Evaluation of Value-based Models for Java Debugging*

Wolfgang Mayer
Database and Artificial Intelligence Group
Institut für Informationssysteme
Vienna University of Technology
mayer@dbai.tuwien.ac.at

December 6, 2001

This document compares the value-based model for Java programs and its loop-free variant. The observations obtained from an evaluation of the models using a set of example programs are presented. In particular, weaknesses, room for improvements and possible enhancements of the models are pointed out.

1 Simple value-based model

The simple value-based model (VBM) [4, 5] is designed to be derived statically from a Java program by translating the program's structure into a set of interconnected components. Although this model allows to find certain faults effectively, there are some classes of programs for which the model is not effective enough. This section describes some of the model's drawbacks and approaches to avoid them.

1.1 Weaknesses

The problems dealt with in this section are mostly related to low precision of the diagnosis results, leaving aside performance-related issues. However, some of the problems indirectly influence performance by causing the search space to enlarge. Limitations and drawbacks presented here include the aliasing between variables, false consistency due to missing object identifiers, duplicated faults, inappropriate fault modes when applying multiple test cases, incompleteness of (simple) hierarchical diagnosis and problems related to loop initializers and granularity of diagnosis.

1.1.1 Aliasing

An analysis of the results obtained on a set of example programs, it was discovered that aliasing between variables significantly influences the quality of diagnosis results.

Due to the undecidability of the aliasing-relation between variables referencing objects on the heap, these variables are represented imprecisely by the model. Each variable pointing to an object does contain an object-identifier that uniquely represents an object in the object-environment, which contains the value of the object's fields. Each time a field of an object is modified, a new version of the object-environment is generated and used subsequently.

* This work was partially supported by the Austrian Science Fund project P12344-INF and project N Z29-INF.

To reduce the copying overhead, the environment is splitted such that each instance variable of a class is represented by a separate connection. This also improves the precision, as modifications of instances of distinct instance variables can be separated during diagnosis.

However, accesses to instances of the same instance variable of separate objects cannot be distinguished by the model. The corresponding statements are modeled as a sequence of components, where each component has one input-port corresponding to the object-environment before the assignment, and an output-port related to the new version. Therefore, the statements form a sequence where each component is dependent upon its predecessor. Consequently, the diagnosis engine cannot distinguish between the statements, as the ab-fault-mode of a component invalidates any values computed by previous statements.

As an example consider the following program:

```
1  class ValueHolder {
2      int value;
3
4      static void demo()
5      /*@testcase : ( true )@*/
6      {
7          ValueHolder a = new ValueHolder();
8          ValueHolder b = new ValueHolder();
9          b.value = 2;
10         a.value = 0;
11         b.value = 1;
12     }
13     /*@testcase : ( a.value == 1) @*/
14 }
```

The example assumes a fault in the statement at line 10. Instead of the value 0, the value 1 should have been assigned. Given the program and the observation at the end of the method, it can be seen that the value of the object pointed-to by the variable a is faulty. Therefore, the statements in lines 7 and 10 are expected as possible locations of the fault.

However, the model cannot distinguish accesses to the two objects and delivers almost every single statement of the method as possible explanation (assuming the value-based model is applied at the statement-level).

The only statement that is not considered faulty is the assignment to $b.value$ in line 9. This is caused by the implementation of the model of assignment-statements that do not require a valid object-environment as input to propagate the assigned value to the output. Consequently, a contradiction can be derived without knowing the value of the object-environment before line 10.

To improve the diagnosis results, additional information about aliasing relations between variables can be introduced into the model. The additional knowledge makes it possible to modify the structure of the model such that components operating on separate objects can be modeled independently and thus to obtain better results. This approach is described in more detail in section 1.2.1.

1.1.2 Missing object identifiers

The approach to model references to objects through indirect accesses using references is well suited to avoid problems related to aliasing between variables. However, this method requires that an object reference is known for each access or modification of a field of an object. As a consequence, nothing can be done without the reference. For the diagnosis process, this observation implies that the statement providing the object reference will often be included in a diagnosis, as without the reference the values dependent on the object cannot be computed and are thus consistent with any observation.

A more detailed description of the relevant aspects of the model and its implementation follows. All components representing accesses to fields of objects or calls to instance methods are given an additional input-port that provides this reference. If no valid reference is given, the component's behavior is equivalent to the ab-behavior, i.e. no outputs are computed. Therefore, whenever the method being debugged includes a new-expression, and some statement located between a contradicting observation of a field and the true location of the fault modifies the same instance variable (this could even be a different object), the new-expression will be considered a diagnosis candidate. If the resulting object-identifier is assigned to a variable, the assignment will also be a diagnosis candidate.

In the example from section 1.1.1, the statement in line 8 is identified as a possible location of a fault because the assignment in line 11 depends on the object-identifier contained in b and modifies the same instance variable as the faulty statement in line 10.

1.1.3 Duplicated faults

When limiting the search for diagnoses to one statement per candidate, faults cannot be found if they are embedded in methods that are called multiple times (directly or indirectly through other methods). This is caused by the fact that the model cannot distinguish a wrong called method from a wrong implementation of the correct method. Therefore, a fault in the implementation of a method manifests as multiple faults if the method is called multiple times. As a consequence, the faults cannot always be located without increasing the diagnosis cardinality.

The following example visualizes the model's inability to locate duplicated faults:

```
1  class Hierarchic {
2      static int x = 0;
3      void demo()
4      /*@testcase : ( true )@*/
5      {
6          modify();
7          int y = x;
8          modify();
9      }
10     /*@testcase :( x==1)@*/
11     /*@testcase :( y==1)@*/
12
13     void modify() {
14         x = 2;  /* 1 */
15     }
16 }
```

When debugging method $demo()$, the single fault in line 14 of method $modify()$ cannot be located. This is caused by the fact that in method $demo()$, both variable x and y are incorrect. The structure of the method ensures that no single component failure in the model of method $demo()$ can explain all observations. Therefore, no single fault can be found in $demo()$, although it clearly can be found at the level of $modify()$.

This observation is not specific to this version of the model. It is also present in the variants of the model. In particular, this problem is worsened in a variant of the loop-free model, where faults in a loop statement are duplicated by unrolling the loop (see section 2.1).

1.1.4 Inappropriate fault modes

To improve precision of the model, multiple test cases can be applied to a method concurrently. This helps to avoid spurious results for many programs, but can also fail to precisely locate faults for some program constructs. Especially for loop statements some of the fault modes are unsuitable for the application of multiple test cases.

