results of a study that confirms our hypothesis. Hence, we propose an adaptation to JML that preserves its language design goals and that allows developers to specify that declarations of reference types are to be interpreted as non-null by default. We explain how this default is safer and results in less writing on the part of specifiers than null-by-default. The paper also reports on an implementation of the proposal in some of the JML tools.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—programming by contract; D.3.3 [Programming Languages]; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

## General Terms

Documentation, Design, Languages, Theory, Verification.

## Keywords

Contracts, Java Modeling Language, JML, reference types, non-null references.

## 1. INTRODUCTION

Null pointer exceptions are among the most common faults raised by components written in mainstream imperative languages like Java. Increasingly developers are able to make use of tools that can detect possible null dereferences (among other things) by means of static analysis of component source. Unfortunately, such tools can only perform minimal analysis when provided with code alone. On the other hand, given that components and their support libraries are supplemented with appropriate specifications, then the tools are able to detect a large proportion of potential null pointer dereferences. The Java Modeling Language (JML) is one of the most popular behavioral interface specification languages for Java [11, 12]. ESC/Java2 is an extended static checker for Java that uses JML as an interface specification language [4].

While writing Java programs and their JML specifications, it has been our experience, and those of peers, that we generally want declarations of reference types to be non-null. Unfortunately JML, like most interface specification and mainstream object-oriented programming languages, assumes that by default declarations can be null. As a result, specifiers must explicitly constrain such declarations to be non-null either by annotating the declarations with `/*@ non_null @*/` or by adding constraints of the form `o != null` to class invariants and/or method contracts. Since most developers tend to write specifications penuriously, in practice this results in module interfaces being left incompletely and inaccurately specified.

In this paper we present the results of a study that confirms the hypothesis that:

> In Java programs, the majority of declarations that are of reference types are meant to be non-null, based on design intent.

For this study we sampled over 150 KLOC out of 450 KLOC of Java source. To our knowledge, this is the first formal empirical study of this kind—though anecdotal evidence has been mentioned elsewhere, e.g. [7, 8]. In light of our study results, we propose that JML be adapted to allow developers to specify that declarations of reference types are to be interpreted as *non-null* by default.

The study method and study results are presented in the next two sections. Our proposal to adapt JML to support non-null by default is presented in Section 4 along with a discussion of the way in which the proposal upholds JML's design goals. We offer a discussion of related work and conclude in Sections 5 and 6 respectively.


*SAVCBS'05*, September 5–6, 2005, Lisbon, Portugal.

```
public abstract class Greeting
{
 private /*@ spec_public non_null */ String nm;

 /*@ public normal_behavior
   @  requires !aNm.equals("");
   @  modifies nm;
   @  ensures nm == aNm;
   @*/
 public void set(/*@ non_null @*/ String aNm) {
   nm = aNm;
 }

 //@ ensures \result.equals(greeting()+nm);
 public /*@ pure non_null @*/ String welcome() {
   return greeting() + nm;
 }

 //@ ensures \result != null;
 //@      && !\result.equals("");
 public abstract /*@ pure @*/ String greeting();

}
```

**(a) `Greeting` class**

```
public class FrenchGreeting extends Greeting
{
 // constructor omitted

 //@ also
 //@ ensures \result.equals("Bonjour ");
 public /*@pure non_null*/ String greeting() {
   return "Bonjour ";
 }
}
```

**(b) `FrenchGreeting` class**

**Figure 1. Sample class specifications**

# 2. STUDY

## 2.1 Metrics

Reference types can be used in the declaration of local variables, fields, method (return types) and parameters. In our study we considered all of these types of declaration except for local variables since the non-null annotation of local variables is not yet fully supported in JML. Unless specified otherwise, we shall use the term *declaration* in the remainder of this article to be a declaration other than that of a local variable.

We have two principal metrics in this study, both of which shall be measured on a per file basis:

- $d$, is a measure of the number of declarations that are of a reference type, and
- $m$ is a measure of the number of declarations specified to be non-null (hence $m \leq d$).

The main statistic of interest, $x$, will be a measure of the proportion of reference type declarations that are non-null, i.e. $m$ / $d$. In the next section we explain how $m$ is computed.

## 2.2 Counting non-null declarations

### *2.2.1 JML core and syntactic sugar*

Like many languages, the definition of JML consists of a core (that offers basic syntactic constructs) supplemented with syntactic sugar that makes the language more practical and pleasant to use. As is often done in these situations, the semantics of JML is defined in terms of a desugaring procedure (that describes how to translate arbitrary specifications into the JML core), and a semantics of the core [13, 17].

As such, it is much simpler to accurately describe how to count non-null declarations relative to the core JML. Unfortunately, such an account would seem foreign to most. Hence, in this paper we have chosen to provide an informal description of counting non-null declarations that is based on "sugared" JML. We refer readers interested in the full details to [5].

### *2.2.2 General rules*

Given a declaration *T* `o`, where *T* is a reference type, we can constrain `o` to be non-null either explicitly or implicitly. We do so explicitly by annotating the declaration with `/*@ non_null @*/` as is illustrated in Figure 1(a). Notice that the field `nm`, the method `welcome()` and the parameter `aNm` of the method `set()` are explicitly declared non-null.

Generally speaking, we consider that `o` is implicitly constrained to be non-null if an *appropriate* assertion is of any one of the following forms, or it contains a conjunct of any one of the following forms:

- `o != null`,
- `o instanceof C`,
- `\fresh(o)` which states that `o` is a reference to a newly allocated object (hence this can only be used in an `ensures` clause), or
- `\nonnullelements(o)`, when `o` is an array reference.

For example, the `greeting()` method of Figure 1(a) is considered to be implicitly declared non-null because the `ensures` clause constrains the method result to be non-null. Next, for each kind of declaration, we describe the circumstances under which we consider declarations of the given kind to be implicitly declared non-null.

### *2.2.3 Fields*

A non-static (respectively, static) field can be implicitly declared non-null if a non-static (static) class invariant contains a conjunct of the form given in Section 2.2.2. For example, given the declarations of Figure 2, we would count `o1`, `o2` and `a` as non-null but not `o3` (because the `o3 != null` term is an argument to a disjunction rather than a conjunction).

```
/*@ non_null @*/ Object o1;
Object o2;
//@ invariant o2 != null;
int i;
Object o3;
//@ invariant i > 0 || o3 != null;
Object a[];
//@ invariant \nonnullelements(a);
```

**Figure 2. Sample field declarations, some non-null**

```
/*@  normal_behavior
  @    requires i == 0
  @    ensures  \result != null
  @          && \result.equals("zero");
  @ also
  @  normal_behavior
  @    requires i > 0;
  @    ensures  \result != null
  @          && \result.equals("positive");
  @ also
  @  exceptional_behavior
  @    requires i < 0;
  @    signals(Exception e) true;
  @*/
/*@ pure @*/ String m(int i) {
   ...
}
```

**Figure 3. Method specification cases separated using 'also'**

#### *2.2.4 Method return types*

The pseudo variable `\result` is used in `ensures` clauses to represent the value returned by a method. A static method or a non-overriding non-static method can be implicitly declared as non-null by constraining `\result` to be non-null in an `ensures` clause as is done for the `greeting()` method of the `Greeting` class—Figure 1(a).

A JML method specification can be given as a list of cases (having different preconditions) separated by the keyword `also` as is illustrated in Figure 3. In such situations, the method can be counted as non-null if and only if: the method is explicitly declared as non-null or, it is implicitly declared as non-null in *every* `normal_behavior` specification case (and *every* `behavior` specification case—not discussed here—for which the `ensures` clause is neither false nor `\not_specified`).

Due to behavioral subtyping, the case of overriding methods is slightly more complicated. In JML, an overriding method like `FrenchGreeting.greeting()` of Figure 1(b) must respect the method specifications of its ancestors—which in this case consists only of one method, `Greeting.greeting()`. As a reminder to readers, all overriding method specifications must start with the keyword `also`. Thus, an overriding method *m* in a class *C* can be counted as non-null if and only if *m* is constrained to be non-null in *C*, as well as in all ancestor classes of *C* where *m* is explicitly declared.

#### *2.2.5 Method parameters*

The case for method parameters is similar to that for method return types. That is, a parameter of a static or non-overriding method is considered non-null if it is constrained as such in a `requires` clause. On the other hand, a parameter of an overriding method can be counted as non-null if and only if it is declared as non-null in the given class and all ancestor classes.

## 2.3 Statistics tool

In order to gather statistics concerning non-null declarations, the Iowa State University (ISU) JML checker was extended. The tool uses heuristics similar to those described in the previous section. The metrics gathered are conservative (i.e. when given the choice between soundness and completeness, the tool opts for soundness). The tool gathers data for normal, ghost, and model references. It also warns the user of inconsistent specifications (e.g. pre- or post-conditions trivially simplifying to false in all method specification cases).

| Overall Project → | ISU Tools | ESC Tools | SoenEA | Koa | Total |
|---|---|---|---|---|---|
| # of files | 831 | 455 | 52 | 459 | 1797 |
| LOC (K) | 243 | 124 | 3 | 87 | 457 |
| SLOC (K) | 140 | 75 | 2 | 62 | 278 |
| **Project subsys. →** | **JML Checker** | **ESC/ Java2** | **SoenEA** | **Koa Tally Subsys.** | **Total** |
| # of files | 217 | 216 | 52 | 29 | 514 |
| LOC (K) | 86 | 63 | 3 | 10 | 161 |
| SLOC (K) | 58 | 41 | 2 | 4 | 104 |

**Table 1 General statistics of study subjects**

## 2.4 Case study subjects

It was actually our work on an ESC/Java2 case study in the specification and verification of a small web-based enterprise application framework named SoenEA [18] that provided the final impetus to initiate the study reported in this paper. Hence, we chose SoenEA as one of our case study subjects. As our three other subjects we chose the ISU JML checker, the ESC/Java2 tool and the tallying subsystem of Koa, a recently developed Dutch internet voting application[1]. We chose these projects because:

- We believe that they are representative of typical designs in Java applications and that they are of a non-trivial size (numbers will be given shortly).
- We were familiar with the source code (and/or had peers that were) and hence expected that it would be easier to write accurate JML specifications for it. Too much effort would have been required to study and understand unfamiliar and sizeable projects in sufficient detail to be able to write correct specifications[2].
- The project sources are freely available to be reviewed by others who may want to validate our specification efforts.
- The sources were at least partly annotated with JML specifications; hence we would not be starting entirely from scratch.

Aside from SoenEA, the other study subjects are actually an integral (and dependant) part of a larger project. For example, the JML checker is only one of the tools provided as part of the ISU tool suite—others include JmlUnit and the JML run-time assertion checker compiler.

Table 1 provides the number of files, lines-of-code (LOC) and source-lines-of-code (SLOC) for our study subjects as well as the projects that they are subcomponents of. Overall the source for all four projects consists of 457 KLOC (278 KSLOC) from over almost 1800 Java source files. Our study subjects account for 161 KLOC from over 500 files.

[1] Koa was used, e.g., in the 2004 European parliamentary elections.
[2] Particularly since projects often lack detailed design documentation.

## 2.5 Procedure

### 2.5.1 Selection of sample files

With the study projects identified, our objective was to add JML specifications to all of the source files, or, if there were too many, a randomly chosen sample of files. In the later case, we fixed our sample size at 35 (as sample sizes of 30 or more are generally considered "sufficiently large"). Our random sampling for a given project was created by first listing the $N$ project files in alphabetical order, generating 35 random numbers in the range 1..$N$, and then choosing the corresponding files.

### 2.5.2 Annotating the sample files

We then added to the selected files JML specifications consisting essentially of constraints on declarations of reference types, where appropriate. In most situations we added `non_null` declaration modifiers.

An example of a field declaration that we would constrain to be non-null is:

```
static final String MSG1 = "abc";
```

Similarly we would conservatively annotate constructor and method return types as well as parameters based on our understanding of the software applications. As an example, consider the following method:

```
String m(int a[]) {
  String result = "";
  for(int i = 0; i < a.length; i++) {
    result += a[i] + " ";
  }
  return result;
}
```

In the absence of any specification or documentation for such a method we would assume that the designer intended `a` to be non-null (since there is no test for nullity and yet the `length` field of `a` is used). We can also deduce that the method will always return a non-null String.

As was previously explained, constraining the method return type or parameters for an inherited method requires adding annotations to the method's class as well as to all ancestors of the class in which the method is declared. This was particularly evident in the case of the JML checker code since the class hierarchy is up to 6 levels of inheritance for some of files that we worked on (e.g. `JmlCompilationUnit`).

## 2.6 Threats to validity

### 2.6.1 Internal validity

We see two threats to internal validity. Firstly, in adding non-null constraints to the sample files we may have been excessive. As was discussed in the previous section, we chose to be conservative in our specification exercise. Furthermore, the code samples (both before the exercise and after) are available for peer review. The JML checker is accessible from SourceForge (sourceforge.net); ESC/Java2 and Koa are available from Joseph Kiniry's GForge site (sort.ucd.ie) and SoenEA is available from the authors[3].

Secondly, our statistics tool may have incorrectly counted a declaration as being non-null. Again, as was previously explained, we chose soundness over completeness during our design of the tool. The tool source (which is part of the ISU JML tool suite) is also available via anonymous CVS for peer review.

[3] At the time of writing, the updated Koa source has not yet been committed to GForge; it is available from the authors.

| | JML Checker | ESC/ Java2 | Soen-EA | Koa TS | Sum or Average |
|---|---|---|---|---|---|
| $n$ | 35 | 35 | 41 | 29 | 140 |
| $N$ | 217 | 216 | 41 | 29 | 503 |
| $\sum d_i$ | 376 | 807 | 231 | 564 | 1978 |
| $\sum m_i$ | 210 | 499 | 177 | 368 | 1254 |
| $\sum d_i / \sum m_i$ | 56% | 62% | 77% | 65% | 63% |
| mean ($\underline{x}$) | 59% | 60% | 72% | 64% | 64% |
| std.dev.($s$) | 0.24 | 0.31 | 0.37 | 0.32 | - |
| $E$ ($\alpha$=5%) | 7.4% | 9.3% | - | - | - |
| $\mu_{min}$ | 52% | 51% | 72% | 64% | 60% |

**Table 2. Distribution of the number of declarations of reference types**

### 2.6.2 External validity

Will we be able to draw general conclusions from our study results? The main question is: can our sample of source files be taken as representative of typical Java applications? There are two aspects that can be considered here: the design style used in the samples, and the application domains.

Modern object-oriented programming best-practices promote e.g., a disciplined (i.e. moderate) use of `null` with the Null Object pattern recommended as an alternative [9]. Of course, not all Java code is written following recommended best practices; hence our sample applications should include such "non-OO-style" code. This is the case for some of the ESC/Java2 core classes (which were designed quite early in the project history); e.g. some of the classes declare their fields as public (a practice that is discouraged) rather than using getters and setters, making it more difficult to ascertain if a field was intended to be non-null. Also, the class hierarchy is very flat, with some classes resembling a module in the traditional sense (i.e. a collection of static methods) more than a class.

With a four sample set, it is impossible to claim that we have coverage in application domains, but we note that the SoenEA sample represents one of the most popular uses of Java—namely, servlet-based web applications.

# 3. STUDY RESULTS

A summary of the output of the non-null statistics tool run on our study samples (after having completed our specification exercise) is given in Table 2. As is usually done, the number of files in each sample is denoted by $n$ and the population size by $N$. Note that for SoenEA, 11 of the files did not contain any declarations of reference types, hence the population size is 41 = 52 – 11; the reason that we exclude such files from our sample is because it is not possible to compute the proportion of non-null references for files without any declarations of reference types. We see that the total number of declarations that are of a reference type ($d$) across all samples is 1978. The total number of such declarations constrained to be non-null ($m$) is 1254. The proportion of non-null references across all files is 63%.

We also computed the mean, $\underline{x}$, of the proportion of non-null declarations on a per file basis ($x_i = d_i / m_i$). The mean ranges from 59% for the JML checker sample, to 72% for the SoenEA

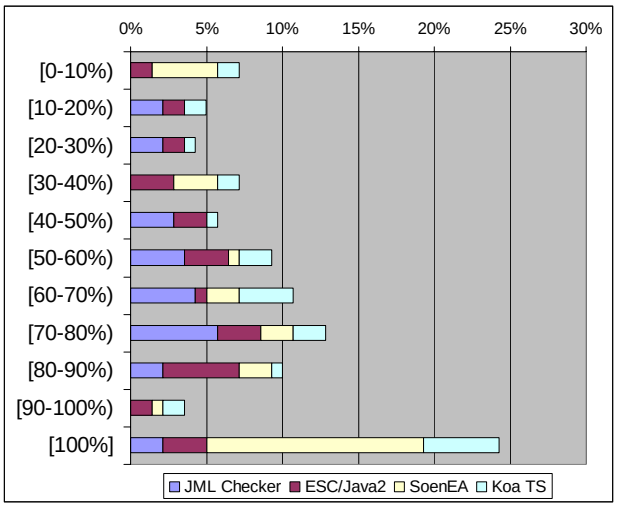



**Figure 4. Percentage of files having a value for *x* (the proportion of non-null declarations) in a given range**

sample. Also given are the standard deviation ($s$) and a measure of the maximum error ($E$) of our sample mean as an estimate for the population mean with a confidence level of $1 - \alpha = 95\%$. Hence we can conclude with 95% certainty that the population means are above 50% in all cases.

It should be noted that for both the JML checker and ESC/Java2 samples, we stopped annotating the files once we had reached a value for $\mu_{min}$ (i.e. $\mu - E$) that was greater than 50% for $\alpha = 5\%$. Hence, it is quite likely that $\mu$ is actually higher for these samples. In the case of SoenEA we essentially completed the annotation exercise for all files, and as a result $\mu$ is 72%.

A distribution of $x$, the proportion of non-null declarations, is given in Figure 4—following standard notation, $[a,b)$ represents the interval of values $v$ in the range $a \leq v < b$. The bar length represents the percentage of files for which $x$ is in the given range. We see that the checker has no files with an $x$ in the range [0-10%). On the other hand, SoenEA has the largest proportion of files in this range as well as for $x = 100\%$.

The mean of $x$ by kind of declaration (fields, methods and parameters) for each of the study samples is given in Figure 5. Almost all samples have a mean for parameters that is higher than

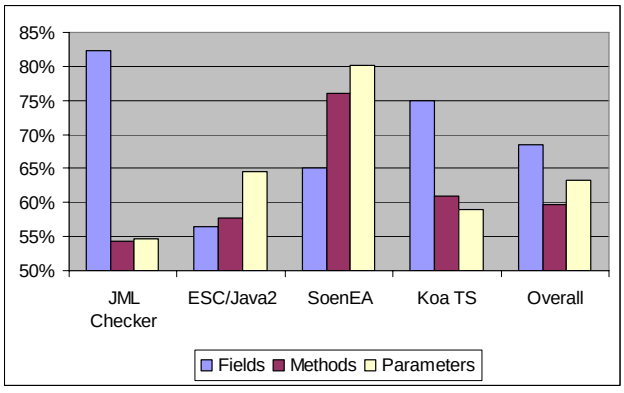



**Figure 5. Mean of *x*, the proportion of non-null declarations, by kind**

for methods. The mean of $x$ for fields is much higher in the case of the JML checker possibly because the checker sample had the smallest number of field declarations.

