Debugging With an Enriched Dependency-based Model or How to Distinguish Between Aliasing and Value Assignment

Rong Chen[1,2] and Franz Wotawa[1]

[1] Technische Universität Graz, Institut for Software Technology, Inffeldgasse 16b/2, A-8010 Graz, Austria
{chen,wotawa}@ist.tu-graz.ac.at}

[2] Institute of Software Research, Zhongshan University, Guangzhou 510275, China

Abstract

This paper introduces a new model for debugging of Java programs. This model is based on previous functional dependency models that have been developed for the same purpose. In contrast the model makes not only use of dependency information but also of aliasing information. Therefore, the results are better for a large class of examples. The model is basically a qualitative model where values of variables are ignored. Hence, the approach can be seen as an application of qualitative reasoning for debugging software. In the paper we further discuss the compilation of Java programs to a constrained value-flow graph, and the mapping from the graph to its logical representation.

1 Introduction

Debugging is a procedure for detecting, locating, and repairing faults in a program throughout the whole software development process. The increasing complexity of software systems has led to an increasing demand for computer aided debugging. However, traditional debugging tools which have been in use for the last couple of decades mainly use testing techniques to detect incorrect behavior for a given program. They are either specific to a programming language, use specialized checking algorithms, or require explicit user-interaction to detect a bug. Thus traditional debugging tools are only partly able to serve the designated purpose of debugging. As a consequence, various approaches have been proposed to build automatic debuggers (see (Ducasse 1993) for an overview). A different line of research follows the concept of Model-based software debugging, which is the application of techniques taken from model-based diagnosis (MBD). Model-based diagnosis (Reiter 1987, de Kleer and Williams 1987) is a process for locating faulty components in a technical system solely on the basis of its structural and behavioral model. The model-based approach provides a general framework that allows for a static, formal, and automatic debugging of programs (see (Stumptner and Wotawa 1998) for an overview and references therein).

The basic idea behind model-based debugging is (1) to automatically build a logical model that describes the structure and the behavior of a program, and (2) to compute the diagnosis candidates by asking the model-based diagnosis engine why the program does not behave as expected. Each diagnosis candidate corresponds to a certain syntactic entity (e.g. a statement or expression) in the original source code. Diagnosis candidates can be further discriminated by using additional measurements, e.g., values of variables that are know at a specific position in the program's source code, or assertions. In the last years our colleagues have developed two categories of models for debugging Java programs: (1) The functional dependency model (FDM) (Mateis, Stumptner and Wotawa 2000a), and (2) the value-based model (VBM) (Mateis, Stumptner and Wotawa 2000b). Since the VBM can eliminate wrong diagnoses by using additional run-time information, e.g., the values of variables, the VBM achieves better results than the FDM (Stumptner, Wieland and Wotawa 2001) in most cases. However, whereas the VBM heavily relies on low-level information, the FDM can reason at a higher level of abstraction. Moreover, the dependency-based representation is expected to scale up well to large-sized programs, as is shown in (Stumptner and Wotawa 2000, Das 2000). In this paper, we focus on extending our previous research on model-based program debugging with the dependency-based representation.

Our main goal is the use of cheap one-level pointer analysis information to achieve better results in the context of dependency-based program debugging.

We can illustrate our approach using a small Java program shown in Figure1. The *demo* method consists of new instance creations, assignments and a conditional. If we know some information about the expected value of *a.value, b.value, c.value,* and *d.value,* then we can ask our debugging tool, JADE debugger to debug this input program. For example, we may know that the values of *b.value, c.value, d.value* are correct, but *a.value* is wrong. Diagnosing with the FDM of the input program, we get 6 diagnosis candidates for this fault. Among these 6 diagnoses, some are unreasonable ones like the conditional (line 5) and assignments (line 2 and line 6), and some are imprecise like assignments in line 1. The reason is that the functional dependency introduced in (Stumptner, Wieland and Wotawa 2001) cannot capture the behavior of a statement block precisely, for example, FDM cannot find that statements in line 1, 3, and 5 show a conditional alias relationship between *a* and *c*. Instead our approach can achieve this by using enriched dependencies with

constraints produced by a points-to analysis which thus lead to an improved result.

