Using abstract dependencies in debugging*

Franz Wotawa and **Safeeullah Soomro**†
Graz University of Technology
Institute for Software Technology,
8010 Graz, Inffeldgasse 16b/2, Austria,
{wotawa,ssoomro}@ist.tugraz.at

Abstract

Fault localization is the next step after detecting faults in programs. Testing and formal verification techniques like model-checking are usually used for detecting faults but fail to locate the root-cause for the detected faulty behavior. This article makes use of abstract dependencies between program variables for localizing faults in programs. It explains the basic ideas, the underlying theory and the limitations. The fault localization model is based on a previous work that uses abstract dependencies for fault detection. Moreover, the paper discusses the relationship between the abstract model and qualitative reasoning.

1 Introduction

Debugging, i.e., removing faults from programs, comprises three parts. Fault detection is used to find a misbehavior. Within fault localization the root-cause for the detected misbehavior is searched for. And finally, during repair the responsible parts of the program are replaced by others in order to get rid of the detected misbehavior. In this paper we focus on fault localization which is based on abstract dependencies that are used by the Aspect system [Jackson, 1995] for detecting faults. Abstract dependencies are relations between variables of a program. We say that a variable `x` depends on a variable `y` iff a new value for `y` may causes a new value for `x`. For example, the assignment statement `x = y + 1;` implies such a dependency relation. Every time we change the value of `y` the value of `x` is changed after executing the statement. Another example which leads to the same dependency is the following program fragment:

```
if ( y < 10) then
    x = 1;
else
    x = 0;
```

In this fragment not all changes applied to `y` cause a change on the value of `x`, although `x` definitely depends on `y`. The Aspect system now takes a program, computes the dependencies and compares them with the specified dependencies. If there is a mismatch the system detects a bug and notifies the user. However, the Aspect systems does not pinpoint the root-cause of the detected misbehavior to the user.

```
    int myMult (int x,y) {
1.    int result = 0;
2.    int i = 0;
3.    while ( i < x ) {
4.      result = result + x ;
          // Should be result = result + y
5.      i = i + 1; }
6.    return result;
    }
```

Figure 1: `myMult` - a faulty implementation of the integer multiplication

We illustrate the basic ideas of fault localization using the faulty implementation of a multiplication operation `myMult` from Figure 1. The bug lies in statement 4 where the variable `x` is used in the right hand side expression of the assignment instead of variable `y`. In order to detect the fault we first have to specify the abstract dependencies for the multiplication where the result should depend on both inputs. Hence, we specify that `result` depends on `x` and `y` which can be written as a rule: `result` $\leftarrow$ `x, y` or as binary relation $\{(\texttt{result},\texttt{x}),(\texttt{result},\texttt{y})\}$.

When approximating the abstract dependencies from the source code of `myMult` using the Aspect system, we finally obtain a dependency relation $\{(\texttt{result},\texttt{x})\}$ which fails to be equivalent to the specified dependency relation. The question now is how the root-cause of this misbehavior can be found. The idea behind our approach is the following. During the computation of abstract dependencies every statement has an impact to the overall dependency set. For example statement 4 says that `result` depends on `result` and `x`. When knowing the dependencies of `result` before statement 4, we can extend the relation. For `myMult` the variable `result` also depends on `i` (statement 3) and a constant 0 (statement 1). The variable `i` itself depends on `i` (statement 5), `x` (statement 3) and a constant 0 (statement 2). Hence, in this case all statements fail to deliver a relation $(\texttt{result},\texttt{y})$ and are

*The work described in this paper has been supported by the Austrian Science Fund (FWF) project P15265-INF and the Higher Education Commission(HEC), Pakistan.

†Authors are listed in reverse alphabetical order

therefore candidates for a root-cause. Let us now extend our example by introducing an additional specified dependency `i` $\leftarrow$ `i`, `x` which is said to be valid for statements 3 to 6. In this case statements 2, 3, and 5 can no longer be candidates for the root-cause because they are necessary to compute dependencies for variable `i` which fulfill the specification. Hence, only 1 and 4 remain as potential root-causes.