We believe that the study results support our hypothesis that in Java code, the majority of declarations that are of reference types are meant to be non-null. It is for this reason that we propose a modification to JML as is explained next.

# 4. ADAPTING JML

Motivated by the study results, we propose that JML be adapted to support

- the module[4] modifier `non_null_ref_by_default` that will allow developers to indicate that reference type declarations in the given module are to be interpreted as non-null by default,
- a `null` declaration modifier, to be used in the context of `non_null_ref_by_default` classes, indicating that a given declaration can be null, and
- a `null_ref_by_default` module modifier.

An example of the use of these modifiers is given in Figure 6. We justify our proposal in the following subsections.

## 4.1 Null vs. non-null by default

The study results support the hypothesis that, in general, designers want more than 50% of declarations of reference types to be non-null. Thus, under the current JML semantics, designers must effectively add `/*@ non_null @*/` annotations to the majority of declarations if he or she wants the non-null constraints to be accurately documented. As was remarked in the introduction, since developers tend to write specifications penuriously, in practice this results in module interfaces being left incompletely and inaccurately specified. Thus, module clients might call methods with null arguments when these should be prohibited, resulting in `NullPointerException`'s—one of the most common programming errors.

It would seem more sensible for declarations to be non-null by default. Adopting this default would allow, on average, over 50% of declarations of reference types in an unannotated source file to be accurately constrained to be non-null. Designers could then gradually add `/*@ null @*/` annotations. A consequence of forgetting or delaying the addition of `null` annotations would, at worst, present a more restrictive interface to a module's clients. This is a safer alternative than null-by-default. Furthermore, since developers must generally provide special code to handle null, it is best for them to be explicitly informed that a value might be null by the presence of an annotation rather than by its absence.

## 4.2 Upholding JML design goals

One of the language design goals of JML is to adhere to the semantics of Java to the extent possible. In those situations where JML semantics differ from Java, it should not come as a surprise to Java developers [14]. In our case, since Java assumes that references are possibly-null by default, it would not be appropriate to simply propose that JML's default be non-null. Instead, there should be an explicit indication in a JML module specification that unadorned references are to be interpreted as non-null. (Of course, it would be preferable for Java to adopt non-null as a

[4] I.e., class or interface.

```
public abstract /*@ non_null_ref_by_default @*/
class Greeting
{
 private /*@ spec_public @*/ String nm;

 /*@ public normal_behavior
   @  requires !aNm.equals("");
   @  modifies nm;
   @  ensures nm == aNm;
   @*/
 public void set(String aNm) {
   nm = aNm;
 }

 //@ ensures \result.equals(greeting()+nm);
 public /*@ pure @*/ String welcome() {
   return greeting() + nm;
 }

 //@ ensures !\result.equals("");
 public abstract
  /*@ pure @*/ String greeting();
}
```

**Figure 6. `Greeting` specification using `non_null_ref_by_default`**

default, as will be done in the next release of Eiffel for example—see Section 5.2.2.)

Hence we propose the addition of a module modifier named `non_null_ref_by_default` that can be placed just ahead of the `class` or `interface` keywords. In this way, a completely unadorned Java module will have the same semantics as Java (with respect to the interpretation of declarations of reference types). An example of the use of this module modifier is given in Figure 6; it is a revised version of the `Greeting` class specification of Figure 1.

As was previously mentioned, we believe that it is safer to have declarations represent non-null references rather than possibly-null references by default. As a final enhancement we suggest that JML tools emit a warning if neither default has been *explicitly* specified for any given Java module[5]. In those situations where a developer wishes to preserve Java semantics, he or she can make use of the `null_ref_by_default` module modifier. Such a behavior will help developers who are new to JML remember to use the safer `non_null_ref_by_default` when appropriate (which should be most of the time).

## 4.3 Implementation

The given proposal has been implemented in the JML checker and the JML Run-time Assertion Checker. Since these tools already supported the notion of possibly-null and non-null declarations, the adaptation was relatively easy (approximately two person-weeks). We expect that the adaptation of other JML tools should be just as straight-forward. Joseph Kiniry and the first author are currently implementing the proposal in ESC/Java2.

# 5. RELATED WORK

## 5.1 Nullity annotations

Most closely related to our current proposal for JML are the nullity annotations supported by Splint [6]. Splint is a static analysis tool for C programs that evolved out of work on Lclint. Lclint was essentially a type checker for Larch/C, a behavioral interface specification language for C [10]. In Splint, all pointer variables are assumed to be non-null by default, unless adored with `@null`. Splint can be used to statically detect potential null dereferences.

[5] Of course, such warnings can be disabled using a command-line option.

```
class GenericGreeting : Greeting {
 string! greeting;
 public GenericGreeting(string! n, string! g) {
   base(n);
   // (1)
 }
 // ...
}
```

**(a) Partially initialized object (Spec#)**

```
class GenericGreeting : Greeting {
 string! greeting;
 public GenericGreeting(string! n, string! g) {
   greeting = g;
   base(n);
   // (1)
 }
 // ...
}
```

**(b) Use of initializer (Spec#)**

**Figure 7. Spec# `GenericGreeting` examples**

## 5.2 Non-null types

In contrast to using assertions or special annotations, some languages have enriched type systems supporting the notion of non-null types. Of the three described here, two are proposing that references be non-null by default.

### *5.2.1 Nice*

The Nice programming language can be seen as an enriched variant of Java supporting parametric types, multi-methods, and contracts among other features [2]. Nice also supports non-null types. By default, a reference type name *T* denotes non-null instances of *T*. To express the possibility that a declaration of type *T* might be null, one prefixes the type name with a question mark [3].

### *5.2.2 Eiffel*

The next major release of the Eiffel programming language [15] will also include support for *attached types* (i.e. non-null types, or non-void types as they might be called in Eiffel) as opposed to *detachable types* (that can be null) [16]. The proposed default for this new release of Eiffel will be attached types. Special consideration has been given to minimizing the migration effort of current Eiffel code.

### *5.2.3 Spec#*

Spec# is an extension of the C# programming language that adds support for contracts, checked exceptions and non-null types. The Spec# compiler statically enforces non-null types and generates run-time assertion checking code for contracts [1]. For reasons of backwards compatibility with C#, a reference type name *T* refers to possibly null references of type *T* whereas *T*! is used to represent non-null references of type *T*.

The introduction of non-null types naturally complicates the type system and leads to other issues. The main issue has to do with partially initialized objects [7]. Consider the example given in Figure 7(a). At point (1) in the constructor code, the field `greeting` will be null—due to the automatic initialization performed on instance fields. To solve this problem, Spec# allows constructors to provide field initializers as is illustrated in Figure 7(b).

In the case of JML, no such special measures are required since a non-null constraint on a field like `greeting` would be translated into a non-null constraint in a class invariant. Class invariants are not assumed to hold on entry to or during the execution of a constructor body. This is not to claim that JML's approach is better—especially given the open issues related to the treatment of invariants—but rather than it is an alternative approach.

## 6. CONCLUSION

In this paper, we report on a novel study of four open projects (totaling over 450 KLOC) taken from various domains of application. The study results support the hypothesis that, by design, the majority of reference type declarations are meant to be non-null in Java.

It would be preferable that Java be adapted so that declarations are interpreted as non-null by default (as will be the case, for example, in the next release of Eiffel). In the meantime, we have suggested an adaptation to JML that would allow specifications to be more concise by interpreting reference types as non-null unless explicitly annotated with the `null` declaration modifier. Our proposal results in a safer default since in the absence of declaration annotations, modules simply present stricter interfaces to their clients.

We have implemented our proposal in the JML checker and run-time assertion checker compiler. As future work, we intend to complete the implementation of the proposal in ESC/Java2, and to conduct further studies in an attempt to measure the effectiveness of our new proposed default for reference type declarations.

## ACKNOWLEDGMENTS

We are grateful to the anonymous referees for their helpful comments. The authors would like to thank Joseph Kiniry for providing access to, and assistance on the Koa source, and for discussions about Eiffel. We thank Kui Dai for implementing the proposed changes to JML in the checker and RAC, as well as Hao Xi for contributing to the non-null metrics tool. Research support was provided by NSERC of Canada (261573-03) and the Quebec FQRNT (100221).

## REFERENCES

[1] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview." In Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004), Marseille, France, LNCS, vol. 3362, 2004.

[2] D. Bonniot. *The Nice programming language*, http://nice.sourceforge.net/, June 2005.

[3] D. Bonniot. *Type safety in Nice: Why programs written in Nice have less bugs*, http://nice.sourceforge.net/safety.html, June 2005.

[4] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer (STTT)*, 2004.

[5] P. Chalin and F. Rioux, *Non-null References by Default in the Java Modeling Language*, Dependable Software Research Group, Concordia University, ENCS-CSE TR 2005-004. June, 2005.

[6] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, vol. 19, no. 1, pp. 42-51, Jan.-Feb., 2002.

[7] M. Fähndrich and K. R. M. Leino, "Declaring and checking non-null types in an object-oriented language," in Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications. OOPSLA'03: ACM Press, 2003, pp. 302-312.

[8] C. Flanagan and K. R. M. Leino, "Houdini, an Annotation Assistant for ESC/Java." In *Proceedings of the International Symposium of Formal Methods Europe*, Berlin, Germany, vol. 2021, pp. 500-517, 2001.

[9] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 1999.

[10] J. V. Guttag and J. J. Horning, *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[11] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: A Notation for Detailed Design," in *Behavioral Specifications of Businesses and Systems*, B. R. Haim Kilov, Ian Simmonds, Ed.: Kluwer, 1999, pp. 175-188.

[12] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs, "JML: notations and tools supporting detailed design in Java," in *OOPSLA 2000 Companion, Minneapolis, Minnesota*, 2000, pp. 105-106.

[13] G. T. Leavens, A. L. Baker, and C. Ruby, *Preliminary Design of JML: A Behavioral Interface Specification Language for Java*, Department of Computer Science, Iowa State University TR #98-06-rev27. April, 2005.

[14] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of JML accommodates both runtime assertion checking and formal verification," *Science of Computer Programming*, vol. 55, no. 1-3, pp. 185-208, 2005.

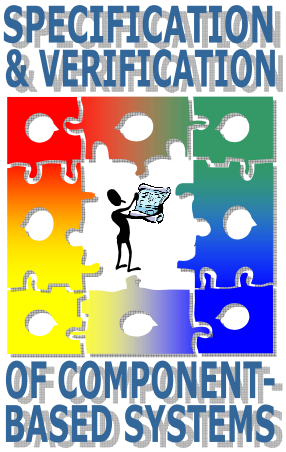
[15] B. Meyer, *Eiffel: The Language*. Object-Oriented Series. New York. Prentice-Hall, 1991.

[16] B. Meyer, *Eiffel: The Language*, Draft of future edition, revision 5.00 (June 2005) ed. Unpublished, 2005.

[17] A. D. Raghavan and G. T. Leavens, *Desugaring JML Method Specifications*, Department of Computer Science, Iowa State University TR #00-03e. May, 2005.

[18] F. Rioux and P. Chalin, "Improving the Quality of Web-based Enterprise Applications with Extended Static Checking: A Case Study." In *Proceedings of the 1st International Workshop on Automated Specification and Verification of Web Sites*, Valencia, Spain, Electronic Notes in Theoretical Computer Science, March 14-15, 2005 (to appear).

# SAVCBS 2005
# POSTER ABSTRACTS



# Constraint satisfaction techniques for diagnosing errors in Design by Contract software*

Rafael Ceballos
Dpto. Lenguajes y Sistemas Informáticos, U. de Sevilla
Avda. Reina Mercedes s/n, CP 41012
Seville, Spain

Rafael Martínez Gasca
Dpto. Lenguajes y Sistemas Informáticos, U. de Sevilla
Avda. Reina Mercedes s/n, CP 41012
Seville, Spain

Diana Borrego
Dpto. Lenguajes y Sistemas Informáticos, U. de Sevilla
Avda. Reina Mercedes s/n, CP 41012
Seville, Spain

## ABSTRACT

Design by Contract enables the development of more reliable and robust software applications. In this paper, a methodology that diagnoses errors in software is proposed. This is based on the combination of Design by Contract, Model-based Diagnosis and Constraint Programming. Contracts are specified by using assertions. These assertions together with an abstraction of the source code are transformed into constraints. The methodology detects if the contracts are consistent, and if there are incompatibilities between contracts and source code. The process is automatic and is based on constraint programming.

## Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Assertion checkers, Correctness proofs, Programming by contract; D.2.5 [**Testing and Debugging**]: Diagnostics; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Assertions Invariants Pre- and post-conditions

## General Terms

Verification, Design by Contract, Diagnosis

## 1. INTRODUCTION

Design by Contract was proposed in [5]. This work specified that the major component of quality in software is the ability to perform its job according to the specification. The software quality is especially important in Object-Oriented (OO) methodology because of the software reusability. In a recent work, [1] it is showed that contracts are useful for fault isolation. By using contracts, the fault isolation and diagnosability is significantly improved.

*This work has been funded by the Ministry of Science and Technology of Spain (DPI2003-07146-C02-01) and the European Regional Development Fund (ERDF/ FEDER).

In this work, a methodology for diagnosing software is proposed, that is, for detecting and locating faults in programs. The main idea is the transformation of the contracts and source code into an abstract model based on constraints. The methodology detects if the contracts are consistent, and if there are incompatibilities between contracts and source code.

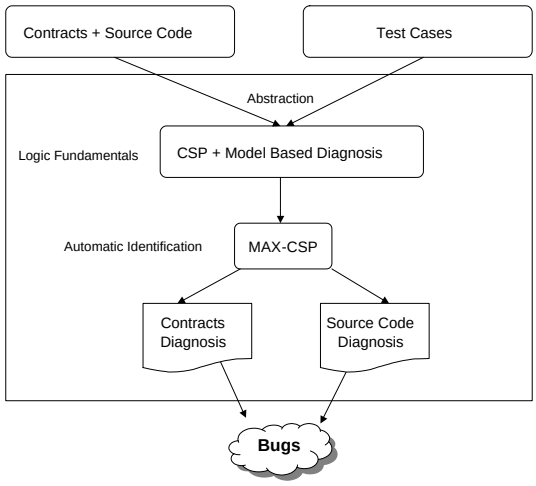



**Figure 1: Diagnosis framework**

## 2. DIAGNOSIS FRAMEWORK

Figure 1 shows the completed diagnosis process. The process obtains an abstract model based on the source code, contracts and test cases. The diagnosis of a program is a set of infeasible assertion and/or erroneous statements. These definitions specify the kind of errors that can be detected.

**Definition 1.** An *infeasible assertion* is a non-viable assertion due to conflicts with previous assertions or statements. The set of assertions of a contract are verified when a program is executed. An infeasible assertion is a wrongly designed assertion that cannot be satisfied and it stops the program execution when it is not specified.

**Definition 2.** An *erroneous statement* is a statement or a set of statements that are faults since they do not allow the

```java
/**
 *  @inv getBalance()>= 0
 *  @inv getInterest >= 0
 */

public interface Account {

    /**
     *  @pre income > 0
     *  @post getBalance() >= 0
     */
    public void deposit (double income);

    /**
     *  @pre withdrawal > 0
     *  @post getBalance() ==
     *      getBalance()@pre - withdrawal
     */
    public void withdraw (double withdrawal);

    /**
     *  @pre interest >= 0
     *  @post getInterest() == interest
     */
    public void setInterest (double interest);

    public double getInterest ();

    public double getBalance ();

}
```

```java
public class AccountImp implements Account {

    private double interest;

    private double balance;

    public AccountImp()  {
       this.balance = 0;
    }

    public void deposit (double income) {
       this.balance = this.balance - income;
    }

    public void withdraw (double withdrawal) {
       this.balance = this.balance - withdrawal;
    }

    public double getBalance() {
       return this.balance;
    }

    public double getInterest() {
       return this.interest;
    }

    public void setInterest(double interest) {
       this.interest = interest;
    }

}
```

**Figure 2: Interface *Account* and class *AccountImp* source code**

correct results to be obtained. The errors under consideration are minor variations of the correct program, such as errors in loop conditions or errors in assignment statements. This paper does not consider errors detected in compilation time (such as syntax errors), nor dynamic errors (such as exceptions, memory access violations, infinite loops, etc).

The following sections explain the phases of the diagnosis process.

# 3. ABSTRACT MODEL GENERATION

In model-based diagnosis approaches, a model of the system components enables detecting, identifying and isolating the reason of an unexpected behaviour of a system. In a Object-Oriented program the methods of different objects are linked in order to obtain the specified behaviour. Each method of a object can be considered as a component, which generates a specified result. The pretreatment of the source code and program contracts enables obtaining an abstract model of a program. This abstract model allows to diagnose errors in programs. The following subsections shows the process for generating abstract model.

## 3.1 Determining basic blocks

Every OO program is a set of classes. In order to automate the diagnosis of a program it is necessary to divide the system into subsystems. Each program classes is transformed into a set of basic blocks. These basic blocks (BB) can be: blocks of invariants (IB), blocks of static class fields (SB), blocks of object attributes (OB), and blocks of object or class methods (MB). A IB includes the set of invariants of a class. A SB includes the set of static field declarations and static code blocks of a class, and a OB includes the set of object field declarations of a class. A MB is the set of all the statements and assertions (such as preconditions, postconditions or loop invariants) included in the method.

Each program class can be transformed into a set of basic blocks ($BB_s$) equivalent to the $Class_i$. When a program is executed, the microprocessor links the basic blocks. The order of these blocks can be represented as a *Control Flow Graph* (CFG). The CFG is a directed graph that represents the control structure of a program. A CFG is a set of sequential blocks and decision statements. A *Path* is the sequence of statements of the CFG that is executed. The following sections will use the basic blocks of an executed path in order to obtain the constrains of the abstract model.

## 3.2 SSA form

The order of the constraints is not important for solving a CSP. But when a program is executed the order of the assertions and statements is very important. It is necessary to maintain this order in the abstract model. The program under analysis is translated into a static single assignment (SSA) form. This step maintains the execution sequence when the program is translated into constraints. In SSA form, only one assignment is made to each variable in the whole program. For example the code *x=a*c; ...x=x+3;... {Post:x =...}* is changed to *x1=a*c; ...x2=x1+3;... {Post:x2 =...}*.