The proposed approach can be seen as a standard qualitative reasoning approach. In contrast to the VBM which requires the knowledge of the variables' values at specific lines in the code, the dependency-based models make only use of qualitative information like a value is correct or two variables are aliased. Hence, debugging using abstract models are good examples for the use of qualitative knowledge to solve real-world problems efficiently and as precise as required.

```
.    class Value {
.        int value;
         Value(int v) {
0.               value = v;          }
.        static void demo(boolean choice)
.          {
1.               Value a = new Value(1); /*l1*/
2.               Value b = new Value(2); /*l2*/
3.               Value c = a;
4.               Value d = b;
5.             if (choice)
5.1                  a = b;
.              else
5.2                  b = new Value(3); /*l3*/
6.             b.value = 4;
7.               c.value = 3;
```

Figure 1: A Small Java Program

2 Modeling Programs for Debugging

In this section, we describe how to model java programs by means of enriched dependencies. We further show the conversion of these dependencies into a logical representation, which can be directly used by a standard model-based diagnosis engine. As a result we obtain diagnoses that correspond to syntactical parts of the program's source code, i.e., statements and expressions.

2.1 Computing Enriched Dependency for Simple Assignment Statements

Our algorithm for computing enriched dependencies is basically an extension of Das's one level flow algorithm (Das 2000), which is a simple extension of Steensgaard's unification-based approach (Steensgaard 1996). In Das's algorithm, flow edges suffice to depict dependencies at expression level that only arise from variable assignments. However, in the setting of java programs some dependencies are possibly conflicting, method calls may give rise to implicit dependencies, and selection statements and loop statements produce inherent conditional dependencies. So we have to take these factors into account to extend Das's algorithm.

Basically our concern in our algorithm is to use an enriched points-to graph to record the semantic information implied by input programs. The nodes of this graph correspond to program variables and memory locations which they point to, and the edges fall into two categories: one represents the value flows between program variables, and the other represents accessing either by reference or selector. In the next section this graph will be augmented with some annotations deriving from expressions.

In order to have a better understanding of our algorithm, we like to illustrate the underlying ideas on simple examples. The main idea of our algorithm is the conversion of an input program into a value-flow graph. We assume that each variable regardless whether it is class, instance, or local has a corresponding node in the induced points-to graph. Since we intend to collect strong knowledge at the low-level location nodes, it is convenient to assume that each node *n* has an edge pointing to its location node *n.lc*. For example, a reference variable *x* induces a structure outlined in Figure 2 (a). The reason is that a variable may refer to an object, and the content of that object may point to other memory locations which can be accessed by field selector. For the sake of uniformity, we also regard objects as memory locations and the edges directing from nodes to their location nodes are uniformly called **access edges** (e.g. solid edges in Figure 2 (a)). We use also **flow edges** to depict basic dependencies that are produced by assignment statements. For example, consider a simple variable assignment, *x=y*. Firstly, reference variables *x* and *y* induce a structure that is outlined in Figure 2 (b). Secondly, we create a dashed edge (i.e. flow edge) because the operand = implies that a value flows from *y* to *x*. This is depicted in Figure 2 (c). Finally, the assignment statement induces the unification of contents of *x* and *y* which is shown in Figure 2 (d).

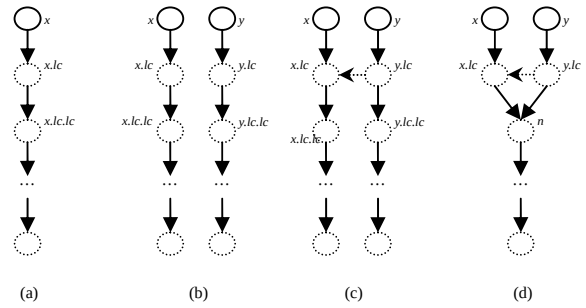

Figure 2: The graph induced by a variable assignment statement.

For the sake of clarity, each time a variable or a constant is used in an assignment statement, we create a **source node** (labeled by the variable or the constant, e.g., node *x* in Figure 2), a corresponding **location node** (represent the memory location, e.g., dashed circles in Figure 2). Some location nodes are called **alias nodes** since they represent the content that is referenced via different variables (e.g., node *n* in Figure 2 (d)). Because of the fact that we collect the knowledge from alias and location nodes with multiple ingoing flow edges we call them uniformly **summary nodes**. For example, the assignment $x \equiv y$ is represented at

the alias node *n* in Figure 2 (d), which means either *"x and y refer to the same object"* or simply *"x equals to y"*.

