DEET for Component-Based Software

Murali Sitaraman
Durga P. Gandi
Computer Science
Clemson University
Clemson, SC 29634-0974, USA
+1-864-656-3444
murali@cs.clemson.edu

Wolfgang Küchlin
Carsten Sinz
Universität Tübingen,
W.-Schickard Institut für Informatik
Tübingen, Germany
+49-7071-29.77047
kuechlin@informatik.uni-tuebingen.de

Bruce W. Weide
Computer Science and Engineering
The Ohio State University
Columbus, OH 43210, USA
+1-614-292-1517
weide.1@osu.edu

Abstract

The objective of DEET (*Detecting Errors Efficiently without Testing*) is to detect errors automatically in component-based software that is developed under the doctrine of *design-by-contract*. DEET is not intended to be an alternative to testing or verification. Instead, it is intended as a complementary and cost-effective prelude. Unlike testing and run-time monitoring after deployment, which require program execution and comparison of actual with expected results, DEET requires neither; in this sense, it is similar to formal verification. Unlike verification, where the goal is to prove implementation correctness, the objective of DEET is to show that an implementation is defective; in this sense, it is similar to testing. The thesis is that if there is an error in a component-based software system either because of a contract violation in the interactions between components, or within the internal details of a component (e.g., a violated invariant), then it is likely—but not guaranteed—that DEET will find it quickly. DEET is substantially different from other static checking approaches that achieve apparently similar outcomes. Yet it builds on a key idea from one of them (Alloy): Jackson's *small scope hypothesis*. Among other things, the DEET approach weakens full verification of component implementation correctness to static checking for errors, in a systematic way that makes it clear exactly which defects could have been detected, and which could have been overlooked.

Keywords

Design-by-contract, error detection, SAT solvers, software component, specification, static analysis, static checking.

1. INTRODUCTION

This paper describes a new approach to detecting errors in component-based software that is developed using the popular paradigm known as *design-by-contract*, and presents results from early experience with a prototype tool. We call the approach DEET for *Detecting Errors Efficiently without Testing*. DEET has the potential to be effective and efficient, and the potential to "scale up" to large component-based software systems. It is intended to offer the following important benefits over early testing:

- DEET can analyze one component at a time in a modular fashion, i.e., it can detect mismatches between a component implementation and its contract, even in isolation from the rest of a component-based system.
- Since DEET does not require program execution or inlining of called procedures, it does not depend on code or even stub availability for other components.
- DEET can detect substitutability bugs, i.e., contract violations that are literally undetectable by testing. Such bugs arise from situations where a particular implementation of a component requires less or delivers more than its contract specifies, and where the correctness of the larger system relies on such incidental behavior of that particular implementation.
- DEET is automated and does not require manual input selection.
- When an error is detected, DEET can pinpoint the origin of the error in the source code. In particular, it can detect internal contract violations among participating components in a larger system—and assign blame. This property makes it suitable for debugging component-based software.

DEET bears some resemblance to other static analysis/checking tools[1], e.g., Alloy [38] and ESC [18]. Section 2 explores connections with these two systems in particular. Section 3 explains the steps of the DEET approach with a detailed example. Section 4 discusses other related work, and Section 5 summarizes the paper.

2. ESC, ALLOY, AND DEET

From the synopsis of features given in the introduction, it may appear that DEET is essentially the same as ESC or Alloy—two well-known efforts in the same general direction. In fact, while DEET shares some common objectives with these approaches, it is complementary in nearly every respect, as explained in this section. Only one technical detail from these systems has been consciously adapted for use in DEET: Jackson's *small scope hypothesis* [37], which is discussed in Section 3.2.3.

2.1 Objectives, Context, and Assumptions

ESC and Alloy seek incremental improvements to current software engineering practice, focusing on "real" languages

[1] But should not be confused with N, N-diethyl-m-toluamide. ABC News (7/3/02) summarized a *New England Journal of Medicine* report as follows: "DEET Is Best Bug Repellent."

and "real" programmers—constraints that impose technical complications which have proved unexpectedly troublesome. The defects that can be detected in practice with these tools are limited by (1) the failure of the "real" programming language to ensure by fiat certain desirable properties of programs, to prevent programming practices that complicate reasoning about software behavior, and in general to have semantics that facilitates modular reasoning; (2) the assumption that "real" programmers are unwilling or even unable to write full specifications of intended functional behavior, and that they will write only certain kinds of annotations that capture part of that intent; and (3) an interest in a tool that can be used with any software—component-based or not—that can be written in the "real" language.[2]

By contrast, DEET is part of a long-term plan to explore the foundations of future software engineering practice as it *could* be. The overall project goal is not to live within the shackles of current practice, but rather to remove them. DEET's context includes (1) a combined specification and implementation language (Resolve [66]) that is expressly designed to support modular reasoning, while still permitting the development of "real" software by strictly disciplined use of "real" languages such as C++ [34]; (2) a recognition, based on teaching experience [71], that tomorrow's software engineers can be taught to understand and even to write formal-language specifications, just as they can be taught to write formal-language implementations; and (3) a focus on component-based software. The project vision is to have an automatic verifier for functional (and performance) correctness of component-based systems. The purpose of an intermediate tool such as DEET is for software engineers to find errors quickly before attempting full verification, as this is likely to remain more efficient than full verification and hence potentially give real-time feedback.

2.2 References and Aliasing

ESC deals with, among other things, "nil-dereference errors" [18]. Among other things, it introduces a "downward closure" rule for *modifies* clauses in contracts. This is used to account for situations where aliases to instance fields of a class could impact an object's abstract state via unintended side effects. This, in turn, leads to a "rep visibility requirement" and another annotation construct, the *depends* clause, that is "a key ingredient of our solution to the problem" [18]. Nonetheless, it is admitted that "one problem in this area that has stumped us is a form of rep exposure that we call abstract aliasing"[18]; see also [17]. The potential for aliasing technically does not always prevent modular reasoning, but the above measures help illustrate that it seriously complicates matters [79].

The Alloy approach "targets properties of the heap" [77] to detect errors in implementations of linked data structures and null dereferences. The extent to which the Alloy approach can scale up to other properties remains an open question: "We expect that the tool will work well for modular analysis of even quite complex classes; how well it scales for analyses amongst classes and whether it will be economical enough for everyday use remains to be seen" [42].

