Introducing Alias Information into Model-Based Debugging

Daniel Köb[1] and **Franz Wotawa**[1 2]

Abstract. Model-based diagnosis applied to computer programs has been studied for several years. Although there are still weaknesses in the used models, especially on dealing with dynamic data structures, the approach has been proven useful for automatic debugging. The weaknesses stem from the fact that heap objects are modeled without considering alias information. Our approach extends the modeling process with a static points-to analysis that reveals the structure and relations between heap objects. This points-to information is then used to improve existing value-based models for Java programs such that the diagnosis engine is able to differentiate between separate data structures. With this extension the set of diagnoses can be reduced for certain types of programs.

1 INTRODUCTION

Automatic debugging of software using model-based diagnosis techniques requires quite sophisticated models in order to achieve reasonable fault localization results. Several models have been proposed with different advantages and drawbacks [2, 5, 7]. Common to all these models is their weakness in diagnosing software that makes heavily use of dynamic data structures. In this paper we propose an extension atop the value-based model [4] for diagnosing Java programs, which overcomes this weakness in some cases.

Value-based models are made up of components that represent the different entities of the programming language and connections between these components representing variables and objects stored on the heap. The term value-based arises from the fact that values are propagated between the components via the connections. The components themselves simulate the data flow of the program entities they represent. Therefore, it is possible to trace forward and backwards to locate those parts of the program, that are responsible for a discrepancy between the values computed by the program, and the expected values. In diagnosis such traces correspond to *conflict sets*, which are the fundamentals for diagnosis computation. An essential part of the definition of a diagnosis is, that a diagnosis for a system must be able to explain *all* observed discrepancies, otherwise it is not a diagnosis. Exactly at this point the weakness in existing models for software diagnosis emerges.

The value-based model represents class instances by a unique identifier and a mapping from this identifier to the member variables of the instance. Operations on data structures are then modeled as modifications of these mappings. Although the shape of the data structures is implicitly stored within the mappings, the value-based model is not capable to profit from this information. Thus, every expression or statement that modifies a single object is immediately suspected to change all objects of the same type, even if it is statically known which object is affected.

In this paper we propose an extension for the value-based model that explicitly stores the shape of data structures and captures the shape transformations performed by the statements of the program. This results in a more precise model according to the language semantics, and therefore provides better fault localization ability.

Section 2 provides a detailed description of the problems that arise in the presence of dynamic data structures and how to manage them. The notations used throughout this paper are described in Section 3. In Section 4 a formal foundation for the model extensions is given. Finally, we discuss related work and some conclusions in Section 5.

2 PROBLEM REPRESENTATION

A frequent issue in model-based diagnosis of software is the reduction of the number of reported (possible) fault locations. The only way to improve diagnosis results without the requirement of providing additional information is to improve the used model. The model provides dependence information from which the diagnosis engine infers responsibilities for misbehavior. Hence, improving a model means extending it with additional dependencies. Finding new dependencies requires a very subtle analysis of the program text and a precise knowledge of the programming language semantics.

The inability of existing value-based models to distinguish between independent data structures may lead to unexpected diagnoses. Thus, if one is able to represent independent data structures in the model and define new dependencies for it, an improvement in diagnosis results can be expected. Suppose the following erroneous code fragment that is used to destructively invert the elements of a stack by adding them to a new stack. The class `LinkedStack` contains a reference to the first element of a linked list that contains integer numbers. The linked list is built by objects of type `ValueHolder` that hold a reference to their predecessor. Methods `size` and `remove` implement the expected behavior, method `item` returns the top element of the stack (i.e. an integer) and method `put` lays the given integer on top of the stack (i.e. add a new element to the list).

```
    LinkedStack invertStack(LinkedStack s) {
 1.    LinkedStack inv = new LinkedStack();
 2.    int items = s.size();
 3.    while (items > 1) {
 4.       inv.put(s.item());
 5.       s.remove();
 6.       items --;
 7.    }
 8.    return inv;
    }
```

The error introduced into the loop condition in line 3 causes the loop to iterate once too little, thus, ending up with one ele-

[1] Institute for Software Technology, Graz University of Technology, Austria email: {dkoeb,wotawa}@ist.tugraz.at
[2] Authors are listed in alphabetical order.