Applying multiple test cases concurrently, the fault modes $loop(\cdot, max)$ in their current implementation are inappropriate because they require the test cases to perform exactly the same number of iterations. When this is not the case, the fault modes cannot be applied and the more general fault mode ab remains as single alternative.

This is easily demonstrated using the following program:

```
class Loop2 {
    void demo(int stop)
    /*@testcase1:(stop==3)@*/
    /*@testcase2:(stop==5)@*/
    {
        int i = 0;
        while (i <= stop) {
            i++;
        }
    }
    /*@testcase1:(i==3)@*/
    /*@testcase2:(i==5)@*/
}
```

When applying the two test cases separately, the assignment and the loop are considered possible locations of the faults. The statement is assigned the fault modes $loop(\cdot, 3)$ and $loop(\cdot, 5)$, respectively. However, when applying both test cases, the loop is assigned the weak fault mode ab. A closer examination of the symptoms indicates that the loop exceeds the desired number of iterations by one, as was implicitly indicated by the results obtained by applying the test cases separately. However, this cannot be expressed by the fault modes in their current form as the number of iterations is different for each test case.

To avoid this problem, the fault-modes of the loop can be changed to stop the execution of the loop a specified number of iterations earlier or later than the original loop does, regardless of the concrete number of iterations the original loop executed. However, this approach can only be taken when the number of iterations of the original loop can be determined, which cannot always the done.

A similar problem is present in the loop-free variant of the model (see section 2.1). There, the default mode of conditional statements may be different when applying multiple test cases, causing explanations for faults to be omitted from diagnoses unless the maximal diagnosis cardinality is increased sufficiently. Therefore, the notion of diagnoses has to be modified to abstract from modes and consider components only. However, this is currently not integrated into the debugging environment and cannot be applied during the diagnosis process.

1.1.5 Hierarchic debugging

When diagnosing hierarchic structures, such as calls to methods or loops, two problematic cases can be observed. First, the current implementation of the hierarchic diagnosis process is incorrect. Second, single faults in a method cannot be found at an abstract level if the method is called multiple times. The latter problem is addressed in section 1.1.3.

It cannot be guaranteed that current implementation of hierarchic diagnosis is correct in every case. This is caused by the fact that – when entering a hierarchic component – the model at a higher level is discarded. Only the values propagated to the input- and output-connections of the entered model are retained. This procedure is correct if the values of all connections are known. However, if the values of some of the connections are unknown, incorrect diagnoses can be computed at the detailed level.

Consider the following program:

```
public class IncorrectDiagnosis {
    static int a = 1,
               b = 2;
```

```
    static void demo()
    /*@testcase:(b==2)@*/
    {
        sideeffect ();
        int c = a * a;
    }
    /*@testcase:(b==4)@*/
    /*@testcase:(c==1)@*/

    static void sideeffect () {
        int i =0;
        while(i < 0) {
            a =3;
            b =4;
        }
    }
}
```

When debugging the method $demo()$, the call to method $sideeffect()$ is computed as the single possible location of the fault. Shifting the focus to the called method, two diagnoses are computed: $ab([i = 0])$ and $loop([while], 1)$. Note that the values of the input-connections representing the variables a and b and the output-connection representing b are known, whereas the value of the output-connection representing a is unknown, as it cannot be computed by the constraint network representing method $demo()$. Therefore, the value 3 that is implied by the second diagnosis does not cause a contradiction within $sideeffect()$. However, the diagnosis is not valid in $demo()$, as it contradicts the observation that c is correct.

This effect does not only arise when values of connections are completely unknown, but can also occur when values of connections can be extended by other values, such as connections representing object-environments.

For single-fault-diagnoses it is sufficient to simulate the diagnoses and check whether a contradiction occurs at some more abstract level. However, when searching for multiple faults, this procedure is not complete and the constraints of the more abstract levels have to be taken into account (without considering them as candidates for diagnoses).

1.1.6 Loop variables

Further imprecision of the model is caused by the fact that values cannot be propagated through loops unless the number of iterations is known. The number of iterations must be determined by forward simulation of the loop and depends on the values of the loop variables before each iteration. In particular, the values before the first iteration of the loop and the values of all variables used in the loop's condition must be known. Otherwise, the component representing the loop cannot compute values for any of its output connections. Consequently, any statement that contributes to the value of the loop variable(s) is considered a diagnosis candidate, even if the statement and the loop cannot explain the observations.

As an example consider the following program:

```
class Loop {
    static void demo()
    /*@testcase: ( true )@*/
    {
        int x =1; /* should be 10 */
        int i =0;
        while ( i < 5) {
            x += 99;
            i++;
```

```
10            }
11        }
12        /*@testcase:(x==10)@*/
13    }
```

Given the observation stated in the postcondition of the method, the assignment to i in line 6 cannot be responsible for the misbehavior (under the single-fault-hypothesis). Caused by the behavioral description of the loop, as explained above, it is nevertheless identified as a possible location of the fault, as the number of iterations of the loop cannot be determined without knowing the initial value of i.

Note that this problem is specific to this version of the value-based model. It is no longer present in the loop-free model. This is done by decoupling the evaluation of the loop from the initial values of the loop variables by determining the number of iterations in a preliminary step. Although this approach avoids the problems for some examples, it causes some deficiencies for other examples (see section 2.1.1).

1.1.7 Granularity

The granularity of diagnosis is still too coarse for programs containing loop statements. This is rooted in the fact that the whole loop statement is considered a single diagnosis component. Consequently, the diagnosis component can be entirely correct or be faulty. No distinction is made between the contained statements. The fault modes $loop(\cdot, k)$ are introduced to improve the results in that certain faults in the condition can be located. However, this is not enough for large classes of programs. In particular, faults in the body of loops cannot be located. A possible solution is to expand the loop in a sequence of nested conditional statements and to perform diagnosis on the expanded program. For more details see the description of the loop-free model in section 2.

1.2 Improvements

This section describes solutions to some of the weaknesses presented in the previous section. An approach to integrate information about the heap structure of the programs and an approach based on strong fault modes are presented.

1.2.1 Points-to information

To enhance the model, additional information about the structure and the relation between heap objects can be integrated into the model. Information about aliasing relations between variables, in particular, which variable possibly points to which object(s), can be exploited. However, due to the fact that the diagnosis engine implicitly modifies the original program by setting specific fault modes of components, only such information may be used that remains valid for all those variants of the program.

To improve the accuracy of the value-based model, the connections representing object environments can be split into separate connections if it is known that the variables used to access objects reference different objects. By doing this, the number of dependencies between components can be reduced and the diagnosis results can be improved, as a single ab-component is dependent on (and thus can discard) less values computed by other components than with the original representation.