## 3.3 Program transformation

The abstract model is a set of constraints which simulate the behaviour of the contracts (assertions) and the source code (statements) of a program. Our approach uses the function *A2C*(assertion to constraints) for transforming an assertion into constraints. The transformation of the source code to constraints appears in previous work [3]. The process is based on the transformation of each statement of the path. The main ideas of the transformation are:

- Indivisible blocks:

  *Assignments*: {*Ident := Exp*}
  The assignment statement is transformed into the fol-

**Table 1: The modified toy problem model**

| *PC*: | *PD*: |
|---|---|
| S1 : int x = a * c | (AB(S1) ∨ (x == a * c))∧ |
| S2 : int y = b * d | (AB(S2) ∨ (y == b * d))∧ |
| S3 : int z = c + e | (AB(S3) ∨ (z == c + e))∧ |
| S4 : int f = x + y | (AB(S4) ∨ (f == x + y))∧ |
| S5 : int g = y + z | (AB(S5) ∨ (g == y + z)) |
| *TC*: Inputs : | {a = 3, b = 2, c = 2, d = 3, e= 3} |
| Outputs : | {f = 12, g = 12} |
| Test Code: | S1 .. S5 |

lowing equality constraint: {Ident = Exp}. If the assignment statement is not a part of the minimal diagnosis then the equality between the assigned variable and the assigned expression must be satisfied.

*Method calls and return statements:* For each method call, the constraints defined in the precondition and postcondition of the method are added. If we find a recursive method call, this internal method call is supposed to be correct in order to obtain the formal verification of the recursive calls.

- Conditional blocks: {*if (cond)* {$If_{Block}$} *else* {$Else_{Block}$}}

  There are two possible paths in a conditional statement depending on the inputs of the condition. The constraints of a conditional statement include the condition and the inner statements of the selected path(only one of the two possible paths is executed).

- Loop blocks: {*while (cond)* {$Block_{Loop}$}}

  In a loop, the number of cycles depends on the inputs. Each cycle is transformed into a conditional statement. The structure of a loop is simulated as a set of nested conditional statements. If the invariant of the loop exists, the diagnosis process is more precise.

### 3.4 Test cases

Testing techniques enables the selection of those observations which are the most significant for detecting bugs in a program. A *Test case* (TC) is a set of inputs (class fields, parameters or variables), execution preconditions, and expected outcomes, which are developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. The values of a test case must satisfy the DbC specification. In our approach when a program is executed by using a test case, the information of the executed basic blocks is stored. This information is necessary for the diagnosis of the system, since it contains which are the statements of the executed path.

## 4. DIAGNOSIS PROBLEM

A diagnosis is a hypothesis for how a program must change in order to obtain a correct behavior. The definition of diagnosis, in Model Based Diagnosis (MDB), is built up from the notion of abnormal [4]: AB($c$) is a boolean variable which holds when a component $c$ of the system is abnormal. For example, an adder component is not abnormal if the output of the adder is the sum of its inputs. A diagnosis specifies whether each component of a system is abnormal or not. In order to clarify the diagnosis process, some definitions must be established.

**Definition 3.** *System model*(SM) is a tuple {PC, PD, TC} where: PC are the program components, that is, the finite set of statements and asserts of a program; PD is the program description, that is, the set of constraints (abstract model, AM) obtained of the PC, PD = AM(PC); and TC is a test case.

**Definition 4.** *Diagnosis*: Let $\mathcal{D} \subseteq$ PC, $\mathcal{D}$ is a diagnosis if PD' ∪ TC is satisfiable, where PD' = PD(PC − $\mathcal{D}$).

The goal of diagnosis is to identify, and refine, the set of diagnoses consistent with the test case.

**Definition 5.** *Minimal Diagnosis* is a diagnosis $\mathcal{D}$ that for no proper subset $\mathcal{D}$' of $\mathcal{D}$ is $\mathcal{D}$' a diagnosis. The minimal diagnoses imply to modify the smallest number of program statements or assertions.

The rules described in the section 3 enables implementing the function AM(Abstract Model) which generates the PD of a program. Table 1 shows the PD of the toy problem program, this is derived from the standard toy problem used in the diagnosis community [4]. The program can not reach the correct output because the third statement is a adder instead of a multiplier.

In order to obtain the minimal diagnosis it is generated a Maximal Constraint Satisfaction Problem (Max-CSP). A Max-CSP is a CSP with a goal function. The objective is to find an assignment of the AB variables that satisfies the maximum number of the PD constraints: Goal Function = Max($N$ i : AB(i) = *false*). A constraints solver will generate the different solutions of the Max-CSP. The diagnosis process by using a Max-CSP is shown previous work [2]. For example, by using a Max-CSP, the minimal diagnoses in the toy program will be: {{S3}, {S5}, {S1, S2}, {S2, S4}}.

## 5. DIAGNOSING PROGRAMS

In order to clarify the methodology the class *AccountImp* is used. This class implements the interface *Account* that simulates a bank account. It is possible to deposit money and to withdraw money. Figure 2 shows the source code and contracts. The method *deposit* has a bug, in that it *decreases* the account balance. In the first phase, assertions are checked in two different ways: without test cases and with test cases. In the second phase the source code with assertions is checked by using test cases.

### 5.1 Diagnosis of assertions without test cases

Two kinds of checks are proposed at this point: 1) a Max-CSP with all the invariants of each class, in order to check if all the invariants of a class can be satisfied together; and 2) a Max-CSP with the assertions of the methods and the invariants, in order to check if the precondition and postcondition of a method is feasible with the invariants of a class. The solutions of these Max-CSP problems enable the verification of the feasibility of assertions.

### 5.2 Diagnosis of assertions by using test cases

It is possible to obtain more information about the viability of the method assertions by applying test cases to the sequence {invariants + precondition + postcondition + invariants } in each method.

**Table 2: Diagnosis of the method *Withdraw***

| | | |
|---|---|---|
| TC | Inputs: | {balance@pre = 0, withdrawal > 0} |
| | Outputs: | {balance = 0} |
| | Test code: | Method Withdraw |
| PD | Inv. | (AB(Inv) ∨ (balance@pre >= 0)) |
| | Pre. | (AB(Pre) ∨ (withdrawal > 0)) |
| | Post. | (AB(Post) ∨ (balance = balance@pre - withdrawal)) |
| | Inv. | (AB(Inv) ∨ (balance >= 0)) |

**Example.** Table 2 shows the PD for the method *withdraw* verification. The test case specified that the initial balance must be 0 units and, when a positive amount is withdrawn, the balance must preserve the value 0. The balance must be equal or greater than zero when the method finishes, but if this invariant is satisfied it implies that the precondition and the postcondition could not be satisfied together. The postcondition implies that *balance = balance@pre - withdrawal*, that is, $0 - withdrawal > 0$, and this is impossible if the *withdrawal* is positive. The problem resides in the precondition, since this precondition is not strong enough to stop the program execution when the withdrawal is not equal or greater than the balance of the account.

## 5.3 Diagnosis of source code and assertions

The diagnosis of a program is the set of those statements which include the errors. We are looking for the minimal diagnosis, that is, it is necessary, by using a Max-CSP, to maximize the number of satisfied constraints of the PD. The constraint obtained by the assertions must be satisfied, because these constraints give us information about the correct behaviour. The diagnosis process result depends on the the final situation of the program:

*Situation 1*: If the program ended up with a failed assertion, and did not reach the end as specified in the test case, the problem can be a strict assertion (the assertion is very restrictive) or one or more erroneous statements before the assertion. In order to determine the cause of the problem, the program should be executed again without the assertion, in order to deduce if the program can finish without the assertion. If this happens, the assertion is very strict. If the program does not finish, the problem is due to the code up to the point of the assertion.

*Situation 2*: If the program ends, but the result is not the one specified by the test case, then the problem can be a wrong statement, or an assertion which is not sufficiently restrictive (this enables executing statements which obtain an incorrect result). If the problem is a wrong statement, the resolution of the Max-CSP problem provides the minimal diagnosis that includes the bugs. If the problem is due to a weak assertion then a deeper study of the assertions is necessary.

**Example.** Table 3 shows an account with an initial balance of 300 units. Two sequential operations are applied: a withdrawal of capital of 300 units, and a deposit of the same quantity. The constraint solver determines that the error is caused by the statement included in the method *deposit*. If the method is examined closely, it can be seen that there is a subtraction instead of an addition. The postcondition of this method is too weak, and did not detect this problem.

**Table 3: Diagnosis of the class *AccountImp***

| | | |
|---|---|---|
| TC | Inputs: | {balance@pre = 300, withdrawal = 300, income = 300} |
| | Outputs: | {balance = 300} |
| | Test code: | S1: account.withdraw(withdrawal)<br>S2: account.deposit(income) |
| PD | Inv. | balance0 >= 0 |
| | Pre. | withdrawal > 0 |
| | Code | (AB(S1) ∨ (balance1 = balance0 - withdrawal)) |
| | Post. | balance1 = balance0 - withdrawal |
| | Inv. | balance1 >= 0 |
| | Inv. | balance1 >= 0 |
| | Pre. | income > 0 |
| | Code | (AB(S2) ∨ (balance2 = balance1 - income)) |
| | Post. | balance2 >= 0 |
| | Inv. | balance2 >= 0 |

# 6. CONCLUSION AND FUTURE WORK

In order to automate the diagnosis of software with contracts, the combination of techniques from different subjects is proposed, such as Constraint Programming, Model-Based Diagnosis, and Design by Contract. This paper is an improvement of previous work [3], and incorporates a more precise way to diagnose software since more characteristics of Design by Contract are incorporated. The methodology detects if the contracts are consistent, and if there are incompatibilities between contracts and source code. A more complex diagnosis process is being developed in order to obtain a more precise minimal diagnosis. We are extending the methodology to include all the characteristics of an Object-Oriented language, such as inheritance, exceptions and concurrence.

# 7. REFERENCES


[1] L. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to support fault isolation in object-oriented code. In *International Symposium on Software Testing and Analysis*, Roma, Italy, 2002.

[2] R. Ceballos, C. del Valle, M. T. Gómez-López, and R. M. Gasca. CSP aplicados a la diagnosis basada en modelos. *Revista Iberoamericana de Inteligencia Artificial*, 20:137–150, 2003.

[3] R. Ceballos, R. M. Gasca, C. D. Valle, and F. D. L. Rosa. A constraint programming approach for software diagnosis. In *AADEBUG*, pages 187–196, Ghent, Belgium, September 2003.

[4] J. de Kleer, A. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 2-3(56):197–222, 1992.

[5] B. Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

# Theory of Infinite Streams and Objects

## [Extended Abstract]

Konstantin Chekin
Institute of Theoretical Computer Science
Dresden University of Technology
D-01062 Dresden
tchekine@tcs.inf.tu-dresden.de

This paper provides a theory of infinite streams and objects, which contains our point of view on the problem of formal modelling of behaviors of objects and their systems with big or infinite number of internal states.

The most closed analogue of our theory are the theory of finite automata, FOCUS theory of M.Broy and K.Stolen [4], and Reo calculus introduced by F.Arbab and J.Rutten [3] together with its alternative semantics given by constraint finite automata in [2].

In FOCUS objects are described in terms of their input/output behaviors given by sets of stream processing functions. In the coalgebraic semantics of Reo objects are modelled by means of relations on timed data streams. The second semantics of Reo uses constrain finite automata to describe objects.

Theory of infinite streams and objects differs cardinally: 1) from the automata theory, FOCUS theory and Reo calculus so, that we use infinite streams for modelling of internal variables as well as for inputs and outputs of objects; 2) from FOCUS and Reo so, that we work with all involved streams concurrently and synchronously. In our theory we give up the representation of an object through its states and switch to the representation through the sets of all admissible behaviors of interface streams and internal streams constituting the object. This provides our theory of infinite streams and objects greater expressive power than theory of finite automata and moreover than push-down automata. The use of streams gives us uniform way for modelling of inputs, outputs and internal variables of objects, simplifies object's definitions, and renders possible our theory to be not afraid of the object's dimensionality increase. A side benefit of the suggested theory is presence of the algorithm for automatic obtaining of the formal model of a system from the given models of objects constituting the system and their topology in the system.

We use the theory of universal coalgebras [5] to describe semantics of our theory of infinite streams and objects. This paper provides detailed description of the formal syntax of objects, of their composition into more complex ones, and outlines corresponding coalgebraic semantics.


*SAVCBS* 2005 Lisbon, Portugal


## Keywords

Theory of objects and streams, systems of objects, set of admissible infinite behaviors, stateless representation

## 1. STREAMS

*Stream* is an infinite sequence of data (elements of the stream) of the certain type. Under notion *stream data type* we will understand a set $A$ of values, which can be taken on by elements of this stream. *The set $A$ may be finite or infinite.*

Infinite streams of elements from the fixed set $A$ are *modelled* by means of the final $F_A$-coalgebra of the functor $F_A : Set \to Set$ defined on sets $M$ by $F_A(M) = A \times M$:

$$(A^\omega; \langle head_{A^\omega}, tail_{A^\omega} \rangle)$$

where the function $\langle head_{A^\omega}, tail_{A^\omega} \rangle$ has form $A^\omega \to A^\omega$. The *carrier set* $A^\omega$ of the final $F_A$-coalgebra

$$A^\omega \stackrel{def}{=} \{ s \mid s : I\!N \to A \}$$

models the set of all streams of this type. It is formally defined as the set of all functions $s : I\!N \to A$ mapping natural numbers $I\!N$ into set $A$ of elements of the stream.

Since elements of the set $A^\omega$ are functions, we can define for every infinite stream $s \in A^\omega$ the operations, which return the first element of the stream $head_{A^\omega} : A^\omega \to A$ and the tail of the stream after removing its first element $tail_{A^\omega} : A^\omega \to A^\omega$, in the following way:

$$head_{A^\omega}(s) = s(0) \qquad tail_{A^\omega}(s) = \lambda x \in I\!N.\ s(x+1)$$

THEOREM 1. *For arbitrary data types $A_1, ..., A_r$ the set of all $r$-tuples of infinite streams $A_1^\omega \times ... \times A_r^\omega$ with head and tail operations executing concurrently on all streams is isomorphic to the set of infinite streams $(A_1 \times ... \times A_r)^\omega$ of $r$-tuples of the corresponding elements of the streams $A_1^\omega, ..., A_r^\omega$.*

We illustrate this theorem by the following example. Let us take two infinite streams: one is the stream of names in the lexicographical order $\alpha \in Lex^\omega$, $\alpha = a_1, b_1, c_1, d_1, e_1, f_1, ...$ and the other one is the stream consisting of natural numbers in the increasing order $\beta \in I\!N^\omega$, $\beta = 0, 1, 2, 3, 4, 5, ....$ The head of the first stream is the element $a_1$, i.e. $head(\alpha) = a_1$, and its tail is the infinite stream $tail(\alpha) = b_1, c_1, d_1, e_1, f_1, ...$

The pair of two streams $\alpha$ and $\beta$ according to the theorem and corollary above is isomorphic to the stream of pairs $\gamma \in (Lex \times I\!N)^\omega$, i.e. $\alpha \times \beta \cong \gamma$. The stream $\gamma = (a_1, 0), (b_1, 1), (c_1, 2), (d_1, 3), (e_1, 4), (f_1, 5)...$ consists of the pairs of the corresponding elements of the streams $\alpha$ and $\beta$.

Later on we will proceed from the requirements, that *names of all used streams and names of all used objects are unique.*

*Semantics* $[\![z]\!]$ of the stream $z : Z^\omega$ in the absence of additional restrictions is the set of all different streams of this type, i.e $[\![z]\!] \stackrel{def}{=} Z^\omega$, where $Z^\omega$ is the type of the stream with name $z$ and simultaneously the carrier set of the final $F_Z$-coalgebra of infinite streams.

*Set* $[\![z]\!]$ of all *admissible behaviors* of the stream $z$ is the set defined by the semantics of this stream. *Instance (variant) of stream behavior* or *behavior* $\alpha_z$ of the stream $z$ is the element from the set $[\![z]\!]$ of admissible behaviors of this stream, i.e. $\alpha_z \in [\![z]\!]$.

We formulate our rule of stream manipulations by the following proposition. *With all involved streams in every model, which is created by our theory, we will work concurrently, i.e. operations head and tail will be executed concurrently on all streams.*

## 2. OBJECT

In our formal theory a fundamental notion is the notion of an object. An object can have several different inputs (input streams), outputs (output streams) as well as several internal streams. Streams describe chronological succession of occurrences of values on the modelled inputs, outputs and internal variables of an object.

*Object* $A = \langle\ X\ ;\ S\ ;\ Y\ ;\ I\ ;\ \Phi \rangle$ consists of the set of names of input streams $X$, the set of names of internal streams $S$, the set of names of output streams $Y$, restriction $I$ on initial values of streams, and invariant property $\Phi$, which describes relation between streams of the object and between current and next values of these streams. The set $X \stackrel{def}{=} \{x_1 : X_1^\omega, ..., x_n : X_n^\omega\}$ is the finite set of names of input streams $x_1, ..., x_n$ and corresponding types $X_1^\omega, ..., X_n^\omega$ of these streams; the sets $S \stackrel{def}{=} \{s_1 : S_1^\omega, ..., s_m : S_m^\omega\}$ and $Y \stackrel{def}{=} \{y_1 : Y_1^\omega, ..., y_k : Y_k^\omega\}$ are the finite sets of names, correspondingly, of internal streams and of output streams.

In case of when the object has no inputs, or no outputs, or no internal streams, we will use the keyword *empty* to denote the absence of streams in the corresponding field in the object's quintuple.

*Syntax of the restriction $I$ on initial values* of streams of the object we define in the following way:

$$I ::= true \mid z(0) = val \mid \neg I \mid I_1 \wedge I_2 \mid I_1 \vee I_2$$

where *true* denotes the absence of any restrictions on initial values of streams; $z \in X \cup S \cup Y$ is one of the streams of the object, and *val* is a value from the set of values of elements of the stream $z$.

The formal *syntax of invariant property* we define as $\forall i \in I\!N.\ \Phi$, where $\Phi$ is a notion of invariant property in its *short form* with implicit universal quantifier. The expression $\forall i \in I\!N.\ \Phi$ represents a complete form of invariant property, which holds infinite number of times, i.e. on all elements of all involved streams. In the further at specification of invariant property we will leave out universal quantifier $\forall i \in I\!N$, nevertheless we will have its presence permanently in mind. By using of the short form of invariant property under notion $\Phi$ we have in mind the notion $\forall i \in I\!N.\ \Phi$.

*Syntax* of the *short form* $\Phi$ of *invariant property* with implicit universal quantifier we define as:

$$\Phi ::= true \mid z_{1(i)} = f(z_{1(i)}, ..., z_{p(i)}) \mid \mathbf{X} z_{(i)} = g(z_{1(i)}, ..., z_{r(i)}) \mid \neg\ \Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2$$

where *true* denotes the absence of any relations between streams in the object; $z, z_1, ..., z_p, ..., z_r$ are streams of the object; $f$ and $g$ are total computable functions, which return values of the same type as the elements of the stream $z$.

In the short form of invariant property with implicit universal quantifier an element $z(i)$ of stream $z$ with index $i$ we call *current element* of the stream, and its value — *current value* of the stream; an element $z(i+1)$ of stream $z$ with index $i+1$ we call *next element* of the stream, and its value — *next value* of the stream. Moreover by the reasons of visual clearness increasing we will denote current element of stream $z$ by $z_{(i)}$ instead of $z(i)$, and next element of the stream by $\mathbf{X} z_{(i)} \stackrel{def}{=} z(i+1)$.

For the given object $A$ the two sets: the set $X$ of the names of input streams and the set $Y$ of the names of output streams form together the set of names of *interface* streams of the object: $Interface(A) \stackrel{def}{=} X \cup Y = \{x_1, ..., x_n, y_1, ..., y_k\}$.