ment still left on the stack `s` and one element missing on stack `inv`. The (single fault) diagnoses retrieved from the loop-free value-based model (see [4]) are [`inv=new LinkedStack()`:1], [`items=s.size()`:2], [`items > 1`:3], [`inv.put(...)`:4], [`s.remove()`:5], and [`items--`:6]. But neither the diagnosis of line 1 nor the ones from lines 4 and 5 are able to really explain the discrepancy in both stacks. That is, they may explain why the values in the two stacks are wrong, but they are not able to explain why there is an element missing in one stack, and why there is an element too much in the other, respectively. For example consider the method call `s.remove` in line 5. It removes an element from stack `s`, thus, it may be responsible for the observations on `s`. But since the connection representing stack `s` is also used to represent stack `inv` it may also be responsible for the values that are put on stack `inv`.

From the above findings emanates that existing models are too imprecise in simulating data flow. Different characteristics of data are joined together and represented by the same entities in the model, therefore loosing the ability to distinguish between them. Hence, model improvements that separate characteristics of data flows are expected to improve diagnosis effectiveness.

The approach we propose in this paper is to expand the value-based model with shape information about data structures. That is we focus on objects created on the heap without referring to the values stored within these objects, except for references to other objects. In addition we divide the heap into independent junks of objects. This means that we are dealing with several object graphs that represent the shapes of independent data structures used throughout the program. For our example program this means that we have two independent shape graphs for variable `s` and `inv` all through the program. The shape graph for variable `s` depends on method calls `s.size`, `s.item`, and `s.remove`, and on the loop condition. Whereas the shape graph for variable `inv` depends on the statement in line 1, the loop condition and the method call `inv.put`. Thus, a value-based model extended with shape graph information will be able to compute the expected diagnoses for this example, namely [`items = s.size()`:2], [`items > 1`:3], and [`items--`:6].

3 TERMINOLOGY AND NOTATION

The major part of the notation used throughout this paper is borrowed from [6]. Our analysis focuses on the set of pointer variables for a given program which is denoted by $PVar$. The term shape nodes used in shape analysis directly corresponds to objects in object-oriented languages like Java. Each shape node has a type, a unique name (e.g. its address in memory), and a set of selectors. The set of selectors of a shape node s (i.e. the member variables of reference type for an object) are denoted by $sel(s)$. The central elements of shape analysis are shape graphs. The following definition provides a formal description of shape graphs.

Definition 1 (Shape Graph). *A shape graph is a finite directed graph that consists of variable nodes and shape nodes and two kinds of edges — variable edges and selector edges. A shape graph is represented by a pair of edge sets, $\langle E_v, E_s\rangle$, where*

- *E_v is the graph's set of variable edges, each of which is denoted by a pair of the form $[x, n]$, where $x \in PVar$ and n is a shape node.*
- *E_s is the graph's set of selector edges, each of which is denoted by a triple of the form $\langle s, sel, t\rangle$, where s and t are shape nodes, and $sel \in sel(s)$.*

Note that the above definition is nondeterministic per se, because it allows multiple edges emanating from a single variable or node selector. In order to precisely represent data structures, it is necessary to define deterministic shape graphs.

Definition 2 (Deterministic Shape Graph). *A shape graph is deterministic if (i) for every $x \in PVar$, $|\{n|[x,n] \in E_v\}| \leq 1$, and (ii) for every shape node s and $sel \in sel(s)$, $|\{n|\langle s, sel, n\rangle \in E_s\}| \leq 1$. The class of deterministic shape graphs is denoted by $\mathcal{DSG}$.*

The set of all shape nodes for a given shape graph (i.e. all objects in memory) are denoted by $shape\_nodes$ which is defined as $shape\_nodes(DSG) = \{n|[*, n] \in E_v\} \cup \{n|\langle n, *, *\rangle \in E_s\} \cup \{n|\langle *, *, n\rangle \in E_s\}$. For simplicity we additionally define $E_v(x)$ as a function returning the node pointed to by variable x or $\emptyset$ if x doesn't point to any node. And we define $E_s(s, sel)$ as function returning the node pointed to by selector sel of shape node s or $\emptyset$ if the selector doesn't point to any node. It will always be clear if the sets or the functions are meant according to whether an argument is provided or not.

