Diplomarbeit

Modellbasierte Diagnose von Java-Programmen Entwurf und Implementierung eines wertbasierten Modells

ausgeführt am Institut für
Informationssysteme
Abteilung für Datenbanken und Expertensysteme
der Technischen Universität Wien

unter Anleitung von Ao.Univ.Prof. Markus Stumptner
und Univ.Ass. Dr. Franz Wotawa
als verantwortlich mitwirkendem Universitätsassistenten

durch
Wolfgang Mayer
A-1140 Wien, Abtsbergengasse 15

Wien, September 2000

Kurzfassung

Diese Diplomarbeit hat zum Ziel, ein Modell für Java-Programme zu entwickeln, welches zum Debugging mittels modellbasierter Diagnose eingesetzt werden kann. Dabei werden insbesondere die objektorientierten Sprachelemente von Java berücksichtigt. Die in der Arbeit entwickelten Modelle sind auf eine Untermenge der Java-Sprache beschränkt, da die Abarbeitungsreihenfolge der Anweisungen bereits während der Bildung der Modelle bekannt sein muß.

Die vorliegende Arbeit betrachtet zunächst ein Modell, welches alle Programmelemente als Komponenten und die benötigten bzw. veränderten Variablen als Verbindungen zwischen den Komponenten darstellt. Es erscheint daher nur für sehr einfache Programme anwendbar. Weniger einfache Programme könnten mit Erweiterungen dieses Modells behandelt werden, dabei erreichen die Modelle jedoch eine überaus hohe Komplexität.

Aufbauend auf dem ersten Modell wird ein weiteres Modell entwickelt, das im Aufbau dem ersten Modell gleicht, die Instanzvariablen von Objekten im Gegensatz zum ersten Modell jedoch gesondert betrachtet. Dadurch wird es möglich, auch rekursive Methodenaufrufe, Arrays und Strings zu modellieren. Weiters wird die hohe Komplexität des ersten Modells vermieden. Ein weiterer Abschnitt dieser Arbeit widmet sich der Implementierung des zweiten Modells in Form eines Constraint-Netzwerkes.

Um die Leistungsfähigkeit des zweiten Modells zu demonstrieren, wird das Debugging mit dem Modell anhand einiger Beispiele veranschaulicht. Die Beispiele zeigen, daß durch Einsatz des Modells zur Diagnose von Java-Programmen vielversprechende Ergebnisse erzielt werden können. Dies gilt auch für objektorientierte Sprachelemente, wie polymorphe Methodenaufrufe oder `new`-Ausdrücke. Auch können in vielen Fällen bessere Ergebnisse erzielt werden als mit nur auf funktionalen Abhängigkeiten basierenden Modellen.

Abstract

Locating faults in programs is an important task within the process of software development, because modern software tends to contain bugs that have to be identified and corrected as soon as possible. Normally, this task is rather difficult, considering size and complexity of recent programs. In addition to this, the programmer trying to locate faults is in many cases not the author of the program. This fact makes the task even more difficult because relations between the specification of the desired behavior and the actual implementation are not given explicitly. They often only exist as a mental model of the author. The first task of the person trying to locate the bug is to understand the program and thus to reconstruct the mental model in order to be able to search for program parts causing the misbehavior. To overcome these difficulties, intelligent debugging tools are needed to assist the programmer in focusing on relevant parts of programs to be examined.

Eliminating those parts of the program that do not contribute to the observed misbehavior can be done in several ways. A lot of different approaches exist to accomplish this task, using various algorithms, modeling paradigms and language-specific knowledge. A common approach is to use functional dependencies between variables and statements to isolate those parts of the program that possibly could influence the computed value of a variable at a particular point of execution. These parts of the program must then be further inspected to determine any incorrect statement. As these methods usually do not require much computational effort and need relatively little information about the expected behavior of the program, they are frequently applied as a pre-processing step to a more detailed analysis. Moreover, these methods do not require specific knowledge about the software under review or the programming language used and thus can be applied to a wide range of programs. One negative aspect becomes obvious in pathological cases where hardly any parts of the program can be omitted and the whole source needs to be analyzed. Another drawback caused by the abstract representation of the program results in the fact that hardly any hints to the type of the fault – and thus to its correction – can be stated.

In this paper we try to overcome the limitations of models being exclusively based on functional dependencies by exactly modeling the execution semantics of the software. Model-based diagnosis is applied to the model in order to detect possible erroneous parts of a Java program. The model described within this paper does not only reflect the functional dependencies between the variables and statements, but reproduces computations of all variables' values ever computed during program execution. To achieve this, the model represents the exact execution-semantics of the program, i.e. given the same input values, not only the execution of the program but also its model computes the same correct results at all points of execution. The value-based model presented in this paper produces better results by considering concrete values rather than by modeling dependencies on their own. In some cases the obtained diagnoses even provide suggestions for correcting the source code.

The described model can be used to build representations of Java programs. Java was chosen as the language to be analyzed because it is an imperative, object-oriented programming language with a rather simple structure and thus is easy to analyze. Furthermore, Java includes

the main features of object-oriented languages such as inheritance and polymorphism. To reduce complexity, the model developed in this paper is restricted to a significant subset of the language. Especially, the execution sequence of the statements and expressions to be modeled must be known in advance, regardless of the concrete input values. As a consequence, threads, exceptions and logical expressions leading to side-effects will not be considered.

As a first step in developing a model that is applicable to model-based diagnosis, the program under consideration has to be converted into a set of components and connections between them. Our model consists of two major parts: (1) the structural representation of the program by means of components and connections and (2) the behavior-description of each component. The representation of the program structure through components and connections between them can be generated automatically from the given source code. This is done separately for each method of the program. The description of the generated component behavior is not derived from the program itself, but is induced by the semantics of Java. Therefore, it is independent of the given program. This approach allows a separate and independent construction of the two parts of the model.

When converting a Java program into a model, the key features are methods and variables. For each method a model of the statements inside the method's body is computed. Additionally, for each statement and its sub-statements and sub-expressions, a component representing the statement or expression is generated. The variables of the program are mapped to connections between the components. Generally, the variables of a Java program can be classified into three categories: (a) local and parameter variables, (b) class variables and (c) instance variables. Local and parameter variables are only visible within a method and cannot be accessed by statements outside this method. Class variables are instantiated once for each class and represent a kind of global variable. Instance variables are created for each instance of the corresponding class and can be accessed through a reference to that instance. Modeling the variables proceeds as follows: during an initialization phase, a connection is created for every variable that can be accessed within the method to be modeled. Whenever the variable is accessed in an expression, the component representing the access is connected to the current connection associated with this variable. Each time a variable is modified, a new connection is created which then replaces the existing one. The value of the created connection is determined by the component representing the modification of the variable. The newly created connection is used in subsequent expressions whenever the variable is accessed until the variable is modified again.

When creating connections for the variables, an attempt could be to represent all variables of the program as separate connections, regardless of their type. Although the idea sounds simple, the resulting model cannot represent object-oriented features, such as aggregation, creation of class instances or dynamic data structures very well. In addition to this, the model has to be supported by an aliasing analysis to correctly represent the semantics of the Java language. This adds enormous complexity to the model and thus makes it difficult to compute.

To avoid these problems, a slightly different approach is used. Contrary to the procedure described above, distinction is made between local, parameter and class variables versus instance variables. The former variables are represented by separate connections, exactly as within the original model. Representation of the latter is modified in such a way that all instance variables, i.e. the status of all objects, are represented in a compound data structure called *object space*. The object space can be seen as dictionary containing associations between the instance variables of the objects and their corresponding values at a particular point of program execution. Accessing these variables is done through a unique identifier representing the object which the variable is part of, coupled with the name of the variable. Modeling of statements that access and modify instance variables is done as described above, but whenever an instance variable is accessed or modified, connections for the object space are used and created, within the model instead of establishing separate connections for each variable. In this way the problem

of dynamic data structures and class instance creation expressions can be avoided. Furthermore, there is no need to apply aliasing analysis because the connections representing local, parameter or class variables being reference-type variables contain identifiers of the addressed objects and thus can be modified separately.

To complete the description of the structural modeling process, the conversion of statements and expressions of a Java program into a component-based model is described below:

- When converting *assignment* statements, it has to be distinguished between assignments to parameter, local or class variables and instance variables. Assignments to the former variables are modeled as a component with one input and one output port. The input port corresponds to the evaluation of the expression on the right side of the assignment and is connected to the output port `result` of the model representing the expression. The output port corresponds to the target variable of the assignment. Assignments to instance variables are represented as a component with two input ports and one output port. The input port `value` corresponds to the evaluation of the expression on the right side of the assignment and is treated the same way as assignments to non-instance-variables. The second input port `objectspace_in` is associated with the object space being valid before execution of the assignment. The output port `objectspace_out` corresponds to the object space after the assignment.

- *Conditionals* are represented as components with several input and output ports. One input port `cond` is associated with the evaluation of the conditional's condition and connected to the output port `result` of the model representing the expression. All remaining input ports correspond to all local, parameter and class variables as well as object spaces that are modified within a branch. For each variable `v` modified inside one of the branches, three ports named `then_v`, `else_v` and `out_v` are created. The ports `then_v` and `else_v` are associated with the value of each variable computed by the then- and else-branches of the conditional and connected to the corresponding connections of the models of the branches. The `out_v` ports correspond to the values of the variables after execution of the entire conditional and are associated with new connections for the variables. If modifications of instance variables occur in one of the branches, ports for the object spaces are added in a similar way.

- *While* statements are mapped to components with several input and output ports. The input ports correspond to the variables (and the object space) used by the condition's model and by the body of the loop. The output ports represent the modified variables (and the object space) in a similar way. Eventually, additional input ports have to be added to ensure that for each modified variable also an input port for this variable exists. This is needed to determine the values of the output connections in case the body is never executed (i.e. the first evaluation of the condition results in *false*). The component is constructed in a hierarchical way with separate models for the loop's condition and body.

- *Return* statements are modeled as assignments to an auxiliary variable `return`. This is possible as the restricted Java language ensures that *return* statements are always the last statements within a method.

- *Method calls* are modeled as hierarchical components with several input and output ports. The input and output ports represent the variables (and object spaces) used and modified with respect to the called method. The model of the method call can be determined from the model M of the called method by substituting all ports associated with the parameter and class variables as well as object spaces in M by ports of the component representing the call. If the method call is dynamic, i.e. the actual method to be invoked is determined

at runtime through the dynamic type of the object the method is invoked on, additional input and output ports may be necessary. Firstly, an input port `object` that corresponds to the object identifier of the object the method is called upon, has to be added. This is necessary to determine the actual type of the object and thus the method to be called. Secondly, the set of input and output ports of the component is determined by computing the union of the model's input and output ports of *all* methods that might have been called. These possibly called methods include all overrides of the called method. Just as within the modeling process of the *while* statements, there must exist input ports corresponding to all variables that are not modified by *all* models of the possibly called methods. This is necessary to determine the values of these variables whenever a method is called that does not modify them.

- *Expressions* are modeled as components with multiple ports. Each component includes at least an output port `result` that is associated with the evaluation of the expression. There are to be distinguished five different cases:
  - *Constants* are mapped to components with only one output port that corresponds to the constant.
  - When representing *variables*, we have to distinguish between two cases: uses of local, parameter or class variables are represented as components with one input and one output port, where the input port corresponds to the accessed variable and connected to the connection associated with the variable and the output port corresponds to the variable's value. Accesses of instance variables are represented as components with two input ports and one single output port. The input port `object` corresponds to the object identifier of the object containing the accessed variable and is connected to the connection of the variable through which access takes place. If no variable is indicated, the auxiliary variable `this`, which is associated with the object identifier of the object the method operates upon, is selected. The second input port `objectspace_in` corresponds to the object space right before the access takes place. The output port corresponds to the value of the variable.
  - Unary and binary *operators* are represented as components with one or two input ports and only one output port. The input port(s) correspond(s) to the evaluation of the operand(s) and is (are) connected to the output port `result` of the corresponding component(s). The output port corresponds to the evaluation of the operator.
  - *Function calls* are modeled in a similar way as method calls. In addition, the output port `return` of the component representing the call, which is associated with the return value of the method, is renamed to `result`.
  - *Class instance creation expressions* are modeled similarly to function calls, but the input port `object` is omitted as its value is created by the component itself. Furthermore, input and output ports representing the object spaces before and after the expression are required, because new instance variables are created.

To be able to apply the model to model-based diagnosis, a description of the correct (and in some cases the faulty) behavior of the component is needed. In this paper, the behavior is specified in first order logic. Here, only an informal description of the behavior is given:

- The behavior of components representing *assignments* to local, parameter or class variables is considered correct if both the values of the input port and the output port are equal. Assignments to instance variables are considered correct if the object space's value after the assignment is equal to the value of the object space before the assignment, where the

value of the assigned variable has been replaced by the value of the input port representing the expression.

- The correct behavior of *conditionals* is specified such that if the condition is *true*, the values of the ports related to the then-branch are propagated to the output ports. If the condition evaluates to *false*, the ports of the else-branch are used instead.

- Components representing *while* statements are considered correct if the model of the loop's body is executed to compute new values as long as the model of the loop's condition evaluates to *true*. The initial values of the input ports of the models for the condition and the body are taken from the input ports of the component representing the loop. The input values for the models used during the subsequent iterations are taken from the output ports of the model of the body that were computed in the previous iteration (or from the input ports of the component representing the loop, if the variable is not modified inside the body). The values of the output ports are taken from the output ports of the body's model of the loop after the last iteration. If the body is never executed, the values of the output ports are taken from the corresponding input ports.

  In addition, there is specified a fault model $loop(C, max)$ that sets the number of iterations of the loop represented by the component C to max, regardless of the outcome of the evaluation of the condition.

- The behavior of *return* statements is equivalent to that of assignment statements to local variables.

- The behavior of *method calls* is obtained by substituting the input and output ports of the called method's model with the input and output ports of the component representing the call. If the method call is dynamic, the model of the called method must first be selected based on the type information that is attached to the object identifier. As the model for the called method must be available when modeling the call, this approach cannot be applied to recursive method calls. These have to be modeled by other (e.g. evaluator-based) representations.

- *Expressions* are subdivided according to the structural part of the model:

  - The behavior of components representing *constants* is correct if the value of the output port is identical with the specified constant.
  - In case of *variable accesses*, separate behavior for the two possibilities has to be specified. Components representing accesses to local, parameter and class variables are considered correct if the output port's value is equal to the value of the input port. In case of accesses to instance variables, the output port's value has to be equal to the value of the variable as stored in the object space for the object with the identifier on the designated port `object`.
  - Components representing *operators* show correct behavior if the value of the output port is equal to the value obtained by applying the operator to its operand(s).
  - The behavior of *function calls* is specified analogously to the behavior of method calls.
  - The behavior of *class instance creation expressions* is specified in a similar way as the behavior of dynamic calls of functions, except that the object identifier is not passed via an input port but computed by the component itself. Further, preceding the execution of the model of the invoked constructor, new instance variables of the object are created and inserted into the object space.

The description of the model elaborated here can be further extended to cope with modeling arrays and strings. Also, the object space can be subdivided to represent each declaration of an instance variable in a separate object space. This reduces the dependencies between the components. Better results can be obtained during diagnosis.

Debugging of Java programs with the model is done by building the model of the program and then selecting a method to debug. The arguments and the expected behavior are specified as observations of connections. Then the model-based diagnosis algorithm is applied to the system and the obtained diagnoses are mapped back to elements of the program's source code. In this way functional faults (i.e. wrong constants, wrong operators, wrong conditions, etc.) can be identified. The model is unable to detect structural faults such as missing statements, wrong order of statements, assignments to wrong variables or non-terminating programs.

In contrast to those approaches that only consider functional dependencies between variables and statements, this model produces better diagnoses as it incorporates more information to exclude impossible diagnoses. Furthermore, the occurrence of a fault mode $loop(C, max)$ for some component C in a diagnosis provides more information about the faults of the program as it suggests possible corrections (modifying the condition of the loop represented by C so that the loop is executed max times).

The model described here was implemented as a constraint network where the components and their behavior are represented by constraints between constraint variables, which constitute the connections of the model. Due to this representation, care has to be taken when specifying observations of instance variables. In some cases these observations have to be represented as constraints as well in order to correctly reflect the semantics of the observations.

Promising experiments with the implementation described above led us to the conclusion that the model presented in this paper is well suited to detecting functional faults in small to mid-sized programs. This includes programs with object-oriented features such as polymorphic method calls and dynamic data structures. Topics for further research are the development of *additional fault models* for components to represent different faults in software and the *diagnosis of hierarchical components* such as loops and method calls. In these cases the models of the components' behavior again constitute diagnosis problems where the observations are derived from the values of the components' input and output ports. Another feature that could be incorporated into the model presented here would be the identification of *replacements*. This would enable the model to give concrete suggestions to the user about how to correct the program's source code. In addition the use of *multiple test cases* could reduce the number of diagnoses and support the programmer to focus on the relevant parts of the program. Finally, the development of an *intuitive user-interface* for a model-based debugger that supports the user in specifying observations and inspecting diagnoses is an open topic for further research.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Die Suche nach Fehlern in Programmen stellt einen wichtigen Teilbereich der praktischen Informatik dar, da aufgrund der Komplexität und Größe moderner Programme diese meist Fehler aufweisen. Das Finden und Beseitigen der Fehler stellt eine aufwendige Prozedur dar, welche möglichst effizient durchgeführt werden sollte. Um dies zu erleichtern, sind intelligente Debugger von Vorteil, welche die Aufmerksamkeit des Benutzers auf eventuelle Fehler im Programm dirigieren können. Dadurch kann der von Benutzer benötigte Aufwand bei der Fehlersuche beträchtlich reduziert werden.

Die vorliegende Arbeit beschäftigt sich mit der Suche nach Fehlern in Java-Programmen mittels modellbasierter Diagnose. Ziel dieser Arbeit ist es, ein wertbasiertes Modell für Java-Programme zu entwickeln, d.h. das Modell bildet die Semantik des Programms nach und erkennt auf diese Weise Abweichungen zwischen den vom Programm berechneten Werten und den vom Benutzer erwarteten Werten. Aus den Abweichungen werden durch den Diagnosealgorithmus möglicherweise fehlerhafte Programmteile berechnet. Durch den Einsatz von wertbasierten Modellen können in vielen Fällen bessere Ergebnisse erzielt werden, als mit Modellen, welche nur die Abhängigkeiten zwischen den einzelnen Anweisungen und Methoden berücksichtigen [MSW99]. Im Zusammenspiel mit einer geeigneten Benutzeroberfläche ermöglicht dies dem Benutzer, seine Aufmerksamkeit zielgerichtet auf die relevanten Teilbereiche des Programms zu richten.