In the following we consider the more complicated case: how to convert statement *1* of method *demo* that is depicted in Figure 1 (i.e. Value a = **new** Value(1)), which is an assignment with a constructor on the right side. For this example we focus on the handling of the constructor. For this purpose we first convert the constructor of the statement. The result of this process is shown in Figure 3 (a). Because an object may influence several variables due to its state change we use named locations to represent a newly-created object; each location *l* has a type, i.e., the class type of the instance modeled by *l*, and a unique index, which non-ambiguously identifies location *l* within a modeled method. To record these object influences, we next link the named location to each instance/class variables of the object (shown in Figure 3 (b)). Afterwards we convert the rest of the assignment which is now simplified to be an assignment of a named location to a variable, i.e. *a=l1* (see Figure 3 (c) for the induced graph).

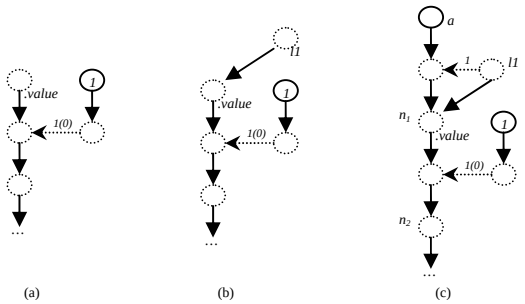

Figure 3: The graph for statement 1 (Value a = **new** Value(1)).

In fact, we can read out some information at alias nodes, for example, in Figure 3 (c) we can read *"a* is an alias of *l1"* and *"a.value = 1"* at node n_1 and n_2 respectively. Moreover we can observe from Figure 3 that:

1: Alias nodes are those with multiple points-to ingoing edges and one outgoing edge, while location node one ingoing edge and one outgoing edge.
2: A source node and a location node can both access their location nodes by either their reference or field selector (e.g. *a*.value).
3: A source node can become a location node. For example, node n_1 in Figure 3 (c) is not only a source node *a.value* but also a location node.

The main idea of our algorithm is the conversion of input statements into such an enriched points-to graph, which comprises of source nodes and summary nodes, as well as access edges and flow edges. Whereas flow edges at non-summary nodes represent the functional dependency between two program variables, edges at summary nodes represent more precise knowledge of multiple program variables. The goal of this representation is to characterize strong knowledge implied by a sequence of relevant statements. And it is computable to gain debugging knowledge by traversing the induced graph, and thus the strong knowledge is used to construct the qualitative model for methods.

2.2 Algorithm for Computing Enriched Dependencies

In the last section, we informally introduced an enriched points-to graph. In this section, we describe an algorithm that converts Java programs to extended points-to graphs. The extension is due to the labeling of edges with constraints. We call the extended graph a constrained value flow graph (CVFG). The constraints can be Boolean expressions that occur as conditional expressions in loops and conditionals as well as conditional assertions.

The CVFG for a given program is constructed by calling two functions. The function **convert** compiles one statement into a basic CVFG. The function **summarize** summarizes the alias information and the conflicting dependencies in the induced CVFG. The function **convert** calls summarize whenever an assignment statement is processed.

To simplify our discussion of these two functions, we introduce a number of functions and notations that are used throughout the discussion. For this purpose we use the following notational conventions:

Id denotes either a variable name or a method identity.

Id.f denotes the field selector *f* of *Id.*

n denotes a node in the induced graph.

nId denotes the corresponding location node for *Id.*

rNode denotes the single returned node.

rNodes denote the list of returned nodes.

lc denotes the abstract reference. We use *n.lc* to follow access edges for each node *n* in the induced graph.

MethodCall denotes the method call.

constraints(*i*) denotes the constraints associated with *nId* with respect to a statement index ***i*.**

sumNodes(*nId*) denotes a set of summary nodes for node *nId*, which are used to record the information to be summarized.

t:j.k(l) denotes the inner statement index, where *j* is the prime statement index (as shown in Figure 1); *k* the sub-index of statements in the block of selection statements; *l* the sub-index of statements in method calls or constructors; *t* is used to label the type of the modeling statement: *t =1*, if it is a loop statement, otherwise *t = 0* for modeling method declarations. It is certain the information in our indexing mechanism corresponds to the statement types; sometimes we will neglect the unimportant part if the context is unambiguous.