Extending the original model to incorporate information provided by a points-to analysis[1], the connections representing object environments have to be split into several connec-

[1] The current implementation uses a graph that approximates the structure of all objects on the heap. The nodes in the graph are given by the variables of the program and locations, which represent instance creation expressions in the program. The edges of the graph represent references to objects and are labeled with the name of the field that holds the reference. Further, summary locations represent collections of objects created by some instance creation expression. This is necessary to model loops and arrays, where the number of created objects cannot be determined at compile-time. To get an idea on how the graph is constructed, see [2, 1].

tions, each representing a set of objects a variable can refer to. As this information changes during method execution (e.g. by assignment statements, etc), the connections have to be merged and split multiple times. This is accomplished by additional components that perform these operations, without being candidates for diagnoses. In particular, a component is required that combines or splits the contents of connections according to given sets of locations. Such components have several input and output connections, each associated with a set of locations. The component's behavior distributes all incoming objects according to the location information attached to their object identifiers (each object identifier stores an identifier of the location which is associated with the statement that created the location). An incoming object is propagated to a connection iff the location-id of the object is included in the set of locations associated with the connection. The output of the component are two connections: one is associated with a given set of desired locations (which are required by the modeling process), whereas the other connection is associated with all locations provided by input connections and are not desired locations. The input connections of the component are all connections that are associated with at least one desired location.

To illustrate the process, consider the following example:

```
1  class Value {
2      int value;
3
4      Value(int v) {
5          value = v;
6      }
7
8      static void demo(boolean choice)
9      {
10         Value a = new Value(1); /*l1*/
11         /* a->l1 */
12         Value b = new Value(2); /*l2*/
13         /* b->l2 */
14         Value c = a;
15         /* c->l1 */
16         Value d = b;
17         /* d->l2 */
18
19         if (choice)
20             a = b;
21         /* a->{l1,l2} */
22         if (choice)
23             b = new Value(3); /*l3*/
24         /* b->{l2,l3} */
25
26         b.value = 4;
27
28         int va = a.value;
29         int vb = b.value;
30         int vc = c.value;
31         int vd = d.value;
32     }
33 }
```

Based on the program's structure, the points-to information of the variables at each position of the program can be computed statically by the heap-structure analysis algorithm. It is assumed that the result of the analysis are associations between variables and sets of abstract locations. The locations represent abstractions of the objects the variable can possibly reference at each point of the program. I.e. the association for the variable b in line 24 denotes that b either points to location $l2$ or to location $l3$ at this point of the program. The information remains valid until a new association for the variable is given.

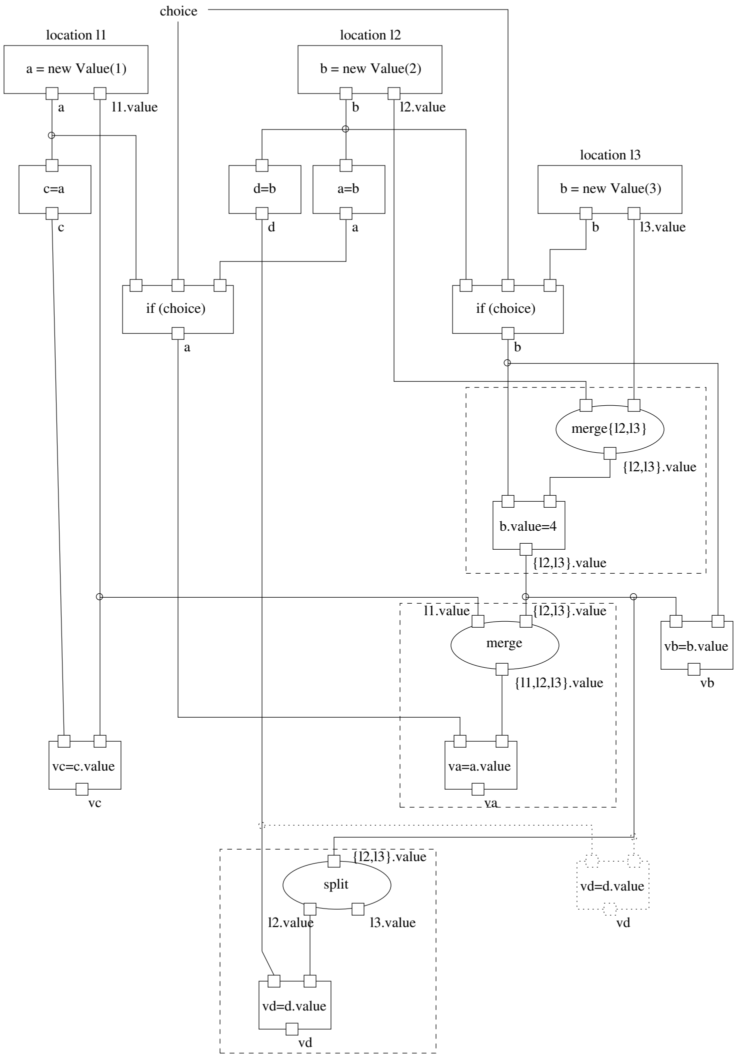
Based on this information, the modeling algorithm splits the object environment into separate connections, each of them representing sets of objects that are represented by one or more abstract locations. In the example, the connections representing the values of the instance variable $value$ of the three created objects are represented by three different connections. Accesses and modifications of these variables can thus be modeled independently, as long as the sets of possibly accessed objects is statically known to be disjoint.

However, after the modeling of the conditional statements, the one-to-one associations between variables and locations is lost, because it cannot be determined statically which branch of the if-statement is executed. Therefore, the variables a and b are associated with both possible locations. When modifying the object referenced by b, both possibilities have to be taken into account. Therefore, both connections representing the values of the objects at locations $l2$ and $l3$, respectively, have to be merged into a single connection. This is done by an additional component that is excluded from being a diagnosis candidate. The resulting connection can then be used as an input connection to the component that represents the assignment. In contrast to the original model, the output port of the component is not associated with the variable b, but with the two possible locations b can point at.

The access of the object field in line 28 is handled similarly. Here, the variable a points to either $l1$ or $l2$. Therefore, a connection representing both of these locations has to be created. However, there is no connection representing the single location $l2$, but only a connection associated with $\{l1, l2\}$, which is used instead. Therefore, the new connection represents the locations $l1$, $l2$ and $l3$ and can safely be used as input to the component modeling the access of $a.value$.