Im folgenden wird die Vorgehensweise beim Software-Debugging mittels modellbasierter Diagnose kurz vorgestellt. Abbildung 1.1 zeigt die Vorgehensweise beim Debugging von Programmen mittels modellbasierter Diagnose. Um modellbasierte Diagnose auf Programme anwenden zu können, muß ein Modell für das Programm erstellt werden. Das Modell beschreibt den Aufbau des gegebenen Programms in Form von Komponenten und deren Verhalten und repräsentiert das Verhalten des Programms. Die Menge der Komponenten-Typen und deren Verhaltensbeschreibungen wird aus der Spezifikation der Programmiersprache, in welcher das zu debuggende Programm verfaßt ist, abgeleitet. Die Struktur des Modells, d.h. die Komponenten und deren Verbindungen untereinander, werden aus dem gegebenen Programm berechnet. Zusätzlich wird eine Menge von Beobachtungen benötigt, welche dem durch das Modell berechneten Verhalten des Programms gegenübergestellt wird. Die Beobachtungen entsprechen dem erwarteten Verhalten des Programms. Werden Abweichungen festgestellt, können daraus Mengen von Komponenten berechnet werden, welche nicht entsprechend ihrem spezifizierten Verhalten arbeiten und damit als fehlerhaft anzusehen sind. Die auf diese Weise erhaltenen Mengen von Komponenten repräsentieren jeweils eine Menge von Programmelementen, welche als mögliche Fehlerursache angesehen werden können. Auf diese Weise können dem Benutzer Hinweise auf möglicherweise fehlerhafte Programmteile und eventuell auch Vorschläge zur Korrektur des Programms gegeben werden. Dieser Prozeß kann in iterativer Weise durchgeführt werden, wobei nach und nach Beobachtungen hinzugefügt werden, bis eine einzelne Fehlerursache isoliert ist, welche sämtliche

Abweichungen zwischen den berechneten und den erwarteten Werten erklärt.

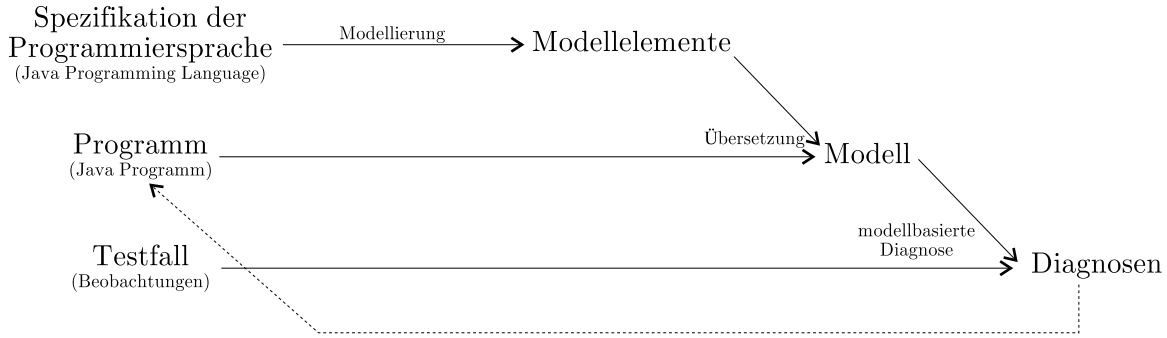

Abbildung 1.1: Debugging mit modellbasierter Diagnose

Das soeben beschriebene Verfahren ist weitgehend von der Programmiersprache unabhängig und kann sowohl für deklarative Sprachen als auch für imperative Sprachen eingesetzt werden. In der vorliegenden Arbeit wurde die Sprache Java gewählt, da diese eine moderne imperative Programmiersprache darstellt, welche alle wichtigen Merkmale einer objektorientierten Sprache aufweist. Hierzu zählen etwa Kapselung von Daten, Vererbung zwischen Klassen und Polymorphismus von Methoden. Darüber hinaus ist die Sprache aufgrund ihrer einfachen Struktur und Semantik relativ einfach zu modellieren.

Die Arbeit entwickelt und analysiert zwei Modelle für Java-Programme, mit deren Hilfe Methoden eines Programms diagnostiziert werden können. Die Modelle werden in der Weise konstruiert, daß das Finden von funktionalen Fehlern möglich ist. Dies umfaßt etwa falsche Operatoren oder Konstanten in einem Programm oder eine falsche Anzahl von Iterationen in Schleifen. Die Modelle sind ungeeignet für Programme, welche fehlende Anweisungen bzw. Variablen aufweisen oder bei welchen die Abarbeitungsreihenfolge der Anweisungen nicht zur Übersetzungszeit angegeben werden kann. Insbesondere werden Programme mit mehreren Threads ausgeschlossen. Weiters können keine Programme diagnostiziert werden, welche Nichtterminationen aufweisen.

Im folgenden Abschnitt werden einige Ansätze zum Software-Debugging aufgelistet. Kapitel 2 betrachtet die geschichtliche Entwicklung der Sprache Java. Weiters wird die Sprache Java der Sprache C++ gegenübergestellt und einige häufige Fehlerursachen in Java-Programmen identifiziert. Kapitel 3 gibt einen Überblick über die Funktionsweise und die grundlegenden Definitionen und Algorithmen der modellbasierten Diagnose nach Reiter [Rei87]. Kapitel 4 stellt den Hauptteil dieser Arbeit dar und entwickelt zwei Modelle für Java-Programme. Das erste Modell repräsentiert sämtliche Variablen und Objekte des Programms als Verbindungen, woraus sich eine Vielzahl von Einschränkungen ergibt. Das zweite Modell umgeht die Limitationen des ersten Modells indem der Zustand von Objekten und deren Instanzvariablen in sog. Objekträumen zusammengefaßt werden. Der Zugriff auf Objekte und Instanzvariablen erfolgt durch den Objekten zugeordnete Identifier. Das Kapitel endet mit einer Beschreibung von Erweiterungen des zweiten Modells, welche die Modellierung von Arrays und Strings bzw. die Verbesserung der Qualität der Diagnosen erlauben. In Kapitel 5 wird die Implementierung des zweiten Modells aus Kapitel 4 in Form eines Constraint-Netzwerkes behandelt. Weiters wird das Vorgehen beim Spezifizieren von Beobachtungen betrachtet und eine Benutzeroberfläche, welche das Debugging von Methoden erlaubt, vorgestellt. Kapitel 6 veranschaulicht die Leistungsfähigkeit des Modells und dessen Implementierung anhand einiger Beispiele. Kapitel 7 enthält eine Zusammenfassung der wichtigsten Merkmale der vorgestellten Modelle und betrachtet einige mögliche Erweiterungen.

1.1 Klassifikation von Verfahren und verwandte Arbeiten

Die Verfahren zur Identifizierung von Fehlern in Programmen können grob in drei Bereiche klassifiziert werden [Duc93]:

- Bei der *Verifikation* von Programmen wird das gegebene Programm mit einer formalen Spezifikation des gewünschten Verhaltens verglichen. Mit dieser Methode können alle Arten von Fehlern entdeckt werden, da eine vollständige Spezifikation des gewünschten Verhaltens vorhanden ist. Nachteil dieses Verfahrens ist der hohe Berechnungsaufwand, welcher dieses Verfahren für die meisten Programme unbrauchbar macht. Weiters ist das Erstellen einer formalen Spezifikation in vielen Fällen mit extrem hohem Aufwand verbunden und zudem ein in hohem Maße fehleranfälliges Unterfangen.

- *Checking* von Programmen besteht darin, im Quelltext des Programms nach für die Sprache typischen Fehlern zu suchen und bei Entdecken eine Warnung auszugeben. Das Wissen um die zu entdeckenden Fehler muß in geeigneter Form (z.B. Regeln) angegeben werden. Nachteil dieser Methode ist, daß viele Fehler auf diese Weise nicht erkannt werden können. Darüber hinaus kann keine Aussage über die Vollständigkeit der Regelmenge getroffen werden, da immer zusätzliche Regeln hinzugefügt werden können. Auch kann in pathologischen Fällen die Anzahl der ausgegebenen Warnungen die Anzahl der Anweisungen des Programms übersteigen. Ein Vorteil dieser Methode ist, daß bei der Entdeckung eines Fehlers aufgrund einer Regel zugleich auch ein Hinweis auf die Ursache des Fehlers geliefert wird.

- *Filtering* versucht, durch Ausschließen von als korrekt angesehenen Teilen des Programms die möglichen Fehlerursachen einzugrenzen. Diese Methode benötigt wenig Information über das gewünschte Verhalten sowie das Programm und kann daher leicht eingesetzt werden. Auch ist der Berechnungsaufwand relativ niedrig. Daraus folgt, daß diese Methode als Vorstufe einer genaueren Analyse eingesetzt werden sollte, um den Umfang des zu analysierenden Programms zu vermindern.

Eine Übersicht über bestehende Implementierungen der einzelnen Verfahren ist in [Duc93] zu finden.

Eine Methode, den Umfang des zu untersuchenden Programms zu verringern, ist Slicing [Wei81, Wei82]. Die Methode stellt eine Variante eines Filtering-Verfahrens dar. Die von diesem Verfahren benötigten Informationen sind eine Programmposition zusammen mit einer Variablen, welche einen falschen Wert aufweist. Das Verfahren bestimmt daraufhin – basierend auf den Abhängigkeiten zwischen den Anweisungen – alle jene Programmelemente, welche möglicherweise an der Berechnung des falschen Wertes beteiligt sind. Alle anderen Programmteile sind irrelevant und können ignoriert werden.

Die im folgenden vorgestellten, auf modellbasierter Diagnose aufbauenden Verfahren, können ebenfalls als Variante der Filtering-Verfahren angesehen werden, da auch bei diesen Verfahren die nicht in einer Diagnose enthaltenen Programmteile als korrekt angesehen werden. Zusätzlich dazu werden die Informationen über die Korrektheit der einzelnen Werte berücksichtigt. Dadurch kann eine Verbesserung der Ergebnisse z.B. gegenüber Slicing-Verfahren erzielt werden.

Das in [Wot96] vorgestellte Verfahren hat zum Ziel, Fehler in VHDL[1]-Programmen zu identifizieren. Das Verfahren basiert auf modellbasierter Diagnose. Es werden drei Modelle mit unterschiedlichem Abstraktionsgrad und Leistungsfähigkeit, sowie Kombinationen derselben untersucht. Die Modelle beruhen vorwiegend auf funktionalen Abhängigkeiten, es können jedoch auch Reparaturvorschläge berechnet werden.

[1] Very High Speed Integrated Circuit Hardware Description Language

Einen weiteren Ansatz stellt das in [Paw96] implementierte Verfahren dar. Die Arbeit untersucht das Debugging von C-Programmen mittels modellbasierter Diagnose. Es wird die Anwendbarkeit der modellbasierten Diagnose auf imperative Programmiersprachen gezeigt, dabei bleiben jedoch manche Probleme ungelöst (z.B. die Diagnose von Schleifen).

In [MSW99] wird ein auf funktionalen Abhängigkeiten basierendes Modell vorgestellt, welches für Java-Programme geeignet ist. Das Verfahren inkludiert u.a. Modelle für Methodenaufrufe und Schleifen.

Ein auf modellbasierter Diagnose beruhendes Verfahren, welches mit Korrekturvorschlägen arbeitet, wird in [SW99] behandelt. Das Modell ist für baumartig strukturierte Systeme geeignet und liefert zusätzlich zu der Information, welche Anweisungen möglicherweise fehlerhaft sind, Vorschläge zur Korrektur des Programms. Weiters wird die Reduktion von Diagnosen bei Vorhandensein von mehreren Testfällen behandelt.

Der in [Jac95] gewählte Ansatz beruht nicht auf modellbasierter Diagnose, sondern auf Zusicherungen über Abhängigkeiten zwischen Parametern und berechnetem Ergebnis einer Methode einer Algol-ähnlichen Sprache. Dieses Verfahren ist unabhängig von den berechneten Werten und erkennt nur fehlende Abhängigkeiten der Implementierung des Programms gegenüber den angegebenen Abhängigkeiten. Auf diese Weise können etwa fehlende Anweisungen erkannt werden.

[BH95] verwenden Wahrscheinlichkeitsmaße, um die Relevanz von Diagnosen zu bewerten. In einem ersten Schritt werden – ausgehend vom Auftreten des zu eliminierenden Fehlers – alle möglichen Pfade im Programm bestimmt, welche zu der den Fehler auslösenden Anweisung führen. Anschließend werden die Pfade mittels eines Bayes'schen Netzes bewertet und so der wahrscheinlichste Pfad ermittelt. Im Gegensatz zu den zuvor behandelten Verfahren ist dieses Verfahren in hohem Maße von der Qualität des Bayes'schen Netzes abhängig und kann daher nur zur Wartung von Programmen benutzt werden, für welche ausreichende statistische Daten über Fehlerursachen bekannt sind, um das Netz zu initialisieren.

Kapitel 2

Java, C++ und Fehler in Programmen

Java [GJS96, SBL96] ist eine objektorientierte, imperative Programmiersprache, die in den letzten Jahren aufgrund ihrer Plattformunabhängigkeit und ihrem Einsatz im Internet in vielen Bereichen der Informatik große Beachtung gefunden hat. Ihre Syntax ist jener von C++ [Str92] sehr ähnlich. Es wurden allerdings einige Sprachelemente von C++ weggelassen, um die Benutzung der Sprache zu vereinfachen und die häufigsten Fehlerquellen zu beseitigen, mit dem Ziel, die Sprache damit insgesamt sicherer zu machen. Ein einfaches Beispiel eines Java-Programms ist in Abbildung 2.1 dargestellt.

```
1  public static void main(String args[]) {
2      int numbers[] = new int[10];
3      for (int i = 0; i < 10; i++) {
4          int number;
5          // read number
6          int j = i;
7          while ((j > 0) && (numbers[j-1] > number)) {
8              numbers[j] = numbers[j-1];
9              j = j - 1;
10         }
11         numbers[j] = number;
12     }
13     // print numbers
14 }
```

Abbildung 2.1: Java-Programm zur Illustration häufiger Fehlerursachen

An dieser Stelle einige Worte zur geschichtlichen Entwicklung von Java. Java wurde ursprünglich von James Gosling und Patrick Naughton bei Sun Microsystems entwickelt, um Applikationen für elektronische Geräte des täglichen Lebens zu entwickeln. Ziel war es, eine Umgebung zu schaffen, die es ermöglichen sollte, solche Applikationen für eine Vielzahl von Geräten – wie etwa Videorecorder, Alarmanlagen oder Mikrowellenherde – plattformübergreifend zu entwickeln und ablaufen zu lassen. Darüber hinaus sollte die Sprache leicht zu erlernen und anzuwenden sein. Die ursprünglich von Gosling und Naughton entwickelte Sprache *Oak* wurde anfänglich nur für Applikationen im Bereich Television und Videosysteme eingesetzt, ein durchschlagender Erfolg blieb jedoch aus. 1994 erhielt die Sprache Oak ihren heutigen Namen: *Java*. Auch wurden der Sprache eine mächtigere Benutzerschnittstelle und Funktionen für Netzwerk- und Internetkommunikation hinzugefügt, was Java für die Entwicklung verteilter Ap-

plikationen geradezu prädestinierte. Ihren endgültigen Durchbruch erzielte die Sprache, als Sun Microsystems im Internet eine einfache, kostenlose Entwicklungsumgebung zur freien Verfügung bereitstellte und die ersten Implementierungen einer Java-Laufzeitumgebung für WWW-Browser verfügbar wurden.

Um die in Java erstellten Programme auf einer möglichst großen Anzahl von Geräten und Betriebssystemen ablaufen lassen zu können, werden Java-Programme beim Übersetzen nicht in architekturspezifischen Maschinencode, sondern in einen plattformunabhängigen Zwischencode, den sog. *Bytecode*, umgewandelt. Das Ausführen von solchen Programmen wird von sog. *virtuellen Maschinen (VM)* bewerkstelligt, die den Bytecode interpretieren. Der Aufbau des Bytecodes und die Funktionsweise der VM sind exakt spezifiziert [LY97], so daß Java-Programme auf allen VM in gleicher Weise und unabhängig von der darunterliegenden Hardware ausgeführt werden. Dadurch wird die Portabilität erhöht und einige in C++ oftmals auftretende Probleme – wie z.B. unterschiedliche Länge eines Maschinenwortes, Darstellung von Zahlenformaten und Adreßräumen, etc. – umgangen. Ein weiterer Vorteil von Java als interpretierte Sprache ist, daß Programme auch in einer Art „sicheren Umgebung“ („*Sandbox*“) zum Ablauf gebracht werden können, wobei sämtliche Zugriffe auf Systemressourcen vom Benutzer beschränkt werden können. Durch diese Maßnahmen soll sichergestellt werden, daß ein Programm keinen Schaden auf dem System anrichten kann, wie z.B. das Löschen von Daten oder Ausspionieren von Passwörtern. Dies ist insbesondere bei Anwendungen in Netzwerken von großer Bedeutung, wo oftmals Programmcode von unbekannten Quellen aus dem Netzwerk bezogen und zum Ablauf gebracht wird.

Im folgenden Abschnitt werden Unterschiede und Erweiterungen von Java gegenüber C++ behandelt, wobei speziell auf mögliche Fehler in Programmen eingegangen wird.

2.1 Unterschiede zur Sprache C++

Ihre Ähnlichkeit zur Sprache C++ verdankt Java dem Umstand, daß C++ zum Zeitpunkt der Entwicklung von Java bereits längere Zeit im Einsatz war und die Entwickler daher mit deren Syntax und Semantik vertraut waren. Um das Erlernen zu erleichtern, wurde die Syntax von Java so weit als möglich an jene von C++ angelehnt, die Semantik wurde aber an einigen Stellen stark vereinfacht. Auch sollte die Sprache möglichst robust sein und daher nur bereits bewährte Technologien und Sprachelemente in die Sprache aufgenommen werden. Aus diesem Grund wurden jene Sprachelemente von C++ weggelassen, die als häufige Ursache von Fehlern identifiziert werden konnten. Im folgenden werden einige dieser Konstrukte näher betrachtet.

An erster Stelle ist die `#define`-Anweisung des C++-Präprozessors zu nennen, da ihr Einsatz Programme in vielen Fällen schwer lesbar macht und durch ihre unterschiedliche Semantik zum Rest der Sprache oft eine Ursache von Fehlern oder nicht portablen Konstrukten ist. Insbesondere kann es bei Verwendung von Ausdrücken mit Nebeneffekten im Zusammenspiel mit Makros zu schwer aufzufindenden Fehlern kommen.

Um in Java erstellten Programmen ein „stärker objektorientiertes“ Design aufzuzwingen, ist es nicht erlaubt, globale Funktionen und Variablen zu definieren und zu verwenden. Vielmehr muß jede Funktion und Variable innerhalb einer Klasse definiert werden. Dies soll die Programmierer dazu anhalten, ihre Programme objektorientiert zu entwerfen und nicht – wie in C++ möglich – nur ein herkömmliches C-Programm mit wenigen objektorientierten Merkmalen zu entwickeln.

Ein weiterer, wesentlicher Unterschied zu C++ besteht darin, daß in Java nur einfache Vererbung von Klassen erlaubt ist. Das Weglassen mehrfacher Vererbung reduziert die Komplexität der Sprache, wodurch viele Probleme von C++ vermieden werden können. Hierzu zählen etwa die Mehrdeutigkeit von Namensauflösungen oder das Problem mehrfach auftretender Basisklas-

sen. Um durch diese Vereinfachung nicht allzusehr eingeschränkt zu werden, bietet Java das Konzept von sog. *Interfaces*. Durch Interfaces ist es möglich, jeder Klasse eine oder mehrere abstrakte Typen zuzuordnen. Die Implementierung der Methoden der Typen muß jedoch in der Klasse selbst erfolgen. Mit Interfaces können also nur Methodensignaturen definiert, aber keine Implementierung angegeben werden. Interfaces können in etwa mit abstrakten C++-Klassen verglichen werden, die nur abstrakte Methoden definieren.