Some auxiliary functions are defined to retrieve values of variables as well as nodes and their associated constraints:

match(*t:j.k(l), i*) returns the value of *t, j, k, l* for a statement index *i*.

access(*Id.f*) goes down along the access edges of node *Id* and return a node corresponding its field *f*.

lookup(*Id*) returns the corresponding location node *nId* for *Id* if it exists.

lookup(*i*) returns the corresponding location node *nId* if statement *i* is in the form "*Id* = Expr".

lookup(*MethodCall*) determines the receiver(s) of the *MethodCall*, returns the corresponding the method identities *Id*(s) of the called method(s).

MethodDec(*Id*) returns the declaration "Type *Id*(FormalParList) {S_1, …, S_n}"of the method *Id*.

output(*Id*) returns nodes related to the returned values of a method call with respect to a method identity *Id*.

input(*Id*) returns a list of nodes for the actual parameters of a method call with respect to a method identity *Id*.

locations(*nId*) returns a list of location nodes of node *nId* if *Id* is a method identity.

copy(*nId*) copies the induced graph, which can be done by traversing from the starting node *nId*, renaming node *nId* as *nId*′ in the copied graph, returning the renamed node *nId*′.

visiable(*nId, i*) returns *true* if either (1) *Id* is an attribute of an object and statement *i* is one of the statements of constructors, or (2) statement *i* is one of the statements in the declaration of a method call and *Id* is an formal parameter; otherwise returns *false*.

The basic functionality of the conversion algorithm can be explained by explaining its behavior for different Java statements. We first describe the conversion of an assignment via the function **convert**.

```
convert(Id = Expr, i)
  tag := false ;
  if (not (nId = lookup(Id))) then
     create a named source node nId and its location
     node nId.lc;
     tag := true ;
  rNodes := convert(Expr, i);
  for each node n ∈ rNodes,
     create a flow edge directed from n to node nId.lc,
     label this edge with the index i and constraints(i);
     if ((tag) and constraints(i)≠ {}) then
        for each constraints con_j in constraints( i)
           create a new location node nId.lc′ for node nId;
           create an access edge directed from nId to
           nId.lc′, label with the index i and con_j;
           create an access edge directed from nId.lc′ to
           n.lc.
     else create an access edge directed from nId.lc to
     n.lc;
     sumNodes(nId) := sumNodes(n.lc);
     for each node n′ in sumNodes(nId)
        if (constraints(i)≠{} or i=0) then
           summarize(i, n′).
  if (visiable(nId, i)) then return nId.lc else return {}.
```

Algorithm 1: Converting an assignment statement.

If an assignment involves a field access (e.g. *Id.f* = Expr), we call **access**(*Id.f*) to return a node corresponding *Id*'s field *f*, afterwards the processing is similar to that of Algorithm 1.

And we convert an expression as follows:

```
convert(Expr, i)
  case Expr=Id:
       if (not lookup(Id)) then
            create a named source node nId, its location
            node nId.lc, and nId.lc 's location node
            nId.lc.lc;
            create access edges directed from each node
            to its location node.
       return nId.lc.
  case Expr=Const:
       create a source node nId , its location node nId.lc,
       and nId.lc 's location node nId.lc.lc;
       create access edges directed from each node to its
       location node;
       return nId.lc.
  case Expr=(Expr1):
       rNode := convert(Expr1, i).
  case Expr= Id.f:
       return access(Id.f).
  case Expr=Expr1 Op Expr2:
       rNodes := convert(Expr1, i) ∪ convert(Expr2,
       i);
       return rNodes.
  case Expr=new MethodCall:
       create a named location node l_i;
        if (not (Id = lookup(MethodCall))) then
            rNodes := convert(MethodDec(Id), i);
            for each n∈rNodes create an access directed
            from l_i to n.
       else
            nId := lookup(Id);
            l_i := copy(nId);
       return l_i.lc.
  case Expr=MethodCall:
       if (not (Id = lookup(MethodCall))) then
            rNodes := convert(MethodDec(Id), i);
       else
```

nId := **lookup**(*Id*);
n := **copy**(*nId*)**;**
rNodes := **locations**(*n*);
for each node $n_j \in$ list **input**(*Id*) and its corresponding node $n'_j \in$ list *rNodes*, create a flow edge directing from $n_j.lc$ to $n'_j.lc$.
return output(*nId*).

Algorithm 2: Converting an expression.