## 3. PARALLEL COMPOSITION AND CONNECTION OF INTERFACE STREAMS

In order to guarantee that our formal theory can be applicable for modelling of system with arbitrary topology and consisting of big number of objects, we have to introduce a formal mechanism of step-by-step successive construction of those systems from their constituting subsystems and objects. In our theory the mechanism of step-by-step successive construction of system of objects will be based on the following proposition. *A result of composition of objects is a system of these objects, which in one's turn is considered in our theory as a bigger object, and with which we syntactically and semantically operate in the same way as with any other object, i.e. we will use it as a constituting part for building of even bigger objects.*

We lay operations of composition of objects under a number of practical-oriented requirements. Operations of composition must have adequate expressive power in order to create big practical-oriented systems by using them. Moreover, operations of composition have to provide an ability of all-around automatization of generating process of formal models of systems from models of objects, which constitute these systems, and from their topology.

In order to satisfy the requirements above we define two operations of composition: the first one is called *parallel composition* and serves for for putting objects in parallel; the second one is called *connection* and serves for connecting of inputs and outputs of an object.

For the first time analogous operations of composition were introduced in the papers of Henzinger on component and interface theories [1] and of Lee on block-diagram languages [6]. Operations of composition, which are used in our theory, simplify the representation and understanding of different kinds of object compositions, and, moreover, they provide us ability to create objects of arbitrary topology and dimension.

For two arbitrary objects $A$ and $B$:

$$A = \langle\, X_A \,;\, S_A \,;\, Y_A \,;\, I_A \,;\, \Phi_A \rangle \quad B = \langle\, X_B \,;\, S_B \,;\, Y_B \,;\, I_B \,;\, \Phi_B \rangle$$

*parallel composition* transforms these two objects into new object $A||B$ in the following way:

$$A||B \stackrel{def}{=} \langle X_A \cup X_B \,;\, S_A \cup S_B \,;\, Y_A \cup Y_B \,;\, I_A \wedge I_B \,;\, \Phi_A \wedge \Phi_B \rangle$$

Sets of names of input, internal and output streams of composition $A||B$ are set-theoretical unions of the corresponding sets of stream names of objects $A$ and $B$: $X_{A||B} = X_A \cup X_B$, $S_{A||B} = S_A \cup S_B$ and $Y_{A||B} = Y_A \cup Y_B$. Consequently, we define the set of names of interface streams of the new object as $Interface(A||B) = Inrface(A) \cup Interface(B)$.

In the next step we introduce the operation of connection, which transforms an arbitrary object $A$ and a well-formed interconnect $\varphi_A$ into new object $A \cdot \varphi_A$.

*Interconnect* $\varphi$ defined on an object $A$ is a finite list (with length $l$)

$$\varphi = [(z_{S1}, z_{T1}), (z_{S2}, z_{T2}), ..., (z_{Sl}, z_{Tl})]$$

of pairs of stream names of the object $A$ such that in every pair $(z_{Si}, z_{Ti})$, where $0 \leq i \leq l$, both streams are interface streams $z_{Si}, z_{Ti} \in Interface(A)$, and moreover the types of these streams must be the same. In every pair $(z_{Si}, z_{Ti})$ the first component $z_{Si}$ we call *source* and the second one $z_{Ti}$ — *target*.

A special case of interconnect is an empty interconnect. *Empty interconnect* is interconnect containing no pairs of stream names, and we will denote it by the empty list [ ]. The empty interconnect is an identity element for the connection operation. Therefore, by the connection of an arbitrary object $A$ with the empty interconnect we obtain the previous object $A$, i.e. $A \cdot [\,] = A$.

The other special case of interconnect is an atomic interconnect, which will be needed in the inductive definition of the connection operation. *Atomic interconnect* is an interconnect containing only one pair of stream names.

In the operation of connection we produce pairwise connection of interface streams of an object according to list of pairs of their names, which is contained in interconnect. There are a number of interconnects admissible from the point of view of the definition above, but which can not be used in the operation of connection in the form, in which it is defined in this paper. Therefore we introduce a notion of "well-formed" interconnect, and later on in the operation of connection we will use only well-formed interconnects. Interconnect $\varphi_A$ defined on an object $A$ is called *well-formed*, if it satisfy the following conditions: 1) the target and source in every pair are different, i.e. in the interconnect $\varphi_A$ there is no pair of the form $(x, x)$; 2) all targets of the interconnect $\varphi_A$ are pairwise different; 3) when the same name of stream occurs in the interconnect $\varphi_A$ in different pairs in source position as well as in target position, then it must appear as target only after all its occurrences as source; 4) in every pair either both stream names denote input streams, or both denote output streams, or the source is a name of output stream and the target is a name of input stream.

In the cases, when at least one of these four conditions is not satisfied, we will consider that such interconnects are not well-formed, and such interconnects we will not use in the operations of connection.

For an arbitrary object $A$ and a well-formed on it *atomic interconnect* $[(z_S, z_T)]$ the operation of *connection* transforms the object $A$ into new object $A \cdot [(z_S, z_T)]$ according to the pair of names of the connected streams, which is contained in the atomic interconnect. Depending on the kind of atomic interconnect there are three possible cases of connection:

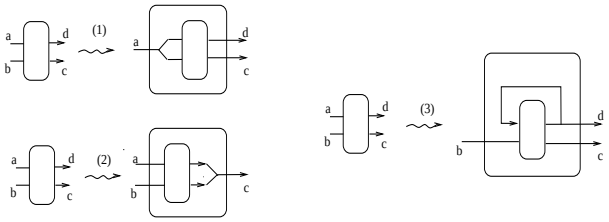



**Figure 1: Three different cases of connection of an object and a well-formed interconnect.**

1. in the case where both elements of the pair $[(z_S, z_T)]$ are names of input streams $z_S, z_T \in X_A$, we remove from the set $X_A$ of names of input streams of the object $A$ the target of the atomic interconnect, i.e. the name of the stream $z_T$: $A \cdot [(z_S, z_T)] \stackrel{def}{=}$

$$\langle X_A \setminus \{z_T\} \,;\, S_A \,;\, Y_A \,;\, I_A\,\{z_T/z_S\} \,;\, \Phi_A\,\{z_T/z_S\} \rangle$$

2. in the case where both elements of the pair $[(z_S, z_T)]$ are names of output streams $z_S, z_T \in Y_A$, we remove from the set $Y_A$ of names of output streams of the object $A$ the target of the atomic interconnect, i.e. the name of the stream $z_T$: $A \cdot [(z_S, z_T)] \stackrel{def}{=}$

$$\langle X_A \,;\, S_A \,;\, Y_A \setminus \{z_T\} \,;\, I_A\,\{z_T/z_S\} \,;\, \Phi_A\,\{z_T/z_S\} \rangle$$

3. in the case where in the atomic interconnect $[(z_S, z_T)]$ the source is a name of output stream $z_S \in Y_A$ and the target is a name of input stream $z_T \in X_A$, we remove from the set $X_A$ of names of input streams of the object $A$ the target of the atomic interconnect, i.e. the name of the stream $z_T$: $A \cdot [(z_S, z_T)] \stackrel{def}{=}$

$$\langle X_A \setminus \{z_T\} \,;\, S_A \,;\, Y_A \,;\, I_A\,\{z_T/z_S\} \,;\, \Phi_A\,\{z_T/z_S\} \rangle$$

In all three cases restriction on initial values $I_A\{z_T/z_S\}$ and invariant property $\Phi_A\{z_T/z_S\}$ of the new object $A \cdot [(z_S, z_T)]$ are obtained, correspondingly, from restriction on initial values $I_A$ and invariant property $\Phi_A$ of the original object $A$ by means of substitution of all occurrences of the name $z_T$ by name $z_S$.

Hypothetic fourth case, where the source is a name of input stream and the target is a name of output stream, is syntactically impossible, since it is filtered out by the requirements of the well-formedness of atomic interconnect.

We define operation of *connection*, which transforms an arbitrary object $A$ and a well-formed on it interconnect $\varphi_A = [(z_{S1}, z_{T1}), (z_{S2}, z_{T2}), ..., (z_{Sl}, z_{Tl})]$ into new object $A \cdot \varphi_A$, in the following inductive way:

$$A \cdot [\,] \stackrel{def}{=} A$$

$$A \cdot [(z_{S1}, z_{T1}), (z_{S2}, z_{T2}), ..., (z_{Sl}, z_{Tl})] \stackrel{def}{=} (A \cdot [(z_{S1}, z_{T1})]) \cdot [(z_{S2}, z_{T2}), ..., (z_{Sl}, z_{Tl})]$$

The sets of names of input and output streams of the connection $A \cdot \varphi$ of an object $A$ and a well-formed interconnect $\varphi = [(z_{S1}, z_{T1}), ..., (z_{Sl}, z_{Tl})]$ represent them self set-theoretical difference of the corresponding sets of stream names of the original object and the sets of targets of the interconnects: $X_{A\cdot\varphi} = X_A \setminus \{z_{Ti} \mid \forall i \in I\!N.\ 1 \le i \le l \wedge z_{Ti} \in X_A\}$ and $Y_{A\cdot\varphi} = Y_A \setminus \{z_{Ti} \mid \forall i \in I\!N.\ 1 \le i \le l \wedge z_{Ti} \in Y_A\}$. The set of names of internal streams of the connection $A \cdot \varphi$ is the set of names of internal streams of the original object $A$, since internal streams of the object are not influenced by operations of connection, i.e. $S_{A\cdot\varphi} = S_A$. Therefore we define the set of names of interface streams of the new object $A \cdot \varphi$ as a set-theoretical difference of the set of names of interface streams of the original object $A$ and the set of targets of the interconnect $\varphi$, i.e. $Interface(A \cdot \varphi) = Interface(A) \setminus \{z_{Ti} \mid \forall i \in I\!N.\ 1 \le i \le l\}$.

Let us illustrate the three cases of connection described above by the following example (see on the Fig. 1). We take the object $A$ with two input $a, b$ and two output $c, d$ streams of the same type:
$A = \langle a, b; empty; c, d; b(0) = 0; c_{(i)} = a_{(i)} + 5 \wedge d_{(i)} = 2 * b_{(i)} \rangle$

We connect in the first case two input streams $a$ and $b$ by means of the interconnect $\varphi_1 = [(a, b)]$, in the second case — two output streams by $\varphi_2 = [(c, d)]$, and in the third case — one input stream with one output $\varphi_3 = [(d, a)]$. As a result we obtain three new objects: first object with one input stream $a$ and two output streams $c, d$:
$A \cdot [(a, b)] = \langle a; empty; c, d; a(0) = 0; c_{(i)} = a_{(i)} + 5 \wedge d_{(i)} = 2 * a_{(i)} \rangle$,
second object with two input $a, b$ and one output $c$ streams:
$A \cdot [(c, d)] = \langle a, b; empty; c; b(0) = 0; c_{(i)} = a_{(i)} + 5 \wedge c_{(i)} = 2 * b_{(i)} \rangle$,
third object with one input $b$ and two output $c, d$ streams:
$A \cdot [(d, a)] = \langle b; empty; c, d; b(0) = 0; c_{(i)} = d_{(i)} + 5 \wedge d_{(i)} = 2 * b_{(i)} \rangle$.

## 4. SEMANTICS

According to the theorem in the first section, a tuple of stream is isomorphic to the stream of tuples of the corresponding elements of these streams. Let a stream of type $Z^\omega$ be isomorphic to all streams, which constitute a structure of an object $A$. We call this stream *generalized* stream of the object. Then semantics $[\![A]\!]$ of the object $A$ is a set of all admissible behaviors of generalized object's stream, which satisfy all requirements imposed on the object. Moreover, every element from the set $[\![A]\!]$ is a concrete instance of behavior of the generalized stream, or in other words, a tuple of concrete instances of behaviors of all streams constituting the object.

Formal *semantics* $[\![A]\!]$ of an object $A = \langle\ X\ ;\ S\ ;\ Y\ ;\ I\ ;\ \Phi \rangle$ is given by:

$$[\![A]\!] \stackrel{def}{=} \{\![I]\!\} \cap \{\![\Phi]\!\}$$

where $\{\![I]\!\} \subseteq Z^\omega$ is the set of generalized streams, which satisfy the restriction on initial values $I$; the invariant $\{\![\Phi]\!\} \subseteq Z^\omega$ is the set of generalized streams, on which the invariant property $\Phi$ is satisfied infinitely long on every element of the stream. Formally $\{\![\Phi]\!\}$ is the carrier set of the subcoalgebra of the invariant constructed by the invariant property $\Phi$.

## 5. COMPARISON WITH AUTOMATA THEORY

THEOREM 2. *Theory of infinite streams and objects has strictly greater expressive power than the theory of push-down automata.*

In our theory we can encode arbitrary push-down automata. We know that the language $\{a^n b^n c^n *^\omega \mid n \in I\!N\}$ can not be recognized by push-down automata . In contrast, in our theory we can create object, which recognizes this language. This object is given by:
$A = \langle x\ ;\ k, l, m\ ;\ empty\ ;\ k(0) = 0 \wedge l(0) = 0 \wedge m(0) = 0\ ;\ \Phi_A \rangle$
with the invariant property $\Phi_A \stackrel{def}{=}$

$$x_{(i)} = a \implies (\mathbf{X}k_{(i)} = k_{(i)} + 1 \wedge \mathbf{X}l_{(i)} = l_{(i)} \wedge \mathbf{X}m_{(i)} = m_{(i)} \wedge (\mathbf{X}x_{(i)} = a \vee \mathbf{X}x_{(i)} = b))$$
$$\wedge$$
$$x_{(i)} = b \implies (\mathbf{X}k_{(i)} = k_{(i)} \wedge \mathbf{X}l_{(i)} = l_{(i)} + 1 \wedge \mathbf{X}m_{(i)} = m_{(i)} \wedge (k_{(i)} = l_{(i)} + 1 \Rightarrow \mathbf{X}x_{(i)} = c) \wedge (k_{(i)} \neq l_{(i)} + 1 \Rightarrow \mathbf{X}x_{(i)} = b)$$
$$\wedge$$
$$x_{(i)} = c \implies (\mathbf{X}k_{(i)} = k_{(i)} \wedge \mathbf{X}l_{(i)} = l_{(i)} \wedge \mathbf{X}m_{(i)} = m_{(i)} + 1 \wedge (l_{(i)} = m_{(i)} + 1 \Rightarrow \mathbf{X}x_{(i)} = *) \wedge (l_{(i)} \neq m_{(i)} + 1 \Rightarrow \mathbf{X}x_{(i)} = c)$$
$$\wedge \quad x_{(i)} = * \implies \mathbf{X}x_{(i)} = *.$$

## 6. ACKNOWLEDGMENT

I would like to thank my supervisor Prof. Horst Reichel and my colleague Dmitri Schamschurko. Without their help, useful comments and criticism this work would not be what it is today.

## 7. CONCLUSIONS

This paper provides formal theory of infinite streams and objects, which contains our point of view on the problem of formal modelling of behaviors of objects and their systems with big or infinite number of internal states. In our theory we give up the representation of an object through its states and switch to the representation through the sets of all admissible behaviors of interface streams and internal streams constituting the object. The synchronous use of all involved streams provides us uniform way for modelling of inputs, outputs and internal variables of objects, simplifies object's definitions, renders possible our theory to be not afraid of the object's dimensionality increase, and gives us enough expressive power. We have shown in this paper that our theory has strictly greater expressive power than push-down automata.

## 8. REFERENCES

[1] L. de Alfaro and T. Henzinger, Interface Theories for Component-based Design, in Proceedings of Workshop on Embedded Software EMSOFT, 2001.

[2] F. Arbab, C. Baier, J. Rutten and M. Sirjani, Modeling Component Connectors in Reo by Constraint Automata, in Proceedings of Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA), 2003.

[3] F. Arbab and J. Rutten, A Coinductive Calculus of Component Connectors, in Proceedings of WADT 2002, Lecture Notes in Computer Science, vol. 2755, Springer-Verlag, 2003, pp. 35-56.

[4] M. Broy, K. Stolen, Specification and development of interactive systems, Monographs in Computer Science, vol. 62, Springer, 2001.

[5] B. Jacobs and J. Rutten, A Tutorial on (Co)Algebras and (Co)Induction, EATCS Bulletin 62, p.222-259, 1997.

[6] A. Lee, Overview of the Ptolemy Project, Technical Memorandum UCB/ERL-M01/11, University of California, Berkeley, 2001.

# Software Product Lines structuring based upon Market Demands

Montse Ereño
Computer Science Department
University of Mondragon
Loramendi 4, 20500,
Mondragon, Spain
(34) 943 79 47 00
mereno@eps.mondragon.edu

Uxue Landa
Computer Science department
University of Mondragon
Loramendi 4, 20500,
Mondragon, Spain
(34) 943 79 47 00
uxuelanda@gmail.com

Dra. Rebeca Cortazar
Computer Science Department
University of Deusto
Avda. de las Universidades, 24.
48007 Bilbao, Spain
(34) 944 139 000
cortazar@eside.deusto.es

## ABSTRACT

Nowadays the increasing demand for customized products and services in traditional areas such as Automotion Manufacturing or Aeronautical Component Engineering is being satisfied with a new approach called "Product Platform". This successful approach is also being considered in the design of software-based components in these areas, which are recognized as complex and critical.

In this paper, we present the research that is being carried out at Mondragon University. This effort focuses on the analysis of existing Product Platform Development methods and the transference of this know-how to Software Product Development.

As a result, a Software Product Line (SPL) development method will be defined and applied in a real case. This method will be based upon market demands, so it should be flexible enough to respond to customer's requests and market pressure. In this paper we will explain in detail one step of the process. This step is concerned with how QFD technique can be used to the specification of components in a SPL.

## Keywords

Software Product Lines, Mass Customization, Market Perspectives, Requirements analysis and specification, Viewpoint-Oriented Requirements, Components.

## 1. INTRODUCTION

Today's highly competitive, global marketplace is redefining the way companies do business. If we focus on the Software Development area, we can see that during the last few years the Software producer's interests have been in contradiction with the customer's interests. Indeed, producers want to maximize their benefits and, thus, to minimize their production's costs and time to market. In opposition, customers ask for better quality and for Software tailored to their individual needs. Furthermore, complexity and size of software products are rapidly increasing due to the market's evolution.

This situation is not new, as the scenario is well known in other sectors such as Automotion Manufacturing or Aeronautical Component Engineering. In these sectors, the development of a family of products—a group of related products derived from a common Product Platform—has provided an efficient and effective mean to implement a sufficient product variety to satisfy a range of customer demands.

Software community has tried to adopt this new approach of product development. They called it "Software Product Lines " (SPL). A SPL is a set of intensive systems of software that share a set of common characteristics, that satisfy the specific needs of a segment of a particular market and that are developed from a set of common assets in a pre-established way [1].SPL represent an innovative and growing concept in Software Engineering. It can also efficiently satisfy the current demand for mass customization of software.

