Diagnosing Program Errors with Light-Weighted Specifications

Rong Chen[1] and Franz Wotawa[2,⋆]

[1] College of Computer Science and Technology, Dalian Maritime University, Linghai 1, 116026 Dalian, China
[2] Technische Universität Graz, Institut for Software Technology, Inffeldgasse 16b/2, A-8010 Graz, Austria
{chen, wotawa}@ist.tugraz.at

Abstract. During the last decade many computer-aided debugging tools have been developed to assist users to detect program errors in a software system. A good example are model checking tools that provide counterexamples in case a given program violates the specified properties. However, even with a detailed erroneous run, it remains difficult for users to understand the error well and to isolate its root cause quickly and cheaply. This paper presents object store models for diagnosing program errors with light-weighted specifications. The models we use can keep track on object relations arising during program execution, detect counterexamples that violate user-provided properties, and highlight statements responsible for the violation. We have used the approach to help students to locate and correct the program errors in their course works.

1 Introduction

Building reliable software is often an onerous task in the real development process. Quite often, bugs in software systems can take days or weeks to debug. To reduce human debugging time, many computer-aided debugging systems [8, 2, 5, 1, 12, 3] have been developed to help users find program errors in various cases. In particular, program verification tools [1, 3] aid users to check whether a software system meets the properties. They detect program errors in various cases and reveal the violation of properties by providing the user with detailed counterexamples. However, manual inspection of program failures is time consuming, even with a detailed trace of a failure in hand.

A tool that helps programmers quickly diagnose program failures is desirable in terms of time to the market and costs for software development. We are interested in *Fault localization* that provides a way to aid users in moving from a trace of failure to an understanding of the error, and even perhaps to a correction of the error. A basic notion shared by researchers in the area of fault localization [4, 12, 1, 6, 9] is that to explain something is to identify its cause[6].

⋆ The work presented in this paper was funded by the National Natural Science Foundation of China (NSFC) Project 60203015 and the Austrian Science Fund (FWF) P15265-N04.

Several approaches have been proposed to localize program errors automatically. Among them are counterexample explanation [1, 6], specification-assisted error localization [4], delta debugging [12], and model-based software debugging (MBSD) [9, 11]. Counterexample explanation identifies the root cause of a detected bug by examining the differences between an erroneous run and the correct run which is close to the erroneous one. In delta debugging [12] possible error locations are highlighted by conducting a modified binary search between a failing and a succeeding run of a program. The Archie system [4] localize the error of data structure inconsistency by minimizing the distance between the error and its manifestation as observably incorrect behavior.

We have developed a Model-Based Software Debugging (MBSD) which applies a model-based diagnosis technique [10] to fault localization. Given a program and a test case to witness the failure, the MBSD compiles the input program into component networks where each statement is mapped to a component whose behavior captures the statement's semantics. The logic behind the MBSD is that components are blamed since assuming the correctness of their statements leads to a failing run. In the MBSD framework, the functional dependency model (FDM) [11] and the value-based model (VBM) [9] handle the code very well and successfully localize the statements responsible for the incorrect program behavior, but they diagnose property violations poorly because they cannot handle the structural properties and their implications very well.

In this paper we propose a program model for diagnosing property violations with light-weighted specifications. This model handles run-time object relations and their compile-time abstractions. It provides users a means to specify structural properties and returns a quality diagnosis of property violations.

The rest of the paper is organized as follows. In Section 2 we introduce our approach by using a motivating example. Then we introduce the specification in Section 3 and the generation of the program model in Section 4. The experimental results given in Section 5 reveal that the program model provides a useful means for diagnosing common structural errors for some classes of programs. Finally, we summarize the paper.

2 A Motivating Example

We present in this section an example to show how a generated program model assist users in understanding the essence of a failing run that violates required properties.

