DIAGNOSING JAVA PROGRAMS WITH STATIC ABSTRACTIONS OF DATA STRUCTURES

Rong Chen [1,2], Daniel Koeb [1] and Franz Wotawa [1] *

[1] *Technische Universitaet Graz, Institute for Software Technology, 8010 Graz, Inffeldgasse 16b/2, Austria;* [2] *Institute of Software Research, Zhongshan University, Xingangxilu 135, 570215 Guangzhou, China*

Abstract: Model-based software debugging helps users to find program errors and thus to reduce the overall costs for software development. In this paper, we extend our previous work to diagnose common data structure errors. The proposed logical program model derives from a collection of indexed object relations, which capture the underlying data structures at the abstraction level of objects. A case study suggests that the counterexample with the diagnoses can help the user to understand the nature of program errors and thus speed up error correction.

Key words: Automatic reasoning, model-based diagnosis, fault localization

1. INTRODUCTION

Model-based software debugging (MBSD) helps users to find program errors and thus to reduce the overall costs for software development [1]. MBSD starts from a description of a working piece of software. This description can be automatically extracted from the software and captures its structure and behavior. Using this model, one can make predictions in terms of values of variables under certain circumstances. If any of these values contradict the test case (i.e., the software does not behave as expected), the diagnostic system will isolate which statement or expression accounts for this contradiction.

* Authors are listed in alphabetical order. The work presented in this paper was funded by the Austrian Science Fund (FWF) P15265-N04, and partially supported by the National Natural Science Foundation of China (NSFC) Project 60203015 and the Guangdong Natural Science Foundation Project 011162.}

To diagnose common data structure errors, we, in this paper, propose a logical program model, derived from a collection of indexed object relations, which capture the underlying data structures at the abstraction level of objects. A case study suggests that the counterexample with the diagnoses can help the user to understand the nature of program errors and thus speed up error correction.

2. ABOUT OBJECT STORE

The Object Store is defined by locations (to represent run-time objects), indexed location relations (defined over locations) and variable assignments (associated with indexed location relations). We briefly introduce the Object Store by using the *List* program shown in the leftmost column of Table 1, where *L* is abbreviated for *List*.

Table 1. The Object Store of the *remove* method in Fig. 1

Program/statements	Object Store	
class List {	Points-to	Indexed location relations
List next;		
int value;		
. remove(int v) {		
1. List c = this;	$(c_1, 0)$	
2. List p = this;	$(p_1, 0)$	$L.next_1 = \{\}_{\{c/0,\ p/0\}}$
3. **while** ((c.next != null)&&(v > c.value)){ 3.1 p = c; 3.2 c = c.next; }		
/* case (a): loop body not executed */		
		$L.next_1 = \{\}_{\{c/0,\ p/0\}}$
/* case (b): unfold loop once */		
3.1 p = c;	$(p_2, 0)$	
3.2 c = c.next;	$(c_2, 1)$	$L.next_2 = \{(0, 1)\}_{\{c/1,\ p/0^*\}}$ (from $L.next_1$)
/* case (c): unfold loop two times */		
3.1 p = c;	$(p_3, 1)$	
3.2 c = c.next;	$(c_3, 2)$	$L.next_3 = \{(0, 1), (1, 2)\}_{\{c/2,\ p/1^*\}}$ (from $L.next_2$)
/* merge location relations for various cases*/		
		$L.next_4 = \phi(L.next_1, L.next_2)$ $L.next_5 = \phi(L.next_2, L.next_3)$
4. **if** (v = p.value) { 4.1 p.next = c; }		
4.1 p.next = c;		$L.next_6 = \{(0, 0)\}_{\{c/0,\ p/0^*\}}$ (from $L.next_1$) $L.next_7 = \{(0, 1)\}_{\{c/1,\ p/0\}}$ (from $L.next_2$) $L.next_8 = \{(0, 1), (1, 2)\}_{\{c/2,\ p/1\}}$ (from $L.next_3$)
/* merge location relations of the if-statement */		
		$L.next_9 = \phi(L.next_1, L.next_6)$ $L.next_{10} = \phi(L.next_2, L.next_7)$ $L.next_{11} = \phi(L.next_3, L.next_8)$
} }		

L -- List

Locations, denoted by positive integers, are abstract representations of run-time objects. A **points-to relation** is a binary relation that relates an object variable and its locations. Since variables/relations might be defined/updated at different program points, we distinguish them by assigning a unique index. In our example, statement *List c = this* is converted into a pair (c_1, 0) in a points-to relation in Table 1, which means variable *c* points to the object at location 0 (i.e. *this*).

A **location relation** is used to relate multiple locations.

Definition 1 A location relation, denoted by *T.f*, is a set of *n*-tuples in the form $(i_1,...,i_n)$ where $i_k(1 \le k \le n)$ are integers (not *nil*) representing locations, *T* is a class name, and *f*, of reference type, is a field of class *T*.

Furthermore, let an *n*-tuple $(i_1,...,i_n) \in T.f$, we say location i_j can **reach** i_k (or i_k is **reachable**) when $1 \le j < k \le n$.