Die Abhängigkeiten zwischen Klassen wird in Java reduziert, da `friend`-Deklarationen ebenfalls nicht unterstützt werden. Es hat sich gezeigt, daß solche Konstrukte nur in Ausnahmefällen sinnvoll eingesetzt werden können, viele Programmierer aber trotzdem auf `friend`-Deklarationen zurückgreifen. Die dadurch bedingte enge Koppelung zwischen Klassen ist meist ein Indiz für schlechtes Design und erschwert die Wiederverwendung der Klassen. In Java können Klassen zu sog. *Packages* zusammengefaßt werden. Innerhalb eines Packages können alle Klassen nahezu uneingeschränkt aufeinander zugreifen, d.h. innerhalb eines Packages gilt eine implizite `friend`-Deklaration zwischen allen Klassen.

Ein weiterer Unterschied besteht darin, daß in Java keine `struct`- und `union`-Konstrukte erlaubt sind, da diese durch Klassen ohne Methoden ersetzt werden können. Auch können `union`s zur nicht portablen – da von der internen Darstellung der zugrundeliegenden Datentypen abhängigen – Umwandlung von Typen verwendet werden. Da dies in Java verhindert werden soll, sind `union`s ebenfalls verboten.

Weitere, in C++ unterstützte, aber in Java fehlende Sprachelemente sind `typedef`, `enum`, Operator-Overloading und Default-Werte für Parameter von Methoden.

Der wohl für C und C++-Programmierer auffälligste Unterschied besteht darin, daß in Java keinerlei Konzept eines Pointers oder einer expliziten Referenz existiert. Damit fallen natürlich auch Pointerarithmetik und verwandte Konstrukte – wie der *address-of*-Operator '`&`' – weg. Zusätzlich erhöht sich die Sicherheit der Programme, da keine undefinierten Operationen, wie etwa `reinterpret-casts` in C++ mit inkompatiblen Typen, möglich sind. In Java sind alle Variablen, die nicht-primitive[1] Datentypen enthalten, implizit nur Verweise auf die enthaltenen Objekte. Dadurch werden die Syntax und die Semantik von Java vereinfacht. Es können sich für ungeübte Programmierer aber auch neue Fehlerquellen auftun (siehe den Abschnitt über 'Aliasing' auf Seite 11).

Um das Entwickeln von komplexen Anwendungen zu erleichtern, wird das Management des Arbeitsspeichers und der Destruktion von nicht mehr benötigten Objekten vom *Garbage-Collector* der Java VM durchgeführt. Der Programmierer wird somit davon befreit, explizit Speicherbereiche anzufordern und wieder freizugeben. Auf diese Weise können viele Fehler durch nicht initialisierte Pointer oder mehrfach freigegebene Speicherbereiche vermieden werden.

Um die Sicherheit von Java-Programmen weiter zu steigern, wurden noch einige Sprachelemente zur Sprache hinzugefügt. Hierzu zählen das Überprüfen der Grenzen bei Zugriffen auf Arrays und das Synchronisieren von Methoden und Datenstrukturen bei gleichzeitigem Zugriff von mehreren Threads.

Trotz aller hier aufgeführten zusätzlichen Sicherheitsmaßnahmen und Modifikationen gegenüber der Sprache C++, bleibt das Finden von Fehlern in Java-Programmen ein schwieriges und zeitraubendes Unterfangen.

Im folgenden Abschnitt werden einige Arten von Fehlern in Java-Programmen näher charakterisiert.

[1] Datentypen in Java können in primitive Datentypen und nicht-primitive Datentypen eingeteilt werden. Primitive Datentypen sind z.B. `int`, `long`, `char`, etc. Nicht-primitive Datentypen repräsentieren Objekte. Beispiele für nicht-primitive Datentypen sind `Vector`, `Hashtable`, etc. Zu beachten ist, daß nicht-primitive Datentypen stets über Verweise angesprochen werden, während primitive Datentypen immer 'by value' übergeben werden.

2.2 Fehler in Java-Programmen

Fehler in Java-Programmen lassen sich grob in zwei Kategorien einteilen:

(a) in Fehler, die bei der Übersetzung des Programms durch den Compiler entdeckt werden können, und

(b) Fehler, die während des Ablaufens des Programms auftreten.

Zu den in (a) angesprochenen Fehlern zählen etwa Syntax-Fehler, Verwenden von undefinierten Klassen und Methoden oder statische Typ-Fehler, wie z.B. wenn der Parameter einer Methode eine Instanz der Klasse `Object` fordert, aber ein primitiver Datentyp, wie z.B. `int` übergeben wird. Diese Arten von Fehlern werden bereits bei der Übersetzung des Programms entdeckt und erfordern eine Modifikation des Quelltextes, um ein gültiges Programm erzeugen zu können. Im weiteren wird diese Klasse von Fehlern nicht näher betrachtet.

Der Schwerpunkt der vorliegenden Arbeit liegt im Bereich der in (b) genannten Fehler. Zu diesen Fehlern zählen etwa falsches Verhalten von Programmen, Runtime-Exceptions oder Nichttermination. Beispiele hierzu sind u.a. falsche Ausgaben von Programmen oder Zugriffe auf Arrays mit Indizes außerhalb der gültigen Grenzen. Ursache nicht terminierender Programme kann z.B. eine falsche Abbruchbedingung einer Schleife sein.

Unbeachtet hingegen bleiben „Fehler“, welche auf der näherungsweisen Repräsentation von reellen Zahlen beruhen. Solche „Fehler“ sind im Bereich der numerischen Mathematik angesiedelt und beruhen meist auf unsachgemäßer Anwendung von Gleitkomma-Datentypen. Beispiele hierfür sind das exakte Testen zweier, durch Berechnungen ermittelter Gleitkommazahlen auf Gleichheit oder das Vernachlässigen von Rundungsfehlern bei Rechenoperationen mit Zahlen unterschiedlicher Größenordnung. Solche „Fehler“ können im allgemeinen nicht auf eine oder einige wenige fehlerhafte Anweisungen zurückgeführt werden, sondern es muß bereits beim Entwurf des Algorithmus auf die speziellen Eigenschaften des Problems und der Repräsentation der reellen Zahlen Rücksicht genommen werden.

Fehler, die aufgrund von Verletzungen von Sicherheitseinstellungen auftreten, werden ebenfalls nicht betrachtet. Dies kann z.B. bei Applets auftreten, die auf ein lokales Filesystem zugreifen wollen, jedoch keine ausreichende Berechtigung dafür besitzen. Diese Fehler sind keine Programmierfehler im herkömmlichen Sinn, sondern es liegt keine ausreichende Berechtigung vor, um die gewünschte Operation durchführen zu können. Das Problem kann z.B. durch Signieren des Applets durch den Entwickler und anschließendem Freischalten der benötigten Operationen in den Sicherheitseinstellungen für dieses Applet durch den Benutzer umgangen werden.

Schließlich sind noch *Synchronisationsfehler* bei gleichzeitigem Zugriff auf gemeinsame Variablen von mehreren Threads aus als Fehlerursache zu nennen. Diese Fehler können ausschließlich bei Programmen mit mehreren Threads auftreten und deuten auf ein Fehlen von `synchronized`-Methoden oder -Blöcken hin. Solche Fehler sind besonders schwer zu finden, da ihr Auftreten meist von bestimmten (äußeren) Bedingungen abhängig ist, die nur in speziellen Situationen auftreten und scheinbar nichtdeterministisch sind. Hier können sich auch unterschiedliche Laufzeiten von Programmteilen und Antwortzeiten des Netzwerks aufgrund unterschiedlicher Belastung des Systems bemerkbar machen. Auch solche Fehler werden in der vorliegenden Arbeit nicht näher betrachtet.
Die verbleibenden Fehler lassen sich in zwei Kategorien unterteilen:

(a) *Fehler, die durch das Laufzeitsystem entdeckt werden können.*
Hierzu zählen etwa undefinierte Operationen, wie eine Division durch Null oder Zugriffe auf Arrays mit Indizes außerhalb des gültigen Bereichs. Auch Zugriffe auf Objekte durch Variable, die `null` enthalten, werden durch das Laufzeitsystem (der VM) abgefangen und

durch eine Exception signalisiert. Dem Programm wird daraufhin die Möglichkeit gegeben, den Fehler zu behandeln, oder – falls der Fehler durch das Programm nicht behandelt wird – mit einer Fehlermeldung zu terminieren.

Solche Exceptions sind aber meist nur eine Folge von Fehlern an einer anderen Stelle im Programm. So kann z.B. die Division durch Null durch eine Vielzahl von Fehlern verursacht werden: möglich wären u.a. eine fehlende Abfrage, eine fehlende Initialisierung oder eine falsche Berechnung. Diese Fehler können jedoch nicht von der VM erkannt werden und müssen vom Benutzer durch Testen und Debugging aufgespürt werden.

(b) *Fehler, die nicht durch das Laufzeitsystem entdeckt werden können.*
In dieser Klasse befinden sich die sog. *funktionalen Fehler* [MSW99]. Dies sind Fehler, die keine unerlaubten Operationen darstellen, d.h. die nicht durch das Laufzeitsystem erkannt werden können, sondern nur vom Benutzer anhand von unerwartetem Verhalten des Programms aufgespürt werden können. Obwohl diese Fehler *syntaktisch* oft nur geringe Ausdehnung haben, können ihre *semantischen* Konsequenzen weitreichende Folgen auf das Programm zeigen. Diese können von falschen Ausgaben bis zu Nichttermination des Programms reichen.

Im folgenden werden einige in Java mögliche funktionale und strukturelle Fehler anhand des Programms in Abbildung 2.1 betrachtet. Die Liste der möglichen Fehler erhebt jedoch keinen Anspruch auf Vollständigkeit.
Zunächst werden mögliche *Fehler in Ausdrücken* betrachtet:

- Der syntaktisch wohl einfachste Fehler ist jener der *falschen Konstante*. Diese Art von Fehler kann leicht durch Tippfehler oder Unaufmerksamkeit des Programmierers auftreten. Ein Beispiel für diese Art von Fehler ist, falls in Programm in Abbildung 2.1 in Zeile 3 anstatt der Konstanten `10` die Konstante `1` aufscheinen würde. Auch ein falsches Vorzeichen einer Konstante zählt zu dieser Fehlerart. Um diesen Fehler zu beheben, muß die fehlerhafte Konstante durch die korrekte Konstante ersetzt werden.

- Eine Variante des Fehlers einer falschen Konstante stellt der Fall dar, daß eine Konstante anstelle einer Variablen in einem Ausdruck aufscheint. Als Beispiel kann der Ausdruck `numbers[0] = number;` betrachtet werden, der durch Ersetzen des Array-Index `j` durch die Konstante `0` in Zeile 11 des Beispielprogramms entsteht.

- Ein weiterer möglicher Fehler ist das *Fehlen eines Operators* oder eines Unter-Ausdrucks in einem Ausdruck. Dadurch wird ein möglicherweise falscher Wert berechnet, abhängig von der Art des Ausdrucks und den Werten der dazu verwendeten Variablen und Konstanten. Als Beispiele können etwa ein fehlendes Dekrement einer Schleifenvariable oder ein fehlender Aufruf einer Funktion genannt werden. Aus dem Programm in Abbildung 2.1 wird z.B. ein nicht terminierendes Programm, wenn man Zeile 9 durch `j = j;` ersetzt. Auch das Fehlen eines unären Negationsoperators kann als Fehler dieser Art angesehen werden.

- Weiters existiert der Fehler des Vorkommens eines *falschen Operators* in einem Ausdruck. Dies kann z.B. die Verwendung der Subtraktion anstelle der Addition sein, oder der Aufruf einer falschen Funktion. Im Programm aus Abbildung 2.1 könnte etwa der Ausdruck `j - 1` in Zeile 9 durch `j + 1` ersetzt worden sein. Auch die Verwendung des Postinkrement-Operators `var++` anstelle des Preinkrement-Operators `++var` zählt zu dieser Fehlerart, da der Wert der Variablen – und daher auch der Ergebniswert – im zweiten Fall um eins höher ausfällt als im ersten Fall. Analoges gilt für die Dekrement-Operatoren.

- Der Fall einer *falschen Variablen* tritt dann auf, wenn in einem Ausdruck eine falsche Variable zur Berechnung verwendet wird. In Programm 2.1 entsteht diese Art von Fehler, wenn man z.B. die Variable `number` in Zeile 11 durch die Variable `i` ersetzte.

- Die bisher betrachteten Fehler treten allesamt in Ausdrücken auf und können daher als Spezialfälle des Fehlers eines *falschen Ausdrucks* angesehen werden. Diese Fehlerart umfaßt alle bisher genannten Fehler, es werden jedoch auch allgemeinere falsche Ausdrücke berücksichtigt. Dieser Fehler tritt auf, wenn man z.B. die rechte Seite der Zuweisung in Zeile 6 des Programms 2.1 durch `sqrt(i*i-3*i+5)` ersetzt. Dies demonstriert zugleich auch den Fehler der *falschen Initialisierung* einer Schleifenvariablen.

Fehler im Kontrollfluß eines Programms treten dann auf, wenn Codeabschnitte unter bestimmten Umständen ausgeführt werden, wenn diese eigentlich nicht ausgeführt werden sollten, oder umgekehrt. Dies kann z.B. durch falsche Bedingungen bei Schleifen oder Abfragen verursacht werden. Neben falschen Unter-Ausdrücken bei Bedingungen, können hier im wesentlichen folgende Fehlerarten unterschieden werden:

- *Falsche Vergleichsoperatoren*
  Diese Art von Fehler entsteht, wenn bei Bedingungen in Abfragen oder Schleifen ein falscher Relationsoperator verwendet wird. Als Beispiel betrachte man etwa Zeile 7 in Programm 2.1. Wird der erste Operator '`>`' durch '`>=`' ersetzt, wird der zweite Teil der Bedingung und der Schleifenrumpf eventuell auch mit `j = 0` ausgeführt. Dies verursacht in der Folge eine Runtime-Exception im zweiten Teil der Bedingung, da der Zugriff auf das Array `numbers` mit Index `-1` illegal ist. Diese Art von Fehler, bei dem eine Schleife einmal zu wenig bzw. einmal zu oft durchlaufen wird und dadurch ein falscher Wert einer Variablen verursacht wird, findet man in der Literatur oft auch als „off-by-one-error".

- *Falsche logische Konnektoren*
  Durch falsche logische Konnektoren in booleschen Ausdrücken können ebenfalls Codeabschnitte unerwünschterweise ausgeführt werden. Häufige Fehler dieser Art sind das Vertauschen von '`&&`' und '`||`' oder das Fehlen des Negationsoperators '`!`'. Wird in Programm 2.1 das '`&&`' in Zeile 7 durch '`||`' ersetzt, so würde das Programm die Zahlen in umgekehrter Reihenfolge ausgeben, anstatt in sortierter Reihenfolge.

- Auch Fehler in der *Struktur* können Ursache eines falschen Kontrollflusses sein. Insbesondere fehlende `break`-Anweisungen in `switch`-Anweisungen können zu unerwarteten Effekten führen, da in diesem Fall automatisch die der nachfolgenden `case`-Anweisung zugeordneten Anweisungen ebenfalls ausgeführt werden. Weiters können fehlende `return`-Anweisungen oder `break`-Anweisungen in Schleifen zu fehlerhaftem Verhalten des Programms führen.

Weitere Ursache inkorrekter Programme sind Fehler in der *Struktur der Programme*. Hierzu zählen insbesondere folgende Fehler:

- Ein dem Fehler der falschen Variablen in einem Ausdruck ähnlicher Fehler ist jener der *falschen Variablen bei einer Zuweisung*. Im Gegensatz zum ersten Fehler ist hier aber nicht der berechnete Ausdruck falsch, sondern die Variable auf der linken Seite der Zuweisung. In Programm 2.1 ensteht solch ein Fehler, wenn z.B. Zeile 9 durch `i = j - 1;` ersetzt wird.

- *Fehlende und überflüssige Anweisungen*
  Häufiges Auftreten dieser Fehlerart sind fehlende Initialisierungen von Schleifenzählern

oder Objektvariablen. Auch fehlende `break`- und `default`-Anweisungen in `switch`-Anweisungen und Schleifen, sowie fehlende `return`-Anweisungen zählen zu dieser Fehlerart.

- Auch durch eine *falsche Reihenfolge* von Anweisungen können Fehler in Programmen verursacht werden. So ist z.B. die Reihenfolge der Anweisungen in einer Anweisungssequenz, mit der die Werte zweier Variablen `a` und `b` vertauscht werden können, essentiell um das erwartete Ergebnis zu erhalten. Wird die Sequenz

  ```
  temp = a;
  a = b;
  b = temp;
  ```

  durch

  ```
  a = b;
  temp = a;
  b = temp;
  ```

  ersetzt, ist die erhaltene Sequenz zu `a=b; temp=b;` äquivalent, was sicherlich nicht dem beabsichtigten Verhalten entspricht.

Weitere Fehler können durch die Tatsache hervorgerufen werden, daß in Java alle Variablen nur Verweise auf die eigentlichen Objekte enthalten. Dies hat zur Folge, daß bei Zuweisungen einer Variablen an eine andere nur die Referenz kopiert wird, und daher beide Variablen auf dasselbe Objekt verweisen. Diesen Effekt nennt man *Aliasing* [Deu94, Ghi98]. Insbesondere C++-Programmierer würden in diesem Fall eher ein Kopieren des Objekts erwarten. Daraus können unerwünschte *Nebeneffekte* entstehen, wenn das Objekt durch eine Variable modifiziert wird, da sich somit auch das über die andere Variable angesprochene Objekt verändert.

Dieser Effekt kann auch bei Argumenten von Methoden auftreten, da auch hier bei nichtprimitiven Datentypen nur Verweise übergeben werden. In Methoden muß also darauf geachtet werden, daß zwei Parameter eventuell auf dasselbe Objekt verweisen.

In diesem Abschnitt wurden vorwiegend einfache, syntaktisch wenig umfangreiche Fehler behandelt, da diese am häufigsten aufzutreten scheinen. Auch sind komplexere Fehler oft aus mehreren, weniger komplexen Fehlern zusammengesetzt. Darüber hinaus sind einfache Fehler bei der modellbasierten Diagnose von Software-Programmen von besonderem Interesse, da hier meist möglichst kleine Programmteile als fehlerhaft identifiziert werden sollen, d.h. die möglicherweise fehlerhaften Programmteile sollen so weit als möglich eingegrenzt werden.

Im nächsten Kapitel werden die Grundlagen der modellbasierten Diagnose nach Reiter [Rei87] behandelt. Weiters wird ein Verfahren zur Berechnung von Diagnosen und eine Charakterisierung von Einfach- und Mehrfachfehlerdiagnosen behandelt. Abschließend wird eine Strategie zur optimalen Selektion von Beobachtungspunkten vorgestellt.