Example 1 (Shape Graph). *Figure 1 depicts a shape graph SG that consists of a single variable s and three shape nodes, where node ls is of type `LinkedStack` and nodes v_1,v_2 are of type `ValueHolder`. The selectors of shape nodes are represented as dots where the names of the selectors are placed on the edges emanating from these dots. Since all variables and selectors at most consist of one edge, it is a deterministic shape graph. Furthermore we can state the following for shape graph SG:*

$$\begin{array}{lll} E_v(s) = ls & E_s(ls, top) = v_1 & E_s(v_1, previous) = v_2 \\ shape\_nodes(SG) = \{ls, v_1, v_2\} & & E_s(v_2, previous) = \emptyset \end{array}$$

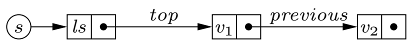


Figure 1. Example of a shape graph

A DSG as defined above represents all objects in memory used by a program. But for our purposes it is required to split it into multiple independent chunks in order to be able to distinguish those parts that are reachable by a variable. The concept of partitions allows us to separate those variables that may point to parts of the same data structure. Since a variable can only point to one object at a time, it can only be part of a single partition. Thus, a partitioning is defined as a collection of pairwise disjunct sets of program variables. For every partition of a given partitioning a unique DSG can be defined that models only a part of the complete storage. The set of DSG's for a given partitioning is called partitioned shape graph (PSG).

Definition 3 (Partitioned Shape Graph). *A PSG for a given partitioning P is a set of DSG's that are mutually disjoint. The DSG's are subscripted by their partition, thus $PSG = \{DSG_X|X \in P\}$ where for all $DSG_X, DSG_Y \in PSG$ with $X \neq Y$ follows that $shape\_nodes(DSG_X) \cap shape\_nodes(DSG_Y) = \emptyset$ must hold. The class of partitioned shape graphs is denoted by $\mathcal{PSG}$.*

Due transformations of a given partitioning of the shape graph it may be necessary to join two shape graph partitions together to form a new one. This is achieved by joining the set of variable edges and the set of selector edges of the two graphs.

$$DSG_{X\cup Y} \overset{\text{def}}{=} \langle E_{v_X} \cup E_{v_Y}, E_{s_X} \cup E_{s_Y}\rangle$$

Based on the above definitions we extend the value-based model with alias information and prepare it for propagation of shape graphs.

4 Diagnosing with Alias Information

The enhancement of the value-based model with shape information first requires a careful analysis of how the various entities of the programming language affect shapes. Based on this information we can analyse the static data flow in a program. The analysis exposes which parts of the program affect a certain data structure. Due to these effects on data structures we can derive new dependencies, that are integrated into the value-based model and used for diagnosis. Our analysis is similar to interprocedural pointer alias analysis [1].

4.1 Extended Alias Analysis for Modeling

Compared to interprocedural pointer alias analysis we are not just interested if two given access paths are may-aliases, instead we need to know if two access paths may refer to parts of the same data structure. With this information we are able to decide if two statements may access or transform the same data structure or not.

Because of branching statements in the programming language, it is not possible to compute the extended aliasing information precisely. Instead only an approximation can be determined by a least fixed point computation in order to preserve as much information as possible. The fixed point computation is based on the domain of partitionings $\mathcal{P}$. This domain is a complete lattice with ordering relation $\sqsubseteq$ defined by

$$P_1 \sqsubseteq P_2 \overset{\text{def}}{=} \forall x \in P_1 : \exists y \in P_2 : x \subseteq y.$$

The join operator $\sqcup$ for P_1 and P_2 is defined over the transitive reflexive closure of the union of P_1 and P_2, where the relation xRy is defined as $xRy = \exists r \in R : x \in r \wedge y \in r$.

$$P_1 \sqcup P_2 \overset{\text{def}}{=} \{a | \forall x, y \in a : x(P_1 \cup P_2)^* y\}$$

The iterative approximation process starts with a model, where every variable of the program resides in its own shape graph partition (i.e. the bottom element $\perp$ of the lattice). Due to the statements and expressions in the program text, the partitioning of the shape graph is changed according to their semantics. For simplicity we preprocess the program text and replace every qualified name, consisting of more than two levels by a fresh variable. Assignment statements that contain the same variable on the left and right hand side are also replaced with a fresh variable, as is shown below.