All arguments for extracting root-causes have been done using only dependencies which are computed by analyzing statements. Hence, a adapted formalization of this process which allows for reasoning about statements and their influences on the computed abstract dependencies should lead to a system which extracts root-causes automatically from the source code of programs and the specified dependencies. During the rest of this paper we provide a framework for this purpose which is based on model-based diagnosis [Reiter, 1987]. Model-based diagnosis provides the means for reasoning about statements and their influences which is necessary for our purpose.

2 Qualitative reasoning in debugging

Although, qualitative reasoning is usually understood as qualitative reasoning of physical systems [Weld and de Kleer, 1989] the basic ideas can be also applied to other domains like software debugging. The underlying idea of qualitative reasoning is to represent the behavior of a system not in terms of quantities but in terms of qualitative values and their relationships. The motivation came from observations of cognitive aspects of reasoning about physical systems, i.e., humans for example require no quantitative model for deriving very meaningful results about the behavior even of complex systems. A similar argument has been used in a different domain, i.e., software analysis. Mark Weiser argues in his paper [Weiser, 1982] that programmers them-self use an abstraction of the program during debugging. Hence, the application of qualitative reasoning if not restricted to some techniques like QSIM [Kuipers, 1986] or QPT [Forbus, 1984] can be used in debugging.

Before recapitulating some related papers in debugging using abstractions of the program's behavior we briefly describe the relationship between the original execution and its abstract variant for a small example program. Consider Figure 2 where the execution of a two line program is depicted. During execution the state of the program is changed. The state in this case is represented by the values assigned to a variable. The program execution starts with a state where variable `y` has the value 1. The execution of statement 1, leads to a new state where in addition `x` has the value 2. This process continues until the end of the program is reached. The states are also called program environments.

The abstract version of the execution of the same program is given in Figure 3. Instead of changing the program environment the computed dependency relations are changed during program execution. Hence, the qualitative version of program execution only changes the semantics function. Instead of having a semantics function of the form $exec : STMNT \times ENV \mapsto ENV$ (where $STMNT$ is the set of possible statements and ENV a program environment),

$\{y = 1\}$
1. `x = y + 1;`
$\{y = 1, x = 2\}$
2. `z = y - 1;`
$\{y = 1, x = 2, z = 0\}$
3. ...

Figure 2: State changes during program execution

$\{\}$
1. `x = y + 1;`
$\{(x, y)\}$
2. `z = y - 1;`
$\{(x, y), (z, y)\}$
3. ...

Figure 3: Dependencies changes during program execution

we now have $exec_A : STMNT \times DEP \mapsto DEP$ (where DEP is the set of possible dependencies). It is clear that because of abstraction we are losing some information. For example, with $exec_A$ it is no longer possible to decide which branch of a conditional to execute. Hence, the obtained result is only an approximation of the concrete execution but represents different execution traces at the same time. Moreover, as an advantage, execution time in this case does not depend on the underlying problem complexity of the implementation.

Previous work that deals with abstraction of the program's behavior include [Cousot and Cousot, 1977] and [Mayer and Stumptner, 2004] where the former introduced the basic concepts and principles and the latter applied it for debugging loops in Java programs. Other work which follows the idea of using dependencies include [Friedrich *et al.*, 1999; Wieland, 2001]. In contrast we use differences between dependencies as starting point for debugging whereas the others make use of observations regarding the correctness or incorrectness of computed values (wrt. the specification).

3 Modeling

In this section we introduce a model which is based on dependencies between variables. For this purpose we first recapitulate Jackson's definitions [Jackson, 1995]. Afterwards, we briefly introduce the basic definitions of model-based diagnosis [Reiter, 1987] and finally introduce the new model.