Kapitel 3

Modellbasierte Diagnose

Modellbasierte Diagnose ist eine leistungsfähige Methode, um unterschiedliches Verhalten eines konkreten Systems gegenüber einem Modell zu erklären. Dies kann zum Beispiel zur Identifizierung von fehlerhaften Komponenten in elektronischen Schaltkreisen verwendet werden, aber auch für abstraktere Systeme, wie Softwareprogramme. Es muß jedoch möglich sein, das System durch eine Menge von Komponenten und Verbindungen zu beschreiben. Ziel der modellbasierten Diagnose ist es, Mengen von Komponenten zu identifizieren, deren Verhalten nicht mit dem Modell übereinstimmt. Um dies durchführen zu können, sind ein Modell des Systems und Beobachtungen am konkreten System notwendig. Durch eventuelle Unterschiede zwischen dem vom Modell vorhergesagten Verhalten und dem tatsächlich am System beobachteten Verhalten können Mengen von Komponenten identifiziert werden, die Abweichungen gegenüber dem Modell aufweisen und somit als fehlerhaft angesehen werden können.

Die Zusammenhänge der einzelnen Prozesse der modellbasierten Diagnose sind in Abbildung 3.1 dargestellt. Essentiell für die Anwendung von modellbasierter Diagnose ist das Vorhandensein einer Beschreibung der Struktur des Systems. Diese ist meist durch Auflistung der Komponenten und deren Verbindungen untereinander gegeben. Neben der Struktur des Systems ist weiters eine Beschreibung des korrekten Verhaltens der einzelnen Komponenten notwendig, um das Verhalten des Systems vorhersagen zu können. Die Beschreibung der Struktur zusammen mit der Beschreibung des Verhaltens der Komponenten bildet die *Systembeschreibung* (das Modell). Weiters ist eine Menge von *Beobachtungen* am zu diagnostizierenden System notwendig. Dies können z.B. Messungen von Signalen in elektronischen Schaltkreisen oder auch Werte von Variablen in Softwareprogrammen sein. Stehen die Beobachtungen mit dem erwarteten, d.h. mit dem vom Modell vorhergesagten Verhalten im Widerspruch, müssen – unter der Annahme der Korrektheit des Modells – eine oder mehrere Komponenten fehlerhaftes Verhalten aufweisen. Ziel der modellbasierten Diagnose ist es, jeder Komponente genau einen Verhaltensmodus zuzuordnen, so daß die *Diagnose* sowohl mit dem Modell als auch mit den Beobachtungen im Einklang steht. Die Diagnose erklärt also das unterschiedliche Verhalten (die *Symptome*) des Systems gegenüber dem Modell.

3.1 Grundlagen

Im folgenden werden die Grundlagen der modellbasierten Diagnose behandelt, wie sie von Reiter [Rei87] formal beschrieben wurden. Weitere Arbeiten zum Thema modellbasierte Diagnose sind in [GSW89] und [dKW87] zu finden.

Um modellbasierte Diagnose möglichst allgemein anwenden zu können, ist die Definition eines zu diagnostizierenden Systems möglichst allgemein gehalten.

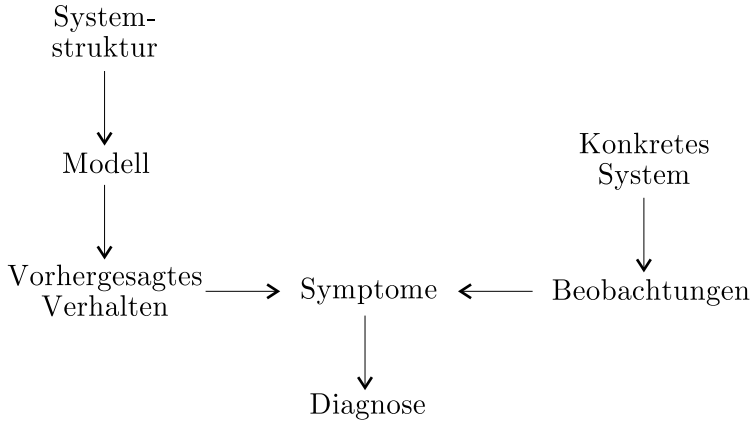

Abbildung 3.1: Funktionsweise der modellbasierten Diagnose

Definition 3.1 (System) Ein *System* ist ein Paar $(SD, COMPS)$, wobei SD die Systembeschreibung und $COMPS$ die endliche Menge von Komponenten des Systems darstellt.

Beispiel 3.2 Sei das zu diagnostizierende System ein einfacher Schaltkreis (siehe Abbildung 3.2). Er setzt sich aus drei Multiplizierern ($\{M_1, M_2, M_3\}$) und zwei Addierern ($\{A_1, A_2\}$) zusammen. Die Zusammenschaltung der Ein- und Ausgänge ist durch die Verbindungslinien gegeben.

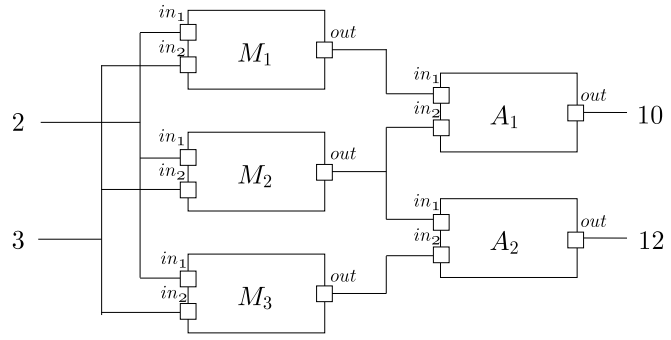

Abbildung 3.2: Schaltkreis mit beobachteten Werten an den Ein- und Ausgängen

Die Menge $COMPS$ enthält die Komponenten des Systems: $COMPS = \{M_1, M_2, M_3, A_1, A_2\}$.

Die Beschreibung der Struktur des Systems kann z.B. in Prädikatenlogik erfolgen. Sowohl Addierer als auch Multiplizierer haben jeweils zwei Eingänge (in_1 und in_2) und einen Ausgang (out). An den Ein- und Ausgängen auftretende Werte einer Komponente C werden durch $in_i(C)$ ($i \in \{1, 2\}$) bzw. $out(C)$ beschrieben. Dann erfolgt die Beschreibung der Struktur des Schaltkreises wie folgt:

$$\begin{aligned} &multiplizierer(M_1), \quad addierer(A_1), \\ &multiplizierer(M_2), \quad addierer(A_2), \\ &multiplizierer(M_3), \\ &out(M_1) = in_1(A_1), \quad out(M_2) = in_1(A_2), \\ &out(M_2) = in_2(A_1), \quad out(M_3) = in_2(A_2). \end{aligned}$$

Das Verhalten der Addierer- und Multipliziererkomponenten kann ebenso durch Ausdrücke in Prädikatenlogik angegeben werden:

$$addierer(C) \wedge \neg ab(C) \Rightarrow out(C) = in_1(C) + in_2(C),$$
$$multiplizierer(C) \wedge \neg ab(C) \Rightarrow out(C) = in_1(C) * in_2(C).$$

Das Prädikat $ab(C)$ ist genau dann wahr, wenn dessen Argument abnormales Verhalten zeigt, d.h. das beobachtete Verhalten nicht mit der Systembeschreibung übereinstimmt.

Die Verhaltensbeschreibung für die Addiererkomponenten kann wie folgt gelesen werden: „Wenn die gegebene Komponente ein Addierer ist und die Komponente korrekt funktioniert, dann muß der Wert am Ausgang gleich der Summe der beiden Werte an den Eingängen sein."

Nur aufgrund der Systembeschreibung sind noch keine Aussagen über das System möglich. Zusätzlich sind noch Beobachtungen notwendig.

Definition 3.3 (Beobachtung) Eine Beobachtung ist eine endliche Menge von Fakten. Ein System mit Beobachtungen ist ein Tripel $(SD, COMPS, OBS)$, wobei $(SD, COMPS)$ ein System aus Definition 3.1 ist und OBS die Menge der Beobachtungen.

Beispiel 3.4 Die in Abbildung 3.2 angegebenen Beobachtungen werden durch folgende Menge repräsentiert: $\{in_1(M_1) = 2, in_2(M_1) = 3, in_1(M_2) = 2, in_2(M_2) = 3, in_1(M_3) = 2, in_2(M_3) = 3, out(A_1) = 10, out(A_2) = 12\}$.

Zu beachten ist, daß die hier angegebenen Beobachtungen dem erwarteten Verhalten des Systems widersprechen, da der Wert am Ausgang des Addierers A_1 nicht 10, sondern ebenfalls 12 ergeben müßte, falls alle Komponenten entsprechend ihrem Modell arbeiten würden. Daraus kann geschlossen werden, daß mindestens eine fehlerhafte Komponente im Schaltkreis existiert. Formal manifestiert sich diese Tatsache dadurch, daß die Menge

$$SD \cup OBS \cup \{\neg ab(M_1), \neg ab(M_2), \neg ab(M_3), \neg ab(A_1), \neg ab(A_2)\}$$

inkonsistent ist.

Definition 3.5 (Diagnose) Eine Diagnose eines Systems $(SD, COMPS, OBS)$ ist eine Menge $\Delta \subseteq COMPS$, so daß

$$SD \cup OBS \cup \{\neg ab(C) \mid C \in COMPS - \Delta\} \cup \{ab(C) \mid C \in \Delta\}$$

konsistent ist.

Eine Diagnose eines Systems $(SD, COMPS, OBS)$ heißt *minimal*, wenn keine Teilmenge derselben eine Diagnose für $(SD, COMPS, OBS)$ ist.

In anderen Worten: eine Diagnose ist eine Menge von Komponenten, die als abnormal betrachtet werden muß, damit das abweichende Verhalten zwischen Beobachtungen und Systembeschreibung erklärt werden kann. Alle anderen Komponenten des Systems werden dabei als korrekt betrachtet. Zu beachten ist, daß Diagnosen stets möglichst minimale Mengen sein sollten, da natürlich immer die Menge aller Komponenten eine gültige Diagnose wäre, dies jedoch meist keine brauchbare Information beinhaltet.

Beispiel 3.6 Im dem hier betrachteten Beispiel ergeben sich folgende minimale Diagnosen: $\{A_1\}$, $\{M_1\}$, $\{M_2, M_3\}$ und $\{M_2, A_2\}$.

Korollar 3.7 Eine Diagnose für ein System $(SD, COMPS, OBS)$ existiert genau dann, wenn $SD \cup OBS$ konsistent ist.

Wäre $SD \cup OBS$ inkonsistent, kann es keine Menge C geben, so daß $SD \cup OBS \cup C$ konsistent ist. Daher kann nach Definition 3.5 auch keine Diagnose existieren.

Korollar 3.8 $\{\}$ ist eine Diagnose für ein System $(SD, COMPS, OBS)$ genau dann, wenn

$$SD \cup OBS \cup \{\neg ab(C) \mid C \in COMPS\}$$

konsistent ist.

Stehen die Beobachtungen mit dem erwarteten Verhalten nicht in Widerspruch, d.h. alle Komponenten verhalten sich entsprechend ihrem Modell, ist die leere Menge die einzige Diagnose.

Die Charakterisierung von Diagnosen kann gegenüber Definition 3.5 noch weiter entwickelt werden. Details dazu finden sich in [Rei87]. Schließlich erhält man folgendes Korollar:

Korollar 3.9 Eine Menge $\Delta \in COMPS$ ist eine Diagnose für ein System $(SD, COMPS, OBS)$ genau dann, wenn

$$SD \cup OBS \cup \{\neg ab(C) \mid C \in COMPS - \Delta\}$$

konsistent ist.

Durch Korollar 3.9 läßt sich das Finden von Diagnosen auf das Generieren von möglichen Kandidaten und einem anschließenden Konsistenztest reduzieren. Diagnosekandidaten eines Systems $(SD, COMPS, OBS)$ sind alle jene Mengen Δ, für die gilt:

$$SD \cup OBS \cup \{\neg ab(C) \mid C \in COMPS - \Delta\} \text{ ist konsistent.}$$

Dies umfaßt alle minimalen Diagnosen und ihre Obermengen. Da die Anzahl der möglichen Kandidaten exponentiell mit der Anzahl der Komponenten im System wächst, ist dieser Ansatz nicht praktikabel. Daher wird im folgenden ein Verfahren dargestellt, welches das Finden von Diagnosen mit Hilfe von *Conflict Sets* durchführt.

Definition 3.10 (Conflict Set) Ein *Conflict Set* für ein System $(SD, COMPS, OBS)$ ist eine Menge $\{C_1, \ldots, C_n\} \subseteq COMPS$, so daß

$$SD \cup OBS \cup \{\neg ab(C_1), \ldots, \neg ab(C_n)\}$$

inkonsistent ist.

Ein Conflict Set für ein System $(SD, COMPS, OBS)$ heißt *minimal*, wenn keine echte Teilmenge des Conflict Sets ein Conflict Set für $(SD, COMPS, OBS)$ ist.

Definition 3.11 (Hitting Set) Sei C eine Menge von Mengen. Ein *Hitting Set* von C ist eine Menge $H \subseteq \bigcup_{S \in C} S$, so daß $H \cap S \neq \emptyset$ für jede Menge $S \in C$.

Ein Hitting Set von C heißt *minimal*, wenn keine echte Teilmenge des Hitting Sets existiert, die ebenfalls ein Hitting Set von C ist.

Beispiel 3.12 Sei C die Menge $\{\{A, B\}, \{A, C\}, \{D, E\}, \{F\}\}$. Daraus ergeben sich folgende minimale Hitting Sets: $\{A, D, F\}$ und $\{A, E, F\}$.

Die folgende alternative Charakterisierung von Diagnosen mittels Hitting Sets bildet die Grundlage für ein Verfahren zur schnelleren Berechnung von Diagnosen.

Satz 3.13 ([Rei87] Theorem 4.4) Eine Menge $\Delta \subseteq COMPS$ ist eine Diagnose für ein System $(SD, COMPS, OBS)$ genau dann, wenn Δ ein Hitting Set für die Menge aller Conflict Sets für das System $(SD, COMPS, OBS)$ ist.

Jede Obermenge eines Conflict Sets für ein System $(SD, COMPS, OBS)$ ist ebenfalls ein Conflict Set für $(SD, COMPS, OBS)$. Daraus ergibt sich folgendes Korollar:

Korollar 3.14 Eine Menge H ist ein minimales Hitting Set für die Menge aller Conflict Sets für ein System $(SD, COMPS, OBS)$ genau dann, wenn H ein minimales Hitting Set für die Menge der minimalen Conflict Sets für das System $(SD, COMPS, OBS)$ ist.

Durch Korollar 3.14 ergibt sich eine weitere Charakterisierung von Diagnosen:

Korollar 3.15 Eine Menge $\Delta \subseteq COMPS$ ist eine minimale Diagnose für ein System $(SD, COMPS, OBS)$ genau dann, wenn Δ ein minimales Hitting Set für die Menge aller minimalen Conflict Sets für $(SD, COMPS, OBS)$ ist.

Beispiel 3.16 Das System in Beispiel 3.4 besitzt folgende minimale Conflict Sets: $\{M_1, M_2, A_1\}$ und $\{M_1, M_3, A_1, A_2\}$. Die minimalen Hitting Sets[1] für diese Menge sind: $\{A_1\}$, $\{M_1\}$, $\{M_2, A_2\}$ und $\{M_2, M_3\}$. Dies sind nach obigem Korollar auch die minimalen Diagnosen für dieses Beispiel.

Im folgenden wird ein Verfahren betrachtet, das mit Unterstützung durch einen Theorembeweiser in der Lage ist, minimale Hitting Sets und damit nach Satz 3.13 auch Diagnosen für ein System $(SD, COMPS, OBS)$ zu berechnen.

Definition 3.17 (HS-DAG) Sei F eine Menge von Mengen. Ein an Knoten und Kanten beschrifteter azyklischer Graph G ist ein HS-DAG („hitting set directed acyclic graph“) genau dann, wenn G der kleinste Graph mit folgenden Eigenschaften ist:

1. Die Wurzel des HS-DAG ist mit $\surd$ beschriftet, falls F leer ist. Ansonsten ist die Wurzel mit einer Menge aus F markiert.

2. Sei n ein Knoten des HS-DAG. Dann ist $H(n)$ die Menge aller Beschriftungen jener Kanten des HS-DAG im Pfad von der Wurzel zu n. Ist n mit $\surd$ beschriftet, besitzt n keine Nachfolger. Ist n mit einer Menge $\Sigma \in F$ beschriftet, dann besitzt n je einen Nachfolger n_σ für jedes $\sigma \in \Sigma$, wobei die Kante zwischen n und n_σ mit σ beschriftet ist. Die Beschriftung von n_σ ist eine Menge $S \in F$, so daß $H(n_\sigma) \cap S = \emptyset$. Existiert keine solche Menge S, ist n_σ mit $\surd$ beschriftet.

Aus obiger Definition folgt unmittelbar:

1. Ist n ein mit $\surd$ beschrifteter Knoten eines HS-DAG, dann ist $H(n)$ ein Hitting Set für F.

2. Für jedes minimale Hitting Set S für F existiert ein mit $\surd$ beschrifteter Knoten n des HS-DAG, so daß $H(n) = S$.

Es ist zu bemerken, daß die Menge aller $H(n)$ für alle n des HS-DAG keineswegs alle Hitting Sets für F enthält, aber alle minimalen Hitting Sets. Dies ist jedoch ausreichend für die Berechnung von Diagnosen.

Weiters sei die Menge F im weiteren die Menge aller Conflict Sets für ein System $(SD, COMPS, OBS)$. Aufgrund der Größe und exponentiellen Berechnungskomplexität der Conflict Sets ist die Menge nicht vollständig explizit gegeben, sondern nur implizit durch das System $(SD, COMPS, OBS)$. Es muß daher versucht werden, die Zugriffe auf die Menge F so gering wie möglich zu halten, da jeder Zugriff einen Aufruf eines Theorembeweisers bedeutet und daher als extrem aufwendig anzusehen ist. Der folgende Algorithmus [GSW89] versucht, durch

[1] Die Berechnung der minimalen Hitting Sets wird in Algorithmus 3.18 beschrieben.

Abbruchkriterien, Wiederverwendung von Knoten und Beschneiden des HS-DAG den Graphen so klein wie möglich zu halten und damit den Berechnungsaufwand zu minimieren.

Die Menge F aus Definition 3.17 sei nun die implizit gegebene Menge der Conflict Sets für ein System $(SD, COMPS, OBS)$. Um den Algorithmus zu vereinfachen, wird angenommen, daß F eine geordnete Menge ist. Der Algorithmus läuft nun wie folgt ab:

Algorithmus 3.18 ($HS-DAG_0$ [GSW89])

1. Sei D der zu generierende HS-DAG. Man erzeuge einen Wurzelknoten von D. Der erzeugte Knoten wird anschließend in Schritt 2 weiter bearbeitet.

2. Man bearbeite die Knoten in D in breadth-first-Reihenfolge, d.h. Knoten mit der geringsten Entfernung zur Wurzel zuerst und bei gleicher Entfernung von links nach rechts. Der zu bearbeitende Knoten sei n.

   (a) Analog zu Definition 3.17 sei $H(n)$ die Menge aller Beschriftungen aller Kanten auf dem Pfad von der Wurzel zu n.