When modeling the field access in line 31, it is known at compile-time that d points to the object represented by location $l2$. However, the active connection containing the value of $l2.value$ is not available directly. Only a connection representing the values of $l2.value$ and $l3.value$ is available. (It was created during the modeling of the assignment to $b.value$ in line 24.) To obtain the desired value, the connection has to be split into two separate connections representing $l2.value$ and $l3.value$, respectively. The connection is subsequently used to model accesses and modifications of $l1.value$, until the points-to information changes. Alternatively, the connection that was created when modeling the access to $a.value$ in line 28 could have been used. Note that splitting the connections is not necessary, as accessing and modifying the field can be modeled as in the original value-based model. The alternative procedure is indicated with dotted lines in the figure above. However, when many subsequent statements are accessing or modifying the same field and precise points-to information is available, it can be advantageous to split in order to avoid unnecessary dependencies between the components.

In summary, the program is compiled into the model below:



Note that the model is closely related to the original model. In particular, the components representing assignments to fields of objects are given the object-identifier and the previous value of the object as inputs, although they can be omitted in many cases. These connection can be eliminated whenever the location of the accessed or assigned-to object can be determined statically and it is know that the location is not a summary location. Otherwise, the connections represent multiple objects and cannot be left out. Also, the behavior of the component might change in the optimized version of the model in case the object-identifier is `null`. The unoptimized version is able to detect this case and raise an exception (conforming to the JLS), whereas the optimized version does not have this information and thus deviates from the expected behavior. As the unoptimized version of the model better conforms to the JLS and it is easier to implement, the optimized version is not implemented in the debugging prototype, but is left for future enhancements.

The modeling of loop statements and method calls can be done in two ways. A solution is to model each method and each loop separately and provide unique interfaces (as in the original value-based model) to the calling method. While this method is advantageous in that each method has to be analyzed only once (fast model building!), several problems have to be dealt with. First, the methods have to be analyzed using conservative assumptions on their environment, leading to imprecise points-to graphs and in sequence to little improvement in the model structure. Further, the locations obtained by the points-to analysis local to the method do not generally match the locations attached to the object identifiers provided by the calling method. Therefore, a mapping between these two locations must be performed for each call.

To avoid these problems, a different approach has been taken. In the current implementation each called method and each loop statement is analyzed separately for each call-site, using the points-to information provided by the calling procedure. This makes it possible to obtain precise points-to information and optimize each call accordingly. Further, the mapping of locations is not necessary, as they are provided by the context of the loop or the method call, respectively. The drawback of this approach is that each method has to be analyzed again for every call, causing a significant slowdown of the model-building process.

Caused by the changes in the structure of the model, the faults that can be detected with the model have changed. For example, faults that cause the object structure to change or that cause variables to point to different objects cannot be found in every case. This is especially true for the optimized version of the model, which omits certain connections from the model. For example, consider the assignment to c in line 14 of the above program. Tough the statement clearly influences the value of vc computed in line 30, the component representing the assignment will not be considered a possible location of a fault unless an observation for the variable c is given explicitly (in the optimized model).

Another limitation of the model is found when analyzing the model's assumptions. The approach presented here is based on the assumption that the points-to information (and thus most of the object structure) remains the same for all diagnosis candidates examined by the diagnosis engine. Otherwise the structure of the model does not correctly represent the program and wrong diagnosis results can be obtained. It is expected that this assumption cannot be guaranteed for a large set of programs. In particular, arrays containing non-trivial data types are expected to lead to low accuracy of the points-to analysis, as the array-indices usually cannot be computed at compile-time. Therefore, it cannot be known which element of the array (and thus which location) is accessed or modified.

For single-component faults, the a method and the observations must conform to the following restrictions in order to guarantee that the points-to information is valid for all potential diagnoses candidates:

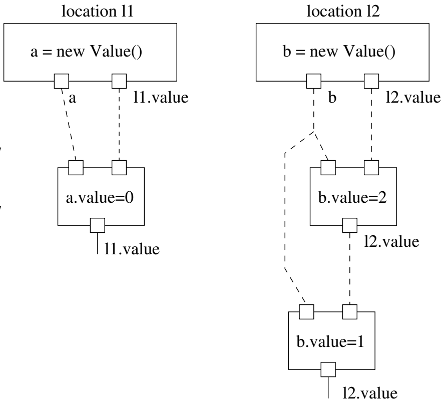
- no reference equality/inequality comparisons may be present in the program (or no backpropagation through these components must occur),
- no observations given for variables of reference type that contradict any value derived by any diagnosis candidate may be given,
- no equality/inequality-assertions for variables of reference type may be specified that contradict any of the diagnosis candidates,
- no precise points-to analysis may be used (e.g. using partial evaluation to eliminate one branch of a conditional statement),[2]
- no exact modeling of array elements in points-to information. In particular it must not be be assumed that the array index of an array access or an array assignment expression remains fixed.

Although the model is expected to work reasonably well for simple methods without method calls, loops and conditional statements, it is not well-suited to represent complex programs whose object structure is unknown at compile-time or whose object structure often changes (for example a balanced search tree). In the worst case, the model is equivalent to the original value-based model, with the addition of splitting and merging components.

Closing the presentation of the model enhancement, the representation of the program of section 1.1.1 using the points-to information is given below. The dashed lines represent connections that can be omitted in the optimized version of the model:

[2] If the program is restricted such that there are no conditional statements and loop statements that modify or create object references, this condition can be omitted.

```
1   class Value {
2       int value;
3
4       static void demo()
5       /*@testcase : ( true )@*/
6       {
7           Value a = new Value(); /*l1*/
8           /* a->l1 */
9           Value b = new Value(); /*l2*/
10          /* b->l2 */
11          b.value = 2;
12          a.value = 0;
13          b.value = 1;
14      }
15      /*@testcase : ( a.value == 1)@*/
16  }
```



The diagnoses obtained with the new model now coincide with the expected results (i.e. the assignment to a in line 12 for the optimized model; in the unoptimized version line 7 is also included) as the accesses and modifications of the two objects are now completely orthogonal.

1.2.2 Diagnosis assumptions

The model presented in the previous section can provide suitable representations of programs in many cases. However, it needs further improvements when the object structure is unknown at compile-time (e.g. arrays or dynamic lists of unknown length), cannot be split for some reason (summary-locations, unsuitable program structure, etc.) or the assumptions on the object references cannot be guaranteed for all diagnosis candidates. In these cases, the model falls back to the representation of the original value-based model.

To enhance the diagnosis results in these cases, additional assumptions on heap data structures and object accesses can lead to improvements in diagnosis quality, even if the approach presented in the previous section is not applicable.