Resolve has value semantics for all variables; there is no aliasing because the language does not permit it [49]. Resolve includes reference-free abstractions for lists, trees, etc., and programs that use these components are not burdened with the complication of reasoning about references or aliasing. However, it remains possible to write programs for situations where explicit aliasing improves efficiency and would be exploited in a language like Java. Using specifications of pointer-like behavior [47], it is possible to reason about these programs formally and to find errors in them using the DEET approach (although the current prototype does not handle this). Techniques used in ESC and/or Alloy to deal with aliasing may prove helpful in such situations, but our initial focus is on typical Resolve programs, in which pointers are not needed or used. And in any case, the use of value semantics for all variables distinguishes DEET from ESC and Alloy.

2.3 Undefined or Invalid Variable Values

ESC reportedly has been successful in detecting failure-to-initialize defects. Indeed, this seems to be one of its primary uses: "… our experience has been that many ESC verifications can be successfully completed with almost no specifications at all about the contents and meanings of abstract types, other than the specification of validity" [18].

The Alloy approach is tied to the Alloy Annotation Language [42] and "is designed for object model properties: namely what objects exist, how they are classified grossly into sets, and how they are related to one another. It is not designed for arithmetic properties…" [37].

In Resolve, every variable has an initial value upon declaration. A variable is never "undefined" or "invalid" and there is no question about whether it "exists", so there is no need for DEET to detect such errors. DEET, rather than avoiding specifications of "the contents and meanings of abstract types" and "arithmetic properties" (of arithmetic types), is intended to find defects related to fully specified component behavior.

2.4 Contracts and Other Assertions

ESC includes contract specification syntax, and most reported examples involve these constructs. However, ESC does not treat such assertions as complete contract specifications, but merely as partial statements of intent (e.g., as specifying only the property that a variable has been initialized). Moreover, ESC seeks to avoid making programmers write loop invariants, for example, because they can be "pedagogical and heavy-handed" [18] and sometimes can be produced automatically. For instance, in a definite **for** loop whose index i ranges from 1 to 10, the invariant $1 \leq i \leq 10$ can be generated by a tool, allowing ESC to detect some range errors without the programmer having to write a loop invariant.

Alloy requires no annotations beyond the property to be checked, and the assertions it checks are not necessarily parts of formal contracts, although the authors "are hopeful that it will extend to the analysis of code in terms of abstract sets and relations specified in an API" [37]. So, the current Alloy approach "expands calls inline" [77] rather than relying on contract specifications, hence requires special translation to handle recursive calls. Alloy relies on loop-unrolling to avoid the need for programmer-supplied loop invariants.

DEET expects full contract specifications for components, and additional internal assertions inherent in the Resolve syntax: loop invariants, representation invariants, and abstraction relations. This is necessary to study the impact of having

[2] ESC handles not just sequential programs but a class of synchronization errors in multi-threaded programs. Alloy and DEET so far are limited to sequential programs.

complete specifications on the quality of checking that can be achieved. If it turns out that experience with DEET suggests that future software engineers would be better off by learning to write specifications than by avoiding them, then it becomes our obligation to teach them how to write specifications. Moreover, because we focus explicitly on component-based software, DEET reasonably expects component libraries to have the specifications and internal assertions needed for full verification. This is because the extra cost of developing these annotations can be amortized over many component uses. The Resolve component catalog is an existence proof that a non-trivial library of fully specified components can be developed.

2.5 Soundness and False Alarms

All the tools here might fail to detect errors. However, they differ in why this is so. In ESC, there is no characterization of which errors might have eluded detection. Moreover, ESC is unsound in the usual logical sense because "verification condition generation is unsound". The verification condition produced is fed to ESC's own refutation-based general theorem prover that is claimed to be "sound, as far as we know". ESC can also produce false alarms, or "spurious warnings". The claim is that neither of these apparent technical shortcomings is a big problem if, on balance, the system in practice finds interesting classes of defects that actually do exist [18].

The idea of Alloy is that if an error is not detected, it is because the "scope", or subset of situations considered in the analysis, has been limited. Specifically, the Alloy approach generates a composite verification condition, and then limits the number of heap cells for each type and the number of loop iterations for each loop to turn this condition into a propositional formula that can be fed to a back-end SAT-solver, which tries to refute it. Any given error might have a witness only outside this scope. Alloy is designed not to give false alarms, i.e., it is not supposed to report errors where none exist [37].

DEET is much like Alloy in this regard, with two major exceptions. First, the soundness of verification condition generation has been established for most constructs of Resolve [22, 32, 67, 68]. The overall approach is still that the verification condition needed for full verification is generated from the relevant specifications and code. But then the scopes of all variables are restricted—not based on the number of heap cells in a data representation, but on the possible abstract mathematical model values for the variables involved. As with Alloy, this allows the verification condition to be recast as a propositional *error hypothesis* that can be fed to a back-end off-the-shelf SAT-solver in an effort to produce a witness to a defect. So DEET, like Alloy, can fail to detect an error if there are no witnesses to that error in the analyzed scope. But DEET does not report errors in code that could be verified as correct.

In overall structure and in many technical details, DEET seems to be a closer cousin of Alloy than of ESC. Nonetheless, there are other significant technical differences between DEET and Alloy. For example, DEET's verification condition generator automatically accounts for the "path conditions" associated with various execution paths. The Alloy approach is based on generating a control flow graph for the program to account for the various execution paths, and is cleverly optimized to do this [77]. A proposal in [42] suggests that loop-free code with method invocations can be handled by generating verification conditions using a logic similar to that used in ESC [17].

Two other differences between DEET and its predecessors are important. In the process of looking for errors, DEET generates the verification condition that would be needed to prove correctness. Proofs of this assertion can be attempted with human-assisted theorem provers (e.g., PVS [60]) when DEET finds no errors. And using the foundations for an extended system for specification and verification of performance (both time and space) [46, 69, 70], in principle DEET might be extended to detect errors relative to performance contracts.