   (b) Ist für alle $S \in F$ $S \cap H(n) \neq \emptyset$, dann wird n mit $\surd$ bezeichnet. Andernfalls wird n mit einer Menge Σ bezeichnet, wobei Σ die erste Menge aus F ist für die gilt: $\Sigma \cap H(n) = \emptyset$.

   (c) Ist n mit einer Menge $\Sigma \in F$ bezeichnet, ist für jedes $\sigma \in \Sigma$ eine neue Kante in D zu erzeugen, die mit σ bezeichnet wird. Die Kante weist auf einen neuen Knoten m in D, mit $H(m) = H(n) \cup \{\sigma\}$. Dieser neue Knoten m wird aufgrund der breadth-first-Strategie erst bearbeitet, wenn alle Knoten in der Ebene von n bearbeitet worden sind.

3. Man liefere den erzeugten HS-DAG D zurück.

Mit diesem Algorithmus können zwar die minimalen Hitting Sets berechnet werden, der Berechnungsaufwand läßt sich durch einige Überlegungen noch weiter herabsetzen.

1. *Wiederverwendung von Knoten*
   Um Aufrufe des Theorembeweisers zu vermeiden, sollten Knoten so oft wie möglich wiederverwendet werden, anstatt einen neuen Knoten m in D anzulegen. Hier sind zwei Fälle zu unterscheiden:

   (a) Existiert ein Knoten n' in D mit $H(n') = H(n) \cup \{\sigma\}$, dann verbinde man die mit σ bezeichnete Kante unter n mit n', anstatt einen neuen Knoten m anzulegen.

   (b) Andernfalls ist ein neuer Knoten m in D zu generieren und wie in Algorithmus 3.18 zu verfahren.

2. *Schließen von Knoten*
   Existiert ein Knoten n' in D, der mit $\surd$ bezeichnet ist, und es gilt $H(n') \subset H(n)$, dann wird der Knoten n geschlossen (d.h. der Knoten wird mit $\times$ bezeichnet). Es werden auch keine Nachfolger in D generiert. Diese Optimierung kann durchgeführt werden, da $H(n') \subset H(n)$ und daher $H(n)$ kein minimales Hitting Set mehr sein kann.

3. *Beschneiden des Graphen*
   Wurde eine Menge $\Sigma \in F$ ausgewählt, um einen Knoten n aus D zu bezeichnen, kann der HS-DAG eventuell beschnitten werden:

   (a) Existiert ein Knoten n' in D der mit $S' \in F$ bezeichnet wurde, wobei $\Sigma \subset S'$, dann ist die Bezeichnung von n' durch Σ zu ersetzen. Anschließend sind alle Kanten

von n', die mit $\alpha \in S' - \Sigma$ beschriftet sind zu entfernen. Alle Knoten unterhalb solcher Kanten werden aus D entfernt, ausgenommen solche, die noch mindestens einen weiteren Vorgänger in D besitzen. Es ist zu beachten, daß dadurch auch der gerade bearbeitete Knoten n aus D entfernt werden kann.

(b) Man vertausche die Mengen S' und Σ in F. Dies hat den gleichen Effekt wie wenn man S' aus F entfernte.

Beispiel 3.19 Sei F die (in diesem Fall explizit bekannte) Menge $\{\{a,b\},\{b,c\},\{a,c\},\{b,d\},\{b\}\}$. Abbildung 3.3 zeigt den vom Algorithmus erzeugten HS-DAG bevor der HS-DAG im Zuge der Bearbeitung von Knoten n_7 beschnitten wird. Es ist zu beachten, daß der Knoten n_3 wiederverwendet wurde, da $H(n_3) = H(n_2) \cup \{a\}$. Wird nun in Knoten n_7 die Menge $\{b\} \in F$ gewählt, kann der Graph beschnitten werden. Abbildung 3.4 zeigt den HS-DAG nach der Operation.

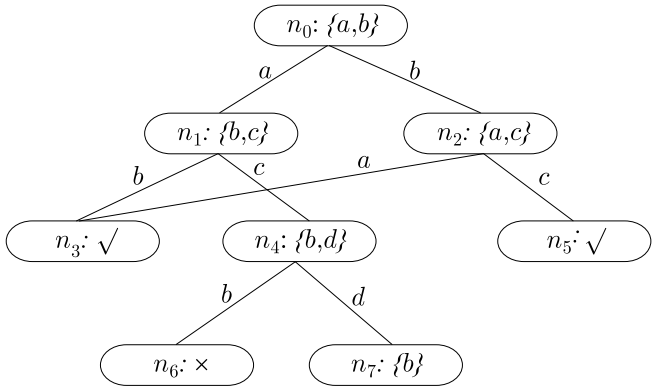

Abbildung 3.3: HS-DAG vor dem Beschneiden in Knoten n_7

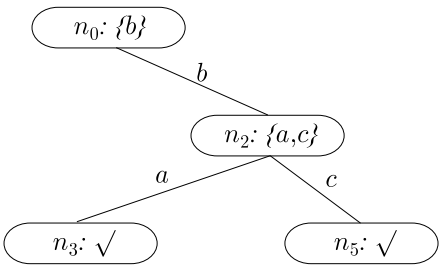

Abbildung 3.4: HS-DAG nach dem Beschneiden in Knoten n_7

Satz 3.20 ([Rei87] Theorem 4.8) Sei F eine Menge von Mengen, und sei T ein vom erweiterten Algorithmus 3.18 für F erzeugter HS-DAG. Dann gilt: Die Menge $\{H(n) \mid n \in T$ und n ist mit $\surd$ beschriftet$\}$ ist die Menge der minimalen Hitting Sets für F.

Mit Hilfe von Algorithmus 3.18 und einem Theorembeweiser kann nun ein Verfahren entwickelt werden, mit dem alle Diagnosen eines Systems berechnet werden können [Rei87]. Da der Algorithmus auf Satz 3.13 beruht, bleibt noch das Problem, daß alle Conflict Sets des gegebenen Systems zu berechnen sind. Durch Anwendung von Algorithmus 3.18 reduziert sich die Menge der zu berechnenden Conflict Sets beträchtlich, da der Algorithmus nur dann auf die Menge F (und damit auf die Menge der Conflict Sets) zugreift, wenn es unbedingt notwendig ist. Durch

diese Vorgehensweise wird es möglich, die Conflict Sets nach und nach einzeln zu berechnen und somit überflüssige Berechnungen im Theorembeweiser zu vermeiden.

In Algorithmus 3.18 können Knoten auf zwei verschiedene Arten mit einer Bezeichnung versehen werden:

1. Durch Wiederverwenden eines bestehenden Knotens. In diesem Fall ist kein Zugriff auf die Menge der Conflict Sets notwendig.

2. Durch Finden eines neuen Conflict Sets. Es wird nach einem Conflict Set S gesucht, für das gilt: $H(n) \cap S = \emptyset$. Existiert eine Menge S mit der gewünschten Eigenschaft, wird diese als Bezeichnung verwendet. Andernfalls wird der Knoten mit $\surd$ bezeichnet.

Um einen HS-DAG zu erzeugen, ist eine Funktion notwendig, die als Argument $H(n)$ erwartet und eine Menge S zurückliefert, so daß $H(n) \cap S = \emptyset$. Falls keine Menge S mit dieser Eigenschaft existiert, wird $\surd$ zurückgeliefert.

Sei nun $TP(SD, COMPS, OBS)$ eine Funktion, die ein Conflict Set zurückliefert, falls eines existiert, d.h. wenn $SD \cup OBS \cup \{\neg ab(C) \mid C \in COMPS\}$ inkonsistent ist. Andernfalls soll $\surd$ zurückgeliefert werden. Die Funktion TP kann z.B. ein Theorembeweiser sein, der die am Resolutionsbeweis beteiligten Annahmen über Komponenten zurückliefert.

Diese Funktion hat folgende Eigenschaft: Sei $C \subseteq COMPS$. Ein Aufruf Funktion $TP(SD, COMPS - C, OBS)$ liefert ein Conflict Set S für das System $(SD, COMPS, OBS)$ mit $S \cap C = \emptyset$. Existiert kein solches Conflict Set, wird $\surd$ geliefert. Diese Funktion kann nun dazu verwendet werden, Conflict Sets (also Bezeichnungen für Knoten des HS-DAG) mit der gewünschten Eigenschaft zu berechnen. Wird eine Bezeichnung für einen Knoten n des HS-DAG gesucht, muß die Funktion $TP(SD, COMPS - H(n), OBS)$ aufgerufen werden. Als Ergebnis erhält man die Bezeichnung für den Knoten n. Auf diese Weise läßt sich ein vollständiger HS-DAG aufbauen, mit dem dann die minimalen Hitting Sets und somit auch die minimalen Diagnosen berechnet werden können. Zusammenfassend ergibt sich folgender Algorithmus:

Algorithmus 3.21 (Diagnose [Rei87])

1. Man erzeuge einen beschnittenen HS-DAG D für die Menge der Conflict Sets F für das System $(SD, COMPS, OBS)$ nach Algorithmus 3.18. Immer dann, wenn auf die Menge F der Conflict Sets zugegriffen werden muß, rufe man die Funktion $TP(SD, COMPS - H(n), OBS)$ auf, wobei n den aktuellen Knoten im HS-DAG bezeichnet.

2. Man liefere die Menge $\{H(n) \mid n \in D$ und n ist mit $\surd$ bezeichnet$\}$.

An dieser Stelle noch einige interessante Bemerkungen zu Algorithmus 3.21:

- Keine zwei Aufrufe der Funktion TP liefern dasselbe Conflict Set zurück. Dies ist eine Konsequenz der Art und Weise wie die Knoten des HS-DAG in Algorithmus 3.18 bezeichnet werden. Daraus folgt sofort, daß die Funktion TP ein Conflict Set niemals auf mehrere verschiedene Arten berechnen muß. Weiters wird die Funktion TP niemals eine Obermenge eines bereits in einem früheren Aufruf berechneten Conflict Sets zurückliefern. Daraus ergibt sich, daß der Algorithmus 3.21 nur einen kleinen Teil der möglichen Conflict Sets berechnet. Dadurch ergeben sich signifikante Einsparungen, da das Berechnen eines Conflict Sets hohen Rechenaufwand erfordert.

- Der Algorithmus setzt nur die Existenz der Funktion TP voraus. Es werden jedoch keinerlei Annahmen über deren Implementierung gemacht. Das hat den Vorteil, daß die Funktion TP auf beliebige Art und Weise implementiert und daher auch auf eingeschränkte Domänen und Problemfälle optimiert werden kann. Es kann z.B. ein Constraint-Propagation-System anstelle des Theorembeweisers verwendet werden.

- Kann garantiert werden, daß die Funktion TP immer minimale Conflict Sets berechnet, wird beim Erzeugen des HS-DAG niemals eine redundante Kante auftreten. Dadurch kann der Algorithmus 3.18 weiter vereinfacht werden.
- Der Algorithmus berechnet die Diagnosen nach einer breadth-first-Strategie. Dies hat zur Folge, daß die Diagnosen mit geringer Kardinalität zuerst berechnet werden. Ergibt sich aus der Anwendung, daß Diagnosen ab einer gewissen Größe sehr unwahrscheinlich zu sein scheinen oder solche Diagnosen aus anderen Gründen ausgeschlossen werden können, kann der Algorithmus einfach nach Bearbeitung der entsprechenden Ebene im HS-DAG abgebrochen werden.

3.2 Einfach– und Mehrfachfehlerdiagnosen

Diagnosen können grob in zwei Klassen eingeteilt werden: in Einfachfehlerdiagnosen und Mehrfachfehlerdiagnosen. Einfachfehlerdiagnosen bestehen aus nur einem Element, d.h. nur eine einzelne Komponente mit abnormalem Verhalten erklärt sämtliche Unterschiede zwischen erwartetem und beobachtetem Verhalten. Mehrfachfehlerdiagnosen bestehen aus mehreren Elementen, d.h. es sind mehrere abnormale Komponenten notwendig, um das beobachtete Verhalten zu erklären. Einfachfehlerdiagnosen sind von besonderem Interesse, da unter der Annahme, daß Komponenten unabhängig von einander ausfallen, die Wahrscheinlichkeit für ein Auftreten bei Einfachfehlerdiagnosen höher ist als bei Mehrfachfehlerdiagnosen.

Korollar 3.22 (Charakterisierung von Einfachfehlerdiagnosen) Nach Satz 3.13 ist $\{c\}$ genau dann eine Einfachfehlerdiagnose des Systems $(SD, COMPS, OBS)$, wenn c in jedem Conflict Set von $(SD, COMPS, OBS)$ enthalten ist.

Aus Algorithmus 3.21 und Korollar 3.22 folgt sofort, daß alle Einfachfehlerdiagnosen aus einem einzigen Conflict Set berechnet werden können:

Satz 3.23 Sei C ein Conflict Set für ein System $(SD, COMPS, OBS)$. $\{c\}$ ist eine Einfachfehlerdiagnose für $(SD, COMPS, OBS)$ genau dann, wenn $c \in C$ und

$$SD \cup OBS \cup \{\neg ab(k) \mid k \in COMPS - \{c\}\}$$

konsistent ist.

Satz 3.23 läßt sich einfach auf mehrere Conflict Sets eines Systems $(SD, COMPS, OBS)$ generalisieren.

Satz 3.24 ([Rei87] Theorem 4.12) Seien $C_1, \ldots, C_n$ $(n \geq 1)$ Conflict Sets für ein System $(SD, COMPS, OBS)$ und sei

$$C = \bigcap_{i=1}^{n} C_i.$$

Dann ist $\{c\}$ eine Einfachfehlerdiagnose für $(SD, COMPS, OBS)$ genau dann, wenn $c \in C$ und

$$SD \cup OBS \cup \{\neg ab(k) \mid k \in COMPS - \{c\}\}$$

konsistent ist.

Angenommen ein System $(SD, COMPS, OBS)$ hat mehr als eine Diagnose. Ohne zusätzliche Information kann keine der Diagnosen ausgeschlossen werden und daher auch kein eindeutiges Ergebnis erzielt werden. Um Diagnosen ausschließen zu können, muß zusätzliche Information in Form von neuen Beobachtungen hinzugefügt werden. Dies können im Fall elektronischer Schaltkreise z.B. zusätzliche Messungen von Signalen sein.

Im folgenden wird aufgezeigt, welchen Einfluß zusätzliche Beobachtungen auf eine bestehende Menge von Diagnosen ausüben können. Sei also $MEAS$ eine zusätzliche Beobachtung. Zu untersuchen bleibt wie sich die Diagnosen im neuen System $(SD, COMPS, OBS \cup \{MEAS\})$ verändern.

Definition 3.25 Eine Diagnose Δ für ein System $(SD, COMPS, OBS)$ sagt Π vorher genau dann, wenn

$$SD \cup OBS \cup \{ab(C) \mid C \in \Delta\} \cup \{\neg ab(C) \mid C \in COMPS - \Delta\} \models \Pi.$$

Beispiel 3.26 In Beispiel 3.2 sagt die Diagnose $\{M_1\}$ $out(M_2) = 6$ und $out(M_1) = 4$ vorher. Die Diagnose $\{M_2, M_3\}$ sagt $out(M_2) = 4$ und $out(M_3) = 8$ vorher.

Korollar 3.27 Sagt keine Diagnose eines Systems $(SD, COMPS, OBS)$ $\neg\Pi$ vorher, so hat das System $(SD, COMPS, OBS \cup \{\Pi\})$ die gleichen Diagnosen wie $(SD, COMPS, OBS)$.

Anders ausgedrückt: Eine Beobachtung, die keine Diagnose ausschließt, beinhaltet keine neue Information.

Korollar 3.28 Eine Diagnose für ein System $(SD, COMPS, OBS)$, die Π vorhersagt, ist auch eine Diagnose für $(SD, COMPS, OBS \cup \{\Pi\})$. D.h. Diagnosen werden beibehalten, falls das beobachtete Verhalten mit dem vorhergesagten Verhalten übereinstimmt.

Korollar 3.29 Keine Diagnose für ein System $(SD, COMPS, OBS)$, die $\neg\Pi$ vorhersagt, ist eine Diagnose für $(SD, COMPS, OBS \cup \{\Pi\})$. D.h. durch eine zusätzliche Beobachtung werden jene Diagnosen ausgeschlossen, die der Beobachtung widersprechen.

Satz 3.30 ([Rei87] Theorem 5.7) Angenommen jede Diagnose eines Systems $(SD, COMPS, OBS)$ sagt entweder Π oder $\neg\Pi$ vorher. Dann gilt

1. Jede Diagnose für $(SD, COMPS, OBS)$, die Π vorhersagt, ist eine Diagnose für $(SD, COMPS, OBS \cup \{\Pi\})$.
2. Keine Diagnose für $(SD, COMPS, OBS)$, die $\neg\Pi$ vorhersagt, ist eine Diagnose für $(SD, COMPS, OBS \cup \{\Pi\})$.
3. Jede Diagnose von $(SD, COMPS, OBS \cup \{\Pi\})$, die keine Diagnose von $(SD, COMPS, OBS)$ ist, ist eine echte Obermenge einer Diagnose von $(SD, COMPS, OBS)$, die $\neg\Pi$ vorhersagt.

Aus Satz 3.30 folgt, daß durch eine Beobachtung, die eine oder mehrere Diagnosen eliminiert, neue Diagnosen entstehen können. Deren Kardinalität ist jedoch größer als jene der eliminierten Diagnose.

Korollar 3.31 ([Rei87] Korollar 5.8) Angenommen $\{\}$ ist keine Diagnose für ein System $(SD, COMPS, OBS)$. Dann folgt aus Satz 3.30, daß jede durch eine Beobachtung neu erzeugte Diagnose eine Mehrfachfehlerdiagnose ist.

Korollar 3.32 ([Rei87] Korollar 5.9) Angenommen {} ist keine Diagnose für ein System $(SD, COMPS, OBS)$. Dann gilt (unter den Annahmen von Satz 3.30): die Einfachfehlerdiagnosen des Systems $(SD, COMPS, OBS \cup \{\Pi\})$ sind genau jene Diagnosen von $(SD, COMPS, OBS)$, die Π vorhersagen.

Beispiel 3.33 Um zwischen den beiden Einfachfehlerdiagnosen $\{A_1\}$ und $\{M_1\}$ aus Beispiel 3.2 zu unterscheiden, kann der Wert an Ausgang $out(M_1)$ beobachtet werden, da $\{A_1\}$ den Wert 6 und $\{M_1\}$ den Wert 4 vorhersagt. Angenommen die Beobachtung ergibt den Wert 6. Dann ist nach Korollar 3.32 $\{A_1\}$ die einzige verbleibende Einfachfehlerdiagnose. Es könnten jedoch auch weitere neue Mehrfachfehlerdiagnosen entstanden sein. Dieser Fall tritt in diesem Beispiel aber nicht auf, und es verbleiben die Diagnosen $\{A_1\}$, $\{M_2, M_3\}$ und $\{M_2, A_2\}$.