Original Code	Preprocessed Code
`x = a.b.sel`	`tmp = a.b;` `x = tmp.sel;`
`x = x.sel`	`tmp = x.sel;` `x = tmp;`

The partitioning transforming semantics for the various statements $s \in \mathcal{ST}$ is defined by the function $[\![]\!] : \mathcal{ST} \times \mathcal{P} \to \mathcal{P}$. For simplicity we only include those partitions in the semantics definition, that are affected by the statement, where P_X generally denotes the partition that includes variable x.

Assignment statements of reference type change the shape graph partitioning depending on the kind of target and source expression. They join two partitions, split a single partition into two new ones, or leave them as they are. An assignment of `null` to a simple variable always removes the variable's dependency to the data structure it pointed to before. Thus, the variable is removed from the partition of the data structure and a new one is created for it.

$$[\![\texttt{x = null}]\!](P_X) = (P_{X-\{x\}}, P_{\{x\}})$$

Contrary if an assignment of `null` to a member variable of some object pointed to by a variable is encountered, we are not able to tell if all variables in its partition still point to the same data structure. Hence, the partition is left as it is.

$$[\![\texttt{x.sel = null}]\!](P_X) = (P_X)$$

Creation of a new object with Java's `new` operator is treated as two independent operations, namely the creation of a new object and the call to the appropriate constructor. The assignment of a new object to a variable causes the variable to definitely point to a different data structure than it did before the assignment. Therefore, the variable is removed from its old partition and put into a new one.

$$[\![\texttt{x = new T}]\!](P_X) = (P_{X-\{x\}}, P_{\{x\}})$$

The result for assignments of simple variables to simple variables depends upon the partitions both variables are in before the statement. If both variables are in different partitions, the target variable is removed from its partition and added to the source partition. Otherwise if both variables are within the same partition, the partitioning is not changed. The same holds for the assignment of a member variable to a simple variable.

$$[\![\texttt{x = y}]\!](P_X, P_Y) = (P_{X-\{x\}}, P_{Y\cup\{x\}})$$
$$[\![\texttt{x = y.sel}]\!](P_X, P_Y) = (P_{X-\{x\}}, P_{Y\cup\{x\}})$$

The last possible form of assignments handles an assignment to a member variable of an object pointed to by a simple variable. In this case the partitions of the source and target variables are joined together as long as they are in different partitions. Otherwise the partitioning is not changed.

$$[\![\texttt{x.sel = y}]\!](P_X, P_Y) = (P_{X\cup Y})$$

Return statements are treated like assignment statements to a reserved variable named `return`. To simplify the analysis we assume that preprocessing assured that the return expression only consists of a simple variable name.

$$[\![\texttt{return x}]\!](P_X) = (P_{X\cup\{return\}})$$

Blocks of statements transform the partitioning statement by statement. Thus, we apply the semantics operator in the order the statements occur, where the operator $\circ$ denotes functional composition.

$$[\![\texttt{s1...sn}]\!](P) = ([\![\texttt{s2...sn}]\!] \circ [\![\texttt{s1}]\!])(P)$$

For method calls it is necessary to map the partitioning at the callers site to an appropriate partitioning for the method's body. In order to simplify this mapping we assume, that the preprocessing step replaced the actual parameters with simple variables. The name of the variable on which the method is called is replaced with `this` and the names of the actual parameters are replaced with the names of the formals. At the end of the method body the partitioning has to be mapped back to the names of the caller. This mapping is achieved by an auxiliary function $m : \mathcal{P} \mapsto \mathcal{P}$ that is bijective, thus, m^{-1} denotes the reverse mapping of variable names. If the method returns a value of reference type the special variable name `return` is used.

$$[\![\texttt{s.method(p}_1\texttt{,...,p}_n\texttt{)}]\!](P) = (m^{-1} \circ [\![\texttt{body}_{method}]\!] \circ m)(P)$$

Branching statements like conditionals and loops consist of join points, where the partitioning information of multiple branches has to be joined together. This is done by the join operator $\sqcup$ defined for the domain $\mathcal{PSG}$.

