The Humane Bugfinder: Modular Static Analysis Using a SAT Solver

Murali Sitaraman, Durga P. Gandi, Wolfgang Küchlin, Carsten Sinz, and
Bruce W. Weide

Technical Report RSRG-03-05
Department of Computer Science
451 Edwards Hall
Clemson University
Clemson, SC 29634-0974 USA

June 2003

The Humane Bugfinder: Modular Static Analysis Using a SAT Solver

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 and Information Science
The Ohio State University
Columbus, OH 43210, USA
1-(614)-292-1517
weide@cis.ohio-state.edu

ABSTRACT

Assertion checking is a widely used technique to discover inconsistencies between specified behavior and actual implementation behavior. A modular, static analysis approach that is suitable for component-based systems is introduced. In the first stage of this approach, using only specifications of reused components and internal assertions in the implementation code (e.g., loop invariants), assertions for verification of correctness are generated. In the second stage, error hypotheses are generated as Boolean formulae—an idea inspired by results on scope restriction from the model checking community. The generated formulae are such that a satisfiable assignment not only indicates an error but provides a directly human-readable trace of a witness to the bug. An example checked using an existing SAT solver suggests that the approach is promising from the practical standpoint.

Keywords: Components, contracts, model checking, SAT solvers, specification, verification.

1. INTRODUCTION

In assertive software development, mathematical assertions are provided as formal documentation to supplement code. *Assertion checking* refers to the general idea of checking that the specified behavior and the actual implementation behavior are consistent. The benefits of writing assertions and using them to detect errors in software are widely known [7][21].

Assertion checking is especially useful in component-based software development to detect contractual violations involving interactions among collaborating components [1][2][6][15]. *Interface violations* occur either because a client component does not satisfy the requirements of the called component, or because the called component does not satisfy its reciprocal obligations to the client. *Internal violations* occur when a component fails to meet its own internal assertions. Such violations result, for example, because an object-based implementation does not adhere to the specified representation invariant or because the code for a loop fails to satisfy its specified loop invariant.

Interface-related and internal assertions can be checked *statically* or *dynamically*. Static assertion checking does not involve execution of code, whereas dynamic assertion checking does. There is considerable previous research on both possibilities. We review only the most closely related research here.

ESC Java facilitates static analysis of Java programs using assertions [4]. Jackson, et al., describe a general static analysis system for detecting errors using relational specifications [11]. The ultimate goal of static analysis is formal verification, i.e., to show that an implementation is correct with respect to its specification. However, the focus of the current paper is (like those above) on bug detection using static analysis.

Eiffel is among earliest systems to popularize runtime assertion checking [15]. iContract, a contract-checking tool for Java programs, has similar objectives [5]. Using an executable industrial-strength specification language, AsmL, Barnett, et al. [1] describe a system for dynamic checking. Cheon and Leavens have used JML (Java Modeling Language) for writing assertions and for runtime assertion checking of component-based Java programs [1]. The benefit of contract checking in commercial development of a component-based C++ software system is described in [10]. Use of wrappers to separate contract-checking code from underlying components is described in [6].

The benefits of static and dynamic analysis approaches are complementary. By not requiring code execution, static analysis makes it possible to test classes of software where execution is expensive, or where it involves rolling back non-trivial aspects of a larger system in which a component is embedded. It may be possible to check statically some subsets of inputs, even faster than by running the program. Dynamic checking also has some benefits. For example, it allows the possibility of incorporating error-recovery code in deployed components, in case errors do arise.

The contribution of this paper is a modular, static analysis approach to detecting errors in component-based software systems. The approach builds on previous work on modular verification, results in model checking, and SAT solvers. The paper contains a detailed example and presents steps for automation. It represents work in progress, raising as many questions as it answers, and hence is something that should be discussed further at a workshop.

Section 2 of the paper summarizes the basics of modular reasoning. Section 3 presents an example. Using the example, Section 4 illustrates the central steps of the approach. Section 5 discusses some preliminary observations.