Rather than modifying the structure of the model itself, the behavior of the components is changed to avoid some undesirable effects when dealing with components in ab-mode. The low accuracy of the model is caused by the loss of all previous values computed for an instance variable if a component modifying the variable is in ab-mode, even if the corresponding statement modifies an unrelated object. Therefore, the theorem prover derives that the model is consistent and reports the statement as explaining the observations.

The solution to the problem proposed here is based on the idea of replacing the ab-fault-mode of certain components by new fault modes relying on stronger assumptions about the component's misbehavior. In particular, assumptions on the structure of the heap data structures and the object references are made. By that, similar results as with the model presented in the previous section can be achieved, without the assumption that the object structure must remain stable for all diagnosis candidates. This is important as it is unclear for which programs such assumptions can be guaranteed. Further, no changes in the (relatively simple) structure of the model have to be made. Therefore, the modeling algorithm can be applied unchanged. Only the behavioral description of the components and the application of the diagnosis engine have to be modified slightly.

When looking at the example program in section 1.1.1, it becomes obvious that the superfluous diagnoses are caused by missed contradictions due to values discarded in ab-components. To overcome this problem, the ab-mode of components representing new-expressions or modifications of instance variables or arrays are replaced by a stronger fault mode. The new mode assumes that the object that is accessed is indeed the object that was intended to be accessed. This enables the fault mode to propagate the values of all other objects and thus enable the diagnosis engine to derive contradictions more often.

To visualize the effects of the new fault modes on the quality of diagnoses, reconsider the example program of section 1.1.1. Assume that structure of the program's model remains the same, whereas the ab-mode of the components representing assignments to the instance variable $Value.value$ has been replaced by stronger mode ab' as explained in the previous paragraph. Consider the component representing the assignment in line 11 as being in ab'-mode. All other components are assumed to behave normally. Based in the component's inputs, it determines the object that would be modified by the statement if the component was not abnormal. In this case, the object referred-to by b is selected. The values of all other objects (in this case: a) are propagated to the component's outputs. In sequence, the diagnosis engine derives a contradiction with the propagated value and the assertion given at the end of the program. Therefore, the fault of the component cannot explain the observations. A similar approach can be applied to the new-expression in line 8. As the assignment in line 9 is not contained in any conflict set, the remaining diagnoses are the new-expression (together with the assignment to a) in line 7 and the assignment in line 10. This is roughly equivalent to the results of the model presented in the previous section and corresponds to the expected results.

The results can be improved further by introducing additional fault modes for some components. For example, it can be assumed that new-expressions are correct in that it really is intended to create a new object of the specified type. The body of the constructor can still be faulty. If it is further assumed that the assignments to local variables of reference type are correct, together with the fault modes for the assignment to instance variables, the statement at line 10 can uniquely be identified as being responsible for the fault. Similar approaches can be applied to components modifying values of arrays. Here, beneath the assumption that the correct heap object is accessed, it is assumed that the array index is correct.

An advantage of this technique is that the diagnosis process can be applied repeatedly, using different fault modes for several kinds of components, until the desired number of diagnosis candidates is obtained. The process can be done incrementally, each iteration based on the results of the previous iteration and using different assumptions and component modes. By that, faults that occur with high probability can be searched first, followed by less probable ones. In particular, if no diagnoses can be found using a specific model, the diagnosis engine switches to another model with changed assumptions to obtain usable results. A drawback of the approach is its incompleteness, i.e. it is no longer guaranteed that the fault is contained within the computed diagnoses.

The list below contains the combinations of fault modes currently used in the debugging prototype (listed in the order they are applied).

$\{$**arrayIndex,objectStructure,instanceCreation,referenceAssignments**$\}$ It is assumed that all expressions representing array indices are correct (and that all other array elements are left unchanged by the array assignment). Further, all components representing instance creations or assignments to variables of reference type are excluded from diagnoses. Considering instance creation expressions, only the part of the behavior that generates the object identifier is assumed to be correct. Faults in the body of the constructor or in instance initializer expressions can still be detected. All components that modify values of objects are assumed to modify the correct object. All other objects provided as input to the component are left unchanged. These assumptions are focused on the detection of faults on the right-hand-side of assignment expressions.

$\{$**objectStructure,instanceCreation,referenceAssignments**$\}$ If no diagnoses can be found using the previous set of assumptions, the assumptions about the correctness of expressions representing array indices is retracted. Therefore, now faults in both the value-part of assignments and the index-part of array-assignments (and combinations of both) can be located.

{**instanceCreation,referenceAssignments**} This set of assumptions excludes assignments to variables of reference type and instance-creation-expressions from diagnoses. The rationale behind this is that the components accessing and modifying objects need the object identifier to execute their behavior. If the identifier is not known, nothing can be done and (spurious) consistency can be achieved in many cases. To avoid this, the model must provide as much object identifiers as possible. The assumptions listed here are used to achieve this while keeping the restrictions on the component's behavior at a minimum.

{**referenceAssignments**} This set of assumptions is comparable to the previous one, except that the restriction on the instance creation expressions is removed. Now faulty instance creation expressions can be detected.

$\emptyset$ No restrictions on the component's behavior is specified. The behavior of the model is equivalent to that of the original value-based model.

The fault-mode-based approach does not rely on the assumption that the structure of all heap objects and references to them remain the same for all diagnoses candidates and conflict sets. The assumption can be relaxed as the connections of the model representing instance variables contain the values of all heap objects, regardless of the actual structure of the heap. The additional dependencies introduced by this representation are partly removed by the strong fault modes. However, these modes can only be effective if the object to be accessed is known by the components, i.e. their object identifier is provided as input. Especially in the case of multiple faults, this is not always the case and the approach loses effectiveness. However, the model is expected to be quite effective when searching for faults with cardinality one.

Another limitation of the approach is its limitation to single component faults. As the various fault modes assume certain properties of programs, and the assumptions are global for the whole program, multiple faults can only be found if the faulty components share the same assumptions. I.e. multiple faults consisting of single faults requiring different assumptions cannot be found with the approach presented in this section.

2 The loop-free model

The value-based model and its variants of the previous section is sufficient to locate faults in most loop-free programs. However, programs that contain loops are not well-suited to be analyzed with these models. The loop-free value-based model (LFM) [6] tries to overcome this limitation and provide means to locate faults within loops. This section provides insight in some of the implications the new representation of programs causes.