Despite the fact that SPL have recently gained research interest, there are only few empirical studies on them. We would expect a rush by the Software industry to exploit the competitive advantages offered by SPL. However, most of the software industry is either unaware of the emerging field of SPL, or if an organization is aware, they don't understand how SPL might be applied in their situation. There is a *need of process* and *quantitative models* to help enterprises in this new way of product development.

The purpose of our research in Mondragon University is the analysis of existing Product Platform Development methods and the transference of this knowledge to Software Product Development. There is an especial interest in product specification and modularization. The Governing Body of Guipuzkoa (Spain) takes part in this research.

The rest of the extended abstract is organized as follows. In section 2, we present the previous work in this field. In Section 3, we illustrate our advances. In this section, the core part of our contribution is given. Finally, in section 4, the paper ends with a conclusion and some words on the future work.

## 2. PREVIOUS WORK

From a mechanical perspective or taking into consideration products of tangible structure, we can affirm that this issue has been studied widely. During the decade of the 90's, multiple studies and investigations appeared that were trying to tackle

these difficulties: how to obtain a great variety of products with a unique design that combines the greatest number of possible similarities [2]. The solution to gain this mass customization passes through the concept of product platform [3]. That is, a unique design for a platform can be personalized in such a way that can extend the variety of different products. We can mention the automation sector [4], aeronautical sector [5], aerospace sector [6] or companies like Hewlett-Packard or Black&Decker[7]]. In all, the challenge is in how to define this platform.

From a Computer Science perspective or intangible structure Product Development, considerable efforts have been made, too. Its origins can go back to the 60's. In a conference titled "Mass Produced Software Components" held in 1968, [8] introduced the *Reusability* concept as the key for the efficient design of new Software Products. The efforts made in the definition of methodologies for the development of Product Lines are considerable: PuLSE [9], KobrA [10], COPA[11], FAST [12], FORM [13], SPLIT [14], etc. From an application point of view, the SPL approach is being adopted by organizations of different sectors [15] [16][17].

As much from the theoretical point of view as form the application one, Product Lines are an interesting and promising approach. Therefore, we can appreciate how very different industries converge towards a new way of design when undertaking a product family development: SPL. Nevertheless, the software development sector is not as advanced as the mechanical industry. Most of the real cases of SPL creation are based upon theoretical methods and a great amount of intuition. There is a need of sound processes and quantitative models to help enterprises in their new way towards Product Development. This is the main motivation for this research.

## 3. SPL development with HOQ

This research analyzes the ability to apply a design methodology for mechanical products in the design of SPL. The base methodology was defined in a project realized in 2002 by the Design Group of Mondragon University with the cooperation of the Governing Body of Guipuzkoa (Spain)

During our research, we are focusing our effort in two main activities. On the one hand, we have been studying the process itself and the suitability to apply it in Software Manufacturing. On the other hand, we have been analyzing the importance of the Voice of the Customers for the analysis and specification of the SPL. During the rest of the paper, we will explain in depth the second subject.

## 3.1 Requirements Engineering for SPL: Getting the Voice of the Customer

In the context of SPL, requirements analysis and specification defines the system to be developed and forms the basis for a contract between a system provider and a customer. There are a number of inherent difficulties in this process. As [18], [19] [20] said, the hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establish the detailed technical requirements. No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later.

It's known by all the Software community that the most critical dimension in Product Development nowadays -and even more in the future- is to develop the products your customers want. During the product development, the Voice of the Customer (VOC) must be accurately defined. Without an accurate and complete definition of what a customer desires, the rest of the process is irrelevant.

In Product Development context, requirements specification is used to formalize and communicate the needs of a real or hypothetical customer to product developers. This is the forward flow of information from concept development to design and implementation. In our research we propose the use of QFD (Quality Functional Deployment ) to deploy the Voice of the Customer throughout the product's design. Specifically, we apply the first part of QFD: the House of Quality (HOQ).

### *3.1.1 QFD in SPL environment*

QFD is a structured approach for defining customer needs or requirements and translating them into specific plans to produce products that meet those needs. QFD can be considered as a tool for requirements analysis and specification. QFD answer the following questions:

- What are the features the customers/market desires?
- What functions must the product serve, and what functions must we use to provide the product or service?
- Based on our available resources, how can we best provide what our customer wants?

QFD is intended to be used by multidisciplinary or cross-functional teams (Marketing, Design Engineering, Finance, Quality Assurance, Manufacturing, Test Engineering, Product Support, etc). This is, of course, to add variety, varying perspectives, and hopefully provide more insight into issues that a single function team would eliminate or not consider relevant.

All these reasons, allow us to believe that QFD will support the different perspectives of the multiple customers, so we will use it to support the analysis and specification of the Product Line Requirements and to obtain the suitable Product Lines Structure based upon stakeholders demands.

## 3.2 Building the HOQ for SPL

The initial step of QFD is determining the voice of the customer (VOC). There is no one monolithic VOC. Customer voices are diverse. All these voices must be considered to develop a successful product. This is accomplished through extensive market research, or other more direct communication methods such as surveys, product complaint history, direct customer feedback, etc. The goal is to find the exact desires of the intended target group and design to provide these aspects, but still remain cost effective for the manufacturer as well as the customer. This

understanding of the customer needs is then summarized in a Product Planning Matrix, or House of Quality (HOQ). We use these matrices to translate higher level "what´s" or needs into lower levels "hows" product requirements or technical characteristics to satisfy these needs.

In software development different roles are found (customers, developers, designers, etc.), where each one provides different needs, with different abstractions levels and with their own perspectives. We have defined four actors. Each actor has his own perspective of the product.

- **Customer or Market**: this group represents all the possible clients around the world.
- **Enterprise**: this group is conformed for all the roles into an enterprise, who are involved on the development and commercialization of the product (Marketing, Design Engineering, Quality Assurance, Manufacturing, Test Engineering, Finance, Product Support, etc.)
- **Designer**: this is a role whose work is concern with the implementation aspects.
- **Stakeholder**: this role groups the previous roles.

When trying to unify all this information and use it in the parts identification (product structuring), we realize that we need to pass twice the first step of HOQ. In each step, different actors play their roles. In each step, we cross different views. In the first round, we cross customer expectations (customer views) against product requirements (enterprise view). During this step, a functional language is used. The result of the first step is a complete identification of the wished functional requirements for the LPS. Nevertheless, we must consider that there are characteristic of a product that affects many others. We call them "Crosscutting concerns". Concretely the term "crosscutting concerns" talks about the quality factors of the Software System (security, real time constraints, usability, persistence, etc.). In the second passage of our method, we establish the correlation between the requirements obtained in step 1 and the "crosscutting concerns". From this step we obtain a quantitative knowledge of the quality characteristics wished by the users, as well as the requirements that are affected and in what degree.

It's interesting to rank the WHATs from the most desired ones to least desired ones, so we can give precedence to some requirements. After this step, all the Product Line requirements are discovered, and we can determinate which of them are common to all the products and which are specific.

In the next step, the abstraction level is lower than in the previous step. This time, technical aspects appear. On the vertical axis we write Product Line requirements (result of the previous step), and on the horizontal axis we write a first proposal of components. This proposal is based upon object oriented concepts (abstraction, inheritance, etc.). The correlation between the proposal of components and the product requirements allow us to detect the exceed or lack of components. The components, may be correlated among them (tiled of the HOQ) based on their behavior, so we can detect the necessity to optimize this proposal of components, adding or eliminating some of them. Figure 1 summarizes all the steps.

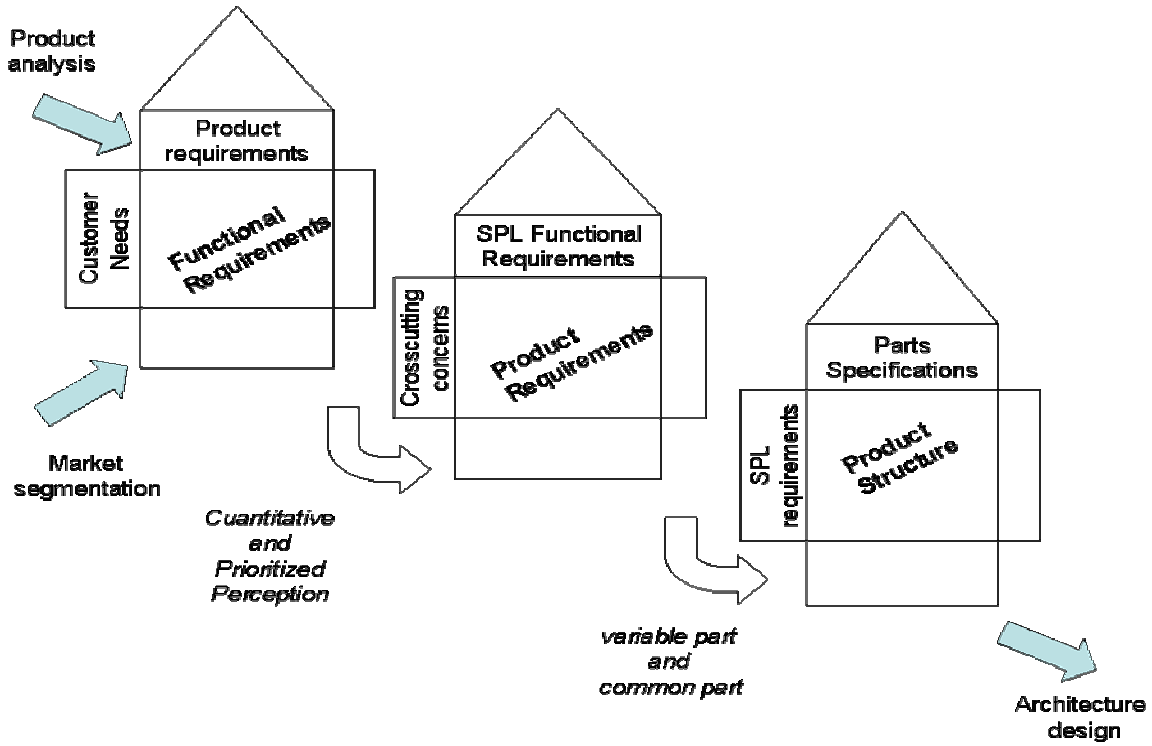



**Figure 1. HOQ for SPL**

## 4. CONCLUSIONS AND FUTURE LINES OF WORK

This paper presents a method whose aims to get the best Product Lines Structure based upon stakeholders demands. This method is integrated in a complete process for SPL development in an enterprise context.

The method applies recognized approaches from Software Engineering discipline such us Object-Oriented principles and Aspect Oriented Software Development ideas (AOSD).

The essence of the method is not a new idea. It is based on the work realized in the area of mechanical manufacture. In our research, we have just analyzed the existing product platform development methods in the mechanical area and the transference of this knowledge to Software Product Development. We must conclude that it's possible to apply mechanical methods and process to SPL as long as we complete some aspects like the adoption of SPL and the consideration of management activities. Also, it's possible to apply techniques like QFD with some adjusts.

As for future lines of work, we think that the application of the method in multidisciplinary products (i.e. products with software and electronic components) would seem to be of great interest. Also, it would turn out interesting to deal with the factor time in explicit form in the method.

## 5. REFERENCES


[1] [Clements 2002] Paul Clements, Linda Northrop, Software Product Lines: Practices and Patterns, Addison Wesley, 2002

[2] [Miller 1998]Thomas D. Miller, Per Elgard, Defining Modules, Modularity and Modularization-Evolution of the Concept in a Historical Perspective, Design for Integration in Manufacturing, 13th Research Seminar, 1998

[3] [Gonzalez-Zugasti 2000] Javier P. Gonzalez-Zugasti, Kevin N. Otto, Modular Platform-Based Product Family Design, ASME Design Engineering Technical Conference and Computers and Information in Engineering Conference, September 2000.

[4] [Bremmer 1999] R. Bremmer, Cutting Edge Platforms, Financial Times Automotive World, pp. 30-88. September 1999.

[5] [Fujita 1998] Kikuo Fujita, Shinsuke Akagi, Tetsu Yoneda, Makbi Ishikawa, Simultaneous Optimization of Product Family Sharing System Structure and Configuratio, ASME Design Engineering Technical Conferences, September 1998

[6] [Gonzalez-Zugasti 1998] Javier P. Gonzalez-Zugasti, Kevin N. Otto, John D. Baker, A Method for Architecting Product Platforms with an Application to Interplanetary Mission Deign, ASME Design Engineering Technical Conferences, September 1998

[7] [Dahmus 2000] Jeffrey B. Dahmus, Javier P. Gonzalez-Zugasti, Kevin N. Otto, Modular Product Architecture, ASME Design Engineering Technical Conferences and Computers and Information in Engineering Conference, September 2000

[8] [McIlroy76] "Mass-Produced software components". In J.M. Buxton, P. Naur, and B.Randell, editors, Software Engineering Concepts and Techniques; 1968. NATO Conference on Software Engineering, pp. 88-98. Van Nostrand Reinhold, 1976

[9] [DeBaud 1998] Jean-Marc DeBaud, Meter Knauber, Applying PuLSE for Software Product Line Development, European Reuse Workshop 98, 1998

[10] [Atkinson 2000] Colin Atkinson, Joachim Bayer, Dirk Muthig, Component-Based Product Line Development: The KobrA Approach, "1st International Software Product Line Conference", 2000

[11] [America 2000] Pierre America, Henk Obbink, Jürgen Müller, Rob van Ommering, COPA: A Component-Oriented Platform Architecting Method for Families of Software Intensive Electronic Products, Tutorial for SPLC1, the First Software Product Line Conference, Denver, Colorado, 2000

[12] [Weiss 1999] David M. Weiss, Chi Tau Robert Lai, Software Product-Line Engineering: A family-Based Software Development Process, Addison Wesley, 1999

[13] [Kang 1998] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures, Annals of Software Engineering, 5:143--168, 1998

[14] [Coriat 2000] M. Coriat, J. Jourdan, F. Boisbourdin, The SPLIT Method, Building Product Lines for Software-Intensive Systems. Proceeding of the SPLC1. Massachusetts, Kluwer Academic Publishers, 147 – 166, 2000

[15] [Brownsword 1996] Lisa Brownsword, Paul Clements, A Case Study in Successful Product Line Development, Technical Report CMU/SEI-96-TR-016, 1996

[16] [Clements 2002b] Paul C. Clements, Linda M. Northrop, Salion, Inc.: A Software Product Line Case Study, Technical Report CMU/SEI-2002-TR-038, 2002

[17] [Clements 2001] Paul Clements, Sholom Cohen, Patrick Donohoe, Linda Northrop, Control Channel Toolkit: A Software Product Line Case Study, Technical Report CMU/SEI-2001-TR-030, 2001

[18] [Brooks87] Brooks, F. "No silver Bullet: Essence and Accidents of Software Engineering". Computer 20, 4 April 1987: 10-19

[19] [Davis90] Davis, A.M. Software Requirements: Analysis and Specification. Englewood Cliffs, NJ: Prentice-Hall 1990

[20] [Faulk97] Faulk, S.R. "Software Requirement: A tutorial", 128-149. Software Requirements Engineering. Los Alamitos, CA: IEEE Computer Society Press, 1997

# A Component-Based Specification Approach for Embedded Systems using FDTs

Abdelaziz Guerrouat
Clausthal University of Technology
Department of Computer Science
Julius-Albert-Str. 4
D-38678 Clausthal-Zellerfeld
Phone: +49-5323-72-7172
aguerrou@in.tu-clausthal.de

Harald Richter
Clausthal University of Technology
Department of Computer Science
Julius-Albert-Str. 4
D-38678 Clausthal-Zellerfeld
Phone: +49-5323-72-7170
richter@in.tu-clausthal.de

## ABSTRACT
This paper presents a framework for specification and testing of component-based embedded systems using formal description techniques (FDTs). We deal with embedded systems from the point of view of communication and thus we propose a communication model for them. We further explain the meaning of component-based embedded systems and how these can be specified using FDTs. FDTs such as Estelle and SDL are based on EFSMs (Extended finite State Machines) and have been widely used in the automation of the development process of protocols and communicating systems, i.e. for specification, analysis and validation purposes. The main goal of this work is to demonstrate the reusability of FDTs for component-based systems.

## Keywords
Formal description techniques, component-based systems, embedded system, specification, testing

## 1. INTRODCTION
Embedded systems are becoming more and more the key technology of any kind of complex technical systems, ranging from telecommunications devices to automobiles and aircrafts. An embedded computer system is a computer system that represents a part of a larger system and performs some requirements of that system.

The growing amount and complexity of requirements on embedded systems regarding properties like safety and real-time behaviour make the software development process a costly and error-prone activity. The cost factor plays, however, a central role in today's industrial competition, for instance, between car manufacturers. The development of competitive and efficient products is imposing more and more constraints to the design of embedded systems. One of the means to reach this goal are formal methods to support the different phases of system development, i.e. specification, synthesis and validation. There are several requirements for those methods that should be among others qualities abstract, understandable, analyzable, scalable and unambiguous specification formalisms.

Component-based development represents an attractive approach in the embedded system area, in particular for the development of many variants of products [10]. While in the last few years component-based software development gained much more attention from both researchers and practitioners, testing such software systems is still however to be more studied [11]. Because one believes that once a component is sufficiently tested, it is not needed to test it again when reused. But, this is generally not true, since components may satisfy a certain application domain and fails in a new environment [11].

Formal description techniques have demonstrated their effectiveness in testing complex requirements like those for communicating systems [1] [2]. They provide a solid mean for unambiguous specification and rigorous analysis and are based on EFSMs (extended finite state machines). They differ from conventional programming languages by providing not only a formal syntax but also a formal semantic. The application of formal specifications increases the confidence in the software and the system. Especially in the area of safety-critical systems, the use of formal techniques is highly recommended [3] [8].

Testing of communication systems based on FDTs mainly concerns conformance testing. The later corresponds to a black box test. But this type of test is the most dominant and important one in component-based systems due to the nature of the constituents and properties of component-based systems. Indeed, components are independent and replaceable parts of a system and should be conform to and provide a set of interfaces. They also consist usually of special components, called COTS (commercial-off-the-shelf) that can be purchased on a component market. These are often delivered without their source code which makes other types of tests like white or grey tests less appropriate.

Actually, the mostly used formalisms to specify requirements for embedded systems are Statecharts and also UML (Statecharts are converted in UML) as semi-formal models. Although Statecharts and UML provide graphical facilities, they might lack formal and unambiguous semantics. Therefore, detecting bugs, incompleteness and inconsistencies becomes a difficult task. To alleviate these lacks many authors try to combine formal notations like Z with state-transition models [5]. Z is based on set theory and first order predicate logic and used for data structuring and abstracting. However, approaches developed around this model do not clearly address test data generation methods for analysis and validation purposes [6] [7] and/or do not deal with component-based systems.

Finite state machines are very popular in the control flow specification of state/transition-based systems and many related analysis methods have been developed [6] [7]. These support a formal test derivation which is used for validation and testing purposes. However, finite state machines lack to deal with the data flow. This shortcoming can be alleviated by using the

extended finite state machine model on which formal description techniques are based.