Based on the information gathered from this static alias analysis, the value-based diagnosis model is extended. Initially a new connection is generated for every variable partition. These connections are used to propagate the respective shape graph partitions through the model. For every statement it is known, which partitions it affects, therefore the component representing the statement is linked with the according connections. For the resulting partitions of the statement, new connections are created and also connected with the component. Furthermore a new meta component must be provided for modeling, that implements the join operation of shape graphs. This auxiliary component, that shall not be suspected for diagnosis, is necessary for join points of execution paths in the program, where different partitionings are summarized to a single new one.

Example 2. *In Figure 2 the new connections created in the model for our stack example program are depicted. Note that in this example only a single loop unrolling is shown. Further instantiations of the loop body are indicated by the dashed lines. The number of loop unrollings is determined by a preliminary simulation of the program. As can easily be seen, the two partitions are affected by different components. Only the loop condition, represented by conditionals, affects both partitions.*

4.2 Value Propagation, Conflicts, and Diagnoses

After building the extended value-based model as described in the previous section, we can apply diagnosis computation. With value-based models diagnosis is performed via propagation of runtime values. For our extended model, in addition to the runtime values, shape graphs are propagated. The shape graphs propagated through the model are modified within the components according to the Java semantics of the statement or expression represented by the components. The operational semantics of the components representing statements $s \in \mathcal{ST}$ is described by a function $[\![]\!] : \mathcal{ST} \times \mathcal{PSG} \rightarrow \mathcal{PSG}$. Again only those DSG's are included in the semantics definition, that are affected by the components.

$[\![\texttt{x = null}]\!](\langle E_{v_X}, E_{s_X}\rangle) \overset{\text{def}}{=} (\langle E_{v_{X-\{x\}}}, E_{s_{X-\{x\}}}\rangle, \langle E_{v_{\{x\}}}, E_{s_{\{x\}}}\rangle)$ where $\begin{aligned} E_{v_{X-\{x\}}} &= E_{v_X} - \{[x,*]\} \\ E_{s_{X-\{x\}}} &= E_{s_X} \\ E_{v_{\{x\}}} &= \emptyset \\ E_{s_{\{x\}}} &= \emptyset \end{aligned}$
$[\![\texttt{x.sel = null}]\!](\langle E_{v_X}, E_{s_X}\rangle) \overset{\text{def}}{=} (\langle E'_{v_X}, E'_{s_X}\rangle)$ where $\begin{aligned} E'_{v_X} &= E_{v_X} \\ E'_{s_X} &= E_{s_X} - \{\langle E_{v_X}(x), sel, *\rangle\} \end{aligned}$
$[\![\texttt{x = new T}]\!](\langle E_{v_X}, E_{s_X}\rangle) \overset{\text{def}}{=} (\langle E_{v_{X-\{x\}}}, E_{s_{X-\{x\}}}\rangle, \langle E_{v_{\{x\}}}, E_{s_{\{x\}}}\rangle)$ where $\begin{aligned} E_{v_{X-\{x\}}} &= E_{v_X} - \{[x,*]\} \\ E_{s_{X-\{x\}}} &= E_{s_X} \\ E_{v_{\{x\}}} &= \{[x, n_{new\langle T\rangle}]\} \\ E_{s_{\{x\}}} &= \emptyset \end{aligned}$
$[\![\texttt{x = y}]\!](\langle E_{v_X}, E_{s_X}\rangle, \langle E_{v_Y}, E_{s_Y}\rangle) \overset{\text{def}}{=} (\langle E_{v_{X-\{x\}}}, E_{s_{X-\{x\}}}\rangle, \langle E_{v_{Y\cup\{x\}}}, E_{s_{Y\cup\{x\}}}\rangle)$ where $\begin{aligned} E_{v_{X-\{x\}}} &= E_{v_X} - \{[x,*]\} \\ E_{s_{X-\{x\}}} &= E_{s_X} \\ E_{v_{Y\cup\{x\}}} &= E_{v_Y} \cup \{[x, E_{v_Y}(y)]\} \\ E_{s_{Y\cup\{x\}}} &= E_{s_Y} \end{aligned}$
$[\![\texttt{x.sel = y}]\!](\langle E_{v_X}, E_{s_X}\rangle, \langle E_{v_Y}, E_{s_Y}\rangle) \overset{\text{def}}{=} (\langle E_{v_{X\cup Y}}, E_{s_{X\cup Y}}\rangle)$ where $\begin{aligned} E_{v_{X\cup Y}} &= E_{v_X} \cup E_{v_Y} \\ E_{s_{X\cup Y}} &= \begin{array}{l} E_{s_X} \cup E_{s_Y} - \{\langle E_{v_X}(x), sel, *\rangle\} \cup \\ \{\langle E_{v_X}(x), sel, E_{v_Y}(y)\rangle\} \end{array} \end{aligned}$
$[\![\texttt{x = y.sel}]\!](\langle E_{v_X}, E_{s_X}\rangle, \langle E_{v_Y}, E_{s_Y}\rangle) \overset{\text{def}}{=} (\langle E_{v_{X-\{x\}}}, E_{s_{X-\{x\}}}\rangle, \langle E_{v_{Y\cup\{x\}}}, E_{s_{Y\cup\{x\}}}\rangle)$ where $\begin{aligned} E_{v_{X-\{x\}}} &= E_{v_X} - \{[x,*]\} \\ E_{s_{X-\{x\}}} &= E_{s_X} \\ E_{v_{Y\cup\{x\}}} &= E_{v_Y} \cup \{[x, E_{s_Y}(E_{v_Y}(y), sel)]\} \\ E_{s_{Y\cup\{x\}}} &= E_{s_Y} \end{aligned}$
$[\![\texttt{return s}]\!](\langle E_{v_S}, E_{s_S}\rangle) \overset{\text{def}}{=} (\langle E_{v_{S\cup\{return\}}}, E_{s_{S\cup\{return\}}}\rangle)$ where $\begin{aligned} E_{v_{S\cup\{return\}}} &= E_{v_S} \cup \{[return, E_{v_S}(s)]\} \\ E_{s_{S\cup\{return\}}} &= E_{s_S} \end{aligned}$