2.1 Weaknesses

As the LFM is an extension of the model presented in section 1.2.1, there are some limitations that are common to both models. These are concerned with false consistency due to missing object identifiers, inappropriate fault modes when using multiple test cases, the inability to locate duplicated faults, the incorrectness of naive hierarchical diagnosis and the problem of misleading diagnoses due to smaller cardinality. These are described in sections 1.1.2–1.1.6 and are not repeated here. It remains to notice that the problems described in sections 1.1.3 and 1.1.6 are even worse in this model as faults in loops are copied for each iteration the loop is executed. Therefore, the assumption that it is sufficient to compute diagnoses of size one or two to obtain reasonable results can no longer be kept.

2.1.1 Decoupled conditionals

In contrast to the original value-based model, the changed behavior of conditional statements allows for better propagation of values backwards through the program, as the selection of a branch of a conditional is no longer dependent on the outcome of the condition, but is determined by the mode of the corresponding component. However, this change has negative effects on the model's performance for some test cases. In particular, it is impossible to locate faults that are embedded in statements representing the initialization part of a loop. For example, consider the following program:

```
1  class LoopInitializer {
2      static void demo() {
3          int k = 2;
4          int i = 0;
5          while(i < k) {
6              i++;
7          }
8          int j = 0;
9          while(j < k) {
10             j++;
11         }
12     }
13 }
```

Assume that the assignment to k in line 3 is faulty and that the value 1 should be assigned. Using observations $i \leftarrow 1$ and $j \leftarrow 1$ for the variables at the end of the of the program a total of 9 diagnoses is derived. Each of the diagnoses indicates a double fault consisting of a statement of the first loop (lines 4–6) and a statement of the second loop (lines 8–10). Thus, the faulty statement is not contained in any diagnosis.

This behavior is caused by the fact that the variable k is not observed (assuming it is a local helper variable) and the loop statements are independent of the value of k when diagnosing. The separation between the statements influencing k and the loop statements is caused by the initial run of the diagnosis system (without observations), where the number of iterations of each loop and the default modes for each conditional statement are derived. In the subsequent diagnosis process the value of k no longer influences the number of iterations or the modes of the conditional statements and therefore the statement is not included in any conflict set.

2.1.2 Expansion depth

When transforming the program including loops into an equivalent loop-free variant, the nesting depth of the conditional statements is derived from the number of iterations of the loop statements during the initial execution of the diagnosis system. For some test cases, this number is less than in the corrected version of the program. Consequently, the fault cannot be explained by simply switching a mode of a conditional statement, but all assignments in the body of the loop that contradict an observation must be included in the diagnoses. Therefore, the minimal diagnosis cardinality increases. As an example consider the following program:

```
class FindPositive {
    static void demo() {
        int [] a = new int[7];
        a[0] = -5;
        a[1] = -3;
        a[2] = -2;
        a[3] =  2;
        a[4] =  3;
        a[5] =  5;
```

```
            a [6] =   7;
            int i = 0;
            int sum = 0;
            int prod = 1;
            int sqsum = 0;
            while (a[i] < 2) {
                sum += a[i];
                prod *= a[i];
                sqsum += a[i]*a[i];
                i++;
            }
        }
    }
```

Assume that the operator in the condition of the loop must be replaced by '$\leq$' to correct the program. In this example, the loop is expanded into three nested conditional statements. Using observations for a, i, sum, $prod$ and $sqsum$, seven diagnoses (with cardinality $\leq$ 3) are derived. Is can be observed that the diagnoses of size two do not include the fault, only the initialization of a and i. In this example a diagnosis of size three includes the faulty statement, together with the initialization and the increment of i. Note that this diagnosis is valid because the index i is crucial to derive values for the observed variables. Therefore, the diagnosis explains the faulty behavior by invalidating the index, which in turn prohibits the computation of values for the variables dependent on the array and thus is consistent with all observations. For other examples, where the computation depends on multiple arrays and index variables the diagnosis cardinality that is required to locate a fault is expected to be larger.

2.1.3 Unknown default mode

For conditional statements and loops (when unrolled more iterations than were executed in the initial run) the default mode of the conditionals and the number of iterations of the loops is unknown. The original loop-free VBM avoids this problem by requiring that the default mode of these components is determined when the branch is executed the first time. However, this is not always possible:

```
class UnknownDefaultMode {
    static void demo() {
        int[] a = new int[3];
        a [0] = 1;
        a [1] = 2;
        a [2] = 3;

        int i = 0;
        int k = 0;

        if (i < 0) {
            if (a[i] < 0) {
                /* do something */
            }
            k = a[i];
        }

        i = i * i;
    }
}
```

Assume the correct behavior of the program is given by the observations $a \leftarrow \{1, 2, 3\}$, $i \leftarrow 2$ and $k \leftarrow 2$. During the initial execution of the diagnosis system the conditional in line 11 is not executed and therefore the default mode of the embedded conditional in line 12 cannot always be derived. For example, consider the mode assignment where the statements in lines 8 and 18 are assumed abnormal and the conditional in line 11 is in *then*-mode. In this case, the value of i is unknown in the body of the conditional and therefore the condition in line 12 cannot be evaluated.

The problem described here is even worse for loops statements. In this case not only the default mode of the conditional statements is unknown, but also the number of conditional statements the loop has to be expanded into.

The current implementation is not able to deal with these issues, as the constraint network requires the structure of the system and the default mode of a component to remain fixed for the whole diagnosis process. As a consequence, the current implementation sets the default mode of components that would otherwise have an undetermined default mode to the mode *#Then*. An approach that can be followed in order to avoid the problem of unknown expansion depth of loops is described in section 2.2.3.

2.1.4 Misleading diagnosis size

The expansion of loops into a hierarchy of conditional statements negatively influences the size of the diagnoses containing the true fault for some examples, if the fault is contained in a loop. This is especially true for programs where a complex loop statement (including many assignment statements and variables) depends on a relatively small initialization part. In these cases, statements belonging to the initialization will often be preferred over statements including the actual fault simply because the diagnosis cardinality is smaller for these statements.[3] As a consequence, it is necessary to consider diagnoses of much larger cardinality to locate the true fault. Note that such spurious diagnoses can be eliminated by heuristic knowledge in some cases, as for example in the `FindPositive` example from section 2.1.2: the assignment to the variable a in line 3 can be excluded from the set of statements to consider, as the variable must hold a reference to an array (the value `null` would cause an exception). This is a similar approach as described in section 1.2.2 for the simple value-based model. Note that the current implementation aims at locating faults at the statement level and cannot locate faults in expressions, as for example in the dimension specification of the new expression.

2.2 Improvements

The following sections describe variants of the model that try to eliminate some of the drawbacks presented in the last section. In particular, an approach that performs diagnosis on a meta-level provides means for both, reducing the search space and smaller diagnosis cardinality. The other approaches deal with the problem of insufficient expansion of loops.