3.3 Selektion des nächsten Beobachtungspunktes

Abschließend bleibt die Problematik, welche Strategien zur Selektion der nächsten Beobachtung geeignet sind. Um die Anzahl der notwendigen Beobachtungen so gering wie möglich zu halten, sollten durch eine Beobachtung möglichst viele Diagnosen ausgeschlossen werden können und möglichst wenig neue Diagnosen hinzugefügt werden. D.h. die Menge der Diagnosen sollte durch die zusätzliche Beobachtung so weit als möglich reduziert werden.

DeKleer und Williams [dKW87] verwenden zur Selektion zwischen den einzelnen Beobachtungspunkten ein Maß basierend auf der Entropie und den Wahrscheinlichkeiten für Komponentenfehler.

Das Verfahren nimmt als Grundlage die Strukturbeschreibung des Systems und identifiziert mögliche Beobachtungspunkte x_i, wobei für jedes x_i die Anzahl der möglichen Werte durch die endliche[2] Menge $\{v_{i1}, \ldots, v_{ik}\}$ gegeben ist. Die Diagnosekandidaten (d.h. die Menge aller Diagnosen und alle Obermengen) lassen sich dann für jeden Beobachtungspunkt x_i in drei Mengen unterteilen:

1. Die Menge R_{ik} enthält alle Diagnosekandidaten, die erhalten bleiben, wenn die neue Beobachtung an der Stelle x_i den Wert v_{ik} ergibt.

2. Die Menge S_{ik} enthält alle Diagnosekandidaten, die für die Stelle x_i präzise den Wert v_{ik} vorhersagen. Das sind nach Satz 3.30 genau jene Diagnosekandidaten, die eliminiert werden, wenn der beobachtete Wert ungleich v_{ik} ist.

3. Die Menge U_i enthält alle Diagnosekandidaten, die keinen Wert für eine Beobachtung an der Stelle x_i vorhersagen. Unabhängig davon, welcher Wert an der Stelle x_i beobachtet wird, kann keiner dieser Diagnosekandidaten eliminiert werden.

Aus dieser Charakterisierung der Mengen R_{ik}, S_{ik} und U_i ergeben sich folgende Zusammenhänge:

1. Die Menge R_{ik} kann durch die Mengen S_{ik} und U_i gebildet werden: $R_{ik} = S_{ik} \cup U_i$.

2. Die Mengen S_{ik} und U_i müssen disjunkt sein: $S_{ik} \cap U_i = \emptyset$.

Basierend auf den Mengen R_{ik}, S_{ik} und U_i wird versucht, jenen Beobachtungspunkt zu identifizieren, der die Menge der Diagnosekandidaten möglichst einschränkt. Dazu wird für jeden Beobachtungspunkt bestimmt, welche Konsequenzen dessen Beobachtung für die Menge der Diagnosekandidaten hätte. Anschließend wird der Beobachtungspunkt mit dem besten Ergebnis als

[2] Es ist möglich, dieses Verfahren auf unendliche Mengen möglicher Werte zu generalisieren. [dKW87] betrachten einige Aspekte zu diesem Thema.

nächster Beobachtungspunkt gewählt. Um die Ergebnisse bewerten zu können, ist ein Maß notwendig, das die Güte eines Ergebnisses auf eine Zahl abbildet. Die Güte eines Ergebnisses kann unter Annahme gleicher Kosten für alle Beobachtungen durch die Anzahl der zusätzlichen Beobachtungen geschätzt werden, die notwendig sind, um die Menge aller Diagnosekandidaten auf eine eindeutige Diagnose zu reduzieren. [dKW87] verwenden dazu die Entropie $H = -\sum p_i \ln p_i$, wobei p_i die Wahrscheinlichkeit bezeichnet, daß der Diagnosekandidat C_i tatsächlich die gesuchte Diagnose ist (*Candidate Probability*).

Durch Eigenschaften der Entropie läßt sich zeigen, daß die optimale Beobachtung jene ist, deren erwartete Entropie

$$H_e(x_i) = \sum_{k=1}^{m} p(x_i = v_{ik}) H(x_i = v_{ik})$$

minimal ist, wobei $\{v_{i1}, \ldots, v_{im}\}$ die möglichen Werte für die Beobachtung x_i angeben und $H(x_i = v_{ik})$ die Entropie ist, wenn die Beobachtung von x_i den Wert v_{ik} ergibt.

Wird von jedem Diagnosekandidaten ein Wert für x_i vorhergesagt, ist $p(x_i = v_{ik})$ die kombinierte Wahrscheinlichkeit aller Kandidaten, die $x_i = v_{ik}$ vorhersagen. Ist U_i nicht leer, kann jedoch nicht für alle Diagnosekandidaten eine Aussage gemacht werden. Daher ist es notwendig, die Wahrscheinlichkeit anzunähern. Es gilt:

$$p(x_i = v_{ik}) = p(S_{ik}) + \epsilon_{ik},$$

wobei

$$0 \leq \epsilon_{ik} \leq p(U_i)$$

und

$$\sum_{k=1}^{m} \epsilon_{ik} = p(U_i).$$

Die Wahrscheinlichkeiten für die Mengen S_{ik} und U_i ergeben sich durch

$$p(S_{ik}) = \sum_{C_j \in S_{ik}} p_j$$

bzw.

$$p(U_i) = \sum_{C_j \in U_i} p_j.$$

Weiters wird angenommen, daß für alle Diagnosekandidaten, die keinen Wert für eine Beobachtung x_i vorhersagen, alle möglichen Werte mit gleicher Wahrscheinlichkeit vorhergesagt werden. Daher folgt

$$\epsilon_{ik} = \frac{p(U_i)}{m}.$$

Die Wahrscheinlichkeiten p_i können unter der Annahme der Unabhängigkeit von Komponentenfehlern durch

$$p_i = \left(\prod_{c \in C_i} p_f(c) \right) \left(\prod_{c \notin C_i} (1 - p_f(c)) \right)$$

berechnet werden, wobei $p_f(c)$ die Wahrscheinlichkeit für einen Ausfall der Komponente c angibt.

Die erwartete Entropie kann nun durch $H_e(x_i) = H + \Delta H(x_i)$ dargestellt werden, wobei H die aktuelle Entropie bezeichnet und

$$\Delta H(x_i) = \sum_{k=1}^{n} p(x_i = v_{ik}) \ln p(x_i = v_{ik}) + p(U_i) \ln p(U_i) - \frac{np(U_i)}{m} \ln \frac{p(U_i)}{m}$$

mit $n = |\{S_{ik} \neq \emptyset\}|$ ist[3]. Daraus ergibt sich, daß jener Beobachtungspunkt x_i gewählt werden sollte, der $\Delta H(x_i)$ minimiert.

Da dieses Modell nicht nur alle minimalen Diagnosen benötigt, sondern alle Diagnosekandidaten berücksichtigt werden müssen, ist dieses Verfahren für größere Systeme sehr aufwendig zu berechnen und damit ungeeignet. Aus diesem Grund wird das Verfahren in der Praxis auf die Menge der minimalen Diagnosen eingeschränkt, was die Qualität der Ergebnisse jedoch meist nicht signifikant herabsetzt.

3.4 Fehlermodelle

Dieser Abschnitt betrachtet die Erweiterung der in Abschnitt 3.1 behandelten Grundlagen der modellbasierten Diagnose auf Fehlermodelle. Neben eines motivierenden Beispiels wird kurz auf die Anwendbarkeit von Fehlermodellen im Bereich der Software-Diagnose eingegangen und Literaturverweise auf weiterführende Arbeiten angegeben.

Die von Reiter in [Rei87] vorgestellten Grundlagen der modellbasierten Diagnose betrachten Komponenten entweder als korrekt oder als fehlerhaft. Wird eine Komponente als korrekt angesehen, arbeitet sie entsprechend der Spezifikation ihres Verhaltens. Andernfalls ist die Komponente fehlerhaft und kann beliebiges Verhalten aufweisen. Diese Unterscheidung ist für viele Anwendungsgebiete ausreichend, in einigen Fällen kann aber eine genauere Unterscheidung sinnvoll sein.

Dies wird durch sog. *Fehlermodelle* bewerkstelligt. Fehlermodelle stellen eine Verallgemeinerung der in diesem Kapitel vorgestellten Definitionen und Algorithmen dar, wobei die Verhaltensbeschreibung der Komponenten auf mehr als zwei Verhaltensmodi erweitert wird. Neben der Spezifikation des korrekten Verhaltens kann eine Menge weiterer Verhaltensmodi angegeben werden, wobei das Verhalten der Komponenten in den einzelnen Modi analog zum korrekten Verhalten spezifiziert wird. Es wird also nicht nur das korrekte Verhalten spezifiziert, sondern auch das Verhalten in Fehlerfällen.

Auf diese Weise kann das Ausschließen von unerwünschten Diagnosen erreicht werden. Als Beispiel sei etwa ein elektrischer Schaltkreis mit einer Stromquelle und drei parallel geschalteten Lampen betrachtet [SD89]. Als Beobachtung sei angenommen, daß eine der drei Lampen leuchtet. Dies steht im Gegensatz zur Erwartung, daß alle drei Lampen leuchten. Die Analyse der Schaltung mittels herkömmlicher modellbasierter Diagnose ergibt zwei Diagnosen: (1) Ein Fehlverhalten der Batterie und der leuchtenden Lampe könnte die Ursache des beobachteten Verhaltens sein. (2) Die beiden nicht leuchtenden Lampen werden ebenfalls als mögliche Fehlerursache angesehen. Hier zeigt sich eine Einschränkung der modellbasierten Diagnose ohne Fehlermodelle: Die Bedeutung der ersten Diagnose kann wie folgt in Worte gefaßt werden: *„Die Batterie ist fehlerhaft und liefert daher keine Energie. Weiters ist die leuchtende Lampe fehlerhaft, da diese ohne Vorhandensein von Energie leuchtet.“* Dies ist jedoch aufgrund der physikalischen Gesetze unmöglich. Aus diesem Grund sollte diese Diagnose nicht als mögliche Erklärung in Betracht gezogen werden. Dies kann erreicht werden, indem die Beschreibung der Batterie und der Lampen in der Art erweitert wird, daß deren Verhalten auch im Fehlerfall vollständig spezifiziert ist. Auf diese Weise kann die unerwünschte Diagnose während der Berechnung der Diagnosen ausgesondert werden, da diese das beobachtete Verhalten nicht erklärt.

Auch im Bereich der Diagnose von Software können Fehlermodelle eingesetzt werden. In dieser Domäne kann durch Spezifizieren zusätzlicher Modi von Komponenten die Modellierung von typischen Fehlern, wie etwa das Negieren einer Bedingung einer Auswahlanweisung, erfol-

[3] In obiger Formel für $\Delta H(x_i)$ wird angenommen, daß die v_{ik} so geordnet sind, daß alle von einer Diagnose vorhergesagten Werte die Indizes $k \in \{1, \ldots, n\}$ besitzen. Die Werte einer Verbindung, welche von keiner Diagnose vorhergesagt werden, sind den Indizes $n+1, \ldots, m$ zugeordnet.

gen. Auf diese Weise können durch die erhaltenen Diagnosen genauere Hinweise auf den Fehler gegeben werden, da der durch die Diagnose angegebene Verhaltensmodus für die Komponente auf die Art des Fehlverhaltens und damit auch auf eventuelle Korrekturen hindeutet.

Aufgrund der erweiterten Verhaltensmodi der Komponenten sind eine Abänderungen der Definitionen und Algorithmen gegenüber Abschnitt 3.1 notwendig. Auch der Berechnungsaufwand erhöht sich, da bei der Suche nach Hitting Sets für jede Komponente C nicht nur die Modi $ab(C)$ und $\neg ab(C)$ betrachtet werden müssen, sondern auch alle weiteren spezifizierten Verhaltensmodelle. Der dadurch verursachte Mehraufwand bei der Berechnung bedingt in vielen Fällen jedoch keine große Geschwindigkeitseinbuße. Speziell bei kleineren Systemen und bei Systemen, welche nur wenige Fehlermodelle einsetzen, kann der zusätzliche Aufwand vernachlässigt werden.

Einen weiteren Effekt, welcher durch den Einsatz von Fehlermodellen bedingt wird, stellt die Tatsache dar, daß in diesem Fall nicht mehr alle Diagnosen eines Systems durch die minimalen Diagnosen charakterisiert werden. Insbesondere kann nicht garantiert werden, daß alle Obermengen einer Diagnose ebenfalls Diagnosen darstellen.

Aus Beschränkungen bezüglich des Umfangs dieser Arbeit werden die notwendigen Erweiterungen und Modifikationen der Definitionen und Algorithmen aus Abschnitt 3.1 hier nicht näher betrachtet, sondern nur auf bereits existierende Literatur verwiesen. Motivierende Beispiele sowie eine Einführung in den Einsatz von Fehlermodellen sind in [SD89] nachzulesen. Erweiterungen des Ansatzes von Reiter auf Fehlermodelle sowie eine mögliche Implementierung werden in [Ung91] vorgestellt. Die wichtigsten der in [Ung91] vorgestellten Definitionen und Algorithmen sind in kompakter Form in [Wot96] zusammengefaßt. DeKleer und Williams [dKW89] betrachten die Berechnungskomplexität von Diagnosen bei Verwendung von Fehlermodellen. Poole [Poo89] analysiert die Zusammenhänge zwischen konsistenzbasierter und abduktiver modellbasierter Diagnose sowie die Rolle von Fehlermodellen.

Das folgende Kapitel betrachtet die Modellierung von Java-Programmen als Komponentenmodelle. Es wird zunächst ein Modell vorgestellt, welches alle Variablen des Programms durch Verbindungen repräsentiert und dessen Beschränkungen und mögliche Erweiterungen betrachtet. Anschließend wird ein weiteres Modell entwickelt, welches mit Objekträumen arbeitet und viele der Probleme des ursprünglichen Modells vermeidet.

Kapitel 4

Modellbildung für Java-Programme

Um Programme mittels modellbasierter Diagnose untersuchen zu können, muß zunächst ein Modell für das gegebene Programm erstellt werden. Dieses Modell kann als Repräsentation der relevanten Aspekte des Programms angesehen werden. Insbesondere werden durch das Modell die Art und Qualität der Ergebnisse – d.h. die möglicherweise fehlerhaften Programmteile – bestimmt, die durch den Diagnosealgorithmus dargestellt und gefunden werden können.

Modelle für Java-Programme können sehr unterschiedlich in der Ausdrucksstärke und der Komplexität gewählt werden. Es sind sowohl einfache Modelle, die nur funktionale Abhängigkeiten von Variablen und Anweisungen betrachten [MSW99], als auch komplizierte Modelle, welche die vollständige Semantik der Anweisungen nachbilden, denkbar.

Durch die Wahl des Modells wird zugleich auch die Klasse von Programmen bestimmt, welche mit Hilfe des Modells analysiert werden können: Komplexe Modelle erlauben genauere Diagnosen, der Berechnungsaufwand hierbei übersteigt jenen eines einfacheren Modells aber meist beträchtlich. Dadurch ergibt sich, daß solche Modelle nur für kleinere Programme oder Programmteile eingesetzt werden können, dafür aber genauere Hinweise auf die Fehlerursache als Ergebnis erhalten werden. Einfachere Modelle hingegen sind auch für größere Programme einsetzbar, wobei aber die Fehlerursache aufgrund des höheren Abstraktionsniveaus oft nicht so genau eingegrenzt werden kann.

Weiters wird auch die Art der Benutzeranfragen beim Berechnen der Diagnosen durch das Modell bestimmt. Bei einfachen, nur auf funktionalen Abhängigkeiten basierenden Modellen ist es oft ausreichend, einen Wert als korrekt oder unkorrekt zu deklarieren. Bei Modellen, die z.B. mit konkreten Werten bei der Programmabarbeitung arbeiten, muß hingegen der genaue Wert bekannt sein. Die Ermittlung des korrekten Wertes für eine bestimmte Variable an einer Programmposition stellt natürlich erhöhte Anforderungen an den Benutzer. Dies gilt insbesondere für Werte, die während der Abarbeitung eines komplizierten Algorithmus auftreten. Hier sind dem Benutzer zwar oft die Effekte des gesamten Algorithmus bekannt (oder zumindest an einigen ausgezeichneten Stellen), die genauen Werte während der Abarbeitung des Algorithmus jedoch oft nicht. Dieses Problem tritt in der Praxis nicht allzu häufig auf, da durch die modellbasierte Diagnose große Teile des zu untersuchenden Programms nur aufgrund der durch das Programm berechneten und der vom Benutzer erwarteten Werte vor und nach der Abarbeitung des Programms als Fehlerursachen ausgeschlossen werden können.

Im weiteren wird, basierend auf [MSW00], ein Modell für Java-Programme behandelt, das zum Auffinden von funktionalen Fehlern in Java-Programmen geeignet ist. Das Modell arbeitet mit konkreten Werten („wertbasiert“) und ist daher bei der Berechnung aufwendiger als bisherige, auf funktionalen Abhängigkeiten beruhende Modelle [MSW99]. Daher kann dieses Modell nur für kleine Programmteile eingesetzt werden. Im Gegensatz zum Modell aus [MSW99] können Fehlerursachen durch dieses Modell genauer eingegrenzt und eventuell sogar Vorschläge zur Kor-

rektur des fehlerhaften Programms berechnet werden. Weiters werden Probleme aufgezeigt, die sich durch die genauere Nachbildung der Semantik von Java-Programmen durch das Modell ergeben und mögliche Lösungen vorgestellt.

4.1 Direktes Modell

Grundlegend für den Einsatz modellbasierter Diagnose zur Analyse von Programmen ist das Bilden eines Modells, welches das gegebene Programm auf Komponenten und Verbindungen zwischen den Komponenten abbildet. Die *Struktur* des Modells, d.h. die Komponenten und ihre Verbindungen, wird durch das gegebene Programm bestimmt und kann automatisch aus dem gegebenen Quelltext generiert werden – ähnlich einem Compiler, der aus dem Quelltext ein ausführbares Programm in einem speziellen Format (z.B. Java Bytecode) erzeugt. Weiters ist eine *Verhaltensbeschreibung* für alle Komponenten notwendig. Diese wird von der Semantik der repräsentierten Sprachelemente bestimmt und muß beim Entwurf des Modells in geeigneter Form angegeben werden. Dies kann z.B. in Prädikatenlogik erfolgen.

Das hier behandelte Modell arbeitet anhand von konkreten Werten und muß daher die Semantik des gegebenen Programms exakt nachbilden, da die erhaltenen Diagnosen sonst möglicherweise falsch wären bzw. nicht alle Diagnosen berechnet werden könnten. D.h. das Modell muß bei gegebenen Eingabedaten an jeder beliebigen Stelle des Programms stets die gleichen Ergebnisse liefern, die auch das gegebene Java-Programm berechnen würde.

Die Komponenten des Modells entsprechen einzelnen Programmteilen (wie ganzen Ausdrücken, Anweisungen oder Methodenaufrufen, aber auch Teilen von Ausdrücken wie z.B. dem Operator '+' oder Konstanten), die Verbindungen modellieren die Abhängigkeiten und den Datenfluß zwischen den einzelnen Programmteilen. Hier muß insbesondere auf die objektorientierten Sprachelemente von Java geachtet werden. Dies umfaßt in diesem Fall u.a. Vererbung, Polymorphismus, Methodenaufrufe mit Nebeneffekten, dynamische und rekursive Datenstrukturen sowie Referenzen auf Objekte durch Variablen, Parameter oder Instanzvariablen von Objekten.