3. DEET APPROACH

This section explains the DEET approach. As in [37], we choose a simple list example to explain in detail, because only with a concise example is it possible to illustrate concretely the variety of technical issues involved. The example helps to demonstrate the additional advantages of both the DEET and Alloy approaches over testing alone. Finally, it highlights some key differences between the DEET and Alloy approaches, because it involves recursive code that is a client of a *List* component contract, rather than being an implementation of a list method that has direct access to the data representation.

```
Concept List_Template( type Entry );
  uses Std_Integer_Fac, String_Theory;

  Type List is modeled by (
      Left, Right: Str(Entry)
    );
    exemplar P;
    initialization
      ensures |P.Left| = 0 and |P.Right| = 0;

  Operation Insert( alters E: Entry;
                    updates P: List );
    ensures P.Left = #P.Left and
      P.Right = ⟨#E⟩ * #P.Right;

  Operation Remove( replaces R: Entry;
                    updates P: List );
    requires |P.Right| > 0;
    ensures P.Left = #P.Left and
      #P.Right = ⟨R⟩ * P.Right;

  Operation Advance ( updates P: List );
    requires |P.Right| > 0;
    ensures |P.Left| = |#P.Left| + 1 and
      P.Left * P.Right = #P.Left * #P.Right;
  …
end List_Template;
```

Figure 1: A Specification of *List_Template*

3.1 Example: A Defective Implementation

Figure 1 shows a skeleton of a contract specification for a *List_Template* component in a dialect of Resolve [66]. In the specification, the value space of a *List* object (with position) is modeled mathematically as a pair of strings of entries: those to the "left" and those to the "right" of an imaginary "fence" that separates them. Conceptualizing a *List* object with a position makes it easy to explain insertion and removal at the fence. A sample value of a *List* of *Integer*s object, for example, is the ordered pair (<3,4,5>, <4,1>). Insertions and removals are explained as taking place between the two strings, specifically at the left end of the right string.

Formally, the declaration of type *List* introduces the mathematical model and, using an example *List* variable *P*, states that both the left and right strings of a *List* are initially empty. A *requires* clause serves as an obligation for a caller, whereas an *ensures* clause is a guarantee from a correct implementation. In the ensures clause of *Insert*, for example, *#P* and *#E* denote the incoming values of *P* and *E*, respectively, and *P* and *E* denote the outgoing values. The infix operator * denotes string concatenation, the outfix operator ⟨•⟩ denotes string construction from a single entry, and the outfix operator |•| denotes string length.

```
Enhancement Reversal_Capability for
                          List_Template;
  Operation Reverse( updates P: List );
    requires |P.Left| = 0;
    ensures P.Left = Rev(#P.Right) and
             |P.Right| = 0;
 end Reversal_Capability;
```

Figure 2: Specification of a List Reversal Operation

```
Realization Recursive_Realiz for
                    Reversal_Capability;
  Recursive Procedure Reverse(
                          updates P: List );
    decreasing |P.Right|;
    var E: Entry;
    if ( Right_Length(P) > 0 ) then
      Remove(E, P); Reverse(P); Insert(E, P);
    end;
  end Reverse;
end Recursive_Realiz;
```

Figure 3: A Defective Implementation of *Reverse*

An interesting aspect of the *Insert* specification is that its behavior is relational. The semantics of *alters* mode for the formal parameter *E* is that the result value of entry *E* is undetermined. This under-specification allows implementations not to have to make expensive copies of non-trivial type parameters, which is an important issue in the design of generic abstractions. It is well known that copying references, while efficient, introduces aliasing and complicates reasoning [33, 49, 79]. The present specification is more flexible. It allows the entry to be moved or swapped into the container structure (efficiently, i.e., in constant time, by manipulating references "under the covers") and thus potentially to alter it, without introducing aliasing [30]. Correspondingly, the *Remove* operation is specified to remove an entry from *P*, and it *replaces* the parameter *R*. Operation *Advance* allows the list insertion position (fence) to be moved ahead. The rest of the specification is in [68]; but it is not needed to understand this example.

Figure 2 contains the specification of an operation to reverse (the right string of) a list. Here, *Rev* denotes the mathematical definition of string reversal. Figure 3 shows an (incorrect) recursive implementation. It uses the *List* operations given in Figure 1. To demonstrate termination, the recursive procedure has a progress metric using the keyword *decreasing*.

3.2 DEET Steps to Detect Errors

3.2.1 Generation of a Symbolic Reasoning Table

As a first step in modular static analysis—either to prove correctness or to find errors—a *symbolic reasoning table* is generated [68]. The soundness and relative completeness of the approach that justifies this step are established in [32]. Figure 4 contains a table for the code in Figure 3. A key observation is that this table can be produced mechanically from the information in Figures 1, 2, and 3, as explained in [32, 68] and summarized below. In the table, each program *State* is numbered. For each state, the *Assume* column lists verification assumptions and the *Confirm* column lists the assertions to be proved. The *Path Condition* column denotes under what condition a given state will be reached.

Reasoning table generation involves the profligate use of variable names, because each program variable name is extended with the name of the state to denote the value of the variable in that state. *P1*, for example, denotes the value of variable *P* in state 1. To prove that the procedure for *Reverse* is correct, we assume that its precondition is true in the initial state and must confirm that its postcondition is true in the final state. For modular analysis, we rely only on the behavioral contracts of the called operations (i.e., *Insert* and *Remove*). In particular, for the calling code to be correct, we must be able to confirm that the precondition of a called operation is true in the state before the call; then we may assume that the postcondition is true in the state after the call. The recursive call to *Reverse* is treated just like any other call. However, before the recursive call, we additionally need to confirm that the progress metric decreases.

State	Path Condition	Assume	Confirm
0		|P0.Left| = 0	
	`if ( Right_Length(P) > 0 ) then`		
1	|P0.Right| > 0	P1 = P0	|P1.Right| > 0
	`Remove(E, P);`		
2	|P0.Right| > 0	P2.Left = P1.Left ∧ P1.Right = <E2> * P2.Right	|P2.Left| = 0 ∧ |P2.Right| < |P0.Right|
	`Reverse(P);`		
3	|P0.Right| > 0	E3 = E2 ∧ P3.Left = Rev(P2.Right) ∧ |P3.Right| = 0	
	`Insert(E, P);`		
4	|P0.Right| > 0	P4.Left = P3.Left ∧ P4.Right = <E3> * P3.Right	
	`end;`		
5.1	|P0.Right| = 0	P5 = P0	P5.Left = Rev(P0.Right) ∧ |P5.Right| = 0
5.2	|P0.Right| > 0	P5 = P4	

Figure 4: A Reasoning Table for the *Reverse* Procedure

The path condition in a given state serves as an antecedent for the implications that are the actual assertions to be assumed

and confirmed in that state. In other words, assume/confirm entries apply only when the path condition holds.