Table 1. Operational semantics for assignment statements

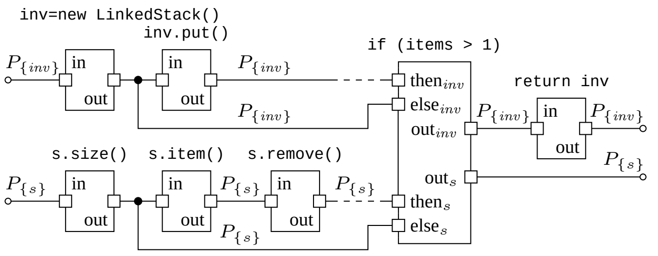


Figure 2. Model extended with shape graph connections

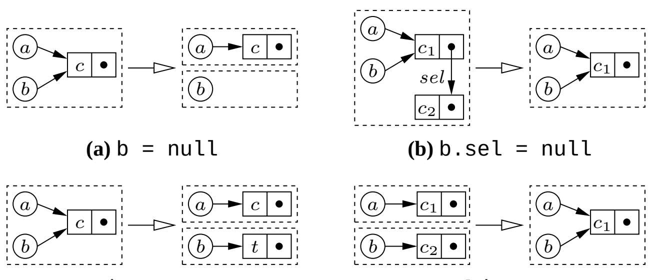
Table 1 outlines the semantics for assignment statements of reference type based on graph manipulation. Note that for the assignments in row 4, 5, and 6 it is assumed, that variables `x` and `y` are in separate partitions before the statement. If this is not the case (i.e. they are in the same partition) the semantics has to be changed slightly such that the sole shape graph for the two variables is changed accordingly. The semantics definition mainly consist of adding and removing edges to the shape graphs and keeping track of the partitioning.