In this paper, we present a framework for testing component-based embedded systems by using formal description techniques (FDTs). We first identify the main components an embedded system consisting of and then show how these can be linked together to constitute the whole embedded system by using the FDT Estelle. The principle of testing such obtained embedded systems is explained, i.e. fault model, test derivation and test execution.

The rest of the paper is organized as follows. Section 2 give the basic structure of embedded systems and explains their basic communication model. The specification and testing framework for component-based embedded systems using the FDT Estelle is presented in Section 3. Finally, Section 4 concludes the paper.

# 2. BACKGROUNDS

## 2.1 Embedded System Components

An embedded system (ES) is any computer system or computing device that performs a dedicated function or is designed for use with a specific embedded software application, e.g. PDA (Personal Data Assistant), Mobile Phone, E-Book (Electronic Book), Robot, etc. That is, an embedded system is a special-purpose system built into a larger device. It is embedded as a subsystem in a larger system which may or may not be a computer system. An embedded system is typically required to meet specific requirements.

Embedded systems must usually be dependable, efficient and must meet real-time constraints. Be 'dependable' means that an embedded system must be reliable, available and safe. The efficiency mostly concerns properties like energy, code-size, run-time, weight and cost. An embedded system is dedicated for a certain application and characterized also by a dedicated user interface. Thus, knowledge about future behavior at design time can be used to minimize resources and to maximize robustness. Many embedded systems must meet real-time constraints. A real-time system must react to stimuli from the controlled object (or the operator) within the time interval dedicated by the environment.

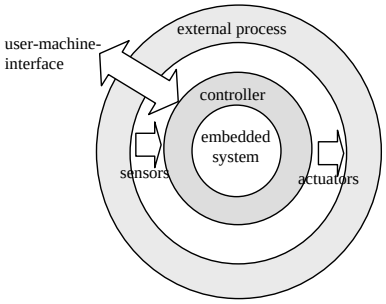



**Figure 1. Main components of an embedded system**

Embedded systems are frequently connected to a physical environment through sensors and actuators. They are typically reactive systems. A reactive system is in continuous interaction with its environment and executes at a pace determined by that environment. The behavior depends on input and current state for which the automata model is often most appropriate.

Figure 1 illustrates the main constituents of an embedded system comprising an *external process*, *sensors*, *actuators*, and a *controller*:

- *The external process* is a process that can be of physical, mechanical, or electrical nature.
- *Sensors* provide information about the current state of the *external process* by means of so-called *monitoring events*. They are communicated to the *controller*. For the *controller*, they represent input events. They are considered as stimuli for the *controller*.
- *The controller* must react to each received event, i.e. input event. Events originate usually from *sensors*. Depending on the received events from sensors, corresponding states of the *external process* will be determined.
- *Actuators* receive the results determined by the *controller* which are communicated to the *external process* by means of so-called *controlling events*.

The external process is usually given in advance. In contrast, the controller is often implemented by real-time hardware and software. This should allow each modification of the controller algorithm in a straightforward way each time this is needed. The controller's behavior is depending on that of the external process. The controller commands the behavior of the external process taking into consideration requirements on the process and its characteristics, such as physical laws, real time and other constraints.

## 2.2 Component-Based Embedded System Development

In the component-based approach for embedded systems one distinguishes a *component repository*, a *composition environment* and a *run-time environment*. The component repository consists of single specifications of the above indicated components of an embedded system: sensors, controller and actuators. These correspond to EFSMs and for each component build functional modules with respect to the used FDT. The composition environment is the embedded system specification which consists of the specification of its environment and its controller. This takes places by linking the single modules to each other by means of channels via interfaces, called interaction points (Figure 2). These modules interact with each other via broadcasting events via these interaction points. However, a sequence has to be respected in this communication. For instance, the direct communication of a module of an actuator with a sensor is not allowed. Run-time environment consists of the instantiated embedded system specification issued from the former step, i.e. the composition. Assuming the FDT Estelle, this builds a tree of linked tasks from which the system is composed. Each subtree rooted in a so-called system process or system activity task represents a subsystem [2]. The number of subsystems and the links between them are fixed once the specification is initialized.

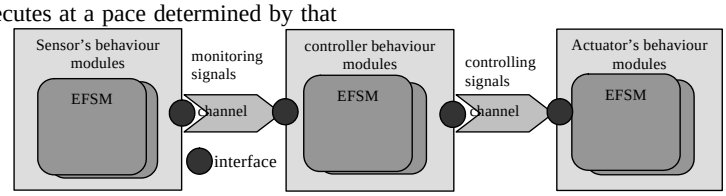



**Figure 2. Overview of the composition environment for embedded system based on EFSMs**

The most important component of an embedded system consists of the controller which communicates with its environment, i.e. sensors and actuators, via signals (i.e. events). To be recognized by all components, these events have to be declared as global variables for adjacent EFSMs. The output events of sensors represent input events for the controller. The events from the controller to the actuators are output events and represent input events for the actuators. They result from new computations performed by the controller that is triggered by the received input events.

Depending on the nature of sensor events (e.g. indicating the power on/of state for an electrical unit, the speed of a mobile object such as a car, etc.) the corresponding EFSM of this component is triggered and the concerned transition(s) are performed. This triggers the EFSMs of the controller whose states change. Depending on the received events, transitions in the FSMs are executed. Note, that transitions in the controller can spontaneously be triggered by other events, e.g. time out. The modeled subsequent state of the external process is computed and communicated as output events via the actuators.

To provide an intermediate specification model which better fits the behaviour part of the considered FDT, i.e. Estelle, we introduce a new EFSM, called p-EFSM (p stands for 'predicated'). This is defined as follows:

**Definition 1** A *predicated extended finite state machine* (p-EFSM) is an 8-tuple $<S, C, I, P, O, T, s_0, c_0>$ where $S$ is a non-empty set of main states, $C=dom(v_1)$ x … x $dom(v_n)$ a non-empty countable set of contexts with $v_i \in V$, $V$ a non-empty finite set of variables, and $dom(v_i)$ a non-empty countable set referred to as the domain of $v_i$, $P$ a countable set of predicates (possibly empty), $I$ a non-empty finite set of inputs, $O$ a non-empty finite set of outputs, $T \subseteq S$ x $C$ x $I$ x $P$ x $O$ x $S$ x $C$ a set of transition relations, $s_0 \in S$ the initial main state, and $c_0 \in C$ the initial context of the p-EFSM.

p-EFSM extends the conventional EFSMs for FDT mapping purposes as we will see later. p-EFSM is similar to EFSM except that in a p-EFSM the conditions on transitions are explicitly specified. This is just a notation facility and functionally and conceptually there is no difference between both models. In the rest of the paper, we indifferently address both models.

This, a transition $t \in T$ of a p-EFSM is a 7-tuple $<s, c, I, p, o, s', c'>$ where $s \in S$ is a current main state, $c \in C$ a current context, $i \in I$ an input, $p \in P$ a enabling predicate which depends on the context $c$, $o \in O$ an output, $s' \in S$ a next main state, and $c' \in C$ a next context.

We consider one or more p-EFSMs for each component of the system and denote them with indices *s*, *c* and *a* for *sensors*, *controller*, and *actuators*.

Interdependencies between these components are described as follows:

- Let be given a transition
  $t_s \in T_s$: $t_s = <s_s, c_s, i_s, p_s, o_s, s'_s, c'_s>$ with
  $s_s \in S_s$, $c_s \in C_s$, $i_s \in I_s$, $p_s \in P_s$, $o_s \in O_s$, $s'_s \in S_s$, $c'_s \in C_s \Rightarrow \exists\ t_c \in T_c$
  $| \ o_s \equiv i_c$
  That is, each output event generated by sensors must trigger a transition of the controller. This event represents an input event for the triggered transition. We assume here that the predicates related to the transitions are satisfied by the actual context.
- Let be given a transition $t_c \in T_c$ with $s_c \in S_c$, $c_c \in C_c$, $i_c \in I_c$, $p_c \in P_c$, $o_c \in O_c$, $s'_c \in S_c$, $c'_c \in C_c$,
  if $i_c \in O_s \Rightarrow \exists\ t_s \in T_s$ and $i_c \equiv o_s$.
  This means that if there exists a transition of the controller whose input event belongs to the set of output events of the sensors then it must exist a transition of the sensors whose output event is identified with the given event.
- Let be given a transition $t_a \in T_a$: $t_a = <s_a, c_a, i_a, p_a, o_a, s'_a, c'_a>$ with $s_a \in S_a$, $c_a \in C_a$, $i_a \in I_a$, $p_a \in P_a$, $o_a \in O_a$, $s'_a \in S_a$, $c'_a \in C_a \Rightarrow \exists\ t_c \in T_c$: $t_c = <s_c, c_c, i_c, p_c, o_c, s'_c, c'_c>$ and $o_c \equiv i_a$.
  Each transition of actuators must be only triggered by the controller and must match the output event of the triggering transition of the controller.

Estelle is a standardized formal description technique (International Standard ISO 9074) based on concepts of structured communicating extended state automata and Pascal. It is oriented towards the specification of complex distributed systems, in particular communicating systems. A specified system is presented as a tree of tasks where each task has a fixed number of input/output access points (interaction points). Within a specified system it exists a fixed structure of subsystems (sub-trees of tasks) and communication links between subsystems.

SDL (Specification and Description Language) is an object-oriented, formal language defined by The International Telecommunications Union Telecommunications Standardization Sector (ITU) (formerly Comité Consultatif International Télégraphique et Téléphonique [CCITT]) as recommendation Z.100. The language is intended for the specification of complex event-driven real-time, and interactive applications involving many concurrent activities that communicate using discrete signals.

Indeed, EFSMs can functionally describe system components that may be blocks or modules depending on the used formal description technique[1].

# 3. SPECIFICATION AND TESTING BASED ON *Estelle*

## 3.1 Specification

A specified embedded system is a tree of tasks (p-EFSMs) which can be categorized in three classes corresponding to controller, sensor and actuator modules. They are organized in an hierarchical structure (parent-son-relationship). Each task has a fixed number of Input/Output access points (interaction points) which can be associated to controller, sensors or actuator modules. Bidirectional communication links may exist between tasks (between their interaction points). Within a specified embedded system exists a fixed structure of subsystems (sub-trees of tasks), corresponding to controller, sensors or actuators, and of communications links (between them) (s. Figure 3). Within a subsystem both structures (of tasks and communication links) may change dynamically. Tasks exchange interactions:

[1] SDL uses the 'block' concept whereas Estelle 'module'.

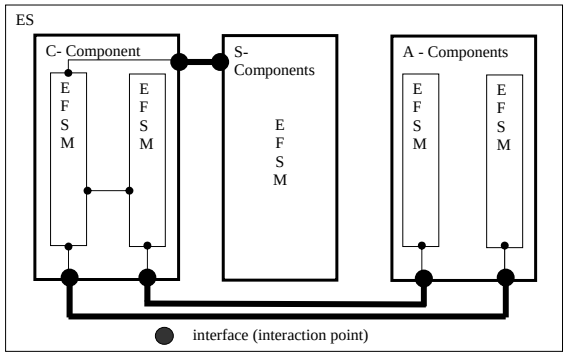



**Figure 3. Structuring and communication in an ES-component-based specification**

- A task may send an interaction through its interaction point to a task linked to it, e.g. from C to A via the interaction points in C and A which are linked to each other (Figure 3).
- An interaction received by a task, as its interaction points, is appended to a FIFO queue associated to this interaction point. A FIFO queue may be either associated to one interaction point (individual queue) or to many interaction points (common queue)

A task may export variables towards its parent which can access them (read and write).

Parallelism: Two kinds of parallelism can be expressed in a ES specification:

- Asynchronous parallelism: only between (actions of) tasks of different subsystems
- Synchronous parallelism: only between different (actions of) tasks of the same subsystem. Synchronous parallelism between actions means that all actions have to complete their parallel execution before other actions can be executed in parallel.

Time notion:

- Execution time of tasks (actions) is assumed unknown because it is implementation dependent.
- Some actions can be specified in such a way that their execution will be delayed. There are two delay values (min and max) which may be specified. The values of these delays are supposed to be modified by an independent process.

## 3.2 Syntax Overview

### *3.2.1 Channel Definition*

```
channel NAME (IP1, IP2)
  by IP1:
      interactionName1 (typed parameters);
      …;
  by IP2:
      interactionName1 (typed parameters);
      …;
  by IP1, IP2:
      interactionName1 (typed parameters);
      …;
```

A *channel* determines two channel types: Channel_NAME (IP1) type and Channel_NAME (IP2) type. A *channel type* defines two sets of interactions: those which can be sent through an interaction point (interface) of this type and those which can be received through an interaction point of this type (Figure 4).

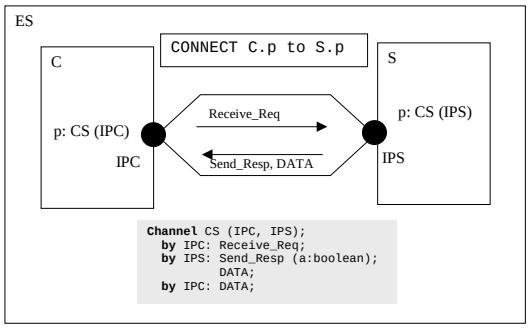



**Figure 4. Interactions via a channel in an ES-component-based specification**

### *3.2.2 Module Definition*

The module definition consists of three main parts: declaration part, initialization part and transition part. The declaration part defines the manipulated objects like constants, types, variables, functions or procedures, state, stateset, channels, (sub) module headers and bodies, module variables etc. The initialization part initializes and controls variables and state variables, creates subtasks (sub-module instances) and establishes communication links. The transition part is the most important because it specifies the embedded system behaviour.

### *3.2.3 Transition Part*

The transition part is composed of a set of transitions. Each transition has two parts: conditions and actions. Conditions are formed by the following clauses: `when, from, provided, delay, priority.` The actions are defined by the clause `TO` and PASCAL-Program with some extensions and restrictions:

```
WHEN clause
       when
interaction_point_id.interaction_name
FROM clause
       from state
       from stateset
PROVIDED clause
       provided Boolean expression
DELAY clause
       delay (integer_expression)
       delay (integer_expr1, integer_expr2)
TO clause
       to state
       to same
       output
```

It is easy to map a p-EFSM specification onto the behavioral part (transition part) of an Estelle module. The *when clause* corresponds to input events in p-EFSM, *from* to edge state, *provided* to predicate, *delay* is a timing special input event, *to* to the tail state and *output* to output event.

If the formal specification is provided in form of p-EFSMs (corresponding to a module) or in FDT many properties (completeness, correctness, consistency, safety, reachability etc.) of an embedded system can be automatically checked. In addition, different phases of the development process (analysis, implementation derivation, test data generation, diagnosis) can be unambiguisly and effectively supported [6] [7] [9].

## 3.3 Test Generation Methods

There exist many test generation methods that are based on FSMs and which can be under some assumptions adapted to

EFSMs [6] [7] [8]. Some of them are able to detect only certain errors classes, whereas other allow to cover all errors classes. All these methods based on FSMs have a common basic idea. A test sequence is a preferably short sequence of consecutive transitions that contains every transition of the FSM at least once and allows to check whether every transition is implemented as defined. To test a transition, one has to apply the input for the transition in the starting state of the transition, to check whether the correct output occurs, and to check whether the correct next state has been reached after the transition. Checking the next state might be omitted (transition tour method) or be carried out by means of distinguishing sequences (checking experiments method), characterizing sequences (W-method), or unique input/output sequences (UIO methods). Some of these methods were also extended to nondeterministic FSMs [7].

# 4. CONCLUSION

In this paper we presented an approach based on formal description techniques for specification of component-based embedded systems. The intermediate model EFSM allows to specify a system component independently of a given FDT, however, easily translatable in a preferred FDT, i.e. Estelle or SDL. We described the basic structure of embedded systems and demonstrate how a component-based approach can be applied for them using Estelle as FDT example. The main goal is to reuse the many well-known methods (automatic analysis, test data generation, validation, diagnosis, formal fault models) that have been since decades developed around state/transition-based models because formal approaches are very recommended in the today's growing complexity of embedded system requirements, especially regarding safety real-time property.

In a future work, we plan to specify a real-life embedded system from the automotive area by using FDTs.

# REFERENCES


[1] Specification and Description Language SDL '92. ITU-T Recommendation Z.100, 1992.

[2] Information processing systems – Open Systems Interconnection – *Estelle: A formal description technique based on an extended state transition model.* International Standard ISO 9074, 1989.

[3] Buessow, R., Geisler, R. and Klar, M. Specifying safety-critical embedded systems with statecharts and Z: A case study. *In Proceedings of Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, 1998.

[4] Mendler, M. and Luettgen, G. Statecharts: From Visual Syntax to Model-Theoretic Semantics. *In K. Bauknecht, W. Brauer, and Th. Mück (editors), Workshop on Integrating Diagrammatic and Formal Specification Techniques (IDFST 2001)*, pages 615-621, Vienna, 2001.

[5] Potter, B., Sinclair, J. and Till, D. Introduction to Formal Specification and Z (2nd Ed.). Prentice Hall PTR; 1996.

[6] Aho, A. V. et al. An optimisation technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours. *In S. Aggarwal and K. Sabnani, editors, Protocol Specification, Testing, and Verification*, New Jersey, 1988.

[7] Fujiwara, S. et al. Test selection based on finite state models. *IEEE transaction on Software Engineering* 17(6): 591-603, 1991.

[8] Richter, H., et al. A Concept For a Reliable, Cost-Effective, Real-Time Local-Area Network for Automobiles. *In Proceedings of Joint conference Embedded in Munich and Embedded Systems*, Munich, 2004.

[9] Henniger, O., Ulrich, A. and König, H. Transformation of Estelle modules aiming at test case derivation. *In A. Cavalli and S. Budkowski (eds.), 8th International Workshop on Protocol Test systems*, Chapmann & Hall, 1995.

[10] Crnkovic, I. Component-based approach for embedded systems. *Ninth International Workshop on Component-Oriented Programming*, Oslo, 2000.

[11] Beydeda, S. and Gruhn, V. Testing Component-Based Systems Using FSMs. In Beydeda and Gruhn (Eds.), Springer-Verlag, 263-280, 2004.

# A Categorical Characterization for the Compositional Features of the # Component Model

Francisco Heron de Carvalho Junior *
Departamento de Computação
Universidade Federal do Ceará
Bloco 910, Campus do Pici
Fortaleza, Brazil
heron@lia.ufc.br

Rafael Dueire Lins †
Departamento de Eletrônica e Sistemas
Universidade Federal de Pernambuco
Rua Acadêmico Hélio Ramos, s/n
Recife, Brazil
rdl@ufpe.br

## ABSTRACT

The # programming model attempts to address the needs of the high performance computing community for new paradigms that reconcile efficiency, portability, abstraction and generality issues on parallel programming for high-end distributed architectures. This paper provides a semantics for the compositional features of # programs, based on category theory.