2. SPECIFICATION-BASED ANALYSIS

The essential role of specifications to enable modular reasoning in component-based systems is well known [23]. Well-designed specifications provide abstraction and separate the essential from the inessential. They serve as firewalls, and avoid expensive implementation-to-implementation coupling. They are implementation-neutral allowing the specified behavior to be implemented in different ways.

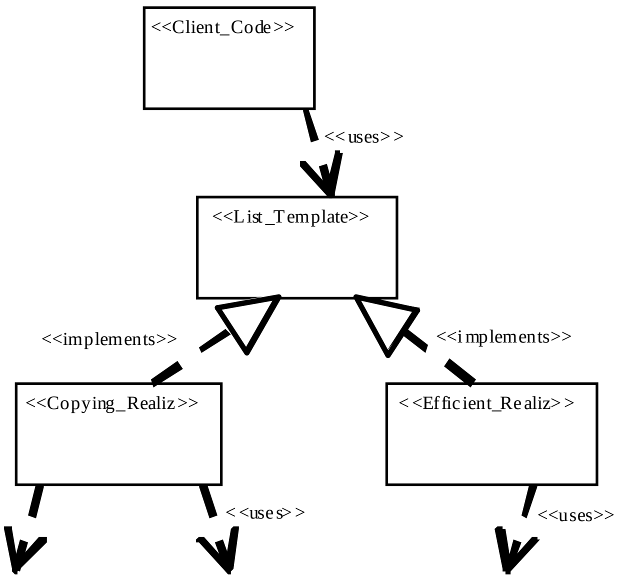

Figure 1: A Design-Time Relationship Diagram

The central role of specifications for this work is illustrated in Figure 1. In a component-based setting, it often becomes necessary to develop and utilize multiple implementations of a single specification (for different component uses even within the same program), because there is usually no one "best" implementation for a specification. The design-time relationship diagram in Figure 1 shows concisely and simultaneously both the relationships among component specifications and implementations, and the potentially very large number of particular component-based systems that could be built using these components [23]. The diagram emphasizes the role of *substitutability*. Client_Code needs to choose an implementation of List_Template—a choice that affects its performance, but not its functionality.

The idea of implementation substitutability leads us to *modular reasoning* or specification-based reasoning: the ability to reason about a component's behavior without knowing anything beyond its specification, and the specifications of the components reused in implementing it. This means that a client should be able to reason about a component without any knowledge of its implementation details. An implementer of a component cannot assume any knowledge of the client environment either, other than those documented explicitly in the component specification. Achieving the modular reasoning property is a central issue in component-based software engineering. It makes it possible to localize understanding, reasoning, and maintenance of systems.

A modular approach to error detection relies only on specifications of reused components. Unlike the situation with execution-based approaches, behaviors of actual implementations do not impact the results of error detection.

3. A COMPONENT-BASED EXAMPLE

3.1 An Example Component Specification

This section contains an example. Figure 2 shows a skeleton of a specification of a bounded version of a parameterized List_Template in a dialect of RESOLVE notation [18]. Here, 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 can be explained as taking place between the two strings, e.g., at the left end of the right string (i.e., just to the right of the fence), as we have done here.

Formally, the declaration of type *List* introduces the mathematical model and, using an example *List*, it 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. In the specification, the infix operator * denotes string concatenation and the outfix operator | | denotes string length.

An interesting aspect of the *Insert* specification is that its behavior is *relational*. The semantics of "alters" mode is that the result value of the entry *E* is undetermined. This under-specification allows implementations not to have to make expensive copies of non-trivial entries, which is an important issue in the specification of generic data abstractions [9]. Copying references, while efficient, introduces aliasing and complicates reasoning [22]. The present specification is more flexible. It allows the entry to be moved or swapped into the container structure (efficiently in constant time by manipulating references) and thus potentially alter it, without introducing aliasing [9]. Correspondingly, the *Remove* operation is specified to remove an entry from the *P* and replace the parameter *R*. Operation *Advance* allows the list insertion position (fence) to be moved ahead. The rest of the specification is discussed in detail in [19].