Um die Komplexität des Modells in Grenzen zu halten, wird nur eine Untermenge des gesamten Sprachumfangs von Java unterstützt. Insbesondere Arrays, Interface-Deklarationen und Typ-Umwandlungen (*casts*) werden nicht unterstützt. Nebeneffekte in booleschen Ausdrücken können ebenfalls nicht modelliert werden. Auch Exceptions und Threads werden nicht durch dieses Modell abgedeckt.

Durch die Beschränkung des Modells auf die eben genannte Untermenge von Java können zwar nicht alle Programme durch dieses Modell dargestellt werden, die getroffenen Einschränkungen erlauben aber eine Vielzahl von sinnvollen Programmen. So können z.B. die fehlenden Arrays durch einfach verkettete Listen oder andere dynamische Datenstrukturen ersetzt werden.

Um ein Java-Programm nachbilden zu können, müssen zunächst die Klassen, Methoden und Variablen des gegebenen Programms und deren Beziehungen zueinander bestimmt werden. Diese Informationen können leicht aus dem Parsebaum des Programms entnommen werden. Bei Klassen werden neben der Vererbungshierarchie zwischen den Klassen auch die in der jeweiligen Klasse definierten Klassen– und Instanzvariablen, sowie die definierten Methoden benötigt. Bei den Methoden muß zwischen `static`- (*Klassenmethoden*) und nicht-`static`-Methoden (*Instanzmethoden*) unterschieden werden. Methoden, die mit dem Schlüsselwort '`static`' definiert sind, können ohne eine konkrete Instanz der Klasse aufgerufen werden. Daraus folgt auch, daß solche Methoden nicht auf Instanzvariablen der Klasse zugreifen können. Bei Instanzmethoden hingegen ist immer eine Instanz der Klasse notwendig, um die Methode zum Ablauf bringen zu können. Im Gegensatz zu Klassenmethoden kann hier aber sowohl auf Instanzvariablen, als auch auf Klassenvariablen der Klasse zugegriffen werden. Die Variablen des Programms lassen sich anhand ihrer Lebensdauer und Ansprechbarkeit in drei Kategorien einteilen: *lokale Variablen*,

Instanzvariablen einer Klasse und *Klassenvariablen*.

- *Lokale Variablen* werden lokal in einer Methode oder einem Block definiert und können nur innerhalb der Methode bzw. des Blocks angesprochen werden. Ihre Lebensdauer endet mit dem Verlassen der Methode bzw. des Blocks bei der Programmabarbeitung. Daraus ergibt sich, daß diese Variablen keine Werte zwischen zwei Methodenaufrufen speichern können.

- *Instanzvariablen* werden erzeugt, sobald ein Objekt der entsprechenden Klasse erzeugt wird. Sie können sowohl durch Referenzen auf das Objekt[1], als auch durch Methoden der Klasse angesprochen werden. Die Lebensdauer der Instanzvariablen endet, sobald das die Variable enthaltende Objekt zerstört wird. Dies erfolgt durch den Garbage-Collector, der das Objekt zerstört, sobald es nicht mehr angesprochen werden kann – d.h. sobald keine Referenz mehr auf das Objekt existiert. Instanzvariablen sind dafür gedacht, den inneren Zustand des Objekts zwischen den einzelnen Methodenaufrufen zu speichern.

- *Klassenvariablen* hingegen existieren solange, bis die zugehörige Klasse von der Java-Virtual-Machine entladen wird. Also auch dann, wenn keine Instanz der Klasse existiert. Da Klassenvariablen auch ohne konkrete Instanz der Klasse angesprochen werden können, stellen diese eine Art „globale“ Variable dar und können von mehreren Programmteilen (also insbesondere auch von Instanzmethoden der Klasse) verwendet werden. Auf diese Weise können etwa Informationen zwischen den einzelnen Instanzen der Klasse weitergegeben werden, ohne daß die einzelnen Instanzen Kenntnis voneinander haben müssen. Die Variable wird also zwischen den einzelnen Instanzen der Klasse geteilt.

Das hier vorgestellte Modell repräsentiert einzelne Anweisungen und Teile von Ausdrücken als Komponenten. Die Variablen werden als Verbindungen zwischen den Komponenten dargestellt, ungeachtet dessen, ob es sich um lokale Variablen, Instanzvariablen oder Klassenvariablen handelt.

Aufgrund der Struktur des Modells können nur funktionale Fehler (wie z.B. falsche Operatoren in Ausdrücken) erkannt werden. Durch dieses Modell unerkannt bleiben strukturelle Fehler, wie z.B. fehlende Anweisungen oder Verwendung von falschen Variablen in Ausdrücken oder Zuweisungen.

Die Funktionsweise des Modells kann näherungsweise wie folgt zusammengefaßt werden: Während der Initialisierung des Modellbildungsprozesses wird für jede existierende Variable eine Verbindung angelegt. Anschließend wird das Programm in ein Modell übersetzt. Wird dabei eine Variable in einer Anweisung oder einem Ausdruck benötigt, wird eine Verbindung zwischen der die Anweisung repräsentierenden Komponente und der der Variablen zugeordneten Verbindung hergestellt. Wird einer Variablen ein neuer Wert zugewiesen – z.B. durch eine Zuweisung oder durch Nebeneffekte eines Methodenaufrufs – wird eine neue Verbindung für diese Variable angelegt und ab diesem Zeitpunkt anstelle der alten Verbindung verwendet.

Beispiel 4.1 Zur Illustration sei folgendes Programmfragment zu analysieren:

```
1  int x, y, z;      /* Anweisung S1 */
2  x = 1;            /* Anweisung S2 */
3  y = x + 1;        /* Anweisung S3 */
4  x = y + 1;        /* Anweisung S4 */
5  z = x - 2;        /* Anweisung S5 */
```

[1] Eventuell wird die Ansprechbarkeit der Variablen durch Access-Specifier eingeschränkt. Innerhalb eines Packages sind jedoch alle Variablen und Methoden einer Klasse von jeder anderen Methode aus ansprechbar. Die einzige Ausnahme stellen Variablen und Methoden dar, die mit einem `private`-Access-Specifier definiert wurden. Sie sind nur durch Methoden derselben Klasse ansprechbar, in der auch die Variable bzw. Methode definiert wurde.

Zunächst werden die Variablendeklarationen für die lokalen Variablen `x`, `y` und `z` in Zeile 1 analysiert und Default-Verbindungen dafür angelegt. Anschließend wird Zeile 2 analysiert und eine neue Verbindung für `x` angelegt, da `x` durch die Anweisung verändert wird. Diese Verbindung wird von nun an bis zur nächsten Veränderung von `x` benutzt, wann immer die Variable `x` benötigt wird. In Zeile 3 wird die Variable `x` benötigt, d.h. die Komponente, welche die Anweisung in Zeile 3 repräsentiert, wird mit der bei der Analyse von Zeile 2 erzeugten Verbindung verbunden. Weiters wird die Default-Verbindung der Variablen `y` durch eine neue Verbindung ersetzt, da `y` verändert wird. In Zeile 4 wird diese Verbindung gleich wieder verwendet und darüber hinaus eine neue Verbindung für die Variable `x` angelegt, welche die Verbindung aus Zeile 2 ersetzt. Schließlich wird diese Verbindung für `x` während der Analyse von Zeile 5 verwendet und die Default-Verbindung von `z` durch eine neue Verbindung ersetzt. Nach Analyse von Zeile 5 ist diese Phase der Modellbildung abgeschlossen, und man erhält die in Abbildung 4.1 dargestellte Struktur.[2] In der Abbildung werden Komponenten durch Rechtecke dargestellt und sind mit dem Label der korrespondierenden Anweisung beschriftet. Verbindungen sind durch kleine Quadrate dargestellt, über denen die korrespondierende Variable dargestellt ist und werden am rechten Rand jener Komponente plaziert, welche die entsprechende Variable verändert. Verbindungen zu anderen Komponenten werden durch Linien zwischen den Verbindungen und den Komponenten dargestellt.

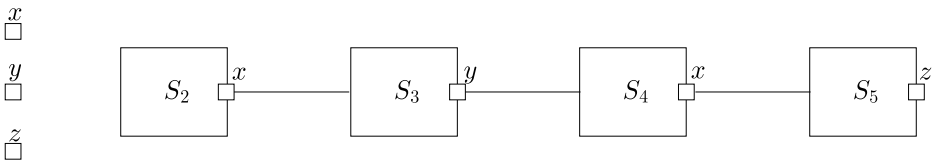

Abbildung 4.1: Modell von Variablen und Anweisungen aus Beispiel 4.1

Die hier beschriebene Konversion von Anweisungen und Variablen in Komponenten und Verbindungen ist nur dann wohldefiniert, wenn die Reihenfolge der Abarbeitung von Anweisungen zur Zeit der Umwandlung bekannt ist. Hier ist anzumerken, daß nicht bekannt sein muß, welcher Zweig einer Auswahlanweisung (wie z.B. `if`-Anweisung) ausgeführt wird oder wieviele Iterationen eine Schleife (z.B. `while`-Anweisung) durchläuft. Durch diese Einschränkung werden aber Programme mit mehreren Threads ausgeschlossen, da hier die Reihenfolge der Abarbeitung der Anweisungen nicht a-priori bestimmt werden kann.

4.1.1 Struktur des direkten Modells

Nachdem die grundlegende Funktionsweise der Konversion von Anweisungen und Variablen in Komponenten und Verbindungen veranschaulicht wurde, bleibt die Frage zu behandeln, in welcher Weise die einzelnen Anweisungen in Komponenten und Verbindungen umgewandelt werden können. Im folgenden werden für alle Klassen von Anweisungen wie Zuweisungen, Auswahlanweisungen, Methodenaufrufe, `return`-Anweisungen, Ausdrücke und `while`-Anweisungen der eingeschränkten Sprache Vorgehensweisen angegeben, wie eine solche Anweisung oder ein Teil derselben in Komponenten und Verbindungen umgewandelt werden kann.

- *Zuweisungen* an Variablen werden als Komponenten mit zwei Ports – einem Input und einem Output-Port – repräsentiert. Der Input-Port ist dem Ausdruck auf der rechten Seite

[2] Die Abbildung dient nur der Veranschaulichung des Beispiels und repräsentiert nicht das endgültige Modell, da Anweisungen durch einzelne Komponenten dargestellt sind. Im endgültigen Modell hingegen werden diese noch feiner in weitere Komponenten und Verbindungen unterteilt.

der Zuweisung zugeordnet. Dieser Port ist mit dem Output-Port `result` des Modells der rechten Seite der Zuweisung, welche einen Ausdruck repräsentiert, verbunden. Der Output-Port der Zuweisung ist mit der Variablen auf der linken Seite der Zuweisung assoziiert. Arbeitet die Komponente korrekt, wird der Wert des Input-Ports an den Output-Port weitergereicht.

- *Auswahlanweisungen* werden als Komponente mit mehreren Input- und Output-Ports modelliert, welche die Auswahl nachbildet. Zunächst besitzt die Komponente einen Input-Port `cond`, der dem Output-Port `result` des Ausdrucks der Bedingung zugeordnet ist.

  Weiters werden für jede Variable v, die in einer der beiden Zweige der Auswahlanweisung verändert wird, drei Ports generiert: die beiden Input-Ports `then_v` und `else_v`, sowie der Output-Port `out_v`. Die Input-Ports sind der Variablen v nach Auswertung des `then`- bzw. des `else`-Zweiges der Auswahlanweisung zugeordnet. Wird die Variable in einem der beiden Zweige nicht verändert, wird der entsprechende Input-Port mit jener Verbindung verbunden, welche der Variablen vor der Abarbeitung der Auswahlanweisung zugeordnet ist. Der Output-Port ist – analog zu Zuweisungen – mit der Variablen v assoziiert und wird in allen folgenden Anweisungen für die Variable v verwendet.

  Zusätzlich zu den veränderten Variablen müssen auch alle Instanzvariablen von in den beiden Zweigen der Bedingung neu erzeugten Objekten (z.B. durch `new`-Ausdrücke oder indirekt durch Methodenaufrufe) in die Modellierung miteinbezogen werden. Dies erfolgt wie soeben für den Fall der veränderten Variablen beschrieben. Wird ein Objekt nur in einem der beiden Zweige erzeugt, werden die mit den Instanzvariablen des erzeugten Objekts korrespondierenden Input-Ports des anderen Zweiges mit einem ausgezeichneten Wert assoziiert, der „kein Wert vorhanden“ oder auch „nicht initialisiert“ repräsentiert. Dies kann z.B. durch das Token `#no_value` repräsentiert werden.

- *Aufrufe von Methoden* werden durch eine Komponente repräsentiert, welche den Aufruf der entsprechenden Methode abstrahiert und die Effekte der Auswertung der Methode zusammenfaßt.

  Methoden lassen sich in zwei Klassen einteilen: (1) Methoden, die kein Ergebnis zurückliefern (ihr Typ des Ergebniswerts ist `void`) und (2) Methoden, die einen Ergebniswert berechnen (ihr Typ des Ergebniswerts ist ungleich `void`). Die beiden Arten von Methodenaufrufen sind sich ähnlich. Methoden ohne Ergebnis können jedoch nicht in Ausdrücken verwendet werden, sondern stellen stets eigenständige Anweisungen dar. Ein weiteres Unterscheidungsmerkmal zu anderen Arten von Anweisungen und Ausdrücken ist, daß Methoden (mit und ohne Ergebniswert) oftmals externe Variablen verändern. Mit externen Variablen werden hier Variablen bezeichnet, die nicht lokal in der Methode definiert sind und daher auch außerhalb der Methode oder in weiteren Methodenaufrufen sichtbar sind. Hierzu zählen alle Variablen, die dem Objekt zugeordnet sind, für welches die Methode aufgerufen werden soll. Weiters sind alle Variablen, welche Teil der Argumente des Methodenaufrufs darstellen, ebenfalls externe Variablen. Auch alle Klassenvariablen sind als externe Variablen anzusehen.

  Die den Methodenaufruf darstellende Komponente besitzt einen oder mehrere Input- und Output-Ports. Die Input-Ports der Komponente sind allen externen Variablen zugeordnet, welche in der entsprechenden Methode verwendet werden. Hierzu zählen *Instanzvariablen des Objekts*, für das der Methodenaufruf durchgeführt wird, *Klassenvariablen* und die *Argumente* des Methodenaufrufs. Die Output-Ports stellen die Nebeneffekte des Methodenaufrufs dar. Das sind alle durch den Methodenaufruf veränderten Variablen, deren Lebensdauer die Dauer des Methodenaufrufs übersteigt. Hierzu zählen insbesondere In-

stanzvariablen des Objekts und Variablen, die den Argumenten[3] des Methodenaufrufs zugeordnet sind bzw. welche über solche Variablen ansprechbar sind. Auch Instanzvariablen von durch Abarbeiten der Methode neu erzeugten Objekten zählen zu den Output-Ports. Bei Methoden mit Ergebnis existiert zusätzlich ein Output-Port `return`, welcher dem durch die Methode berechneten Wert zugeordnet ist.

Das Modell des Methodenaufrufs einer Methode m kann ermittelt werden, indem das Modell von m als Grundlage verwendet wird und darin alle Verbindungen von externen Variablen und Parametern (und allen dadurch erreichbaren Variablen) durch die Verbindungen der tatsächlich im Aufruf verwendeten externen Variablen und Parameter ersetzt werden. Das auf diese Weise erhaltene Modell entspricht jenem des Methodenaufrufs.

Das bisher beschriebene Modell für Methodenaufrufe funktioniert nur dann korrekt, wenn die zu modellierende Methode bereits zur Übersetzungszeit eindeutig bestimmt werden kann. Ist dies nicht der Fall, z.B. bei Vorhandensein einer Vererbungshierarchie mit mehreren konkreten Klassen mit Aufrufen von polymorphen Methoden, muß das Verfahren erweitert werden. Zunächst ist ein Mechanismus zu schaffen, um die zu repräsentierende Methode auswählen zu können. Um dies durchführen zu können, muß der Komponente die tatsächliche Klasse des durch die Variable angesprochenen Objekts durch einen zusätzlichen Input-Port `class` zugänglich gemacht werden. Durch diesen Port kann bei der Auswertung die korrekte Methode ausgewählt und deren Modell abgearbeitet werden. Dieser Port ist mit einer zusätzlichen, vom Übersetzer automatisch generierten Instanzvariablen der Klasse verbunden, welche einen Identifier der tatsächlichen Klasse des Objekts enthält. Diese Variable simuliert in gewisser Weise eine beschränkte Form des Reflection-API von Java. Hier ist zu bemerken, daß diese zusätzliche Input-Variable nur bei Aufrufen „normaler“ Methoden auftritt. Bei Aufrufen von Konstruktor-Methoden in `new`-Ausdrücken ist das nicht sinnvoll, da das Objekt erst durch den Aufruf des Konstruktors angelegt wird und darüber hinaus die jeweilige Klasse ohnedies aus dem `new`-Ausdruck entnommen werden kann. Auch bei Klassenmethoden ist dies nicht sinnvoll, da bei solchen Methodenaufrufen die entsprechende Klasse bereits zur Übersetzungszeit bestimmt werden kann.

Weiters können die benötigten und veränderten externen Variablen in den unterschiedlichen Methoden abweichen. Daraus ergibt sich, daß alle möglicherweise aufzurufenden Methoden berücksichtigt werden müssen. Das Modell des Methodenaufrufs wird somit als Kombination der einzelnen Modelle dieser Methoden gebildet. Die Menge der Input- bzw. Output-Ports der Methodenaufrufs-Komponente wird jeweils durch die Vereinigungsmenge der Input- bzw. Output-Ports der Komponenten der Modelle der einzelnen Methodenaufrufe gebildet. Hier ist zu beachten, daß für jeden Output-Port immer der korrekte Wert ermittelt werden können muß, unabhängig davon, welche Methode tatsächlich aufgerufen wird. Wird eine Variable durch eine Methode nicht verändert, muß deren Wert unverändert durchgereicht werden. Daraus ergibt sich, daß diese Variable zu der Menge der benötigten Input-Variablen hinzugefügt werden muß. Folglich muß auch ein Input-Port für diese Variable generiert werden. Auf diese Weise können Input-Ports für die Komponente entstehen, welche in keiner Methode benötigt werden, jedoch für die Korrektheit des Modells unerläßlich sind. Bei Instanzvariablen neu erzeugter Objekte ergeben sich ähnliche Effekte. In diesem Fall kann den entsprechenden Output-Ports ein ausgezeichneter Wert, wie z.B. `#no_value`, zugeordnet werden. Dies ist möglich, da Methodenaufrufe in der Folge möglicherweise eine Verbindung zu diesen Ports aufbauen. Aufgrund des Auswahlmechanismus

[3] Hier müssen nur Argumente mit nicht-primitivem Datentyp betrachtet werden, da diese als Referenz übergeben werden und daher etwaige Veränderungen auch außerhalb der Methode sichtbar sind. Primitive Datentypen hingegen werden als Kopie übergeben und eventuelle Veränderungen dieser Argumente sind nicht außerhalb der Methode sichtbar.

für Methoden werden aber in keinem Fall Methoden aufgerufen, welche diese Variablen tatsächlich benutzen. Die auf diese Weise erhaltene Komponente repräsentiert also alle möglichen Aufrufe einer der in Frage kommenden Methoden, sowie das Durchreichen nicht in jedem Fall benutzter Variablen.