To motivate and illustrate our technique, we use a Java program in Figure 1, which operates on a linked list. The list is implemented by the class *LinkedList* that provides methods to insert elements, remove elements, and reverse elements. This simple data structure comes with a structural constraint as follows:

Property 1. *List l is always acyclic.*

The code is truly simple. However we have already seeded a bug in the code. What can go wrong is that the $insert(v)$ does not respect Property 1; it creates a cyclic list when it is ever called on a list with a single element.

```
class LinkedList {
  LinkedList next;
  Object value;
  ...
  LinkedList(Object o){
    next = null;
    value = o;
    }
  boolean nextIsEmpty(){
    return (this.next == null)
    }
  ...
  void insert(Object v) {
      LinkedList c = this;
      LinkedList p = this;
      while (!c.nextIsEmtpy()&&(v>c.value)){
        p = c;
        c = c.next;
      }
      if (p.nextIsEmpty()||(v!=p.value)){
        p.next = new LinkedList();
        p = p.next;
        p.value = v;
        p.next = c;
      }
    }
}
```

Fig. 1. A Java example of *LinkedList*

Invoking the buggy $insert(v)$ method on a list possibly corrupts the list. However, the corrupted list can grow further with new elements. So we have to wait even longer until the corrupted list manifests itself as observably incorrect behavior.

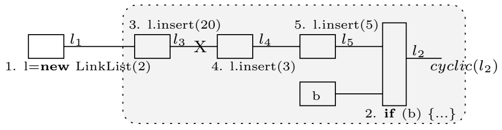
To see where it goes wrong, we write a $demo(b)$ method in Figure 2(a), where the corrupted list l manifests itself as an infinite loop because the $size()$ method is going through the entire list to calculate the length.

Our approach on fault localization is an application of the standard Model-Based Diagnosis [10]. Formally, a diagnosis system is a tuple $(SD, STMNTS, SPEC)$, where SD is a logical description of the program behavior, $STMNTS$ the set of statements, and $SPEC$ denotes the light-weighted specification of correctness. For instance, a test case specifies the input data and the expected output data. The program fault on the other hand is a set of system components, i.e., statements or expressions, which are responsible for the failure.

To illustrate the resulting description, Figure 2(b) displays the graphical representation of the system description of the statements $1 \sim 5$, where each statement

```
void demo(boolean b){
1. LinkedList l = new LinkedList(2);
2. if (b) {
     3. l.insert(20);
     4. l.insert(3);
     5. l.insert(5);
     }
/*@l.next.next.next.next!=l@*/
/*@¬ cyclic(l) @*/
6. l.size();
}
```

(a) The code



(b) The program model

Fig. 2. A Java example of *LinkedList*

is mapped to a component, and connection l_i holds the value of the list l after statement i. These components are connected because they manipulate the same list. In this way, the program model is defined by the structure of a component network and behaviors of all components. While the structural part corresponds to syntactic entities, the behavior part implements the language semantics.

The diagnosis process is a process of searching for possible bug locations by assuming how statements might behave. We represent the *correctness assumptions* about the behavior of statements in terms of predicates assigning appropriate modes to the statements. Formally, the diagnosis process is a searching process to find a set of assumptions that is consistent with the given specification:

Definition 1 (Mode Assignment)
A mode assignment for statements $\{S_1, ..., S_n\} \subseteq STMNTS$, each having an assigned set of modes ms and a default mode $default$ such that $default \in mc(S_i)$ for each $S_i \in STMNTS$, is a set of predicates $\{m_1(S_1), ..., m_n(S_n)\}$ where $m_i \in mc(S_i)$ and $m_i \neq default\ (S_i)$.

Consider the $demo(b)$ method, statements have the modes $\neg AB$ (not abnormal) and AB (abnormal), referring to the assumption of correct and incorrect behavior respectively. The goal of the diagnosing process is to find a set of assumptions that is consistent with the given specification.

Definition 2 (Diagnosis)
A set $\Delta \subseteq STMNTS$ is a diagnosis for a diagnosis problem $(SD, STMNTS, SPEC)$ iff $SD \cup SPEC \cup \{\neg AB(\mathrm{S}) \mid S \in STMNTS \setminus \Delta\}$ is consistent.

Given the test case, input and output values are propagated forward and backward throughout the network. A contradiction is raised in the diagnosis system when (1) a variable gets two or more different values from different components, or (2) a certain property is violated at the output ports but not at the input ports. A conflict is defined by a set of components causing the contradictions.