When modeling the selection statement and loop statement, we assume that we have pre-processed their corresponding expressions by introducing additional statements with auxiliary variables if expressions contain constructors like arithmetic operator or method calls. That is, we assume that expressions are simple conditions. So the selection statement (e.g. an if-then-else statement) and the loop statement (e.g. a while-statement) are converted by Algorithm 3 and Algorithm 4 as follows:

convert(if (Expr) {S_1, …, S_n} else {S_1', …, S_n'}, *i*)
constraints(*i.1*) := **constraints**(*i*)∪{Expr};
rNodes := **convert**({S_1, …, S_n}, *i.1*);
constraints(*i.2*) := **constraints**(*i*)∪{¬Expr};
rNodes := **convert**({S_1', …, S_n'}, *i.2*).

Algorithm 3: Converting a selection statement.

convert(while (Expr) {S_1, …, S_n}, *i*)
constraints(*1:i*) := **constraints**(*i*)∪{ Expr};
rNodes := **convert**({S_1, …, S_n}, *1:i*);

Algorithm 4: Converting a loop statement.

Since we convert a statement sequence statement by statement, we can in the same way convert the body of a method declaration and record the conversion of this method by creating an identity node as well as some access edges (as shown in Algorithm 5).

convert(**MethodDec**(*Id*), *i*)
create node *nId* for the method identity *Id*.
rNodes := **convert**({S_1, …, S_n}, *0:i*);
for each *n*∈*rNodes* create an access directed from *nId* to *n*.
return *rNodes*.

Algorithm 5: Converting a method declaration.

Since method calls might modify external variables, thus we have to consider side effects which stem from statements in the method to be modeled. We achieve this in by focusing on the nodes that can be reached through input formal parameters and the returned variables (as shown in Algorithm 2). And we prune any unreachable nodes in the induced graph.

Algorithms 1~5 formally define the compilation process of Java programs into CVFGs. These algorithms might (implicitly) call the function **summarize** (e.g. in the conversion of assignment statements). The basic idea behind **summarize** is to collect knowledge at summary nodes that is represented by consistent constraints. The collected knowledge, represented by equalities, is strong because it represents several dependencies coming from the nodes that are one level higher than the summary node. We can always create such summary nodes by creating the node and a flow edge directing to itself. For this purpose we have to be sure that there are no constraints annotated and that there are no conflicts in related flow edges (as shown at the summary nodes in Figure 2). Whenever two flow edges are directed to the same summary node, we call them *conflicting*. To resolve any conflicts, we compare related statement indexes at a summary node and select the one that has been executed at the latest. Regarding the summary nodes of a while statement, we can also obtain strong knowledge by using some sort of fix-point computation, i.e., calling **summarize** until no new knowledge can be obtained.

Using the above mentioned **convert** and **summarize**, we can convert Java programs into a constrained value-flow graph. As an example, Figure 4 shows the final AVFG of our example in Figure 1. Note that this AVFG contains all dependencies that can be computed in the *demo* method.

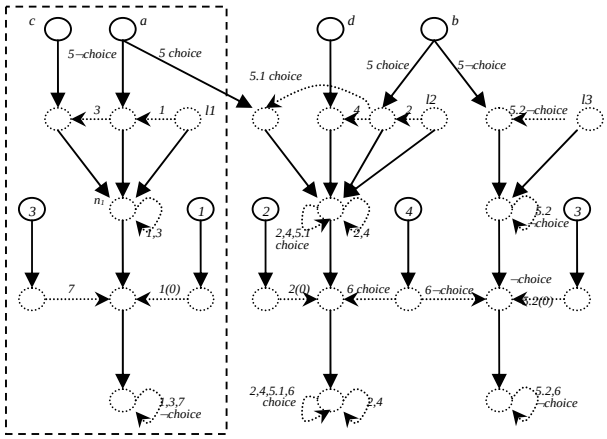

Figure 3: The induced graph for the small Java program in Figure 1.

2.3 Constrained Value-Flow Model

After computing all dependencies, i.e., the constrained value-flow graph, we map them to a logical representation, which can be directly used by a standard model-based diagnosis engine. For this purpose, we have to introduce a predicate $\neg Ab(i)$ to denote that a statement *i* is assumed to be correct where *i* is the index of this statement. The mapping of dependencies to their logical model was described elsewhere (Mateis, Stumptner and Wotawa 2000a). In the context of CVFGs this mapping is similar to the mapping of dependencies at non-summary nodes to logical sentences. From the standard functional dependency model we receive the following (simplified) rules for variables *a* and *c*:

$$\neg Ab(1) \rightarrow ok(a_1)$$
$$\neg Ab(3) \wedge ok(a_1) \rightarrow ok(c_3)$$
$$\neg Ab(7) \wedge ok(c_3) \rightarrow ok(c_7) \wedge ok(c_7)$$

where v_i denotes the state of the variable v after executing the statement in line i. If we now assume that $ok(c_7)$ and $\neg ok(a_7)$ is valid, i.e., c is computed correctly by the program *demo* whereas a is not, then we can compute 3 single-fault diagnosis candidates: {*1*}, {*3*}, {*7*}, which is not a good result as explained previously.