3.2.2 Generation of Error Hypotheses

To prove the correctness of the code, then, entails confirming each obligation in the last column, using the assumptions in the states above and including the state where the obligation arises (but, critically for soundness, not the states below it in the table [32]). Rather than attempting the non-trivial process of verification using a general theorem-proving tool, DEET instead looks for a witness to a bug in the code. In particular, it attempts to find values for the variables that satisfy all relevant assumptions but that fail to satisfy something that needs to be confirmed. This is done by conjoining the assumptions and the negation of the assertion to be confirmed, and then seeking a satisfying assignment for the variables in this *error hypothesis*—a witness to a bug.

To illustrate the idea, consider the assertions that need to be confirmed in state 5 (arising from the postcondition of *Reverse*). In particular, consider the recursive case when the path condition $|P0.Right| > 0$ holds. The code is defective if there is a set of assignments to the variables that satisfies the assertion in Figure 5. In the figure, the conjunct numbered I is the path condition, conjuncts II through VII are assumptions from states 0 through 5, and conjunct VIII is the negation of the assertion to be confirmed in state 5.

Error hypothesis generation also can be mechanized. There are four error hypotheses for the present example, one each corresponding to the confirm clauses in states 1 and 2, and two for state 5 (one for the base case 5.1 and one for the recursive case 5.2). If a satisfying assignment exists for an error hypothesis arising from an intermediate state (e.g., state 1 or 2 here), then the code fails to live up to its part of the contract for an operation it calls. It is possible that the error hypothesis arising from the final state at the end of the code (in state 5 in the table) cannot be satisfied, even though intermediate errors (e.g., violation of preconditions of called operations) are found. The code still should be deemed defective under design-by-contract because the calling code violates a requirement of a called operation.

```
(|P0.Right| > 0) ∧
  I
(|P0.Left| = 0) ∧
  II
(P1 = P0) ∧                                  III
(P2.Left = P1.Left ∧
  P1.Right = <E2> * P2.Right) ∧              IV
(E3 = E2 ∧ P3.Left = Rev(P2.Right) ∧
  |P3.Right| = 0) ∧                          V
(P4.Left = P3.Left ∧
  P4.Right = <E3> * P3.Right) ∧              VI
(P5 = P4) ∧                                  VII
(¬ (P5.Left = Rev(P0.Right) ∧
  |P5.Right| = 0))                           VIII
```

Figure 5: Error Hypothesis for Confirm Clause 5.2

3.2.3 Restriction of Scope

The search for a witness to an error hypothesis relies on Jackson's small scope hypothesis (where "scope" is, loosely speaking, a measure of the size of the input space to be searched). Jackson notes that even though, for any given scope, one can construct a program with a bug whose detection requires a strictly larger scope, in practice, many bugs will be detectable in small scopes [37]. If a bug is found within a small scope, then the code is not consistent with the verification conditions. If none is found in the given scope, then there are no inconsistencies in that scope; yet, inconsistencies might exist in a larger scope.

For DEET, we have explored restricting the scopes of participating variables by restricting their mathematical spaces, instead of placing bounds on loop iterations or heap cells. It is reasonable to begin with the most stringent restrictions. In the example, for instance, we start by looking for a witness to the error hypothesis in which all variables of type *Entry* have exactly one value, and in which strings of type *Entry* are either empty or contain just a single *Entry* with that value. Without loss of generality, we use *Z0* to stand for the single value of type *Entry*. This in turn restricts the scope of the search for strings to the two-element set {*Str_Empty*, *Str_Z0*}, where *Str_Empty* denotes the empty string and *Str_Z0* denotes the string <*Z0*>.

These restrictions on scope lead to a (possibly large, but finite) propositional formula corresponding to each error hypothesis generated from the code and the specifications, e.g., the one in Figure 6. Each satisfying assignment for this formula identifies a particular witness to a particular error hypothesis. To conserve space, we have shown only a part of the formula to use as a means of explaining how it can be generated. In the conjuncts listed in Figure 6, the names of all (Boolean) variables can be generated automatically. The variable *P0_Left_equals_Str_Empty* being true, for example, denotes that the left string of the program variable *P* in state 0 is equal to the empty string. In addition to the variables that correspond directly to the symbols in Figure 5, variable names corresponding to mathematical expressions involving string length, reverse, and concatenation are needed as well. Given this, the first two conjuncts in Figure 6 correspond directly to those in Figure 5.

To assert that *P1* = *P0* (conjunct III in Figure 5), the formula has to assert that the left strings of the two lists are equal and that the right strings are equal. However, each string may have only one of two values because of scope restriction: *Str_Empty* or *Str_Z0*. The left strings of *P0* and *P1* will be equal if they are both *Str_Empty* or if they are both *Str_Z0*. This observation leads to conjuncts in III in Figure 6. The rest of the conjuncts are derived similarly. A list of additional conjuncts needs to be generated to complete the propositional formula generation, and only some of these additional conjuncts are shown in Figure 6. For example, we need to assert that the right string of a list cannot be both empty and contain a single entry (although it could be longer), i.e.:

```
(¬ P0_Right_equals_Str_Empty ∨
 ¬ P0_Right_equals_Str_Z0)
```

The formula needs to make this assertion for the left and right strings of each *List* variable in each state. Another set of assertions is based on mathematical string length, e.g.:

```
(Len_P0_Right_equals_Zero ⇔
 P0_Right_equals_Str_Empty)
```