3.1 Dependencies

Jackson [Jackson, 1995] defines dependencies over state changes which come from program execution, i.e., a variable `x` depends on `y` if different pre-states that are distinguishable only in their `y` component lead to post-states having different `x` components. In other words `x` depends on `y` if the computation of `x` makes use of the value of `y`. For example, the statement `x = y + z` impose a dependency relation $\{(\mathtt{x}, \mathtt{y}), (\mathtt{x}, \mathtt{z})\}$. Using similar arguments as for slicing [Weiser, 1982; 1984] the computation of dependency information from the source code can only be approximated. This approximation may lead to additional entries of the resulting dependencies. Before we discuss an example to illustrate

this observation we introduce rules for computing dependencies for alias-free programs. We make use of two functions. The function D maps dependencies of the every statement. Function M maps statements to a set of variables which are defined in the statements.

The model for the assignment statement is defined by the following definitions:

$$D(x \texttt{ = } e) = \{(\texttt{x}, v) | v \in vars(e)\}$$
$$M(x \texttt{ = } e) = \{\texttt{x}\}$$

In the above rule the function $vars$ is assumed to return all variables which are used in the parameter expression e.

For conditional statements the dependencies not only come from the dependencies of the branches but also from the used conditional expression. This expression influences which branch is to be executed. Since, it is not possible to determine which branch to go at compile-time we have to consider all possible branches.

$$D(\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2) =$$
$$= D(S_1) \cup D(S_2) \cup ((M(S_1) \cup M(S_2)) \times vars(e))$$
$$M(\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2) = M(S_1) \cup M(S_2)$$

The loop statement can be represented as an infinite nested conditional statement, i.e., `while` e { S } = `if` e `then while` e { S } `else` NOP where NOP represents the empty statement. Hence, its model can be expressed by:

$$D(\texttt{while } e \texttt{ \{ } S \texttt{ \} }) = D(S; \texttt{while } e\{ \ S\}) \cup$$
$$D(NOP) \cup$$
$$((M(S; \texttt{while } e\{ \ S\}) \cup M(NOP)) \times vars(e))$$
$$M(\texttt{while } e \texttt{ \{ } S \texttt{ \} }) = M(S)$$

Considering $M(NOP) = \{\}$ and $D(NOP) = I$ with $I = \{(x, x) | x \in VARS\}$ we obtain:

$$D(\texttt{while } e \texttt{ \{ } S \texttt{ \} }) = D(S; \texttt{while } e\{ \ S\}) \cup$$
$$I \cup (M(S; \texttt{while } e\{ \ S\}) \times vars(e))$$

Statement sequences can be modeled by composing their dependencies

$$D(S_1; S_2) = D(S_1) \circ D(S_2)$$
$$M(S_1; S_2) = M(S_1) \cup M(S_2)$$

where $\circ$ stands for the following operator which is not equivalent to the original composition operator on relations.

$$R_1 \circ R_2 = \begin{array}{l} \{(x,y) | (x,z) \in R_2 \wedge (z,y) \in R_1\} \cup \\ \{(x,y) | (x,y) \in R_1 \wedge \nexists (x,z) \in R_2\} \cup \\ \{(x,y) | (x,y) \in R_2 \wedge \nexists (y,z) \in R_1\} \end{array}$$

Hence, the $\circ$ operator ensures that no dependencies which come from previous statements are ignored. Only dependencies for variables that are defined in the last statement are changed.

Using the conversion rules for statement sequences we further can rewrite the rules for loop statements and finally obtain:

$$D(\texttt{while } e \texttt{ \{ } S \texttt{ \} }) =$$
$$= D(S)^* \cup (M(S) \times vars(e)) \circ D(S)^*$$

In the above rule $D(S)^*$ is the reflexive and transitive closure of $D(S)$. Note that the rule represents the least solution of the rule after applying the sequence rule.

The following example not only shows how dependencies are computed but serves as one example for additional dependency relations. Consider the following program fragment:

```
1.   x = y + r;
2.   x = x - r;
```

In this program `x` only depends on variable `y` and not on `r`. However, the computation leads to:

$$D(\texttt{ x = y + r}) = \{(\texttt{x}, \texttt{y}), (\texttt{x}, \texttt{r})\}$$
$$D(\texttt{ x = x - r}) = \{(\texttt{x}, \texttt{x}), (\texttt{x}, \texttt{r})\}$$

and finally:

$$D(\texttt{ x = y + r; x = x - r}) = \{(\texttt{x}, \texttt{y}), (\texttt{x}, \texttt{r})\}$$

where the rightmost entry represents an additional dependency.