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

 Type List is modeled by (
   Left: Str(Entry);
   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.Right =
        #P.Left * #P.Right  and
       |P.Left| = |#P.Left| + 1;

  Operation Right_Length
     ( restores P: List ): Integer;
   ensures Right_Length = |P.Right|;

  Operation Reset ( updates P: List );
   ensures |P.Left| = 0 and
      P.Right = #P.Left * #P.Right;
  . . .
end List_Template;
```

Figure 2: A Specification of List_Template

3.2 Specification and Implementation of an Example Operation

Figure 3 contains the specification of an operation to reverse (the right string of) a list. In the specification, *Rev* denotes the mathematical definition of string reversal.

Figure 4 contains an (incorrect) recursive procedure to implement the specification. This realization is written using the primary *List* operations given in Figure 2. To demonstrate termination, the recursive procedure is annotated with a progress metric using the **decreasing** keyword.

4. MODULAR STATIC ANALYSIS

4.1. Step 1 – Reasoning Table Generation

As a first step in modular static analysis—either to prove correctness or to find errors—we first generate a symbolic reasoning table [19]. Figure 5 contains a table for the code in Figure 4, which is similar to the one given in [19]. A key observation for the current paper is that this table can be produced mechanically from the information in Figures 2, 3, and 4, as explained in [19] and summarized below.

```
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 3: 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 4: An Implementation of List Reversal Operation

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 to demonstrate correctness. The *path condition* denotes under what condition a given state will be reached.

A 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 verification, we rely only on the behavioral contracts of the reused operations (i.e., *Insert* and *Remove*). In particular, for correct calling code we must be able to confirm that the requires clause of a reused operation is true in the state before the call; then we can assume that the ensures clause 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 $\wedge$ P1.Right = <E2> * P2.Right	$\|P2.Left\| = 0 \wedge$ $\|P2.Right\| < \|P0.Right\|$
	`Reverse(P);`		
3	$\|P0.Right\| > 0$	E3 = E2 $\wedge$ P3.Left = Rev(P2.Right) $\wedge$ $\|P3.Right\| = 0$	
	`Insert(E, P);`		
4	$\|P0.Right\| > 0$	P4.Left = P3.Left $\wedge$ P4.Right = <E3> * P3.Right	
	`end;`		
5.1	$\|P0.Right\| = 0$	P5 = P0	P5.Left = Rev(P0.Right) $\wedge$ $\|P5.Right\| = 0$
5.2	$\|P0.Right\| > 0$	P5 = P4	P5.Left = Rev(P0.Right) $\wedge$ $\|P5.Right\| = 0$

Figure 5: A Reasoning Table for List Reverse Procedure

The path condition in a given state serves as an antecedent for the assertions that can be assumed and that must be confirmed in that state. In other words, the assumptions apply only when the path condition holds. Similarly, the obligations need to be confirmed only when the path condition holds.

4.2. Step 2 – Error Hypothesis Generation

To prove the correctness of the code, then, we need to confirm each obligation in the last column, using the assumptions in the states before and including the state where the obligation arises. Before attempting the non-trivial process of verification using a general theorem-proving tool, it is useful to look for errors.

In the current approach, we look for a witness to a bug in the code. In particular, we attempt to find values for the variables that satisfy all relevant assumptions but that fail to satisfy something that needs to be confirmed. We do this by conjoining the assumptions and the negation of the assertion to be confirmed, and seek a satisfying assignment for the variables in this *error hypothesis*—a witness to a bug.

To illustrate the idea, consider one of the assertions that needs 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. We wish to find a set of assignments to the variables that satisfies the assertion in Figure 6. This would show that the code is defective. In the figure, the conjunct numbered I is the path condition, conjuncts II through VII are assumptions in states 0 through 5, and conjunct VIII is the negation of the assertion to be confirmed in state 5.

Error hypothesis generation is automatable. 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 to an operation it calls. It is also 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, although intermediate errors are found. In this case, the code is still deemed wrong because of the modularity property discussed in Section 2.