Choosing $\neg AB$ mode as the default mode, we cannot assume statements 1 $\sim$ 5 in our example work correctly because the output list l_2 after statement 2 violates Property 1, i.e., $\neg cyclic(l)$ in Figure 2(b). To diagnosis this failing execution trace [1, 2, 3, 4, 5], we start from the last statement 5, go back through all statements [1, 2, 3, 4, 5], compute the witness $cyclic(l)$ at each statement, and thus to see where the data structure inconsistency actually originates. A contraction is thus raised at connection l_3, marked by X in Figure 2(b). This is because component $l.insert(20)$ receives an acyclic list but sends a cyclic list. So we cannot assume statements 1, 2 and 3 work correctly at the same time. Thus we have three single fault diagnoses $\{AB\ (1)\}$, $\{AB\ (2)\}$, and $\{AB\ (3)\}$. The diagnoses pinpoint the flaw in the $insert(v)$ method, i.e., the list become cyclic when it is used to insert the second element. This is informative for the user, giving a hint on how the flaw could be corrected.

In contrast, the VBM is less informative. An assertion $l.next.next.next.next\,! = l$ is used to specify the expected behavior of the $demo(b)$ method. Surely this assertion is violated. The VBM's diagnosis is that all statements $1, 2, 3, 4, 5$ are possibly faulty because they influence the value of the assertion. Of course, the diagnosis is not false, but it obscures the original source of error.

3 Specification over Object Store

Structural properties are formulas defined over object relations. Since objects have types, object relations are thus typed.

Definition 3 (Object Relation)
An object relation R with type $(T_1 \rightarrow ... \rightarrow T_k)$, denoted by $R : T_1 \rightarrow ... \rightarrow T_k$, is a set of tuples $(o_1, ..., o_k)$ such that for $1 \leq i \leq k$, object o_i is of type T_i.

Let's call an object relation with k-tuples a *k-relation*. 1-relations and 2-relations are said to be *unary* and *binary*.

Example 1. *We think of a unary relation as a table with a single column, a binary relation as a table with two columns.*

1. *Let x be a program variable that references an object o of type T. Then $x : T$ is a unary relation of type T, which is a singleton set $\{(o)\}$.*
2. *Let A be a set of objects of type T. Then $A : T$ is a unary object relation of type T, which is a set $\{(o) \mid o \in A\}$.*
3. *The next field of a LinkedList makes a binary relation next : LinkedList $\rightarrow$ LinkedList.*

A data structure explicitly declares various binary relations. For any field f of an object x, f is a binary relation because $x.f$ can access at most one object. So we have:

Corollary 1. *Let $x.f$ be a field access that represents an object. f is a binary relation.*

Set operators and relational operators provide us a means to derive new relations. The relational operators in our concern are *concatenate* and *join*[1].

Definition 4 (Concatenate Operator)
Let $p : T_1 \rightarrow ... \rightarrow T_k$ and $q : T'_1 \rightarrow ... \rightarrow T'_m$ be two relations. The concatenate $p \oplus q$ of relations, with type $(T_1 \rightarrow ... \rightarrow T_k \rightarrow T'_2 ... \rightarrow T'_m)$, is a set $\{(p_1, ..., p_k, q_2, ..., q_m) \mid (p_1, ..., p_k) \in p$, there is a tuple $(q_1, ..., q_m) \in q$, such that $p_k = q_1$ and $T_k = T'_1\}$.

Definition 5 (Join Operator)
Let $p : T_1 \rightarrow ... \rightarrow T_k$ and $q : T'_1 \rightarrow ... \rightarrow T'_m$ be two relations. The join $p \circ q$ of relations, with type $(T_1 \rightarrow ... \rightarrow T_{k-1}, \rightarrow T'_2 ... \rightarrow T'_m)$, is a set $\{(p_1, ..., p_{k-1}, q_2, ..., q_m) \mid (p_1, ..., p_k) \in p$, there is a tuple $(q_1, ..., q_m) \in q$, such that $p_k = q_1$ and $T_k = T'_1\}$.