Other sets of assertions are generated for string reversal and concatenation within the restricted scope. Notice that similar

conjuncts for, e.g., reversal of the left string of a list, are not generated because they do not arise in the conjuncts corresponding to the assertions in Figure 5. The complete formula is at:

http://www.cs.clemson.edu/~resolve/reports/RSRG-03-05.pdf

```
(¬Len_P0_Right_equals_Zero)                          I
( Len_P0_Left_equals_Zero)
  II
((P1_Left_equals_Str_Empty ∧
P0_Left_equals_Str_Empty)                            III
     ∨ (P1_Left_equals_Str_Z0 ∧
P0_Left_equals_Str_Z0)) ∧
  ((P1_Right_equals_Str_Empty ∧
P0_Right_equals_Str_Empty)
     ∨ (P1_Right_equals_Str_Z0 ∧
P0_Right_equals_Str_Z0))
((P2_Left_equals_Str_Empty ∧
P1_Left_equals_Str_Empty)                            IV
     ∨ (P2_Left_equals_Str_Z0 ∧
P1_Left_equals_Str_Z0)) ∧
  ((P1_Right_equals_Str_Empty ∧
    Cat_E2_P2_Right_equals_Str_Empty)
       ∨ (P1_Right_equals_Str_Z0 ∧
            Cat_E2_P2_Right_equals_Str_Z0))
(E3_equals_Z0 ∧ E2_equals_Z0) ∧                      V
  ((P3_Left_equals_Str_Empty ∧
        Rev_P2_Right_equals_Str_Empty)
     ∨ (P3_Left_equals_Str_Z0 ∧
            Rev_P2_Right_equals_Str_Z0)) ∧
  (Len_P3_Right_equals_Zero)
((P4_Left_equals_Str_Empty ∧
P3_Left_equals_Str_Empty)                            VI
     ∨ (P4_Left_equals_Str_Z0 ∧
P3_Left_equals_Str_Z0)) ∧
  ((P4_Right_equals_Str_Empty ∧
      Cat_E3_P3_Right_equals_Str_Empty)
     ∨ (P4_Right_equals_Str_Z0 ∧
            Cat_E3_P3_Right_equals_Str_Z0))
((P5_Left_equals_Str_Empty ∧
P4_Left_equals_Str_Empty)                            VII
     ∨ (P5_Left_equals_Str_Z0 ∧
P4_Left_equals_Str_Z0)) ∧
  ((P5_Right_equals_Str_Empty ∧
     P4_Right_equals_Str_Empty)
     ∨ (P5_Right_equals_Str_Z0 ∧
P4_Right_equals_Str_Z0))
(¬ ((( P5_Left_equals_Str_Empty ∧
  VIII
      Rev_P0_Right_equals_Str_Empty) ∨
     (P5_Left_equals_Str_Z0 ∧
        Rev_P0_Right_equals_Str_Z0)) ∧
     (Len_P5_Right_equals_Zero)))
```

Additional Assertions

Unique Values (sample: P0.Right)

```
(¬ P0_Right_equals_Str_Empty ∨
 ¬ P0_Right_equals_Str_Z0)
```

String Length (sample: |P0.Right|)

```
(Len_P0_Right_equals_Zero ⇔
 P0_Right_equals_Str_Empty)
```

String Reverse (sample: Rev(P0.Right))

```
(Rev_P0_Right_equals_Str_Empty ⇔
 P0_Right_equals_Str_Empty) ∧
(Rev_P0_Right_equals_Str_Z0 ⇔
 P0_Right_equals_Str_Z0)
```

*String Concatenate (sample: <E2> * P2.Right)*

```
(¬ Cat_E2_P2_Right_equals_Str_Empty) ∧
(Cat_E2_P2_Right_equals_Str_Z0 ⇔
     (E2_equals_Z0 ∧
      P2_Right_equals_Str_Empty))
```

Figure 6: Selected Conjuncts Corresponding to Figure 5

The number of variables in the formula is bounded by the product of the size of the restricted scope, the number of program variables and expressions in the original verification conditions, and the number of rows in the tracing table (i.e., the number of lines of code). The number of conjuncts depends on the mathematical models and the assertions involved, along with the number of generated variables.

3.2.4 Error Detection

The example illustrates that the formulas generated during this process are not in conjunctive normal form (CNF). We do not convert them to CNF, but rather apply a SAT-solver that can handle arbitrary propositional formulas [41]; other state-of-the-art SAT solvers such as BerkMin [28] or Chaff [54] could be used by converting the formulas to CNF. The solver we have used, developed by the co-authors at Tübingen, is based on a Davis-Putnam-style [16] algorithm. It can handle formulas involving several thousand variables. For example, when the formula in Figure 6 was (translated into the required input format and) supplied to this solver, it produced the assignment given in Figure 7 within a fraction of a second. In addition, it concluded that this is the *only* solution.

```
Len_P0_Left_equals_Zero
P0_Left_equals_Str_Empty
P0_Right_equals_Str_Z0
Rev_P0_Right_equals_Str_Z0
P1_Left_equals_Str_Empty
P1_Right_equals_Str_Z0
P2_Left_equals_Str_Empty
E2_equals_Z0
P2_Right_equals_Str_Empty
Cat_E2_P2_Right_equals_Str_Z0
Rev_P2_Right_equals_Str_Empty
P3_Left_equals_Str_Empty
E3_equals_Z0
P3_Right_equals_Str_Empty
Cat_E3_P3_Right_equals_Str_Z0
Len_P3_Right_equals_Zero
P4_Left_equals_Str_Empty
P4_Right_equals_Str_Z0
P5_Left_equals_Str_Empty
P5_Right_equals_Str_Z0
```

Figure 7: Only Solution (*true* Vars) for Formula in Figure 6

The solution gives the value of each program variable in each state. For example, the following variables are true in the witness: *P0_Left_equals_Str_Empty*, *P0_Right_equals_Str_Z0*, *P5_Left_equals_Str_Empty*, and *P5_Right_equals_Str_Z0*. This corresponds to a *List* input value of P = (< >, <*Z0*>) and an output value of P = (< >, <*Z0*>). The code is defective because

the output value as required by the specification is *P* = (<*Z0*>, < >). A problem with the code is identified here with a severely restricted scope because the lengths of the left and right strings resulting from the code and specification do not match. (If no satisfying assignments were found, the scopes would have to be enlarged and the process repeated.)