In order to improve this result we made some slight changes to the model. We only consider the following rules:

$$\neg Ab(1) \rightarrow ok(a_1)$$
$$\neg Ab(7) \rightarrow ok(c_7)$$

and add some additional rules that are for handling the aliasing relationship between a and c. The first rule that can be easily extracted from the CVFG captures aliasing.

$$\neg Ab(3) \wedge \neg choice \rightarrow a \equiv c$$

We further add some rules that allow us to derive new information from the aliasing knowledge.

$$a \equiv c \wedge ok(a_i) \rightarrow ok(c_i)$$
$$a \equiv c \wedge ok(c_i) \rightarrow ok(a_i)$$

When using these rules and the above observations and the fact $\neg choice$, we now obtain the diagnosis {*3*} as the only single-fault diagnosis.

In general the process of generating the aliasing rules can be expressed as follows. For each summary node n where there exists a flow edge between parent nodes add the following rule to the system description:

$$\neg Ab(i) \wedge Cns(n) \Rightarrow v \equiv w$$

where i denotes the maximum value of the statement indices $IDX(n)$ that are associated with n, v and w denotes the variables that corresponds to the parent nodes, and $Cns(n)$ denotes the constraints that are associated with n.

This sentence basically means that v and w denotes the same memory location if the constraints are fulfilled.

For example, the summary node n_1 in Figure 4 (in the dashed rectangle) will lead to the rule:

$$\neg Ab(3) \wedge \neg choice \rightarrow a \equiv c$$

because $IDX(n_1) = \{1, 3\}$, $3 = max\{ IDX(n_1) \}$, and $Cns(n_1) = \{\neg choice\}$.

In (Mateis, Stumptner and Wotawa 2000a) the authors conclude that a model based purely on dependencies is too weak to discriminate between all possible program errors. As shown with our running example the new model really helps to improve the diagnosis results whenever aliasing occur within a program. A more detailed analysis of the model and a formal comparison with previous research is left for future research.

3 Related Work

Our algorithm for dependency analysis is an extension of Das's one level flow algorithm. Das's algorithm (Das 2000) is a simple extension of Steensgaard's unification-based approach (Steensgaard 1996), which is described as a set of type inference rules over a language of pointer related assignments. The focus of Das's approach is to provide a practical method for obtaining better points-to information on large program. Unfortunately, the Das's algorithm cannot fully meet our needs that are required for debugging Java programs. For example, in some cases faults in programs rely heavily on the order of statements, however flow-insensitive analysis assume that statements can be execute in any order (Das 2000), i.e., the control structures of the languages are irrelevant (Steensgaard 1996). Moreover, some flow-insensitive dependencies are possibly conflicting. Method calls may give rise to implicit dependencies, and selection statements and loop statements produce inherent conditional dependencies. All these facts have to be taken into account for debugging. Hence, our algorithm extends the Das's algorithm in this respect and presents a constrained value-flow graph to model Java programs in an appropriate way.

Several automatic debugging approaches have been proposed so far to help programmers solving the debugging task. Among them are program slicing(Weiser 1982, Weiser 1984), algorithmic debugging (Shapiro 1983), dependency-based techniques (Jackson 1995, Kuper 1989), and others (see (Ducasse 1993) for an overview). Another approach that makes use of a model checker to produce error trace in order to find a fault is described in (Ball, Naik and Rjajamani 2003). In this paper the authors discuss the localization of a fault in the source code of a program using error traces. Finally, there is a body of work on model-based debugging approaches, which makes use of model-based diagnosis for locating faults in software (Stumptner and Wotawa 1998). In (Wotawa 2002) F. Wotawa shows that model-based debugging in the context of functional dependencies provides at least the same capabilities than program slicing for locating bugs. Moreover, in the same paper the author proves that the model-based debugging approach can provide better results.