$(\lvert P0.Right \rvert > 0) \wedge$	I
$(\lvert P0.Left \rvert = 0) \wedge$	II
$(P1 = P0) \wedge$	III
$(P2.Left = P1.Left \wedge P1.Right = \langle E2 \rangle * P2.Right) \wedge$	IV
$(E3 = E2 \wedge P3.Left = Rev(P2.Right) \wedge \lvert P3.Right \rvert = 0) \wedge$	V
$(P4.Left = P3.Left \wedge P4.Right = \langle E3 \rangle * P3.Right) \wedge$	VI
$(P5 = P4) \wedge$	VII
$(\neg\ (P5.Left = Rev(P0.Right) \wedge \lvert P5.Right \rvert = 0))$	VIII

Figure 6: Error Hypothesis Corresponding to the Obligation (Case 5.2) in State 5

4.3. Step 3 – Scope Restriction and Boolean Formula Generation

In the search for a witness to an error hypothesis, we appeal to Jackson's small scope hypothesis (where "scope" is, loosely speaking, a measure of the size of the input space to be searched). It claims 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 [11]. Restriction of scope allows us to check all the valid inputs in a given scope to find a witness to an error hypothesis. If one is found, then we can conclude that the code is not consistent with the assertions. If none is found in the given scope, then we only know that there are no inconsistencies in the scope; inconsistencies may be found if the scope is increased.

We restrict the scopes of participating variables instead of placing bounds on individual objects, recursive calls, and loops. We begin with the most stringent restrictions. In the example, we start by looking for a witness to the error hypothesis in which all variables of type *Entry* have one particular 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 call the single value of type *Entry* Z0. This in turn restricts the scope of our search for strings to be the two-element set {Str_Empty, Str_Z0}, where Str_Empty denotes the empty string and Str_Z0 denotes the string <Z0>.

With these restrictions on scope, we can create a (possibly large, but finite) Boolean formula to correspond to each error hypothesis generated from the code and the specifications, e.g., the one in Figure 6. Each satisfying assignment for this Boolean formula identifies a particular witness to a particular error hypothesis. To condense space usage, we have shown only a part of the formula in Figure 7.

In the conjuncts listed in Figure 7, the names of all Boolean variables can be generated automatically (although they are sanitized here to be somewhat "meaningful" for human reading). The Boolean variable *P0_Left_equals_Str_Empty* being true, for example, denotes that the left string of the list *P* in state 0 is equal to the empty string. In addition to the variables that correspond directly to the symbols in Figure 6, variable names corresponding to mathematical expressions involving string length, reverse, and concatenation are needed as well. Given this, the first two conjuncts in Figure 7 correspond directly to those in Figure 6.

To assert that *P1 = P0* (conjunct III in Figure 6), the Boolean 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 7. The rest of the conjunctions IV through VIII in Figure 7 are derived similarly.

A list of *additional conjunctions* needs to be generated to complete the Boolean formula generation; only some of these additional conjunctions are shown in Figure 7. 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 a list in each state. One set of assertions is generated based on the mathematical definition of 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 6. The complete formula is given in the Appendix.

I
(¬Len_P0_Right_equals_Zero)

II
(Len_P0_Left_equals_Zero)

III
((P1_Left_equals_Str_Empty ∧ P0_Left_equals_Str_Empty)
  ∨ (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))

IV
((P2_Left_equals_Str_Empty ∧ P1_Left_equals_Str_Empty)
  ∨ (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))

V
(E3_equals_Z0 ∧ E2_equals_Z0) ∧
((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)

VI
((P4_Left_equals_Str_Empty $\wedge$ P3_Left_equals_Str_Empty)
$\vee$ **(**P4_Left_equals_Str_Z0 $\wedge$ P3_Left_equals_Str_Z0)) $\wedge$
((P4_Right_equals_Str_Empty $\wedge$
Cat_E3_P3_Right_equals_Str_Empty)
$\vee$ (P4_Right_equals_Str_Z0 $\wedge$
Cat_E3_P3_Right_equals_Str_Z0))

VII
((P5_Left_equals_Str_Empty $\wedge$ P4_Left_equals_Str_Empty)
$\vee$ **(**P5_Left_equals_Str_Z0 $\wedge$ P4_Left_equals_Str_Z0)) $\wedge$
((P5_Right_equals_Str_Empty $\wedge$
P4_Right_equals_Str_Empty)
$\vee$ (P5_Right_equals_Str_Z0 $\wedge$ P4_Right_equals_Str_Z0))