2.2.1 Model variants

Experiments have shown that the model's precision can be improved by special handling of insufficiently expanded loop components. In particular, the insertion of a further conditional statement after the last expanded iteration of the loop can reduce the number of cardinality-minimal diagnoses. The body of the conditional is not derived from the loop, but contains an auxiliary component that always causes a contradiction when executed. In this way, some diagnoses can be omitted where the diagnosis implies more iterations to be executed than are actually present in the model.

[3] The table in the appendix shows that the diagnosis performance of the model decreases with ascending diagnosis size. Except for very small programs, the percentage of the source code that is left for further inspection is generally higher when larger diagnosis cardinality is required.

2.2.2 Meta modeling

In section 1.1.3 the duplication of faulty statements has been identified having a negative effect on the size of the diagnoses including the actual fault. To avoid this problem and reduce the size of diagnoses, only one component representing the mode is generated for each statement contained in a loop. Therefore it is no longer possible to select a mode for a component representing the execution of a statement in a specific iteration. Rather a single component selects the mode for *all* iterations. This makes it possible to speedup the diagnosis process, as the number of different mode assignments is now proportional to the number of statements of the program and not dependent on the number of iterations each loop executed. Furthermore, the size of diagnoses decreases for programs where copies of the same statement belonging to different iterations would otherwise be considered independently.

Conditional statements are treated differently, as their default mode may not be the same for each iteration. This prohibits the representation of the modes of all copies in different iterations through a single component. However, meta components may also be introduced for conditional statements, which set the mode of the conditional to either *#Then*, *#Else* or to the inverse of the default mode. This allows to detect faults where the conditional statement should always execute the Then-branch, the Else-branch or where the condition has been inverted. Note that it is not clear how to determine the default modes of these meta components. One possibility is to select one arbitrarily or to take the mode that the majority of the conditionals consider their default mode.

An alternative procedure is to perform diagnosis using the original model of the loop-free program and then map the diagnoses back to the original program to get the set of statements that are possibly faulty. Although this approach is able to provide more precise results for some programs, first experiments with a set of example programs indicate that the results obtained when using the meta-model are sufficiently accurate. Also, this approach requires much more computational effort, as diagnoses have to be computed up to a much larger cardinality.

2.2.3 Simulate additional iterations

To cope with the problem of insufficient expansion of loops, an additional component is inserted after the expansion of the last iteration that is used to simulate additional iterations (bounded by some number k of iterations). This approach is preferred over the alternative approach – expanding the loop k more times – because the latter approach has some difficulties to determine the number of iterations of embedded loops and to choose the default modes of conditional statements. A negative effect of the chosen representation is the fact that loops that contain faults in the body which in turn cause the loop to be executed less times during the initial run of the program cannot be found, as the body of the loop is assumed to behave correctly.

The behavioral description of the inserted component is equivalent to the behavior of an ordinary while-loop, with the exception that the component only allows $loop(\cdot, k)$-modes and that the condition of the loop is not executed.

2.2.4 Combined models

In section 2.1.1 it is pointed out that the decoupling of the conditional from the variables the condition depends on is not always a good idea, as some diagnoses can no longer be found. To remain the degree of completeness of the original VBM as much as possible, a hierarchical approach should be applied: First the program is analyzed using the simple VBM. If a loop component is included in a diagnosis, it is transformed into a loop-free variant and the diagnosis process is repeated using the LFM, with the focus restricted to the components of the expanded loop. This process continues until the current diagnosis does not contain a loop statement any more. The above procedure ensures that no diagnoses are lost due to the changed behavior of the conditionals. In addition, backtracking has to be performed when no reasonable diagnosis can be found using the LFM. In this case the current diagnosis is discarded and the whole process is repeated using another diagnosis selected from the set of initial diagnoses.

3 Conclusion

This document summarizes the evaluation of several variants of the VBM for Java programs. The weaknesses and strengths of each of the models are pointed out and for some ideas for improvement are presented. It turns out that the simple VBM is unsuited for programs containing loop statements because its abilities to reason backwards are not strong enough for these programming constructs. The LFM avoids these limitation, however, requires an increased amount of computational effort to achieve the goal. Furthermore, some new issues regarding completeness of the approach are raised which remain to be solved in order to get a universally applicable tool. Concluding, the value-based models already provide more accurate results than models based on functional dependencies [3] for most example programs studied so far, but are not suited to be applied alone. Rather, they need to be combined with other strategies to locate bugs more quickly. Especially when the focus of attention has been restricted to a relatively small region of a program, the value-based models provide a quite effective tool to precisely locate faults.

References

[1] David R. Chase, Mark Wegman, and F. Keneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN'90 on Programming Language Design and Implementation*.

[2] James C. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. Technical report, Department of Information and Computer Science, University of Hawaii, 1998.

[3] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Debugging of Java programs using a model-based approach. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis*, Loch Awe, Scotland, 1999.

[4] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. A Value-Based Diagnosis Model for Java Programs. In *Proceedings of the Eleventh International Workshop on Principles of Diagnosis*, Morelia, Mexico, June 2000.

[5] Wolfgang Mayer. Modellbasierte Diagnose von Java Programmen – Entwurf und Implementierung eines wertbasierten Modells. Master's thesis, Technische Universität Wien, 2000.

[6] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-based Debugging or How to Diagnose Programs Automatically. Submitted for Publication.

A Evaluation results

This section contains diagnosis results for the set of test cases used for evaluation. For each of the examples several parameters are listed:

Stm	Number of statements in the program.
C	Number of components in the model.
F	Number of components subject to mode assignments.
D	Diagnoses of minimal cardinality that were derived.
H	Number of cardinality-minimal diagnoses that included the faulty statement.
S	Cardinality of minimal diagnoses that include the faulty statement.
%	Percentage of statements that have to be analyzed until the fault is located, provided that the diagnoses are presented such that diagnoses with smaller cardinality are listed first. For the LFM, the percentage is computed after the diagnoses are mapped back to the original program. Note that this can be improved with heuristic knowledge, such as estimates on a statement's fault probability based on runs of test cases etc.

For the VBM, the columns H and S are omitted (the value is always 1). When numbers are prefixed with '$>$' or '$\geq$' no exact solution has been found due to huge amount of time required for diagnosis. In these cases, the results are based on the diagnoses that were actually computed (for '$\geq$' the fault is already included in the set of diagnoses (but it is unknown how many additional statements are also included), whereas for '$>$' the fault couldn't be located.)