The functional dependency model (FDM) (Stumptner and Wotawa 2000a, Wieland 2001) and value-based model (VBM) (Mateis, Stumptner and Wotawa 2000b) have successfully been applied to debug Java programs in the Jade project. A comparison of the models and their effectiveness was given in (Stumptner, Wieland and Wotawa 2001). The empirical results show that the VBM achievers better results than the FDM in general. The main reason is the VBM can eliminate wrong diagnoses by using additional run-time information like the values of variables (Stumptner, Wieland and Wotawa 2001).

4 Conclusion

In this paper we introduce the use of a cheap one-level pointer analysis for debugging in the context of model-based diagnosis. This approach achieves better results than previous model-based approaches that only make use of dependencies. Our model is a purely qualitative model that captures not only the dependencies between variables but also the aliasing relationship between the variables. Thus it provides more information that can be successfully used to reduce the number of computed diagnoses during debugging. The necessary aliasing information can be obtained by traversing the CVFG which is a graph that represent the flow of values within a program.

Also we introduce an algorithm that compiles Java programs into a CVFG. Moreover, we show how the CVFG can be used in order to obtain a logical model that can be directly used by a standard model-based diagnosis engine. Although, we have currently almost no empirical results that would allow to draw a statement regarding the usefulness of the approach in a real-world setting, we have analyzed the model using examples where other functional dependency model does not provide good results.

In summary, the contributions of this paper include:

- The use of one-level pointer analysis for debugging.
- An extension of basic value-flow representations that allow extracting useful debugging information.
- Handling of aliasing information in the context of debugging which leads to better debugging results in several cases.

Future research should include an empirical analysis of the proposed model and an alternative representation of the underlying ideas using abstract program semantics. Moreover, the development and handling of alternative models for debugging should be in the focus of attention.

Acknowledgment

The work presented in this paper was funded by the Austrian Science Fund (FWF) Project P15265-N04, and partially supported by the National Natural Science Foundation of China (NSFC) Project 60203015 and the Guangdong Natural Science Foundation Project 011162. And the authors would like to thank Roderich Bloem for his comments.

References

Ball T.; Naik M.; and Rajamani S. K. 2003. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proceedings of the Seventh ACM Symposium on Principles of Programming Language (POPL).* New Orleans, USA.

Das M. 2000. Unification-based Pointer Analysis with Directional Assignments, In *PLDI 2000*, Vancouver, Canada.

de Kleer, J.; Williams, B. C. 1987. Diagnosing multiple faults. *Artifical Intelligence* 32(1): 97–130.

Ducassé, M. 1993. A pragmatic survey of automatic debugging. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging*, AADEBUG '93, 1–15, Springer LNCS 749.

Jackson D. 1995. Aspect: Detecting Bugs with Abstract Dependences. *ACM Transactions on Software Engineering and Methodology* 4(2):109–145.

Kuper R. I. 1989. Dependency-directed localization of software bugs. Technical Report AI-TR 1053, MIT AI Lab.

Mateis C.; Stumptner M.; and Wotawa F. 2000a. Modeling Java Programs for Diagnosis, *In the 14th European Conference on Artificial Intelligence*, Berlin, Germany.

Mateis C.; Stumptner M.; and Wotawa F. 2000b. A Value-Based Diagnosis Model for Java Programs, In *the Eleventh International Workshop on Principles of Diagnosis (DX)*, Morelia, Mexico.

Reiter, R. 1987. A theory of diagnosis from first principle. *Artifical Intelligence* 32(1): 57–95.

Shapiro E. 1983. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts.

Steensgaard B. 1996. Points-to analysis in almost linear time. In *Conference Record of the Twenty-Third ACM Symposium on Principles of Programming Languages.*

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

Stumptner M.; Wotawa F. 1998. A Survey of Intelligent Debugging, *The European Journal on Artificial Intelligence (AICOM)* , 11(1).

Stumptner M.; and Wotawa F. 2000. Using Model-Based Reasoning for Locating Faults in VHDL Designs. *Künstliche Intelligenz*, 14(4):62–67.

Weiser M. 1982. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452.

Weiser M. 1984. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357.

Wieland D. 2001. Model-Based Debugging of Java Programs Using Dependencies. PhD thesis, Vienna University of Technology.

Wotawa 2002. On the Relationship between Model-Based Debugging and Program Slicing. *Artificial Intelligence* 135(1–2):124–143.