VIII
($\neg$ (((P5_Left_equals_Str_Empty $\wedge$
Rev_P0_Right_equals_Str_Empty) $\vee$
(P5_Left_equals_Str_Z0 $\wedge$
Rev_P0_Right_equals_Str_Z0)) $\wedge$
(Len_P5_Right_equals_Zero)))

Additional Assertions:

Unique Values (sample: P0.Right)
($\neg$ P0_Right_equals_Str_Empty $\vee$ $\neg$ P0_Right_equals_Str_Z0)

String Length (sample: |P0.Right|)
(Len_P0_Right_equals_Zero $\Leftrightarrow$ P0_Right_equals_Str_Empty)

String Reverse (sample: Rev(P0.Right))
(Rev_P0_Right_equals_Str_Empty $\Leftrightarrow$
P0_Right_equals_Str_Empty) $\wedge$
(Rev_P0_Right_equals_Str_Z0 $\Leftrightarrow$ P0_Right_equals_Str_Z0)

*String Concatenate (sample: <E2> * P2.Right)*
($\neg$ Cat_E2_P2_Right_equals_Str_Empty) $\wedge$
(Cat_E2_P2_Right_equals_Str_Z0 $\Leftrightarrow$
(E2_equals_Z0 $\wedge$ P2_Right_equals_Str_Empty))

Figure 7: Selected Conjuncts Corresponding to Figure 6

The number of Boolean variables in the entire formula is bounded by the product of the size of the restricted scope, the number of program variables and mathematical expressions in the original verification conditions, and the number of states in the implementation. The number of conjuncts depends on the sizes of the mathematical assertions involved and the number of generated Boolean variables.

4.4. Step 4 – Use of a SAT Solver

The example illustrates that the formulae generated during this process are not in conjunctive normal form (CNF). While it is possible to convert the formula into CNF, the result is a formula that is much longer and that does not correspond to the code directly. We therefore applied a SAT checker that can handle arbitrary propositional formulae [12]. This solver overcomes the CNF limitation of other state-of-the-art SAT checkers such as BerkMin [8] or Chaff [16]. The solver uses a Davis-Putnam-style [3] algorithm to compute satisfying assignments, and can handle formulae involving several thousand variables. While the current solver is fast, we expect parallel implementations to allow further improvements [17]. Thus, the solver is potentially suitable for handling assertions resulting from non-trivial specifications and implementations.

When the formula in Figure 7 was supplied to the SAT solver, it produced the assignment given in Figure 8 within a fraction of a second. In addition, the solver 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 8: Only Solution to the Formula in Figure 7

The solution essentially gives the value of each variable in each state. Here, 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 erroneous because the output value as required by the specification is *P* = (<Z0>, < >).   A problem with the code is identified here with a severaly restricted scope because the lengths of the left and right strings resulting from the code and specification do not match.   (If no assignments were found, the scopes would 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 8, a reader can infer how to fix the code.

5. DISCUSSION

5.1. Generality of the Approach

While we have illustrated details of the approach using a simple recursive procedure in this paper, the potential significance of the approach is its generality. Elsewhere, we have discussed handling a proof system for verification of data abstractions (including those where abstraction relations are necessary) [20] and verification of both time and space constraints [14], even in the presence of dynamic memory management and loops. With the proof rules in those papers, a reasoning table similar to the one in Figure 5 can be generated for functionality and performance verification, and then the process of finding errors described in this paper can be applied to check the assertions.

It is important to note that the complexity and the capability of this error checking method depend on the assertions that are specified. A user of the system may choose to specify simplified assertions (e.g., a specification of the Insert operation that merely states that the length of a list is incremented by one), in which case simplified formulae will be generated with less error-catching potential but possibly faster analysis.