A key benefit of the modular error detection approach is that it is relatively easy to debug the code from the given solution. Based on the finding in Figure 7, especially with the help of a tool to improve the presentation, the programmer of *Reverse* can infer how to fix the code. In particular, based on the input that revealed a defect (*P0*), it is easy to see that the program is erroneous when it is given a list *P.Left* = < > and *P.Right* = <*Z0*>. The assignment from the SAT solver gives the values of each variable in each state, making it relatively easy to debug.

3.3 Effectiveness and Efficiency of DEET

DEET should need to deal with a large number of statements only rarely, because it examines not just one component, but only one component operation, at a time. Still, to check scalability in this dimension, we mechanically generated an error hypothesis formula for a "synthetic" procedure body with 2000 statements, using operation specifications similar to the ones given in the example [72]. The resulting formula involved 6000 variables and twice as many conjuncts. The solver found two solutions (witnesses to errors) in less than 2 seconds on a 1.2 MHz Athlon PC.

Much more experimentation is needed with this and other solvers before we can reach any conclusions on the effectiveness or efficiency of DEET. There is significant potential for further improvements to take advantage of the kinds of formulas that arise from the DEET process, including parallelization and specialized computer algebra techniques.

4. OTHER RELATED WORK

The idea of error detection within a small "scope"—borrowed by DEET from Alloy—differs from most related work in fundamental ways, as noted in [29, 37, 42, 77], and we summarize only additional differences here.

The benefits of static analysis are widely acknowledged, even more so recently as a result of the extensive work in model checking research and industrial practice [10, 14, 36]. Though model checking has its origins in hardware verification, an impressive collection of results spans a spectrum of programming languages and software systems. Given that it is difficult to summarize even the most important work in this area, we discuss only a representative sample.

Finite-state systems are the focus, though there have been efforts to extend model checking to minimize the impact of this inherent limitation (e.g., [5]). Holzman has employed SPIN to detect numerous bugs in the PathStar processing system developed in C. Java Pathfinder at NASA has been used successfully to locate a variety of heap-related errors [31]. To limit the search space, Bandera, a tool for analyzing Java code, employs user-supplied abstractions [15, 58] whereas Smith *et al.* have described a system that assists in property specification [74]. The fundamental difference between DEET and such uses of model checkers is in the way a finite-state model of program execution is devised, i.e., by combining Jackson's small scope hypothesis with assertions that arise from verification conditions that are generated from the code and component contract specifications.

Symbolic execution of programs, where concrete inputs used in testing are replaced with symbolic values to generate constraints between inputs and outputs, have been used for debugging and testing [12, 45] and verification [19]. Early work on symbolic execution was limited by its inability to handle complex types, loops, and dynamic data structures. Coen *et al*. have shown that symbolic execution can be useful for verification of safety-critical properties in an industrial setting, but this requires severe limitations to be placed on the code [13]. More recently, using symbolic execution for model checking, the SLAM project [1] has shown how to handle recursive calls in C code. Khurshid *et al*. have addressed properties of the heap and dynamic data structures [43]. Unlike these efforts, whose focus is on verification, PREfix is a tool based on symbolic execution for error detection [6]. While the tool has been shown to reveal errors in large-scale C/C++ systems, it cannot handle properties such as invariants and it can produce false alarms.

With user-supplied loop invariants (similar to the DEET approach for handling loops), in [39] Jensen *et al*. have discussed how to prove heap-related properties and find counterexamples. Their program has been shown to be quite effective in practice. Their work differs from traditional pointer analyses because they can answer more questions that can be expressed as properties in first-order logic. While this work focuses on linear linked lists and tree structures, more recently Moller and Schwartzbach have extended the results to all data structures that can be expressed as "graph types" [53]. There is also significant work in shape analysis, including recent work on parametric shape analysis that allows more questions to be answered concerning heaps [62]. Ramalingam *et al.* describe how to check client conformance with component constraints [61] using abstract interpretation. The goals and methods of these related efforts are quite different from ours because our focus is on the total correctness of component-based software based on design-by-contract, not on verifying heap properties.

Ernst provides an overview of the complementary merits of dynamic and static analysis approaches for error detection in [24]. While the benefits of writing assertions and using them to detect errors in software are widely known [26, 78], assertion checking is especially useful in component-based software development to detect contractual violations among collaborating components [2, 8, 21, 27, 52]. Eiffel is among the earliest systems to popularize runtime assertion checking [52]. iContract, a contract-checking tool for Java programs, has similar objectives [20]. Using an executable industrial-strength specification language, AsmL, Barnett *et al.* describe a system for dynamic checking [2]. Cheon and Leavens have used JML for writing assertions and for runtime assertion checking of component-based Java programs [7, 8, 9]. The benefit of contract checking in commercial development of a component-based C++ software system is described in [34]. Use of wrappers to separate contract-checking code from underlying components is described in [21, 22]. However, runtime checking is difficult to modularize, requires that implementations of not just the unit being checked but all reused components be available, detects only errors that arise from particular implementations rather than their contracts (so substitutability bugs are not revealed), and requires manual input selection—all problems that DEET avoids.

There is considerable work on making SAT solvers efficient. But that work is orthogonal to DEET, which is intended to use an off-the-shelf solver (i.e., based only on its functional specification). Experimentation with different solvers for DEET is necessary to develop an effective tool because of potentially significant performance differences among solvers.

5. SUMMARY

The ultimate objective of formal verification techniques is to prove that a piece of code (in our case, a software component) is correct with respect to its specification. Experience shows, however, that before attempting to prove correctness, it is usually cost-effective to look for behavioral errors that can be found by simpler means. DEET is our first effort toward a modular, static analysis approach for discovering errors of this sort, including some that are not revealed by testing—which is the usual approach to finding code defects—or by existing static analysis/checking tools. Some aspects of the DEET approach have been automated at the time of writing, and others are work in progress.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grant CCR-0113181. We thank the reviewers for observing, quite correctly, that some related work deserved a more detailed discussion than in our original submission.