The semantics of branching statements and method calls is defined by their nested statement blocks. The first line in Table 2 shows that the semantics of a block of statements is defined by applying the semantics of each statement successively. For conditional statements and loops it is necessary to deterministically decide which branch has to be selected for value propagation. Thus, it is necessary to evaluate the conditional expressions. Fortunately this information is provided by the present value-based model. Note that in Table 2 the condition of branching statements has to be considered in the semantics too, since it may contain side effects.

For method calls again a mapping for variable names is needed. This time the mapping function $m : \mathcal{PSG} \mapsto \mathcal{PSG}$ replaces the variable names within each DSG of the according PSG. Obviously not every variable defined at the callers site is visible in the called method. Thus, these variables have to be removed from the DSG's before the call and are reinserted afterwards.

The semantics definition for shape graphs is given in forward direction only. For most of the statements (e.g. assignments) no deterministic backward semantics can be specified. Therefore, we will omit it in the further discussion.

Example 3. *Figure 3 depicts the assignment of* `null` *to a simple variable and a member variable. Note that in the first case (Figure 3a) the partition of variable* `b` *is split into two new partitions, whereas in the second case (Figure 3b) they are not split. In addition Figures 3c and 3d show shape graph transformations for non-null assignment statements. The other types of statements not shown follow the same patterns.*



(a) `b = null` **(b)** `b.sel = null` **(c)** `b = new T` **(d)** `b = a`

Figure 3. Shape-graph transformations for various assignments

The diagnosis algorithm is based on conflict detection. Therefore, the input and output values provided by a test case are propagated into the value-based model. A conflict is raised if two different values are assigned to the same connection. For example the propagated output value for a variable does not match the value assigned to the connection from the test case. In case of propagation of shape graphs it is necessary to compare two shape graphs with each other. The inconsistencies between two shape graphs may be missing variable edges, missing selector edges, edges pointing to different nodes, and different node (sub-) types. In order to detect all these kinds of inconsistencies a simultaneous depth-first search on both shape graphs is performed. During this search the type of nodes in the two graphs are compared in order to find type conflicts. If the types are equal, the nodes are both labeled with the same fresh sequence number. The search on a path is stopped, if no further nodes can be reached via selectors. Furthermore the search on a path is stopped, if an already visited node is encountered. If the sequence numbers of the currently visited nodes do not match or if only one node wasn't visited yet, then a conflict is generated. In addition, a conflict is generated if a selector of a node in one graph points to a node, but not in the other graph. Two shape graphs are equal if all nodes reachable from all variables in both graphs are labeled equally.

Example 4. *Suppose a test case for our running example that specifies a stack with two elements for variable* `s`*. The expected outputs then are a stack with two elements for variable* `inv` *and an empty stack for variable* `s`*. Due to the propagation of shape graphs from the input connections to the outputs we will detect two different conflicts, that will be used by the diagnosis engine. Finally we will get the expected diagnoses [*`items = s.size()`*:2], [*`items > 1`*:3], and [*`items--`*:6].*

$[\![\texttt{s1...sn}]\!](PSG) \overset{\text{def}}{=} ([\![\texttt{s2...sn}]\!] \circ [\![\texttt{s1}]\!])PSG$
$[\![\texttt{if (c) \{s1\} else \{s2\}}]\!](PSG) \overset{\text{def}}{=} PSG'$ where $PSG' = \begin{cases} [\![\texttt{c; s1}]\!](PSG) & \text{if c is true} \\ [\![\texttt{c; s2}]\!](PSG) & \text{if c is false} \end{cases}$
$[\![\texttt{while (c) \{s\}}]\!](PSG) \overset{\text{def}}{=} PSG'$ where $PSG' = \begin{cases} ([\![\texttt{while (c) \{s\}}]\!] \circ [\![\texttt{c; s}]\!])PSG & \text{if c is true} \\ [\![\texttt{c}]\!](PSG) & \text{if c is false} \end{cases}$
$[\![\texttt{s.method(p)}]\!](PSG) \overset{\text{def}}{=} PSG'$ where $PSG' = (m^{-1} \circ [\![\texttt{body}_{method}]\!] \circ m)PSG$

Table 2. Operational semantics for branching statements

In order to verify the improvements of our approach we created a prototypical implementation of the extended value-based model. We were able to improve diagnosis results for various Java programs that operate on dynamic data structures, but we also encountered that there are still some deficiencies in our approach. The main problem is that the computation of the heap structures depends on the values computed in the model (e.g. conditionals, target objects for method calls, etc.). Since the value-based model summarizes all objects of the same type within a single connection, every component that is put in abnormal mode by the diagnosis engine prevents propagation of object references. Due these abnormality assumptions it's possible that obvious contradictions of heap structures can't be derived.