[1] Essentially they generalize the standard *product* and *join* operators.

If we apply the *concatenate* operator and *join* operator on the same input relations, the *join* contains less columns. For example, a *Tree* class declares two binary relations: $\mathit{left} : \mathit{Tree} \rightarrow \mathit{Tree}$ and $\mathit{right} : \mathit{Tree} \rightarrow \mathit{Tree}$. Whereas $\mathit{left} \oplus \mathit{right}$ is a 3-relation that concatenates *left* and *right*, $\mathit{left} \circ \mathit{right}$ is a binary relation. Moreover, let $A : \mathit{Tree}$ be a unary relation holding a set of objects of type *Tree*, the joint $A \circ \mathit{left}$ is a set of objects accessed by the objects in A through the field *left*.

By repeatedly applying the *join* operator, we can compute the **transitive closure** of a binary relation R, denoted by R^*. Given binary relations, we can create new k-relations by concatenating them and joining them. Therefore, we just keep track on binary relations when modeling the input program.

Putting them together, we write structural properties as formulas. For example, Property 1 is represented by:

$$\forall x \in l \circ \mathit{next}^* \Rightarrow x \notin x \circ \mathit{next}^* \tag{3.1}$$

which says that a list l is acyclic if there is no element in list l that can access itself.

Object relations have their origins and histories; they are like variables in that they have different values at different program points. So a relation is said to be a **parent relation** if it is the origin of other relations. An object relation is a **child** if it has a parent relation. Moreover, we introduce the term **relation variable**.

Definition 6 (Relation Variable)
A relation variable is a variable $T.f$, where f is a binary relation $f : T \rightarrow T'$. The value assigned to a relation variable is a set of pairs in the form (i_1, i_2) where i_1 and i_2 are of type T and T' respectively.

Since we implement a set of methods to perform a consistency checking on a relation variable, the violation of structural properties is detected if we know the variable value.

Definition 7 (Object Store)
An Object Store is a collection of relation variables and their values.

4 Model Building Process

In this section, we present the algorithm for generating object store models. Throughout this section, L is abbreviated for *LinkedList*.

To compile the program into models, we assume each syntactic entity has a function *buildOSM* which maps itself into a component, links its input ports, possibly propagate forwards static information, and returns its output connection. Statement by statement, we convert classes and methods successively and return a set of components defining the diagnosis system.

The static information in our model are location pairs, denoted by pairs of numbers that abstract the run-time objects and approximate the semantics of

```
ObjectCreation ::=       $buildOSM(v = \textbf{new}\ C, env)$
    $v$ = new C              $c = newComp(assignment, \textbf{new}\ C, env)$
                             for all $v_p \in parent(env, v)$
                                 $c' = newRelation(c, pointstoRelation, v_p, env)$
                                 $propagate(c')$
                             endfor

ObjectAssignment ::=     $buildOSM(v = w, env)$
    $v$ = w                  $c = newComp(assignment, w, env)$
                             for all $v_p \in parent(env, v)$
                                 $c' = newRelation(c, pointstoRelation, v_p, env)$
                                 $propagate(c')$
                             endfor

FieldAccess ::=          $buildOSM(v.f, env)$
    $v.f$                    $c = \textbf{new}\ C_{fieldaccess}$
                             $addComp(env, c)$
                             $in(c) = conn(env, v)$
                             $out(c) = conn(env, v.f)$
                             for all $v_p \in parent(env, v)$
                                 $c' = newRelation(c, v_p, objectRelation, env)$
                                 $propagate(c')$
                             endfor

FieldAssignment ::=      $buildOSM(v.f = w, env)$
    $v.f = w$                $c = newComp(fieldassignment, w, env)$
                             for all $v_p \in parent(env, v)$
                                 $c' = newRelation(c, objectRelation, v_p, env)$
                                 $propagate(c')$
                             endfor

IfStatement ::=          $buildOSM(\textbf{if}\ (Exp)\ S_1\ \textbf{else}\ S_2, env)$
  if Expr                    $c = \textbf{new}\ C_{if}$
      S$_1$                  $addComp(env, c)$
  else                       $cond = buildOSM(Exp, env)$
      S$_2$                  $oldpath = path(env)$
                             $path(env) = path(env) + 1$
                             Let $env'$ be a copy of $env$
                             $buildOSM(S_2, env')$
                             $path(env) = path(env) + 1$
                             Let $env''$ be a copy of $env$
                             $buildOSM(S_1, env'')$
                             create an input port of $c$ and connect it to $cond$
                             Let $A = modifiedConn(env') \cup modifiedConn(env'')$
                             for all $x \in A$
                                 if $x \in modifiedConn(env')$
                                     create an input port of $c$ and connect it to $x$
                                 endif
                                 if $x \in modifiedConn(env'')$
                                     create an input port of $c$ and connect it to $x$
                                 endif
                                 create an output port of $c$ and connect it to a new
                                     connection $x'$ named by $x$
                                 remove $x$ from $modifiedConn(env)$
                                 add $x'$ into $modifiedConn(env)$
                             endfor
                             $path(env) = oldpath$
```