Program	Stm	VBM				LFM						Meta LFM					
		C	F	D	%	C	F	D	H	S	%	C	F	D	H	S	%
BinSearch	27	54	16	6	63	253	63	1	1	2	8	261	43	1	1	2	8
BoundedSum	16	50	14	4	38	76	22	9	3	2	25	79	19	1	0	2	38
BubbleSort	15	20	10	6	93	235	54	1	1	1	7	245	34	7	1	1	47
FindPair	5	6	4	4	100	34	11	5	2	2	100	40	10	1	0	2	80
FindPositive2	17	43	13	3	41	111	27	10	0	3	35	116	20	2	1	1	12
FindPositive3	17	43	13	3	41	111	27	6	0	3	24	116	20	2	1	1	12
Library	24	83	21	6	38	161	56	4	0	2	34	169	36	5	0	2	34
Matrix	71	96	21	21	100	970	199	67	3	1	46	1030	127	37	1	1	52
MaxSearch2	21	58	16	3	38	264	69	1	0	2	14	268	37	2	0	2	19
MultLoops	21	21	12	2	19	92	33	4	0	> 5	> 43	104	27	4	2	3	24
Permutation	24	54	17	14	96	118	32	4	2	1	13	126	29	3	1	1	13
Permutation0	26	62	19	12	69	150	38	1	1	1	4	160	33	1	1	1	4
Permutation1	26	64	19	12	69	130	32	8	0	3	$\geq$96	135	32	8	0	3	100
Permutation2	26	64	19	15	85	152	38	12	1	1	35	162	33	9	1	1	35
Permutation3	24	64	19	12	67	148	38	1	0	3	71	158	33	2	0	3	50
SkipEqual	5	6	4	4	100	36	14	1	1	1	20	42	11	2	1	1	40
Stat	23	59	17	3	39	413	87	1	0	4	$\geq$35	414	42	2	0	4	48
Sum	5	6	4	3	80	34	11	2	1	1	40	40	10	3	1	1	40
SumPowers	21	25	12	8	81	200	61	2	1	1	10	211	36	5	1	1	24
average	22	46	14	7	66	194	48	7	0.9	> 2.0	> 35	204	33	5	0.7	1.7	36
Binomial_f2	76	60	26	9	42							1429	255	24	1	1	32
Hamming_f1	27	53	19	11	70							576	95	9	1	1	33
Huffman_f1	64	70	22	9	80							1656	161	9	0	> 2	> 25
Huffman_f2	64	70	22	6	59	diagnosis system too large						1698	164	12	1	1	19
Intersection	95	119	31	12	84							1971	155	8	1	1	5
MultiSet_f7	97	157	55	8	28							1652	283	1	0	> 2	> 11
Polynom	120	278	64	14	24							1305	189	26	0	> 3	> 13
SearchTree_f4	84	309	41	41	100							1385	140	45	0	> 1	> 54
average	39	74	20	9	65	(194)	(48)	(7)	(0.9)	(> 2.0)	(> 35)	576	77	8	0.6	> 1.6	> 32

Analyzing the evaluation results reveals some interesting properties of the models:

- The performance of the LFM and the Meta-LFM are dependent on the size of diagnoses. Especially, the assumption that it is sufficient to restrict the diagnosis process to single and double faults can no longer be kept for the LFMs, where the diagnosis size is not only dependent on the program's structure but also depends on the number of iterations each loop is executed.

  For larger cardinality it can be observed that the percentage of the program that is left for further inspection is generally higher than for programs with smaller diagnoses. This is due to the increasing number of combinations of statements that prevent large parts of the program to compute values (e.g. unknown values of array index expressions and object identifiers, etc). Also, structural faults and faults in the initialization of variables used in the condition of conditional statements are causes for larger diagnoses cardinalities (see section 2.1).

- The accuracy of the of the loop-free models is better than the simple VBM for most example programs. While the LFM is not well-suited for programs with large sized diagnoses and a huge search space, the Meta-LFM avoids these problems in most cases. However, there exist programs where the loop-free models are less accurate than the simple VBM. These include programs which consist of a set of loops that are dependent on a faulty variable or include conditional statement that have to be inverted for each iteration in order to explain the observations. In these cases, a hierarchic approach using combinations of the VBM and the LFMs (section 2.2.4 seem to reach good solutions more quickly than than each of the models alone. In summary, neither of the models always provides better results than any of the other models unless combinations of models are used.

- The Meta-LFM is able to provide meaningful results for programs where the LFM is too computationally intensive to be applied. Further, the Meta-LFM seems to provide similar accuracy than an extended version of the LFM, where in a subsequent step the diagnoses are mapped back to the program containing loops. However, both of the models generally need to compute diagnoses up to a larger cardinality to locate a fault, whereas for the VBM it is sufficient to compute diagnoses of cardinality one. Also, for the LFMs the probability that the cardinality-minimal diagnoses include the actual fault is lower than for the VBM.

- Comparing the VBM with the Meta-LFM, it is obtained that the Meta-LFM usually provides more precise results than the VBM. However, there are some exceptions (the examples `MultLoops`, `Permutation1` and `Stat`). For these cases the more detailed resolution of the LFMs becomes a drawback. In particular, the loops' independence of the loop variables require that additional loop statements in `MultLoops` and `Permutation` must be included in the diagnoses, as they are dependent on the counters modified in the faulty loops. For `Stat`, the diagnosis size is increased due to the separate modeling of the conditional in the loop's body for each iteration. Together with separate assumptions for each statement of the loop, additional diagnoses including the initialization of variables modified in the body of the loop are computed. Note that these diagnoses are not derivable with the VBM, as a larger diagnosis size is required, comprising all initialization statements.

- Comparing the LFM with the Meta-LFM, it is obtained that the results for the two models coincide on most examples, the LFM being slightly more accurate than the Meta-LFM. Again, there are some programs where the Meta-LFM provides better results (`FindPair`, `FindPositive`, `MultLoops` and `Permutation3`). For `Permutation3`, the better results for the Meta-VBM are caused by the smaller diagnosis cardinality, preventing the algorithm from computing supersets. Although the faulty statement is not contained in the small diagnoses, the inclusion of unnecessary statements is prevented. For the other examples, the inability of the LFM to model additional iterations of loops is responsible for the higher percentage of statements. In these cases the diagnoses consist of combinations of statements in the loop's body and their initialization part. As the bodies of the loops are fault-free, the Meta-LFM is able to simulate the additional iterations by the auxiliary loop component (see section 2.2.3) and thus to avoid the diagnoses derived by the LFM.