REFERENCES

1. T. Ball and S. K. Rajamani. The SLAM toolkit. *CAV* 2001, pp. 260-264.
2. M. Barnett, W. Grieskamp, C. Kerer, W. Schulte, C. Szyperski, N. Tillmann, and A. Watson. Serious specification for composing components. In *Proc. Sixth ICSE Workshop on Component-Based Software Engineering*, May 2003, pp. 31-36.
3. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *Tools and Algorithms for the Analysis and Construction of Systems* (TACAS'99), LNCS 1579, Springer-Verlag, 1999.
4. W. Blochinger, C. Sinz, and W. Küchlin. Parallel propositional satisfiability checking with dynamic learning. *Parallel Computing,* 29(7), 2003, pp. 969–994.
5. T. Bultan, R.Gerber, and W. Pugh, Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4), July 1999.
6. W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7), 2000, pp. 775–802.
7. Y. Cheon and G.T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Magnusson, B., editor, *ECOOP 2002 – Object-Oriented Programming*, *16th European Conference, Malaga, Spain, Proceedings*, LNCS 2374, Springer-Verlag, Berlin, June 2002, pp. 231-255.
8. Y. Cheon and G.T. Leavens. A runtime assertion checker for the Java modeling language (JML). In *Proc. Int'l Conf. Software Engineering Research and Practice*, CSREA Press, June 2002, pp. 322-328.
9. Y. Cheon and G.T. Leavens, M. Sitaraman, and S. H. Edwards. *Model variables: Cleanly supporting abstraction in design by contract*. Technical Report 03-10a, Department of Computer Science, Iowa State University, September 2003; available from archives.cs.iastate.edu.
10. D. Clarke, O. Grumberg and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency - Reflections and Perspectives*. LNCS 803, Springer-Verlag, 1994.
11. D. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7{34}, 2001.
12. L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3), September 1976, pp. 215-222.
13. A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. Using symbolic execution for verifying safety-critical systems. In *Proc. 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2001, pp. 142–151.
14. D. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In Gerard Berry, Hubert Comon, and Alan Finkel, editors, *Proc. Computer Aided Verification*, LNCS 2102, Springer-Verlag, 2001, pp. 435-453.
15. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, H. Zheng, H. Bandera: extracting finite-state models from Java source code. *Proceedings of the 22nd International Conference on Software Engineering*, Limeric, Ireland, 2000.
16. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM 7*, 1960, 201-215.
17. D. L. Detlefs, K. R. M. Leino, and G. Nelson. *Wrestling with Rep Exposure*. Research Report 156, Compaq Systems Research Center, July, 1998.
18. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. *Extended Static Checking.* Research Report 159, Compaq Systems Research Center, December, 1998.
19. L. K. Dillon. Using symbolic execution for verification of Ada tasking programs. *ACM Transactions on Programming Languages and Systems*, 12(4), 1990, pp. 643-669.
20. A. Duncan and U. Hölzle. *Adding Contracts to Java with Handshake*. Technical Report TRCS98-32, Univ. of California at Santa Barbara, Dec. 1998.
21. S. H. Edwards, G. Shakir, M. Sitaraman, B.W. Weide, and J. Hollingsworth. A framework for detecting interface violations in component-based software. In *Proc. 5th Int'l Conf. Software Reuse*, IEEE, June 1998, pp. 46-55.
22. S. H. Edwards, M. Sitaraman, B.W. Weide, and J. Hollingsworth. Contract-Checking Wrappers for C++ Components. *IEEE Trans. On Software Engineering*, 2004, to appear.

23. G. W. Ernst, R. J. Hookway, and W. F. Ogden. Modular verification of data abstractions with shared realizations. *IEEE Trans. Software Eng.*, 20(4), Apr. 1994, 288-307.

24. M. D. Ernst. Static and dynamic analysis: synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, Portland, OR, May 2003, pp. 24-27.

25. J. Esparza, A. Kucera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. *Information and Computation*, 186(2), November 2003, pp. 355–376.

26. R.B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *Proc. 8th European Software Engineering Conference*, ACM Press, New York, NY, 2001, pp. 229–236.

27. R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proc. ACM SIGPLAN 2001 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2001, pp. 1-15.

28. E. Goldberg, E. and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proc. Design, Automation, and Test in Europe Conference and Exposition (DATE)*, IEEE Computer Society Press, 2002, 131-149.

29. O. Grumberg, D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, v.16 n.3, pp.843-871, May 1994.

30. D.E. Harms and B.W. Weide. Copying and swapping: influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5), 1991, pp. 424-435.

31. K. Havelund and T. Pressburger. Model checking Java programs Using Java Pathfinder. *International Journal on Software Tools for Technology Transfer,* 2(4), Springer-Verlag, April 2000.

32. W. D. Heym. *Computer Program Verification: Improvements for Human Reasoning.* Ph.D. Dissertation, Department of Computer and Information Science, The Ohio State University, Columbus, OH, 1995.

33. J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. *The Geneva Convention On The Treatment of Object Aliasing*, http://gee.cs.oswego.edu/dl/aliasing/aliasing.html, 1997.

34. J.E. Hollingsworth, L. Blankenship, and B.W. Weide. Experience report: Using RESOLVE/C++ for commercial software. In *Proc. ACM SIGSOFT 8th Int'l Symposium on the Foundations of Software Engineering*, ACM, Nov. 2000, pp. 11-19.

35. H.Hoos. SAT-encodings, search space structure, and local search performance. *Proc. 16th Intl. Joint Conf. On Artificial Intelligence (IJCAI'99)*, Stockholm, Sweden, Morgan Kaufmann,1999, pp. 296–303.

36. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997, pp.279-295.

37. D. Jackson and M. Vaziri. Finding bugs with a constraint solver. *ACM SIGSOFT Software Engineering Notes*, Sept. 2000, pp. 14-25.

38. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2), April 2002, pp.256-290.

39. J. L. Jensen, M. E. Jorgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, 1997.

40. C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

41. A. Kaiser. *A SAT-based Propositional Prover for Consistency Checking of Automotive Product Data*. Technical Report WSI-2001-16, W.-Schickard Institut für Informatik, Universität Tübingen, Tübingen, Germany, 2001.

42. S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. *Procs. 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, Seattle, WA, 2002.

43. S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Procs. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2003)*, Warsaw, Poland, April 2003.