3.2 Model-based diagnosis

The basic idea behind model-based diagnosis (MBD) [Reiter, 1987; de Kleer and Williams, 1987] is to use a model of a system directly to compute diagnosis candidates. The prerequisite of MBD is the availability of a logical model of a system which comprises different components. The outcome of diagnosis is a set of components that may causes an observed unexpected behavior. In debugging for example we are interested in statements that contribute to the computation of wrong values for some variables. Hence, statements serve as components. In our case we are interested in finding statements that cause the computation of wrong dependencies, i.e., dependencies that are not specified. Hence, the behavior of components must be expressed in terms of dependencies.

Before showing how models can be derived from dependencies we recapitulate the basic definitions of MBD. In MBD a diagnosis problem is a triple $(SD, COMP, OBS)$ where SD is a logical representation of the structure and behavior of the system, $COMP$ is a set of components, and OBS is a set of given observations, i.e., itself a set of logical sentences. In our case OBS will be the set of specified dependencies. A subset Δ of $COMP$ is a diagnosis iff $SD \cup OBS \cup \{\neg Ab(C)\ C \in COMP \setminus \Delta\} \cup \{Ab(C) | C \in \setminus \Delta\}$ is consistent. In other words a diagnosis is a set of components that when assumed to behave not as expected, i.e., abnormal, does not contradict the given observations. The system description SD comprises logical sentences that correspond to the behavior of the components and the structure of the system. At least the correct behavior has to be element of SD. This behavior is given as sentences of the form $\neg AB(C) \rightarrow \ldots$ where the right-hand-side of the rule specifies the behavior.

Computation of diagnosis can be done by checking all subsets of $COMP$ which is obviously highly inefficient. For one basic algorithm that computes diagnoses usually faster we refer the reader to Reiter [Reiter, 1987; Greiner *et al.*,

1989]. Beside optimization techniques which avoid some re-computations Reiter's algorithm can be easily adapted to search for diagnoses up to a given size. For example, first the single diagnoses are computed, and later on double, triple diagnoses and so on. Since, in most cases we are only interested in single fault diagnoses this algorithm works sufficiently good.

3.3 The dependency model

The dependency model makes use of Jackson's dependency computation. It allows for locating a bug within a program which manifests itself in missing or additional dependencies.

The first part of the model SD_D comprises rules for the correct abstract behavior of the statements. This behavior is given by the rules for computing the dependencies. The only addition to the original rules is to introduce the negated AB predicates. Figure 4 shows the rules that represent the correct abstract behavior.

The other rules for computing the functions D and M remain the same and are assumed to be member of our dependency model SD_D. When we take SD_D together with the assumption that all components are correct, i.e., provide the correct dependency relations, we obtain the same dependencies from SD_D and the original definitions of D and M. However, it remains unclear what happens in cases when assuming a statement to be faulty and as a consequence delivering the wrong dependencies. The idea behind the model for incorrect statements is the following. An incorrect statement contributes in the computation of the dependencies. In cases of assignment statements there are relation entries which are wrong. Hence, if we assume a statement to be incorrect we have to remove these entries. In order to strengthen the result we do not remove all entries but we say that an assignment statement contributes with an arbitrary number of entries. The only fixed part of all entries is the left side which holds the target variable of the assignment. The right side is represented by a model variable χ_ι which represents variables from $VARS$ for the statement at line ι.

$$Ab(x \texttt{ = } e) \rightarrow D(x \texttt{ = } e) = \{(x, \xi_\iota)\}$$

For the model of the incorrect statement we assume that target variables of assignments are correct but the bug lies in an incorrectly used variable on the right-hand-side of the assignment.

The models of incorrect conditional and loop statements are similar. The only difference is that these two statements can provide wrong contributions to the dependency computation only because of the used variables in the conditions. Hence, we introduce a model variable for the condition. Figure 5 summarizes the rules of the incorrect abstract behavior.