5 RELATED WORK AND CONCLUSION

A similar approach for alias information in value-based models for debugging is presented in [3]. Instead of adding new dependencies (i.e. new connections) in terms of heap partitions, the existing connections representing objects are split up. Hence, a connection for object identifiers now represents only those objects, that may be aliased. In addition connections that represent member variables of primitive types (i.e. integers, etc.) are also split into multiple connections. For certain kinds of programs this approach provides better diagnosis accuracy, but it also implies several restrictions. Programs that result in structural errors in the model can't be debugged correctly anymore. That is, the real error in the program may not be among the set of single fault diagnoses, instead those errors only appear within multi-fault diagnoses. On one hand this is the case for assignment statements with a wrong target or source variable. But such cases can be handled with our approach, because it only affects diagnosis results, if heap structure information is provided within the observations. And in this case erroneous assignments affect the partitioning of the heap, and therefore are suspected as faults. On the other hand, structural errors caused by calling methods on the wrong objects reveal a weakness of our approach in the sense that single fault diagnoses may be missed similarly as in [3]. This weakness arises because method calls don't change the partitioning necessarily. Thus, a method call on an object in a different partition is not suspected for errors in other partitions. In order to prevent these missing diagnoses, further research is necessary that allows us to rule out this weakness.

The work presented in this paper is an extension to existing value-based models for diagnosing Java programs. Its main advantage is that it is able to improve diagnosis results without additional observations needed. The additional analysis step that is needed in order to extend the value-based model requires extra time, but this is negligible compared to overall diagnosis time. Furthermore, due to the simple analysis used for the first evaluation of our approach, the benefit for diagnosis is limited to a special class of programs. The programs must operate on at least two separate data structures, and errors in the shape of data structures must be observed.

Although the advantages of its application are limited to a special class of problems, the improvements are remarkable and worth the extra effort. In addition we believe that based on this work it is possible to further improve diagnosis of programs that make use of dynamic data structures. The shape analysis we used in this work only focuses on the local behavior of statements. A global analysis of shape graph transformation for statement blocks is believed to further improve the results and possibly also widen its applicability.

ACKNOWLEDGEMENTS

This work was partially supported by the Austrian Science Fund (FWF) project P15265-INF and the European Coordinating Committee for Artificial Intelligence (ECCAI).

REFERENCES

[1] M. Hind, M. Burke, P. Carini, and J.-D. Choi, 'Interprocedural Pointer Alias Analysis', *ACM TOPLAS*, **21**(4), 848–894, (July 1999).

[2] C. Mateis, M. Stumptner, and F. Wotawa, 'A Value-Based Diagnosis Model for Java Programs', in *Proceedings of the 11th International Workshop on Principles of Diagnosis*, Morelia, Mexico, (June 2000).

[3] W. Mayer, 'Evaluation of Value-Based Models for Java Debugging', Technical report, Technische Universität Wien, Institut für Informationssysteme 184/2, (December 2001).

[4] W. Mayer, M. Stumptner, D. Wieland, and F. Wotawa, 'Can AI help to improve debugging substantially? Debugging experiences with value-based models', in *ECAI*, pp. 417–421, Lyon, France, (2002).

[5] W. Mayer, M. Stumptner, D. Wieland, and F. Wotawa, 'Towards an Integrated Debugging Environment', in *ECAI*, pp. 422–426, Lyon, France, (2002).

[6] M. Sagiv, T. Reps, and R. Wilhelm, 'Solving Shape-Analysis Problems in Languages with Destructive Updating', *ACM TOPLAS*, **20**(1), 1–50, (1998).

[7] M. Stumptner, D. Wieland, and F. Wotawa, 'Comparing Two Models for Software Debugging', in *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (KI)*, Vienna, Austria, (2001).