We distinguish location relations by assigning a unique index (thus called **indexed location relations**) and a **variable assignment**, which is a set of variable-location associations that describe the points-to pairs at a certain program point. In table 1, $L.next_2 = \{(0, 1)\}_{\{p/0,c/1\}}$ is an indexed location relation, where a location pair (0, 1) is placed because locations 0 and 1 are of type *List*, and the object at location 1 is reachable from the object at location 0. Moreover, the variable assignment {*p*/0, *c*/1} means variables *p* and *c* point to locations 0 and 1 respectively.

Note that locations are introduced for modeling a reference variable, a field access and a parameter of method call. A **used location** is a location which is explicitly used in class creation statements, or when its content is explicitly used. In our example, location 0 becomes a used location because its content *next* is explicitly used when modeling the field access expression *c.next* on statement 3.2 (In Table 1 a star is used for marking used locations).

With the notations that we have already introduced, we can formally define the Object Store as follows:

Definition 2 Object Store is a collection of points-to relations and indexed location relations.

The Object Store in Table 1 is extracted from the input program *List* automatically by accessing the structure of the data and converting the statements into points-to relation and indexed location relations. Since there are various control flows that may update location relations, we adopt the static single assignment form (SSA, see [2]); that is, we insert extra pseudo-assignments (known as ϕ-functions) at control flow merge points. In our example, $L.next_1$ arises from the then-branch of if-statement 4, and $L.next_6$ from the else-branch. Then we insert a pseudo-assignment $L.next_9 = \phi(L.next_1, L.next_6)$ right after the if-statement 4. If the path of execution comes from the then-branch, the ϕ-function takes the value of $L.next_1$; otherwise, it takes the value of $L.next_6$.

3. DIAGNOSING WITH LOGICAL RULES FROM THE OBJECT STORE

Since our diagnostic system uses a theorem prover to find conflict sets and thus to compute diagnoses [1,3], we map each certain indexed location relation into a logical rule. In our example, the Object Store is mapped into the following logical rules:

$$\neg AB(1) \wedge \neg AB(2) \rightarrow ok(L.next_1) \quad (1)$$
$$\neg AB(3) \wedge ok(L.next_1) \rightarrow ok(L.next_2) \quad (2)$$
$$\neg AB(3) \wedge ok(L.next_1) \wedge ok(L.next_2) \wedge \neg ok(L.next_4) \rightarrow \perp \quad (3)$$
$$\neg AB(3) \wedge ok(L.next_2) \rightarrow ok(L.next_3) \quad (4)$$
$$\neg AB(3) \wedge ok(L.next_2) \wedge ok(L.next_3) \wedge \neg ok(L.next_5) \rightarrow \perp \quad (5)$$
$$\neg AB(4) \wedge ok(L.next_1) \rightarrow ok(L.next_6) \quad (6)$$
$$\neg AB(4) \wedge ok(L.next_1) \wedge ok(L.next_6) \wedge \neg ok(L.next_9) \rightarrow \perp \quad (7)$$
$$\neg AB(4) \wedge ok(L.next_2) \rightarrow ok(L.next_7) \quad (8)$$
$$\neg AB(4) \wedge ok(L.next_2) \wedge ok(L.next_7) \wedge \neg ok(L.next_{10}) \rightarrow \perp \quad (9)$$
$$\neg AB(4) \wedge ok(L.next_3) \rightarrow ok(L.next_8) \quad (10)$$
$$\neg AB(4) \wedge ok(L.next_3) \wedge ok(L.next_8) \wedge \neg ok(L.next_{11}) \rightarrow \perp \quad (11)$$

Where predicate $AB(i)$ means statement i is abnormal. Moreover, we expect the following properties:

Property 1 *If the list is not empty before, then one cell with the target value is removed afterward.*

Property 2 *If the list is acyclic before, then it is acyclic afterwards.*

The Object Store provides an appropriate computing environment for checking user-expected properties. In our example, $L.next_6$ violates Property 2 because location 0 can reach itself. Since $L.next_9 = \phi(L.next_1, L.next_6)$, one observation is $\neg ok(L.next_9)$. By calling the diagnosis engine with rules (1)~(11), we get one conflict set $\{\neg AB(1), \neg AB(2), \neg AB(4)\}$, which tells us that one of statements 1, 2 and 4 is faulty (i.e., diagnoses, see [3]).

Furthermore, $L.next_7$ and $L.next_8$ are counterexamples of Property 1 because they are the same as their parent locations $L.next_2$ and $L.next_3$. Therefore, the diagnoses and the counterexamples $L.next_6$, $L.next_7$ and $L.next_8$ give the user some hints, help him to understand the nature of program misbehavior, and then correct the program error quickly.

REFERENCES

1. C. Mateis, M. Stumptner, and F. Wotawa. Modeling Java Programs for Diagnosis, ECAI 2001, Berlin Germany.
2. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph, ACM TOPLAS, 13(4): 451-490, 1991.
3. R. Reiter. A theory of diagnosis from first principles, AI, 32(1):57-59, 1987.