The model for alias-free programs now allows for computing dependencies which may comprise model variables ξ_ι. After the computation we have to compare the computed dependencies with the specified ones to check consistency of our assumptions whether a component is correct or not. We perform this comparison by first grounding all model variables ξ_ι and second comparing the relations. If the computed and grounded dependency relation is equivalent to the specified relation the assumptions do not contradict the given observations. Otherwise, we obtain an inconsistency. Note that this definition is quite strong and requires the specification of all dependencies in advance.

The grounding step is done as follows. Assume that we compute the dependency $(\texttt{x}, \xi_1)$ and that we specified the dependencies $\{(\texttt{x}, \texttt{y}), (\texttt{x}, \texttt{z})\}$. We know that ξ_1 can hold every variable. In order to make the dependency relation equivalent, we only have to replace ξ_1 by `y` and `z`. Hence, we remove $(\texttt{x}, \xi_1)$ from the set of computed dependencies and add the two entries $(\texttt{x}, \texttt{y})$, $(\texttt{x}, \texttt{z})$. If for example ξ_1 is used in another relation entry, e.g., $(\texttt{w}, \xi_1)$, we again ground this entry by replacing it with new entries $(\texttt{w}, \texttt{y})$, $(\texttt{w}, \texttt{z})$.

We now illustrate the computation of single fault diagnoses using a small example program.

```
1.  r = d/2;
2.  a = r*r;
3.  c = 2*r*p;
```

We further assume the following specification $\{(\texttt{r}, \texttt{d}), (\texttt{a}, \texttt{d}), (\texttt{a}, \texttt{p}), (\texttt{c}, \texttt{d}), (\texttt{c}, \texttt{p})\}$. The program does obviously not fulfill the specification because statement 2 does not provide all dependencies, i.e., $(\texttt{a}, \texttt{p})$ is missing. We compute all single fault diagnoses by making the appropriate assumptions.

1. Assume $Ab(\texttt{r=d/2})$: Using the rules we obtain the following dependency relation: $\{(\texttt{r}, \xi_1), (\texttt{a}, \xi_1), (\texttt{c}, \xi_1), (\texttt{c}, \texttt{p})\}$. After grounding the statements where ξ_1 is replaced by `d` and `p` we have $\{(\texttt{r}, \texttt{d}), (\texttt{r}, \texttt{p}), (\texttt{a}, \texttt{d}), (\texttt{a}, \texttt{p}), (\texttt{c}, \texttt{d}), (\texttt{c}, \texttt{p})\}$ the computed dependency relation contradicts the specified one. Hence, $\{\texttt{r=d/2}\}$ is not a single diagnosis.
2. Assume $Ab(\texttt{a=r*r})$: For this assumption we obtain $\{(\texttt{r}, \texttt{d}), (\texttt{a}, \xi_2), (\texttt{c}, \texttt{p})\}$. After grounding where ξ_2 is replaced by `d` and `p` the computed dependency relation is equivalent to the specified one. Therefore $\{\texttt{a=r*r}\}$ is a diagnosis.
3. Assume $Ab(\texttt{c=2*r*p})$ from which we compute $\{(\texttt{r}, \texttt{d}), (\texttt{a}, \texttt{d}), (\texttt{c}, \xi_3)\}$. Grounding by replacing ξ_3 with `d` and `p` again leads to a set which contradicts the given specification. `c=2*r*p` is not a single fault diagnosis.

From the above derivations we see that the statement in line 2 remains to be the only single fault diagnoses.

3.4 A diagnosis example

We represent the `myMult` example program from (Fig. 1) to illustrate our approach. The ideas behind our approach is the following. During the computation of abstract dependencies every statement has impact to overall dependency set. Here we specify that `result` depends on `x` and `y` which is written as $\{(\texttt{result}, \texttt{x}), (\texttt{result}, \texttt{y})\}$.

The computed dependencies for `myMult` are depicted in Figure 6. The dependency in line 3 is the summarized dependency for the while-statement.