Fig. 3. Algorithm for model building

four syntactic entities: class creation, object variable assignment, field access and field assignment. To describe the history of a relation, we introduce indexed relation variables as follows:

Definition 8 (Indexed Relation Variable)
An indexed relation variable is $NAME[PATH]_{IDX}$, where $NAME$ is a relation variable name, $PATH$ is a sequence of numbers denoting execution branches, and IDX is an index.

Similar to [9], the algorithm maps loops and method calls to hierarchic components with inner sub-models. The loop component contains two sub-models: M_C and M_B, where M_C denotes the sub-model of the loop condition, and M_B the sub-model of the loop body (represented as a nested if-statement[2].

We further assume that *env* represents the working environment of components, connections, and the indices assigned to variables. The algorithm is summarized in Figure 3, where c denotes a component and the following auxiliary functions are used:

- Function $addComp(env, c)$ adds a component c into the environment env.
- $in(c)$ denotes the input connections of c.
- $out(c)$ denotes the output connections of c.
- Function $propagate(c)$ receives the static information from the input ports of a component c, stores the input instance, unifies it with the value of the parent relation, and propagates them to the output port of c.
- Function $conn : (ENV, EXP \cup Var) \mapsto CONNS$ maps expressions or variables to connections by using an environment.
- Function $parent : (ENV, EXP) \mapsto CONNS$ looks up in the environment for a set of parent relations named by the input expression[3]. If none, a parent relation is created.
- Function $modifiedConn(env)$ returns a set of connections denoting variables with new values.
- $path(env)$ is a number denoting the current branch.
- $newComp(type, exp, env)$ is a function that returns a new component of *type*, which is initialized by the following steps:
  ```
  1: $c = \textbf{new}\ \ C_{type}$
  2: $addComp(env, c)$
  3: $out(c) = conn(env, exp)$
  4: $in(c) = buildOSM(exp, env)$
  5: $return\ c$
  ```
- $newRelation(c, v_p, type, env)$ is a function that returns a new relation component of *type*, which is initialized by the following steps:

[2] The nesting size is obtained by computing all pairs shortest path in a dependency graph (see [11]).

[3] For an object variable w, the name is $T.pt$, where T is w's class type. For a field access $v.f$, the name is in the form of $T.f$ where T is w's class type.

```
1: $c' =$ new $C_{type}$
2: $addComp(env, c')$
3: $out(c') = conn(env, v_p)$
4: $in(c') = \{v_p\} \cup out(c) \cup in(c)$
5: $return\ c'$
```

5 Experimental Results

The experiments are performed on students' programs for the identical assignments in a programming course. Most of the assignments requires various data structures such as linked list, stack, tree, etc. All programs involve various control flows, virtual method invocation, and object-oriented language notations, such as multiple objects, class creations, instance method calls, class and instance variables, etc.

Given properties and test cases, fifty students are grouped into three groups: G1, G2 and G3, where G1 members are asked to locate and eliminate the errors in their programs using traditional debugging tools, while G2 and G3 are assisted with the debugging tool we developed, using the VBM and object store models respectively. Table 1 presents the average number of minutes used by each group to locate and correct the identical program errors.