## 1. INTRODUCTION

Due to the advent of clusters and grids, the processing power of large-scale distributed architectures is now accessible for a wider number of academic and industrial users, most of them non-specialists in computers and programming. This new context in high-performance computing (HPC) has brought new challenges to computer scientists. Contemporary parallel programming technologies that can exploit the potential performance of distributed architectures, such as message passing libraries like MPI and PVM, still require a fair amount of knowledge on the architecture and the strategy of parallelism used. This knowledge goes far beyond the reach of naive users [7]. The high-level approaches available today do not join efficiency with generality. The scientific community still looks for a parallel programming paradigm that reconciles portability and efficiency with generality and a high-level of abstraction [4].

The # component model attempts to meet the needs of the HPC community [5], by moving parallel programming from a process-based perspective to a concern-oriented one based on components, separating concerns of specification of computations from concerns related to their coordination, and providing a number of features for abstraction in topological composition of components. This paper provides a categorical [2] foundation for the # component model, focused on the semantics of its compositional features.

## 2. THE # COMPONENT MODEL

The # component model moves parallel programming from a *process-based* perspective towards a *concern-oriented* one. Without any loss of generality, in the former perspective, a parallel program may be seen as a set of processes that synchronize by means of communication channels. Application *concerns* [10] are scattered across the implementation of processes. In fact, a process may be decomposed in a set of *slices*, each one describing the role of the process with respect to a concern. In the latter outlook, ***components*** are programming abstractions that address concerns. In # programming, a component is described as a set of units organized in a network topology through synchronization channels that connect their interfaces. The interface of a unit comprises a set of input and output ports, whose activation order is dictated by a *protocol*, specified using a formalism with the expressiveness of labeled Petri nets. Component units have direct correspondence to processes slices. In fact, a # process is defined by the unification of a set of units from distinct components. Each unit from a component corresponds to a slice that describes the role of a process with respect to its concern. It is not difficult to see that *processes are orthogonal to concerns* (Figure 1) and that concern-oriented parallel programming fits better modern software engineering methodologies.

In # programming, the concerns about computations and the ones related to their coordination are separated in composed and simple components, respectively. Composed components comprise the ***coordination medium*** of # programs, while simple components comprise their ***computation medium***. Components may be combined with other components yielding new components, through *nesting* or *overlapping* composition. Nesting composition occurs when a simple/composed component is assigned to a unit of another composed component. Overlapping composition occurs when units from disjoint composed components are unified. Component models of today allow only nesting composition [1, 3] and does not support separation of cross-cutting concerns. The # model also brings the support to *skeletal programming* [6] to component-based programming. The essence of # programming is to provide compositional features for raising the level of abstraction in dealing with basic channel-based parallel programming. It is supposed that any parallel programming artifact may be defined in terms of # programming abstractions. This section provided only an overview of the # component model features. Details about compositional and abstraction issues in # programming will be given in the categorical semantics presented in the next sections, where concepts like *compositional interfaces* and *interface*

*Supported by FUNCAP and CNPq.
†Supported by CNPq.

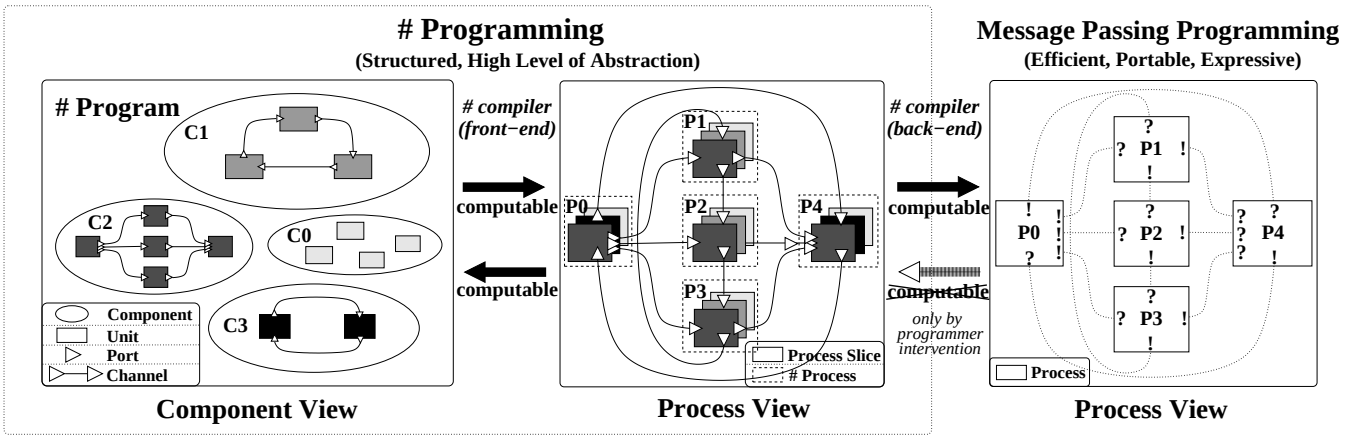



**Figure 1: Components versus Processes**

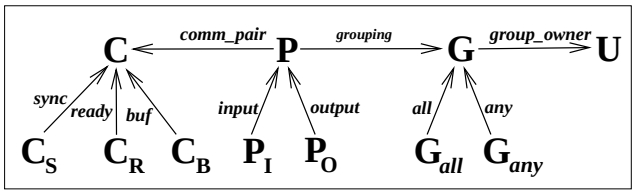



**Figure 2: $G_{\mathcal{H}}$ (Graph of Sketch $\mathcal{H}$)**

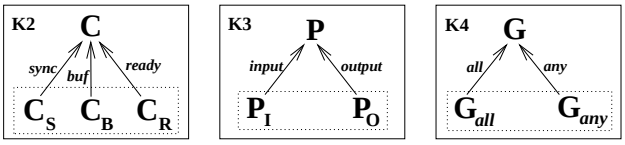



**Figure 3: $L_{\mathcal{C}}$ (Cocones of Sketch $\mathcal{C}$)**

*abstractions* are introduced.

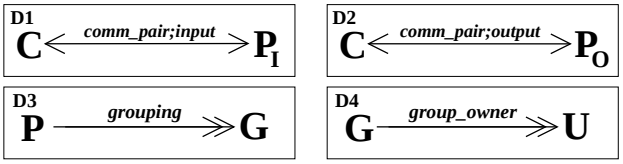



**Figure 4: $D_{\mathcal{C}}$ (Diagrams of Sketch $\mathcal{C}$)**

## 3. THE CATEGORICAL # MODEL

For better understanding of this section, knowledge about basic category theory and graph theory concepts are required. However, some intuition behind the introduced formal concepts will be provided whenever possible. The concepts of *sketch* [2] and *institution* [8] are the only advanced categorical concept employed herein. Sketches were firstly proposed for the specification of certain mathematical structures. They play the same role as traditional techniques from first-order logic and universal algebra, but they seem to be more appropriate for dealing with multi-sorted structures and with models in categories other than sets. Institutions were proposed by Goguen and Burnstal for providing a unified theory for algebraic specification systems, which in general differs by the underlying logical system used for expressing properties.

### 3.1 The Category of Units

The sketch $\mathcal{H}$ is defined by $(G_{\mathcal{H}}, D_{\mathcal{H}}, \emptyset, K_{\mathcal{H}})$. The graph $G_{\mathcal{H}}$, the diagrams $D_{\mathcal{H}}$, and the cocones $K_{\mathcal{H}}$ are presented in Figures 2, 4, and 3, respectively. A # component is defined by a *model* (sketch homomorphism) of the sketch $\mathcal{H}$ on SET, the category of sets, satisfying the commutative diagram in Figure 5, where UNIT is the category of units, defined further, and M, M' are # components. Therefore, a homomorphism $\mu$ between # components $M$ and $M'$ is a homomorphism of models, defined by a natural transformation $\mu : M \rightarrow M'$. The category of # components, named HASH, has components as objects and homomorphisms between components as morphisms. As usual for categories of functors (components are functors), the vertical composition of natural transformations defines composition in HASH.

Let $M$ be a component. $M(\mathbf{U})$ is a set of *units*. $M(\mathbf{G})$ is a set of references to *groups of ports*. $M(\mathbf{P})$ is a set of *ports*. $M(\mathbf{C})$ is a set of *communication channels*. $M(\mathbf{C_B})$, $M(\mathbf{C_R})$, and $M(\mathbf{C_S})$ are sets of *buffered*, *ready* and *synchronous* channels, respectively. The Cocone $K1$ states that $M(\mathbf{C_B})$, $M(\mathbf{C_R})$, and $M(\mathbf{C_S})$ are disjoint subsets of $M(\mathbf{C})$. $M(\mathbf{G}_{any})$ and $M(\mathbf{G}_{all})$ are sets of references to groups of ports of kind *any* and *all*, respectively. The Cocone $K3$ yields that $M(\mathbf{G}_{any})$ and $M(\mathbf{G}_{all})$ are disjoint subsets of $M(\mathbf{G})$. $M(\mathbf{P_I})$, and $M(\mathbf{P_O})$ are sets of *input* and *output* ports, respectively. The Cocone $K2$ states that $M(\mathbf{P_I})$, and $M(\mathbf{P_O})$ are disjoint subsets of $M(\mathbf{P})$. The function $M(\textbf{\textit{grouping}})$ associates ports to

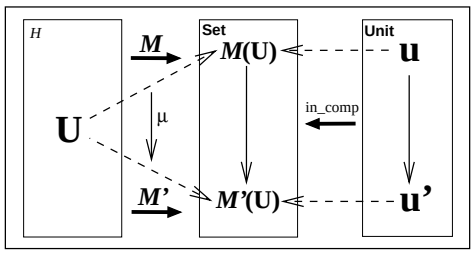



**Figure 5: Sketch Restriction**

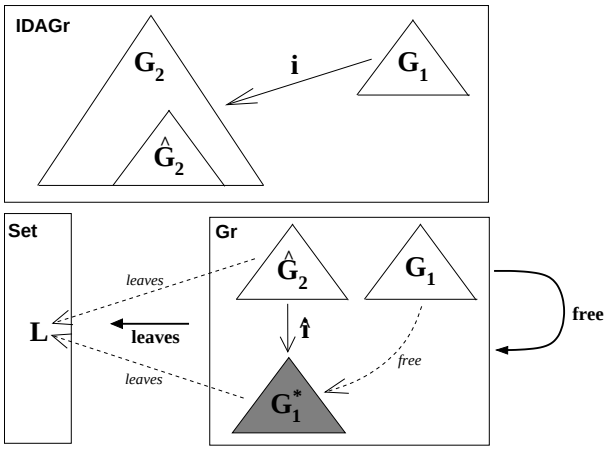



**Figure 6: The functor $i_G$**

their groups of ports. Diagram $D3$ is an epimorphism, forbidding empty groups. The function $M(\textbf{\textit{group\_owner}})$ defines the unit for which a group of ports belongs to. Diagram $D4$ ensures that it is an epimorphism too, forbidding units with an empty set of ports. The function $M(\textbf{\textit{comm\_pair}})$ defines the channel where a port is a communication pair. Together, $D1$, $D2$ and $K3$ ensure that channels are unidirectional. The restriction in Figure 5 ensures that the mapping between units induced by HASH-homomorphisms obeys morphisms in the category of units (UNIT).

The next sections define categories for *units* and *interfaces*. The relation between interfaces and units resembles the relation between signatures and algebras in universal algebra. For this reason, an *institution* will be employed for characterizing the relation between *interface signatures* and *units*.

## 3.2 The Category of Interface Signatures

The objects of category INTERFACESIG represent *interface signatures*. They are defined as tuples $\langle G, E, \nu \rangle$. $G$ is a finite, connected, directed, and acyclic graph $\langle V, A, \partial_0, \partial_1 \rangle$ with exactly one root node. Graphs of this kind are referred as IDAG's. *Leave nodes* represent ports and *branch nodes* represent *interface slices*. The category of IDAG's is IDAGR, with a special notion of morphism that will be defined further on. As usual, GR denotes the category of graphs. $E$ is the set of *exposed nodes* of G ($E \subset \textbf{nodes}(G)$). **Each path in $G$ must have exactly one exposed node**. $\nu : V \to \mathbb{N}$ is a total function that maps nodes of G onto a *stream nesting factor*.

Let $I_1$ and $I_2$ be interface signatures. A morphism $\imath : I_1 \to I_2$, in INTERFACESIG, is defined by a tuple $\langle i_G, i_E, i_\nu \rangle$. The IDAGR-morphism $i_G : G_1 \to G_2$ maps $G_1$ to a branch (also a IDAG) of $G_2$, called $\hat{G}_2$, in such way that there is an GR-morphism $\hat{\imath}_G : \hat{G}_2 \to \textbf{free}(G_1)$ that preserves leaves, e.g. $\textbf{leaves}(\hat{G}_2) = \textbf{leaves} \circ \textbf{free}(G_1)$. This is illustrated in Figure 6. The function $i_E$ maps exposed nodes from the interface signatures. It must satisfy $i_E(E_1) \subseteq E_1$, which is a sufficient condition to ensure the preservation of exposed nodes between $I_1$ and $I_2$. $i_\nu(\nu_1) = \nu_2$, satisfying the commutativity of the diagram in Figure 7 (preservation of stream nesting factors).

## 3.3 The Institution of Units

In terms of the theory of institutions, a unit is essentially a model for an interface signature, as well as $\Sigma$-algebras are models for an algebra signature $\Sigma$. Units augment interface signatures with a notion of *behavior*, defined by a protocol that generates a formal language whose alphabet is composed by the set of exposed nodes of the interface signature of the unit.

Let $I$ be an arbitrary interface signature. The functor $Sen$ : INTERFACESIG $\to$ SET maps $I$ to the set $E^*$ (Kleene closure of E). A unit is defined by a tuple $\langle P, R, \delta, \pi \rangle$, where $P$ is a set of ports, $R$ is a set of slice references, $\delta : P \to \{input, output\}$ is a total function defining direction of ports, and $\pi$ is a protocol. A protocol expression is defined by the syntactic class $\Pi$, whose definition is

$$\Pi_i ::= \textbf{seq}\ \{\Pi_1, \Pi_2, \ldots, \Pi_k\} \mid \textbf{par}\ \{\Pi_1, \Pi_2, \ldots, \Pi_k\} \mid \textbf{alt}\ \{\Pi_1, \Pi_2, \ldots, \Pi_k\} \mid p_i? \mid p_o! \mid \textbf{do}\ \mathrm{r} \mid s+ \mid s-$$

where $p_i \in \{p \in P \mid \delta(p) = Input\}$, $p_o \in \{p \in P \mid \delta(p) = Input\}$, $r \in R$, and $s$ is a semaphore symbol. The protocol combinators **seq**, **par** and **alt** denote respectively *sequence*, *concurrency* and *alternative*. The symbols ! and ? are the basic primitives for sending and receiving data in distributed synchronization channels, respectively. The primitive **do**, the *unfolding* primitive, denotes the protocol underlying the slice reference $r$. It can be viewed as a macro expansion operator. The symbols + and - mean the usual signal (V) and wait (P) semaphore primitives. Protocols of units may generate a terminal Petri net formal language on the alphabet $P$ [9]. The motivation for Petri nets is to build dynamic models for the behavior and interaction of units, making possible the analysis of formal properties and performance evaluation of programs using Petri net tools and their variants. The formal language generated by a protocol $\pi$ is denoted by $\Lambda(\pi)$.

A unit $U$ complies with an interface $I$ if the following condition holds: (1) $P = E \cap \textbf{leaves}(G)$ (ports are exposed leaf exposed nodes of G), and (2) $R = E - \textbf{leaves}(G)$ (slice references are exposed non-leaf nodes of G). The covariant functor $Mod$ : INTERFACESIG $\to$ CAT$^{op}$ maps each interface signature $I$ onto the category of units that complies with $I$.

Let $I$ be an interface signature. The relation $\models_I$: $Sen(I) \times Mod(I)$ associates units with the activation sequences supported by their protocols. For instance, let $w \in Sen(I)$ be a

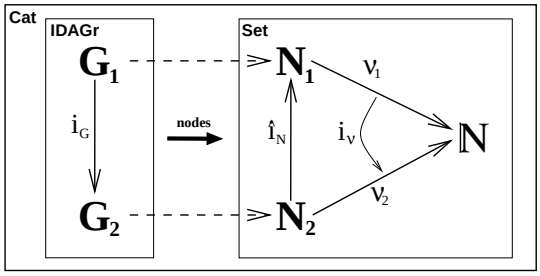



**Figure 7: Commutative Diagram for $i_\nu$**

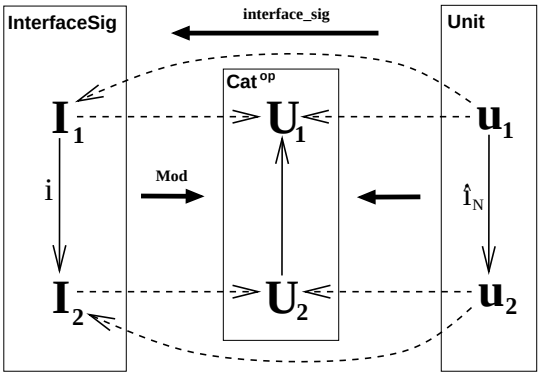



**Figure 8: The Category of Units**

word in $E^*$, $U$ be a unit, and $\pi$ be the protocol of $U$. Then, $U \models_I w$ iif $w \in \Lambda(\pi)$.

The category InterfaceSig, the functors $Sen$ and $Mod$ and the relation $\models_I$, as defined before, forms an **institution**. In fact, for some morphism $i : I \rightarrow I'$ in InterfaceSig, the required *satisfaction condition* $u' \models_{I'} Sen(i)(w) \Leftrightarrow Mod(i)(u') \models_I w$ holds for each $u' \in |\ Mod(I')\ |$ and for each $w \in Sen(I)$

## 3.4 The Category of Units

The *institution* of units splits the category of units in classes of units that complies to the same signature. Also, it naturally expresses the unusual **behavior preservation property between units**. The functor **interface_sig** : Unit $\rightarrow$ InterfaceSig maps units to their interfaces. The commutative diagram in Figure 8 must be satisfied.

## 3.5 Composition of Components

The category HashFDDiag has all discrete diagrams in Hash as *objects* and the graph homomorphisms between them as *morphisms*. Let $h : \mathbf{D}_1 \rightarrow \mathbf{D}_2$ be a morphism in HashFDDiag , and let $\langle C_1, C_2 \rangle$ be a pair of nodes in $\mathbf{D}_1$ and $\mathbf{D}_2$, respectively. Then $h(C_1) = C_2$ implies that $f : C_1 \rightarrow C_2$ is a morphism in HashFDDiag.

Let $\boldsymbol{D}$ be a HashFDDiag-object formed by $n$ components. $\boldsymbol{D}$ is the *start diagram* for overlapping them. The vertex of its co-limit is conventionally called $M_0^D$. For the sake of simplicity, $M_0$ may be used instead of $M_0^D$ whenever this does not cause confusion. $M_0$ is called *initial component* of $\boldsymbol{D}$, obtained from the disjoint overlapping of the components in $\boldsymbol{D}$. From $M_0$, new components are formed by applying the composition operations *unification*, *factorization*, *replication* and *superseding*. The functor ***overlap*** : HashFDDiag $\rightarrow$

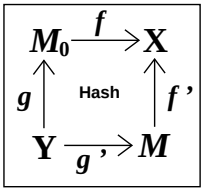



**Figure 9: A Commutative Diagram for *overlap***

Cat associates a diagram $\boldsymbol{D}$ with the sub-category of Hash containing all objects $M$ such that there are epimorphisms $f : M_0 \twoheadrightarrow X$, $f' : M \twoheadrightarrow X$, $g : Y \twoheadrightarrow M_0$, and $g' : Y \twoheadrightarrow M$, for some pair of objects $X$ and $Y$, such that the diagram in Figure 9 commutes.