The computation of all single diagnoses is done in the same way as before. We assume a statement to be incorrect and the

$$\neg Ab(x \texttt{ = } e) \rightarrow D(x \texttt{ = } e) = \{(\texttt{x}, v) | v \in vars(e)\}$$

$$\neg Ab(\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2) \rightarrow$$
$$D(\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2) = D(S_1) \cup D(S_2) \cup ((M(S_1) \cup M(S_2)) \times vars(e))$$

$$\neg Ab(\texttt{while } e \texttt{ \{ } S \texttt{ \} }) \rightarrow$$
$$D(\texttt{while } e \texttt{ \{ } S \texttt{ \} }) = D(S)^* \cup (M(S) \times vars(e)) \circ D(S)^*$$

Figure 4: The correct abstract behavior

$$Ab(x \texttt{ = } e) \rightarrow D(x \texttt{ = } e) = \{(x, \xi_\iota)\}$$

$$Ab(\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2) \rightarrow$$
$$D(\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2) = D(S_1) \cup D(S_2) \cup ((M(S_1) \cup M(S_2)) \times \{\xi_\iota\})$$

$$Ab(\texttt{while } e \texttt{ \{ } S \texttt{ \} }) \rightarrow$$
$$D(\texttt{while } e \texttt{ \{ } S \texttt{ \} }) = D(S)^* \cup (M(S) \times \{\xi_\iota\}) \circ D(S)^*$$

Figure 5: The abstract behavior for buggy statement

```
    int myMult (int x,y) {
1.    int result = 0; // D(1)={}
2.    int i = 0;// D(2)={}
3.    while ( i < x ) {
// D(3)={(result,result),(result,x),
//      (result,i),(i,i),(i,x)}
4.      result = result + x ;
// D(4)={(result,result),(result,x)}
5.      i = i + 1; }
// D(5)={(i,i)}
6.    return result;
// D(6)={(result,result),(result,x),
// (result,i),(i,i),(i,x)}
      // Specified dependencies:
      // {(result,x),(result,y),
      // (result,result),(result,i),
      // (i,i),(i,x)}
    }
```

Figure 6: The computed dependencies for the integer multiplication example

remaining statements to be correct. Compute the dependencies, ground them and compare them with the specified dependencies. If a contradiction arises the assumption that the selected statement is incorrect is wrong. The computational complexity is polynomial in the number of statements provided a polynomial algorithm for computing dependencies.

When using our model we obtain two single fault diagnosis. Statement 1 or 4 can be the root cause of the detected differences between the dependencies. For example, statement 5 cannot be responsible for the following reasons.

Assume statement 5 to be abnormal, i.e., $Ab(5)$. From this we derive the dependency $D(5) = \{(\texttt{i}, \xi_5)\}$ which leads to the summarized dependency

$$D(3) = \{(\texttt{result}, \texttt{result}), (\texttt{result}, \texttt{x}), (\texttt{result}, \texttt{i}), (\texttt{result}, \xi_5)(\texttt{i}, \xi_5), (\texttt{i}, \texttt{x})\}$$

of statement 3. After grounding we obtain to dependency relations (result,y) and (i,y) where the former is an element of the specified dependencies but the latter is not. We obtain a contradiction and conclude that our assumptions cannot be true anymore. With similar computations we can rule out statement 2 from the list of candidates. Statement 3 can also not be the a candidate because it would lead to a dependency (i,y) which is not a specified one.

4 Limitations

The limitations of the proposed approach are the same as for Aspect system. Additional computed dependencies may not be the result of a wrongly written statement but is caused of the used approximation algorithm. For example consider again the code fragment:

```
1.  x = y - r
2.  x = x + r
```

where x only depends on y and not on r. If we specify $OBS = \{(\texttt{x}, \texttt{y})\}$ as the expected dependency relation, the contradiction relation $\{(\texttt{x}, \texttt{y}), (\texttt{x}, \texttt{r})\}$ would be computed. Using the given model we finally obtain a double-fault diagnosis which comprises both statements. This result is not correct when assuming diagnosis candidates to be always the cause of a faulty behavior. However, in terms of differences between a given specification and a computed specification the result is correct because it says only that the cause of the difference lies in both statements. Hence, the approach is correct with respect to the model. Because of the used abstraction, results maybe not correct with respect to the program's semantics. This limitation can be seen as an effect of the used abstraction.