We also compare the performance of the VBM and object store models running against the identical programs with seeded errors in Table 2, where we depict the elapse time for modeling (M-G column), the elapsed time for computing diagnoses (T-D column), and the number of diagnosis (N-D column). The right column lists the result obtained by diagnosing with a VBM. Compared with the VBM, it is shown that the number of diagnosis candidates is reduced and all diagnosis candidates are in the VBM's diagnosis. This accounts for why members in G3 use less time than those in G2 to locate and eliminate the identical program errors.

Table 1. Error corrections

Group	Error 1 in Shape	Error 2 in Stack	Error 3 in ExpressionTree
G1	5	3	11
G2	3.5	1	8
G3	2.5	0.5	4

Table 2. Comparison of the program models

Program	M-G [sec.]	T-D [sec.]	N-D [#]	VBM N-D
LinkedList	0.6	0.3	3	10
Stack	0.6	0.4	1	4
	0.9	0.4	2	4
Shape	2.3	0.3	1	5
ExpressionTree	7.2	2.5	4	9

6 Related Work

During the last decade many computer-aided debugging tools have been developed to assist users to find program errors in a software system. The error detection techniques used by these tools are static and dynamic analysis, program slicing, symbolic execution and model checking.

In [8] Jackson introduces Aspect, an efficient specification-assisted approach for error detection. The Aspect specification is in the form of abstract dependencies, and its scheme is to check dependencies required by the specification against those implied by the source code. It is good at catching errors of missing variables.

The PREfix tool [2] detects anomalies by symbolic execution of code. It uses path-sensitive analysis to explore multiple execution paths in a function, with the goal of finding path conditions under which undesirable properties like null pointers hold. Carefully heuristics are needed to detect errors without generating too many spurious reports.

ESC [5] uses a powerful tailored theorem prover to check code against user-supplied annotations. It has been successfully applied to a particular class of program errors such as out-of-bounds array access, null pointer dereferencing and unsound use of locks.

Using program slicing and shape analysis, the Bandera project [3] is developing a toolkit that extracts finite state machines from code, which can thus be used by model checkers. The SLAM project combines symbolic execution and model checking to produce error traces in order to localize the fault in the source code [1]. Groce and Visser [7] attempt to extract information from a single counterexample produced by model checking in order to facilitate the understanding of malfunctioning systems.

The Archie system [4] successfully localize the error of data structure inconsistency. But there is no guarantee that the original source of all data structure corruption errors are captured in Archie because the consistency checker is invoked periodically.

7 Conclusion

In this paper, we present object store models to diagnose data structure inconsistencies. Our approach handles both the structural properties and their implications by reasoning about object relations arising from the program execution. We have used the approach to help students to locate and correct the program errors in their course works.

We will work on extending the model to handle programs with exceptions, threads and recursive method calls, and exploring how static analysis can assist us to rank user-provided properties.

References

1. T. Ball, M. Naik, and S.K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proc. of POPL*, pages 97–105. ACM Press, 2003.
2. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7): 775–802, 2000.

3. James C. Corbett. Using shape analysis to reduce finite-state models of concurrent Java programs. *ACM Transactions on Software Engineering and Methodology*, 9(1):51–93, January 2000.
4. Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. *ACM SIGPLAN Notices*, 38(11):78–95, 2003.
5. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report SRC-RR-159, HP Laboratories, 1998.
6. A. Groce. Error explanation with distance metrics. In *TACAS*, volume 2988 of *Lecture Notes in Computer Science*. Springer, 2004.
7. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *10th International SPIN Workshop on Model Checking of Software*, 5 2003.
8. Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *ACM TOSEM*, 4(2):109–145, 1995.
9. W. Mayer, M. Stumptner, D. Wieland, and F. Wotawa. Can ai help to improve debugging substantially? debugging experiences with value-based models. In *Proc. ECAI*, pages 417–421. IOS Press, 2002.
10. Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
11. D. Wieland. *Model-Based Debugging of Java Programs Using Dependencies.* PhD thesis, Vienna University of Technology, Institute of Information Systems (184), Nov. 2001.
12. Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), 2002.