5.2. Benefits of Static Analysis

One benefit of static analysis is in checking performance constraints, which is difficult to do using execution-based approaches (though members of the research group are exploring that possibility as well). In addition, the modularity of the approach offers other advantages as well. For example, suppose that there are two implementations of *List_Template*, as shown in Figure 1: one in which *Insert* copies *E* (leaving *E* = #*E*) and one based on swapping, where #*E* and *E* may be unrelated.   Now consider the assertive client code in Figure 9 that is intended to retrieve the "next" entry of a list. In the figure, *P* is of type *List* and *Next* is of type *Entry*. For the code to work, it must copy *Next* before it is inserted back on the list.

```
Assume P = <α,β> and |β| ≠ 0;
    Remove (Next, P);
    Insert (Next, P);
Confirm (∃γ: Str(Entry) ∋ β = <Next> ∗ γ) and P = <α,β>;
```

Figure 9: Error in "Get Next" Code That Goes Undetected in Runtime Assertion Checking

Based on the specification of *Insert* alone, we cannot confirm the assertion at the end, because the value of *Next* may be arbitrary. However, this error will not be revealed in runtime assertion checking or testing, if the copying implementation of *List_Template* is used. It does not matter how many test points are employed, because the client code (inadvertently or intentionally) is relying on the unspecified behavior of this particular implementation. In typical dynamic analysis, the error will be revealed only if different implementations of underlying components are substituted. Alternatively, the error is likely to be detected in a runtime verification approach that includes a non-deterministic choice construct [1]. Execution of that construct might lead to any one (of a finite) set of alternatives for *E* after *Insert*, and hence, the assertion to be confirmed at the end of the code in Figure 9 will fail.

This kind of problem is quite common in component-based software, and has nothing at all to do with the particular example. The merits of relational specification, and in general, under-specification, are well documented, for example in [13] and more recently in [11][18]. Optimization problems often have relational specifications to allow any of several "tied" answers to be produced. Under-specification need not always occur in ensures clauses. For example, the requires clause of a call to another operation following the code in Figure 9 might be satisfied in runtime checking, if *Next* indeed happens to be the front entry of β, but not otherwise. Finally, weaker internal assertions such as representation invariants and loop invariants, that are insufficient to prove correctness of data abstractions, will go undetected if the corresponding code does more than what is documented in those assertions.

5.3. Summary

The ultimate objective of formal verification techniques is to prove that a piece of code is correct with respect to its specification. Before attempting to prove correctness, however, it might be cost-effective to check for errors. We have described a modular, static analysis approach for discovering some such errors.

Some aspects of the approach have been automated at the time of submission, and some others are work in progress. While we have already identified a suitable SAT solver that can handle the types of formulae resulting from our mechanization, currently we are developing a too that can generate Boolean formulae given assertions from a reasoning table.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grants CCR-0113181 and CCR-0081596. We would like to thank John Hunt and Greg Kulczycki for reading and commenting on an earlier draft of this paper.

REFERENCES

[1] 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.

[2] 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.

[3] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM 7*, 1960, 201 - 215.

[4] 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.

[5] A. Duncan and U. Hölzle. *Adding Contracts to Java with Handshake*. Technical Report TRCS98-32, Univ. of California at Santa Barbara, Dec. 1998.

[6] 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 CS Press, June 1998, pp. 46-55.

[7] 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.

[8] 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.

[9] 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), 424-435.

[10] 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.

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

[12] 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.

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

[14] 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.

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

[16] 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.

[17] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT – Parallel SAT – Checking with lemma exchange: Implementation and applications. In *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, Elsevier Science, 2001.

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

[19] 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, 266-283.

[20] 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), 157-170.

[21] J. M. Voas. Quality time: How assertions can increase test effectiveness. *IEEE Software 14*, 2 (Feb. 1997), 118-122.

[22] 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.

[23] B.W.Weide. Component-based systems. *Encycloaedia of Software Engineering*, ed. J. J. Marciniak, John Wiley & Sons, 2001.