Another limitation is the due to the used model. The model allows to deal with differences between dependencies. However, it does not allow to deal with situations where we know that the outcome of a program is faulty. Other models are available for this purpose and we refer the reader to [Wotawa, 2002; Wieland, 2001].

The described model can only be used for alias-free programs. An extension to pointer handling, structures and func-

tion calls is straightforward. The underlying techniques are described by Jackson [Jackson, 1995].

5 Related research

The used model strongly depends on the concept of dependencies between variables. This concept of dependencies has been used for verification purposes [Jackson, 1995] but also for debugging, i.e., fault localization, [Kuper, 1989; Wieland, 2001]. In contrast to these approaches we do not use detected differences in variable values at a certain line in the code, but make use of differences between specified and computed dependencies. Hence, our model is an extension to previous research.

In [Zeller and Hildebrandt, 2002] Zeller introduced a method for reducing the input to a minimal subset which still entails a faulty behavior. His approach can also be used for locating a bug using a cause effect chain but the computation of faults is limited to one diagnosis because of the underlying binary search algorithm. We use model-based diagnosis which always guarantees to find all possible bugs with respect to the given model.

Other approaches like [Johnson, 1986] focus on novice programmers and make use of methods that help to find faults in the code by comparing the code with pre-specified problem formulations. If the given problem cannot be realized by the novice's code, then the reason for this difference is given back as a result. This approach can only be used in situations where the problem is formulated in a formal way which is unlikely for general programs.

6 Conclusion and future research

In this paper we introduced a novel model that helps to locate bugs in a program. The model is based on dependencies between variables that can be extracted at least approximately directly from the source code. These dependencies are compared with pre-specified dependencies. In case of unexpected differences the model allows to compute diagnosis candidates. These candidates explains possible causes for the detected differences. The approach is different to other available dependency-based models and provides better results for some examples as explained in the paper.

Future research has to extend the model to handle pointers, function calls, data structures, and other programming language specific features. Moreover, an empirical evaluation has to be carried out in order to prove the usefulness of the approach.

References

[Cousot and Cousot, 1977] Patrick Cousot and Radhia Cousot. Abstract interpreation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *in Proc. POPL'77*, pages 238–252, Los Angeles, 1977. ACM.

[de Kleer and Williams, 1987] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[Forbus, 1984] Kenneth D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

[Friedrich *et al.*, 1999] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999.

[Greiner *et al.*, 1989] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

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

[Johnson, 1986] W. Lewis Johnson. *Intention-Based Diagnosis of Novice Programming Errors*. Pitman Publishing, 1986.

[Kuipers, 1986] Benjamin Kuipers. Qualitative simulation. *Artificial Intelligence*, 29:289–388, 1986.

[Kuper, 1989] Ron I. Kuper. Dependency-directed localization of software bugs. Technical Report AI-TR 1053, MIT AI Lab, May 1989.

[Mayer and Stumptner, 2004] Wolfgang Mayer and Markus Stumptner. Debugging program loops using approximate modeling. In *Proc. ECAI'04*, pages 843–847, Valencia, Spain, August 2004.

[Reiter, 1987] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

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

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

[Weld and de Kleer, 1989] D. Weld and J. de Kleer, editors. *Readings in Qualitative Reasoning about Physical Systems*. Morgan Kaufmann, 1989.

[Wieland, 2001] Dominik Wieland. *Model-Based Debugging of Java Programs Using Dependencies*. PhD thesis, Vienna University of Technology, Computer Science Department, Institute of Information Systems (184), Database and Artificial Intelligence Group (184/2), November 2001.

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

[Zeller and Hildebrandt, 2002] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), feb 2002.