# 4. CONCLUSIONS

This paper introduced a categorical interpretation for the compositional features of # component model. Further works will use the formal framework introduced herein for formalizing concepts and proving properties regarding the structure of # components. Another source of work is to study the expressiveness of component models, by mapping # components onto components from other component models using functors and natural transformations. In fact, this is the main motivation for adopting category theory as an underlying mathematical foundation.

# 5. REFERENCES

[1] R. Armstrong et al. Towards a Common Component Architecture for High-Performance Scientific Computing. In *The Eighth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society, 1999.

[2] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[3] F. Baude, D. Caromel, and M. Morel. From Distributed Objects to Hierarchical Grid Components. In *International Symposium on Distributed Objects and Applications*. Springer-Verlag, 2003.

[4] Bernholdt D. E. Raising Level of Programming Abstraction in Scalable Programming Models. In *IEEE International Conference on High Performance Computer Architecture (HPCA), Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, pages 76–84. Madrid, Spain, 2004.

[5] F. H. Carvalho Junior and R. D. Lins. The # Model for Parallel Programming: From Processes to Components with Insignificant Performance Overheads. In *Workshop on Components and Frameworks for High Performance Computing (CompFrame 2005)*, June 2005.

[6] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30:389–406, 2004.

[7] J. Dongarra, *et al. Sourcebook of Parallel Computing*. Morgan Kauffman Publishers, 2003.

[8] J. Goguen and R. Burnstal. Institutions: Abstract Model Theory for Specification and Programming. *Journal of ACM*, 39(1):95–146, 1992.

[9] T. Ito and Y. Nishitani. On Universality of Concurrent Expressions with Synchronization Primitives. *Theoretical Computer Science*, 19:105–115, 1982.

[10] H. Milli, A. Elkharraz, and H. Mcheick. Understanding Separation of Concerns. In *Workshop on Early Aspects - Aspect Oriented Software Development (AOSD'04)*, pages 411–428, March 2004.

# Specification and Design of Component-based Coordination Systems by Integrating Coordination Patterns[1]

Pedro L. Pérez-Serrano
QUERCUS Software Engineering Group,
Computer Science Department,
University of Extremadura
Escuela Politécnica, Avda. Universidad, S/N
10071 Cáceres (Spain)
+34-927257173
plperez@unex.es

Marisol Sánchez-Alonso
QUERCUS Software Engineering Group,
Computer Science Department,
University of Extremadura
Escuela Politécnica, Avda. Universidad, S/N
10071 Cáceres (Spain)
+34-927257807
marisol@unex.es

## ABSTRACT
Rewriting logic has been revealed as a powerful tool to represent concurrent and state-transitions aspects in a declarative way, providing an adequate environment to specify and execute system representations. Moreover, rewriting logic is reflective, allowing for the definition of operations that transform, combine and manipulate specification modules by making use of the logic itself. Taking advantage of these capabilities, this paper presents a set of tools based on the rewriting logic language Maude to express the specifications of component-based systems with important coordination constraints, where coordination aspects are treated as separate components from functional ones. This representation allows for the testing of the system behavior from the early stages in the development process by executing the specifications. In addition, the development of basic coordination patterns using UML is presented to describe the coordination relationships between components in any system, providing a standard notation that complements the tools of the proposal.

## Categories and Subject Descriptors
D.2.1 Requirements/Specifications (D.3.1)

D.2.2 Design Tools and Techniques

D.2.4 Software/Program Verification (F.3.1)

## General Terms

Performance, Design, Standardization, Languages, Verification.

## Keywords
Coordination Requirements, Behavior Simulation, Accordance Checker, Coordination Patterns.

## 1 INTRODUCTION
The need to develop more and more complex systems has enabled the improvement in languages and models to manage the coordination constraints between system components, promoting reusability, and the flexibility to change the interaction policies between components by means of the separate treatment of functional and coordination concerns. However, a serious limitation of these models, with regard to their usability, is that they do not provide support to manage the coordination constraints from the early stages in the software life cycle. That makes the adoption of a particular coordination model or language more difficult during the detailed design or implementation phases, due to the fact that coordination aspects are implicit and dispersed along all the components present in the system model, and now it is necessary to express explicitly the coordination aspect in a separate way from the functional one, to be able to apply a specific coordination model. Dealing with the specification of complex systems, coordination models should be encompassing a methodology that supports the separation of concerns throughout the whole software development process. Such methodology should be based on the use of formal techniques providing strictness and allowing the demonstration of properties that the specifications must satisfy.

Among the varieties of logics on which the formal specification techniques are based, rewriting logic has been revealed as a well suited base to express the system specifications in concurrent and state transition environments. Moreover, rewriting logic supports a wide spectrum of applications for developing prototypes, parallel execution and transformations. In addition, the reflective capability of this logic makes it a powerful tool to express the system specifications and to develop applications by transforming, checking and manipulating the logic itself.

In particular, this work focuses on the execution of specifications from the early stages of the life cycle, to validate and test the system behavior imposed by the coordination constraints between components. The use of Maude language [1] supporting rewriting logic is proposed. This choice is motivated by Maude's capabilities not only to specify and execute the system coordination requirements, but also to construct tools that manipulate the system specifications and transform them from a representation which is closer to designers to a more detailed and

[1] This work has been supported by the project CICYT under grant TIC 02-04309-C02-01.

complex representation that specifies the mechanisms needed to perform the coordination concerns.

This supposes accepting specifications which adopt the syntax of a coordination model, and which transform them into representtations detailing how the coordination mechanisms act to simulate the coordinated behavior by executing this last representation.

As the detailed representation can be refined along the software development process with features of design and implementation, a tool to check the accordance between representations of different abstraction levels is required. This tool manipulates specifications, and it is developed in the rewriting logic itself, making use of the reflective capability.

With these aims, we have developed COFRE (Coordination Formal Requirements Environment) [2], a set of tools considering all the above features, providing a methodology to make the system specification easy based on formal and graphic techniques, and dealing with coordination constraints from the early stages in the software development process. As regards benefits, changing and reusing components and coordination patterns from requirements are more systematic and pleasant tasks; in addition, the system behavior can be simulated by executing the formal specifications, simplifying the validation process, and the use of a model checker allows for the verification of the accordance between specifications in different abstraction levels.

In order to provide a standard notation to specify the system coordination constraints, we have developed a set of basic coordination patterns in UML that can be combined to express any kind of coordination constraints between the components of a system, even the most complex. Although these patterns are proposed to complete COFRE, they can also be used independently of this set of tools and the coordination model or language in which the system will be implemented.

The structure of this paper is organized as follows: In Section 2 the motivation for this work is presented. The steps of our proposal which makes use of a language based on rewriting logic to represent the system model along the software development process are described in Section 3. Section 4 gives an overview of the work in progress, describing the design of coordination patterns and their integration into the proposal. Finally, Section 5 provides the conclusions.

## 2 MOTIVATIONS

In recent years a wide range of tools combining both graphical and formal techniques have been developed with the aim of making software development easier. These tools can express in a detailed way the static and dynamic aspects of the system. But, initial requirements making use of these techniques are expressed in a global way which makes it difficult to adopt an architectural or design technique based on the separation of concerns (including coordination constraints). Particularly, in coordination environments, the adoption of a coordination language requires the separation of the specification of the coordination behavior from the functional one. But this task is delegated to designers or programmers starting from a conceptual model where these concerns are mixed. By avoiding this problem, the development processes are made more agile and consistent. Consequently, we proposed a set of tools named COFRE, based on the use of a rewriting logic language that tries to separate the functional concerns described for each component from concerns related to the interactions between components starting from the requirements definition in the development process.

Simulation by means of the execution of the model is the technique that best permits the observation and testing of the system dynamic properties. Often the simulation techniques by formal specifications execution, named *animation* techniques, required the translation of the specification to an imperative programming language like C++ or Java to be executed [10,11]. However, the different abstraction levels of the languages used to specify and to animate the model can provoke lack of precision and fidelity between both representations. This justifies the use of formal techniques allowing the execution of specifications to check the system behavior when implementation details have not yet been described.

Particularly, in coordination systems, special attention must be paid to guaranteeing that the final behavior obtained in the composed application is semantically coherent. That means verifying whether gluing together a coordination policy and a set of components in an application (which can have been coded and checked separately) will produce the expected behavior, and whether the addition or change of coordination constraints will produce conflicts with the current behavior.

Because the adoption of a coordination model means specifying the system constraints in a more detailed way, it is necessary to guarantee the accordance of this representation with regard to the initial requirements definition by means of a verification process checking that the interaction between the system components are maintained.

## 3 COFRE: SPECIFYING COORDINATION SYSTEMS

In this section, the above topics are focused on, proposing COFRE to make the specification and validation process easier for both software engineers and users. This proposal is based on the use of the formal language Maude to express the different representations of the system specifications and as a language in which the tools that manipulate and transform these specifications are implemented.

Maude is an executable algebraic language based on rewriting logic, that describes the specifications in a concurrent and non-deterministic way. The specifications can be executed by means of reducing terms in equations and rewrite rules performed by the interpreter provided by the language, which facilitates its use for prototyping and for checking the specifications behavior.

Maude allows for the definition of functional modules containing operations and equations, system modules also containing rewrite rules and object modules that are system modules allowing the specification of features concern to O-O paradigm. The use of rewrite rules is particularly appropriate to represent state transitions in systems of concurrent objects, where a configuration formed by the object instances and the current messages present in the system determine the system state in each moment. Maude provides specific definitions and operations to efficiently manage system configurations.

These are the capabilities that make the use of Maude appropriate to represent the different abstraction level specifications of the

system as well as to manipulate the specifications themselves, being the language in which part of the tools composing COFRE are implemented.

The methodology under COFRE proposes the use of IRD diagrams to represent the system components and their interactions from requirements analysis. IRDs have a corresponding specification in Maude language. However, this representation is not executable, because a specific coordination model needs to be adopted, but it is necessary to check the accordance of subsequent detailed specifications with regard to the initial requirements. COFRE adopts a specific coordination model and generates, starting from this initial specification, the equivalent specification making use of the coordination model syntax. This specification can be transformed in a more detailed Maude representation, expressing all the artifacts to represent the coordination aspects. This more detailed representation can be executed to simulate the system coordinated behavior [3], contributing to the system validation. Moreover, the specification adopting the coordination model can be verified with respect to the initial formal representation of the IRD to determine their accordance making use of the accordance checker, developed for this purpose. Figure 1 shows a schematic representation of the method.

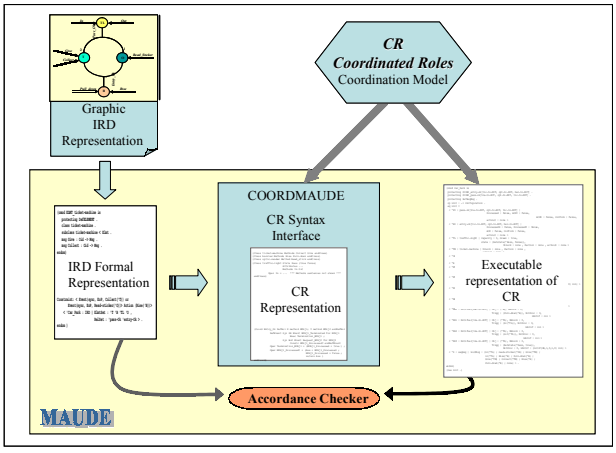



**Figure 1. Schematic representation of COFRE method steps**

Figure 1 shows the main steps of COFRE:

1. The main system components, their external interface and the interaction rules of the system are expressed using Interelement Relation Diagrams (IRDs). IRDs are used to specify the cooperation rules (coordinated interactions) between components in a graphical way.

2. The system IRD has a Maude representation allowing for verification of the agreement of a detailed Maude specification with regard to the original requirements expressed in the IRD.

3. The system specification can adopt a specific coordination model making use of its syntax, and this specification has a Maude representation of the coordination details permitting the model execution.

4. The behavior simulation of the system can be tested and validated after each iteration in the refinement of the development process.

5. The accordance with regard to the requirements expressed in the system IRD can be checked.

The transition to the design stage is made by adopting the exogenous coordination model Coordinated Roles (CR) [4]. CR is inspired by *IWIM* model [5] and based on the *Event Notification Protocols* (*ENP*) mechanism. This mechanism allows a coordinator component to ask for the occurrence of an event in another component, and the notifications can be asked for in a *synchronous* and an *asynchronous* way. The process must be transparent for the components to be coordinated.

Each coordination component imposes a coordination pattern structured as a set of roles. A role represents each of the characters that can be played in a coordination pattern. Behavior components will have to adopt these roles in order to be coordinated. For each role, coordination components specify the set of events required to represent the desired coordination constraints. The binding between coordinators and components to be coordinated is done at run-time via composition syntax.

# 4 COORDINATION PATTERN DESIGN

Design patterns [6] are common solutions accepted as being correct for specific design problems. They constitute an appreciated tool to improve of the quality of software development.

In order to take the advantage of using design patterns and applying them to the coordination aspects, we are working on the definition of a set of coordination patterns. These coordination patterns are specified using some diagrams of UML [7], to provide a standard representation that facilitates their use. We are starting with the definition of patterns representing basic coordination events and patterns representing the coordinators to develop the solution to problems in coordinated environments, in a level of abstraction independent of the programming language or the platform used.

The coordination patterns are developed with the aim of being widely applied, and with the purpose of being integrated in developing tools making use of UML. However, we propose to integrate coordination patterns in COFRE, to provide a standard notation that improves this set of tools and facilitates its use and its comprehension.

The coordination pattern integration is divided into three phases that are shown in figure 2:

**Phase 1:** This phase includes several steps:

- Specification of the events and the coordinators in a Class Diagram. Thus, it will define the static part of the system, the functional and singular aspects of each of its components of it. Also, the coordination aspects will be defined, that is, the system coordinated behaviour with the Interaction Diagrams of UML, composed of two kind of diagrams:
  - The Sequence Diagrams, showing the messages temporary ordination of the different objects in the system. The behavior and the role of the coordinator or coordinators in an action will be shown in these diagrams.

– The Collaboration Diagrams, showing the structural organization of the objects, that is, the set of messages sent and received between different objects in a collaboration.

- Generation of the formal definition of the system in Maude. Starting from this representation the sequence of steps proposed in COFRE can be applied to obtain a representation of the system in CoordMaude and to simulate the system behaviour.
- A detailed system specification is obtained independent of the platform and the programming language to be implemented. This solution will be saved in a repository together the documentation generated. The coordinated patterns are proposed to be reused and the documentation generated could be exploited to help in their reutilization.

**Phase 2:** This phase will consist of the development of a tool (black arrow number 1 in the figure 2) that transforms the Class and the Interaction Diagrams of Phase 1 to the IRD diagrams of COFRE. In this way, the development of the coordinated system is performed in the analysis phase with COFRE and in the design phase with the tools developed in phase 1. Thus, it will be necessary to develop a new tool (black arrow number 2 in figure 2) to check the accordance between the results of both phases.

**Phase 3:** In this phase, a tool will be developed to allow the inverse process to the explained one in phase 2; that is, a tool that converts the IRD diagram to the corresponding UML diagrams.

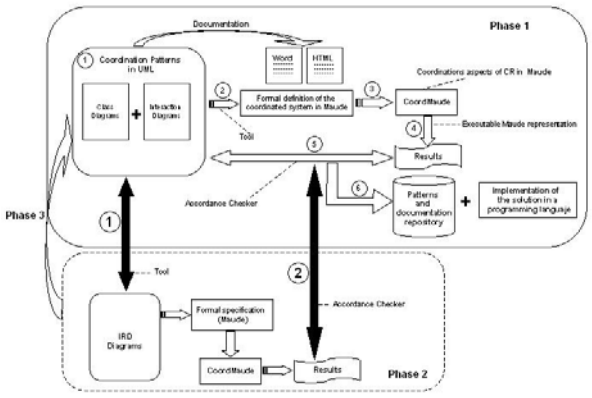



**Figure 2. Integration of the Coordination Patterns in COFRE**

# 5 CONCLUSIONS

The system requirements representation must be done performed making use of formal techniques allowing the validation of system requirements from the early stages in the development process, by executing specifications. This is especially important in coordination environments, where it is necessary to guarantee that interactions between system components work properly.

COFRE has been developed for this purpose, providing these advantages:

1. The coordinated interactions between components can be specified independently from component functionality By using IRDs. The formal representation of IRDs in Maude avoids ambiguity, adding precision to the model.
2. The adoption of a specific coordination model is facilitated from early stages in the development process due to the separation of functional and coordination concerns.
3. The features and event notification protocols are represented in Maude to make use of the rewrite engine of the language and with the aim to simulate the system behavior with different configurations.
4. The system representation making use of the CR syntax is possible, using *CoordMaude* that extends Maude, to accept specifications made on CR and generating all the mechanisms needed for the representation of the coordination model in Maude, in order to execute the system specifications.
5. Formal representation of IRDs can be checked with the specifications resulting from applying the coordination model to determine whether the accordance between both representations is maintained.

The definition of basic and generic coordination patterns in a standardized notation like UML allows the system specification to be expressed in an independent way from the tools used for the developing process and the coordination model adopted for the implementation of the system. The specification of complex coordination constraints can be made by combining basic coordination patterns, maintaining the separation of the coordination and the functional aspects of the systems. This separation contributes to making the software development and modification easier.

# 6 REFERENCES


[1] Clavel, M. Durán, F. Eker, S. Lincoln, P. Martí-Oliet N. Meseguer, J. and Quesada, J. Maude: Specification and Programming in Rewriting Logic. Computer Science Laboratory. SRI International. March'99.

[2] Sánchez-Alonso, M. Murillo, J.M. and Hernández J. COFRE: Environment for Specifying Coordination Requirements using Formal and Graphical Techniques. Journal of Research and Practice in Information Technology, Vol. (36) pp: 231-246, Australian Computer Society Inc. 2004.

[3] Sánchez-Alonso, M. and Murillo, J.M. Specifying Cooperation Environment Requirements using Formal and Graphical Techniques. In *WER'2000, 5th. Workshop on Requirements Engineering,* 2000.

[4] Sánchez-Alonso, M. Clemente, P.J. Murillo, J.M. and Hernández, J. CoordMaude: Simplifying Formal Coordination Specifications of Cooperation Environments. *2nd Workshop on Languages Description Tools and Applications (LDTA'03)*. ENTCS nº 82.

[5] Arbab, F. (1996): The IWIM Model for Coordination of Concurrent Activities.. *1st Int. Conf. on Coordination'96.* LNCS 1061. Springer-Verlag.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional, 1995.

[7] J. Rumbaugh, I. Jacobson, G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.