Appendix: A Complete Boolean Formula Corresponding to Figure 6

(not Len_P0_Right_equals_Zero) and
(Len_P0_Left_equals_Zero) and
((P1_Left_equals_Str_Empty and P0_Left_equals_Str_Empty)
or (P1_Left_equals_Str_Z0 and P0_Left_equals_Str_Z0)) and
((P1_Right_equals_Str_Empty and P0_Right_equals_Str_Empty)
or (P1_Right_equals_Str_Z0 and P0_Right_equals_Str_Z0)) and
((P2_Left_equals_Str_Empty and P1_Left_equals_Str_Empty)
or (P2_Left_equals_Str_Z0 and P1_Left_equals_Str_Z0)) and
((P1_Right_equals_Str_Empty and
Cat_E2_P2_Right_equals_Str_Empty)
or (P1_Right_equals_Str_Z0 and
Cat_E2_P2_Right_equals_Str_Z0)) and
(E3_equals_Z0 and E2_equals_Z0) and
((P3_Left_equals_Str_Empty and
Rev_P2_Right_equals_Str_Empty)
or (P3_Left_equals_Str_Z0 and
Rev_P2_Right_equals_Str_Z0)) and
(Len_P3_Right_equals_Zero) and
((P4_Left_equals_Str_Empty and P3_Left_equals_Str_Empty)
or (P4_Left_equals_Str_Z0 and P3_Left_equals_Str_Z0)) and
((P4_Right_equals_Str_Empty and
Cat_E3_P3_Right_equals_Str_Empty)
or (P4_Right_equals_Str_Z0 and
Cat_E3_P3_Right_equals_Str_Z0)) and
((P5_Left_equals_Str_Empty and P4_Left_equals_Str_Empty)
or (P5_Left_equals_Str_Z0 and P4_Left_equals_Str_Z0)) and
((P5_Right_equals_Str_Empty and
P4_Right_equals_Str_Empty)
or (P5_Right_equals_Str_Z0 and P4_Right_equals_Str_Z0)) and
(not (((P5_Left_equals_Str_Empty and
Rev_P0_Right_equals_Str_Empty) or
(P5_Left_equals_Str_Z0 and
Rev_P0_Right_equals_Str_Z0)) and
(Len_P5_Right_equals_Zero))) and

(not P0_Left_equals_Str_Empty or not P0_Left_equals_Str_Z0) and
(not P1_Left_equals_Str_Empty or not P1_Left_equals_Str_Z0) and
(not P2_Left_equals_Str_Empty or not P2_Left_equals_Str_Z0) and
(not P3_Left_equals_Str_Empty or not P3_Left_equals_Str_Z0) and
(not P4_Left_equals_Str_Empty or not P4_Left_equals_Str_Z0) and
(not P5_Left_equals_Str_Empty or not P5_Left_equals_Str_Z0) and
(not P0_Right_equals_Str_Empty or not P0_Right_equals_Str_Z0) and
(not P1_Right_equals_Str_Empty or not P1_Right_equals_Str_Z0) and

(not P2_Right_equals_Str_Empty or not P2_Right_equals_Str_Z0) and
(not P3_Right_equals_Str_Empty or not P3_Right_equals_Str_Z0) and
(not P4_Right_equals_Str_Empty or not P4_Right_equals_Str_Z0) and
(not P5_Right_equals_Str_Empty or not P5_Right_equals_Str_Z0) and
(Len_P0_Left_equals_Zero iff P0_Left_equals_Str_Empty) and
(Len_P0_Right_equals_Zero iff P0_Right_equals_Str_Empty) and
(Len_P3_Right_equals_Zero iff P3_Right_equals_Str_Empty) and
(Len_P5_Right_equals_Zero iff P5_Right_equals_Str_Empty) and
(Rev_P0_Right_equals_Str_Empty iff
P0_Right_equals_Str_Empty) and
(Rev_P0_Right_equals_Str_Z0 iff P0_Right_equals_Str_Z0) and
(Rev_P2_Right_equals_Str_Empty iff
P2_Right_equals_Str_Empty) and
(Rev_P2_Right_equals_Str_Z0 iff P2_Right_equals_Str_Z0) and
(not Cat_E2_P2_Right_equals_Str_Empty) and
(Cat_E2_P2_Right_equals_Str_Z0 iff
(E2_equals_Z0 and P2_Right_equals_Str_Empty)) and
(not Cat_E3_P3_Right_equals_Str_Empty) and
(Cat_E3_P3_Right_equals_Str_Z0 iff
(E3_equals_Z0 and P3_Right_equals_Str_Empty))