44. S. Khurshid, D. Marinov, I. Shlyakhter, and D. Jackson. A case for efficient solution enumeration. *Procs. 6th International Conference on Theory and Applications of Satisfiability Testing* (SAT), Portofino, Italy, May 2003.

45. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, vol. 19 (7), July 1976, 385-394.

46. J. Krone, W. F. Ogden, and, M. Sitaraman. *Modular Verification of Performance Constraints.* Technical Report RSRG-03-04, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, May 2003, 25 pages; available at www.cs.clemson.edu/~resolve.

47. G. Kulczycki, M. Sitaraman, W. F. Ogden, and J. E. Hollingsworth, *Component Technology for Pointers: Why and How*, Technical Report RSRG-03-03, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, April 2003, 19 pages; available at http://www.cs.clemson.edu/~resolve.

48. G. Kulczycki, M. Sitaraman, W. F. Ogden, and G. T. Leavens, *Preserving Clean Semantics for Calls with Repeated Arguments*, Technical Report RSRG-04-01, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, April 2003, 35 pages; available at http://www.cs.clemson.edu/~resolve.

49. G. Kulczycki. *Direct Reasoning.* Ph.D. Dissertation, Department of Computer Science, Clemson University, Clemson, SC, May 2004.

50. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, 37(5), 2002, pp. 246-257.

51. D. Marinov and S. Khurshid. TestEra: a novel framework for automated testing of Java programs. *Procs. 16th IEEE Conference on Automated Software Engineering* (ASE), San Diego, CA, 2001.

52. B. Meyer, *Object-oriented Software Construction*, 2nd Edition, Prentice Hall, Upper Saddle River, NJ, 1997.

53. A. Moller and M. I. Schwartzbach, The pointer assertion logic engine. *ACM SIGPLAN Notices*, 36(5), May 2001, pp.221-231.

54. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*. ACM, 2001, 530-535.

55. P. Muller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency, Computation Practice and Experience*, 15, 2003, pp. 117-154.

56. J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 2001.

57. J.W. Nimmer and M.D. Ernst. Invariant inference for static checking: an empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, Charleston, SC, November 2002, pp. 11-20.

58. C. Pasareanu, M.B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, 2001.

59. S. Prestwich. Local search on SAT-encoded coloring problems. *Proc. 6th Intl. Conf. On Theory and Applications of Satisfiability Testing (SAT 2003),* Santa Margherita Ligure, Italy, Springer, 2003, pp. 105–119.

60. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification of fault-tolerant architectures: prolegomena to the design of PVS. *IEEE Trans. Software Engineering*, 21(2), Feb. 1995, 107-125.

61. D. Ramalingam, A. Warshavsky, J. Field , D. Goyal, M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. *ACM SIGPLAN Notices*, 37(5), May 2002.

62. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Tran. on Programming Languages and Systems 24*, 3 (2002), 217-298.

63. C. Sinz, W. Blochinger, and W. Küchlin. PaSAT - parallel SATchecking with lemma exchange: implementation and applications. In H. Kautz und B. Selman, Hrsg., *LICS'2001 Workshop on Theory and Applications of Satisfiability Testing (SAT'2001)*, Electronic Notes in Discrete Math., 9, Elsevier, Boston, MA, June 2001.

64. C. Sinz, T. Lumpp, J. Schneider, and W. Küchlin. Detection of dynamic execution errors in IBM System Automation's rulebased expert system. *Information and Software Technology*, 44(14), November 2002, pp. 857–873.

65. C. Sinz. Verifikation regelbasierter Konfigurationssysteme. Dissertation, Fakultät für Informations- und Kognitionswissenschaften, Universität Tübingen, 2003.

66. M. Sitaraman and B.W. Weide. Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes 19*, 4 (1994), pp. 21-67.

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

68. M. Sitaraman, S. Atkinson, G. Kulczycki, B.W. Weide, T. Long, P. Bucci, S. Pike, W. Heym, and J.E. Hollingsworth. Reasoning about software-component behavior. In *Proceedings of the 6th International Conference on Software Reuse*, LNCS 1844, Springer-Verlag, 2000, pp. 266-283.

69. M. Sitaraman. Compositional performance reasoning. *Procs. Fourth ICSE Workshop on Component-Based Software Engineering: Component-Certification and System Prediction*, Toronto, CA, May 2001.

70. M. Sitaraman, J. Krone, G. Kulczycki, W. F. Ogden, and A. L. N. Reddy. Performance specification of software components. *ACM SIGSOFT Symposium on Software Reuse*, May 2001.

71. M. Sitaraman, T. J. Long, B. W. Weide, J. E. Harner, and L. Wang. A formal approach to component-based software engineering: education and evaluation. In *Procs. of the International Conference on Software Engineering*, IEEE, Toronto, Canada, May 2001, pp. 601-609.

72. M. Sitaraman, D. P. Gandi, W. Küchlin, C. Sinz, and B. W. Weide. *The Humane Bugfinder: Modular Static Analysis Using a SAT Solver*. Technical Report RSRG-03-05, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, May 2003, 18 pages; available at http://www.cs.clemson.edu/~resolve.

73. M. Sitaraman, B. W. Weide, and W. F. Ogden. *Design, Specification, and Analysis of Software Components*. CS 372 Course Notes, Clemson University, Clemson, SC 29634-0974, 2003.

74. R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. PROPEL: an approach supporting property elucidation. *Proceedings of the 24th International Conference on Software Engineering*, May 2002.

75. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995, pp. 121–189.

76. M. Vardi. On the complexity of modular model checking. In *Proc. 10th IEEE Symposium on Logic in Computer Science*, 1995, pp. 101-111.

77. M. Vaziri and D. Jackson. Checking heap-manipulating procedures with a constraint solver. *TACAS'03*, Warsaw, Poland, 2003.

78. J. M. Voas. How assertions can increase test effectiveness. *IEEE Software 14*, 2 (Feb. 1997), pp. 118-122.

79. B.W. Weide and W.D. Heym. Specification and verification with references. In *Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems*, ACM, 2001.

80. B.W.Weide. Component-based systems. In *Encyclopaedia of Software Engineering*, ed. J. J. Marciniak, John Wiley and Sons, 2001.

81. J. M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 29(9), Sep. 1990, pp. 8-24.