- `return`-*Anweisungen* in Methoden werden durch Komponenten mit jeweils einem Input- und einem Output-Port dargestellt. Der Input-Port ist dem zu retournierenden Ausdruck zugeordnet und mit dessen `result`-Port verbunden. Der Output-Port `return` repräsentiert den zurückgelieferten Ausdruck.

  Hier ist zu beachten, daß das Modell in dieser Form nur eine `return`-Anweisung in einer Methode erlaubt, da Sprünge im Programmfluß (wie `return`, `break`, `continue` oder Exceptions) nicht dargestellt werden können. Daher würde der Programmfluß nach der `return`-Anweisung normal bis zum Ende der Methode weiterlaufen. In der Folge könnten unerwünschte Effekte entstehen, wenn dadurch Nebeneffekte auftreten, welche auch außerhalb der Methode oder in folgenden Aufrufen der Methode sichtbar wären. Darüber hinaus ist die Wahl des zu retournierenden Wertes unter Umständen nicht mehr eindeutig.

  Um die Beschränkung auf eine einzige `return`-Anweisung pro Methode zu umgehen, muß ein Mechanismus geschaffen werden, um alle nachfolgenden Anweisungen, welche Werte von lokalen Variablen verändern, Nebeneffekte aufweisen oder `return`-Anweisungen darstellen, zu unterdrücken. Dies kann durch eine lokale Hilfsvariable `return` erfolgen, welche jeder solchen Komponente als Input-Variable hinzugefügt wird. Die Verhaltensbeschreibung der Komponenten muß dahingehend verändert werden, daß – falls am Input-Port `return_in` bereits ein Wert anliegt – alle weiteren Effekte der Komponente unterdrückt werden und anstelle der berechneten Werte an den Output-Ports die ursprünglichen Werte vor der Ausführung der Anweisung ausgegeben werden. Dadurch werden alle der ersten `return`-Anweisung nachfolgenden Anweisungen unterdrückt. An dieser Stelle sei darauf hingewiesen, daß eventuell zusätzliche Input-Ports zu den Komponenten hinzugefügt werden müssen, welche den ursprünglichen Werten der mit den Output-Ports assoziierten Variablen zugeordnet sind.

- *Ausdrücke* sind Sprachelemente, welche als Teile von Anweisungen auftreten können, jedoch selbst keine vollständige Anweisung darstellen. Eine Ausnahme bilden Funktionsaufrufe, die durch Ignorieren der Ergebniswerte auch als Methodenaufrufe – und somit als vollständige Anweisung – angesehen werden können.

  Ausdrücke werden durch Komponenten mit mehreren Ports dargestellt, wobei immer mindestens ein Output-Port existiert. Die Input-Ports repräsentieren die benötigten Unter-Ausdrücke (oder Argumente bei Funktionsaufrufen), die Output-Ports repräsentieren die durch die Abarbeitung des Ausdrucks verursachten Nebeneffekte. Mit Ausnahme von Funktionsaufrufen können in der eingeschränkten Sprache keine Nebeneffekte auftreten. Weiters existiert ein Output-Port `result`, der mit dem Ergebnis des Ausdrucks assoziiert ist. Dieser wird z.B. bei Zuweisungsanweisungen mit der Komponente, welche die Zuweisung repräsentiert, verbunden.

  Da die hier verwendete eingeschränkte Sprache keine Exceptions und auch keine Nebeneffekte in booleschen Ausdrücken zuläßt, werden Ausdrücke immer bis zuletzt ausgewertet und ein Ergebniswert berechnet. Tritt während der Auswertung ein Fehler auf, z.B. eine Division durch Null, wird ein Fehlerwert (z.B. das Token `#error`) weitergegeben.

  Ausdrücke lassen sich in vier Kategorien unterteilen: *Konstanten*, *Variablen*, *Operatoren* und *Funktionsaufrufe*.

  - *Konstanten* werden durch Komponenten mit einem Output-Port `result` dargestellt. Der Port repräsentiert das Ergebnis des Ausdrucks, welches dem Wert der Konstanten entspricht. Die Aufgabe der Komponente ist es, den Wert der im Programm angegebene Konstanten an den Output-Port weiterzureichen.
  - *Variablen* werden in ähnlicher Weise wie Konstanten modelliert, es ist aber ein zusätzlicher Input-Port notwendig. Es wird eine Komponente mit einem Input- und einem Output-Port verwendet. Der Input-Port ist hier mit der zu modellierenden Variablen assoziiert. Der Output-Port `result` ist wieder mit dem Ergebnis des Ausdrucks assoziiert. Bei korrekter Funktionsweise der Komponente wird der Wert der Variablen, der am Input-Port anliegt, an den Output-Port durchgereicht – und umgekehrt.
  - Die in der eingeschränkten Sprache verfügbaren *Operatoren* sind unäre sowie binäre Operatoren. Prefix- und Postfix-Operatoren werden nicht unterstützt, obwohl diese problemlos durch Kombination mehrerer einfacherer Elemente nachgebildet werden können. Da die hier betrachteten Operatoren neben dem berechneten Ergebnis keinerlei Nebeneffekte verursachen, können diese durch Komponenten mit ein- oder zwei Input-Ports und einem Output-Port dargestellt werden.

    Unäre Operatoren werden durch Komponenten mit jeweils einem Input-Port `in` und einem Output-Port `result` dargestellt. Der Input-Port ist mit dem `result`-Port der Komponente verbunden, welche den Ausdruck modelliert, auf den der Operator angewendet wird. Der Output-Port ist dem Ergebnis der Anwendung des Operators auf den Ausdruck zugeordnet.

    Binäre Operatoren werden als Komponenten mit jeweils zwei Input-Ports modelliert. Die Input-Ports `in1` und `in2` sind mit den `result`-Ports der Komponenten der beiden Ausdrücke verbunden, auf welchen der Operator angewendet wird. Der Output-Port `result` ist dem Ergebnis aus der Auswertung des Operators, angewendet auf die beiden Ausdrücke, zugeordnet.
  - Schließlich kann auch ein *Aufruf einer Funktion oder eines Konstruktors* (durch einen `new`-Ausdruck) einen Ausdruck darstellen. Ein Funktionsaufruf bzw. Konstruktoraufruf wird analog zu einem Methodenaufruf modelliert, der resultierenden Komponente wird der Output-Port `result` hinzugefügt. Dieser ist dem Ergebnis der Auswertung der Funktion zugeordnet und entsteht durch das Umbenennen des `return`-Ports der den Funktionsaufruf modellierenden Komponente. Im Fall eines Konstruktoraufrufs durch einen `new`-Ausdruck kann der Port `result` nicht durch Umbenennen des Ports `return` des Modells des aufgerufenen Konstruktors gewonnen werden, da Konstruktoren kein Ergebnis zurückliefern und daher auch keinen Port `return` aufweisen. Stattdessen ist dem Port `result` das gesamte, neu erzeugte Objekt zugeordnet.

    Stellt der Ausdruck einen `new`-Ausdruck dar, sind der Komponente Output-Ports für alle neu angelegten Instanzvariablen hinzuzufügen. Weiters muß in einigen Fällen auch ein zusätzlicher Output-Port `class` generiert werden, welcher die tatsächliche Klasse des erzeugten Objekts speichert. Diese Information kann für spätere Methodenaufrufe benötigt werden, falls die aufzurufende Methode nicht bereits zur Übersetzungszeit eindeutig bestimmt werden kann. Dem generierten Port ist ein Identifier der Klasse zugeordnet, anhand der eine Methodenaufrufs-Komponente später die korrekte Methode auswählen kann.

- *`while`-Anweisungen* werden (ähnlich wie bei Methodenaufrufen) durch jeweils eine Komponente dargestellt, welche die Effekte der gesamten Schleife zusammenfaßt. Wie bei Methodenaufrufen besitzt jede Komponente einen oder mehrere Input- und Output-Ports. Die

Input-Ports repräsentieren alle in der Schleifenbedingung oder im Schleifenrumpf benötigten Variablen: für jede benötigte Variable v wird ein Input-Port `in_v` generiert. Die Output-Ports sind mit den im Schleifenrumpf veränderten Variablen assoziiert. Analog zu den Input-Ports wird für jede veränderte Variable v ein Output-Port `out_v` angelegt. Hier ist zu beachten, daß der Komponente in einigen Fällen auch zusätzliche Input-Ports hinzugefügt werden müssen, welche weder in der Schleifenbedingung noch im Schleifenrumpf benötigt werden. Dies ist notwendig, da der Fall eintreten kann, daß der Schleifenrumpf kein einziges Mal abgearbeitet wird und daher anstelle der durch den Schleifenrumpf berechneten Werte jene Werte an die Output-Ports weitergereicht werden müssen, welche vor der Schleifen-Anweisung gültig waren. Diese Werte müssen der Komponente durch zusätzliche Input-Ports zugänglich gemacht werden. Bei Schleifen-Anweisungen ist also stets sicherzustellen, daß die Menge der mit den Output-Ports assoziierten Variablen eine Teilmenge der mit den Input-Ports assoziierten Variablen ist.

Als weitere Voraussetzung wird wiederum angenommen, daß während der Auswertung der Schleifenbedingung keine Nebeneffekte auftreten. Dies stellt keine allzu starke Einschränkung dar, da solche in der Praxis nicht allzu häufig vorkommen und darüber hinaus jede Schleife mit Nebeneffekten in der Bedingung durch Aufteilen der Bedingung auf mehrere Anweisungen und Einführen von temporären Variablen in eine Schleife mit nebeneffektfreier Bedingung (und einigen zusätzlichen Anweisungen vor der Schleife und am Ende des Schleifenrumpfs) transformiert werden kann.

Für die Bedingung und den Schleifenrumpf werden eigenständige Modelle angelegt. Dies erfolgt wie bisher in den vorangegangenen Abschnitten für Anweisungen beschrieben. Im weiteren wird angenommen, daß für die Schleifenbedingung ein Modell M_C existiert und für den Schleifenrumpf ein Modell M_B. Weiters wird angenommen, daß für jede Variable v, welche in M_C verwendet wird, ein entsprechender Input-Port `ci_v` der Komponente, welche M_C repräsentiert, existiert. Weiters wird ein Output-Port `cond` angenommen, welcher dem Ergebnis der Auswertung der Bedingung zugeordnet ist. Für den Schleifenrumpf gilt ähnliches: hier werden Input-Ports `bi_v` vorausgesetzt, welche die benötigten Variablen repräsentieren. Schließlich werden hier Output-Ports `bo_v` vorausgesetzt, welche den im Schleifenrumpf veränderten Variablen zugeordnet sind. Hier ist anzumerken, daß die Menge der im Schleifenrumpf veränderten Variablen (Ports `bo_*`) äquivalent zu der Menge der mit den Output-Ports der `while`-Komponente assoziierten Variablen ist (unter der Annahme, daß in der Bedingung keine Nebeneffekte auftreten).

Das hier vorgestellte Modell eignet sich gut zur Diagnose von Methoden, in welchen vorwiegend primitive Datentypen verwendet werden. Auch einfache Methoden mit nicht-primitiven Datentypen lassen sich mit diesem Modell gut darstellen. Abbildung 4.2 zeigt ein Beispiel eines Java-Programms, das eine iterative Berechnung durchführt. Es berechnet einen Punkt der Mandelbrot-Menge. Weiters ist das zugehörige Komponentenmodell für die Klassenmethode `iterate(double,double)` dargestellt (Abbildung 4.3). Die Modelle für die anderen Methoden der Klasse sind nicht dargestellt, da deren Aufbau sehr einfach ist und daher auf die Darstellung verzichtet wird. Auch das Modell des Schleifenrumpfs ist in der Abbildung nicht dargestellt. Zum Modell ist zu bemerken, daß das zusätzliche Attribut `class` der Klasse `Complex` nicht notwendig ist, da dieses Beispiel nur aus einer einzigen Klasse besteht und daher die auszuwertende Methode stets bekannt ist. Zur Verdeutlichung des Konzepts ist die Variable jedoch trotzdem in der Graphik dargestellt.

Abschließend kann gesagt werden, daß dieses direkte Modell für Programme, die nur auf primitiven Datentypen operieren, sehr leistungsfähig ist. Bei Programmen, welche mit Klassen arbeiten und eventuell sogar verschachtelte Datenstrukturen behandeln, werden die Grenzen des Modells jedoch bald offensichtlich.

```
1  public class Complex {
2      double re, im;
3      Complex(double r, double i) {
4          re = r;
5          im = i;
6      }
7      double getRe() { return re; }
8      double getIm() { return im; }
9      void square() {
10         double r = re * re - im * im;
11         im = 2 * re * im;
12         re = r;
13     }
14     void add(Complex c) {
15         re = re + c.getRe();
16         im = im + c.getIm();
17     }
18     double abs() {
19         return re * re + im * im;
20     }
21     public static int iterate(double r, double i) {
22         Complex c = new Complex(r,i);
23         Complex z = new Complex(r,i);
24         int it = 100;
25         while ((it > 0) && (z.abs() <= 4)) {
26             z.square();
27             z.add(c);
28             it = it - 1;
29         }
30         return it;
31     }
32 }
```

Abbildung 4.2: Programm zur Illustration der Struktur des direkten Modells

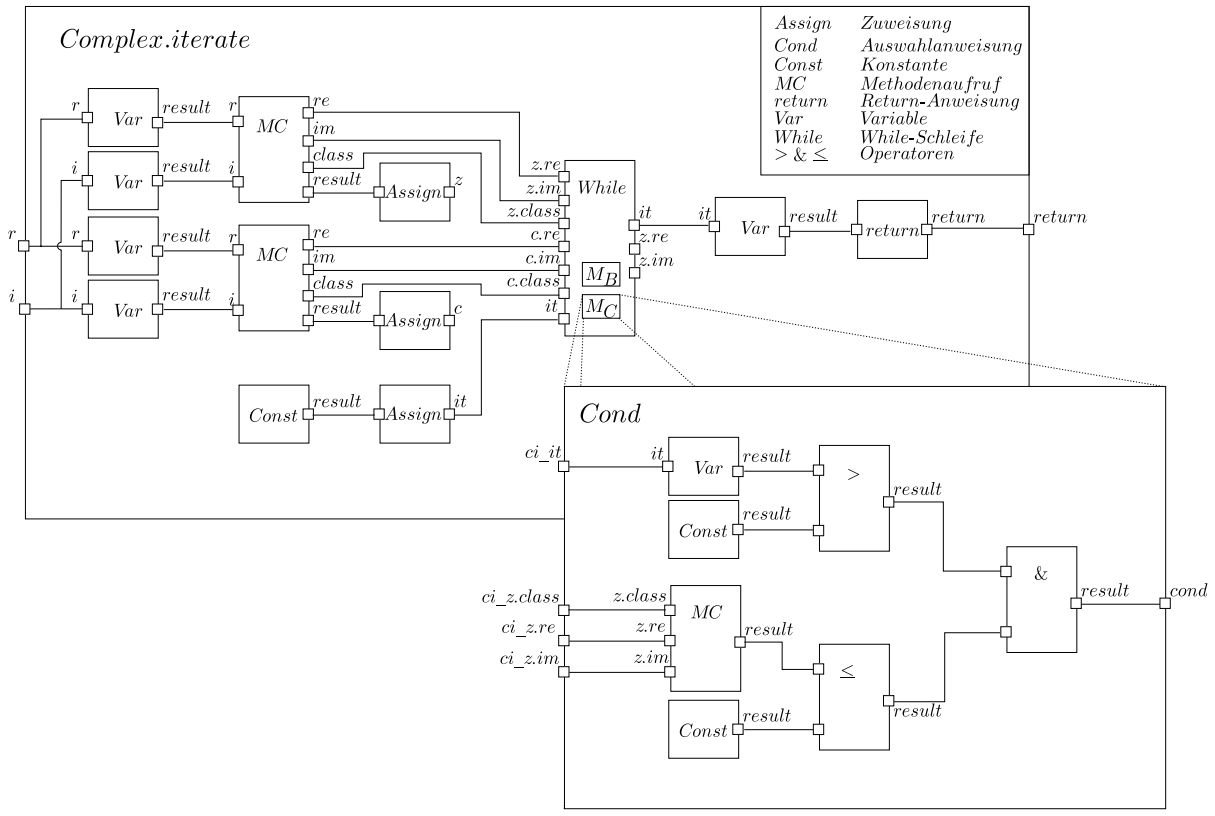

Abbildung 4.3: Partielles Modell der Methode `iterate(double,double)` aus dem Programm in Abbildung 4.2

Im nächsten Abschnitt werden einige Probleme des Modells und mögliche Lösungsansätze aufgezeigt.

4.1.2 Grenzen des Modells

Zur Illustration betrachte man das Programm in Abbildung 4.4. Dieses Beispiel kann mit dem oben behandelten Modell nicht modelliert werden. Man betrachte das partielle Modell in der Abbildung 4.5. Die Modellierung verläuft problemlos, bis vor Zeile 23. An dieser Stelle benötigt das Modell des Methodenaufrufs von `getValue()` die beiden Variablen `vh.class`[4] und `vh.value`. Aufgrund der Struktur des bisher gebildeten Modells kann zu diesem Zeitpunkt nicht eindeutig festgestellt werden, welche Ports mit diesen beiden Variablen assoziiert sind. Dies wird durch die Zuweisungen in der Auswahlanweisung verursacht, welche den Wert der nicht-primitiven Variablen `vh` verändern. Nach der Auswahlanweisung kann zwar der mit `vh` assoziierte Port (der Output-Port der Komponente, welche die Auswahlanweisung repräsentiert) eindeutig bestimmt werden, für andere Variablen, welche über diese Referenz angesprochen werden, ist das nicht möglich. Daher lassen sich keine geeigneten Verbindungen für die Input-Ports des darauffolgenden Methodenaufrufs finden.

```
1  class ValueHolder {
2      int value;
3      ValueHolder(int v) {
4          value = v;
5      }
6      int getValue() { return value; }
7      void setValue(int v) { value = v; }
8  }
9
10 public class ImpossibleConditionSample {
11     public static void demo(boolean cond) {
12         ValueHolder vh;
13         ValueHolder vh1 = new ValueHolder(1);
14         ValueHolder vh2 = new ValueHolder(2);
15         int v;
16
17         if (cond) {
18             vh = vh1;
19         }
20         else {
21             vh = vh2;
22         }
23         v = vh.getValue();
24     }
25 }
```

Abbildung 4.4: Nicht mit dem direkten Modell darstellbares Programm

Die Lösung dieses Problems kann auf drei Arten erfolgen:

(a) Es kann versucht werden, bei Zuweisungen in Auswahlanweisungen, `while`-Schleifen und Methoden nicht nur Ports für die veränderten Variablen zu generieren, sondern auch für alle über diese Referenzen ansprechbaren Variablen.

[4] Diese Variable kann hier weggelassen werden, da – wie in dem vorangegangenen Beispiel – die Klasse `ValueHolder` keine Subklassen aufweist.

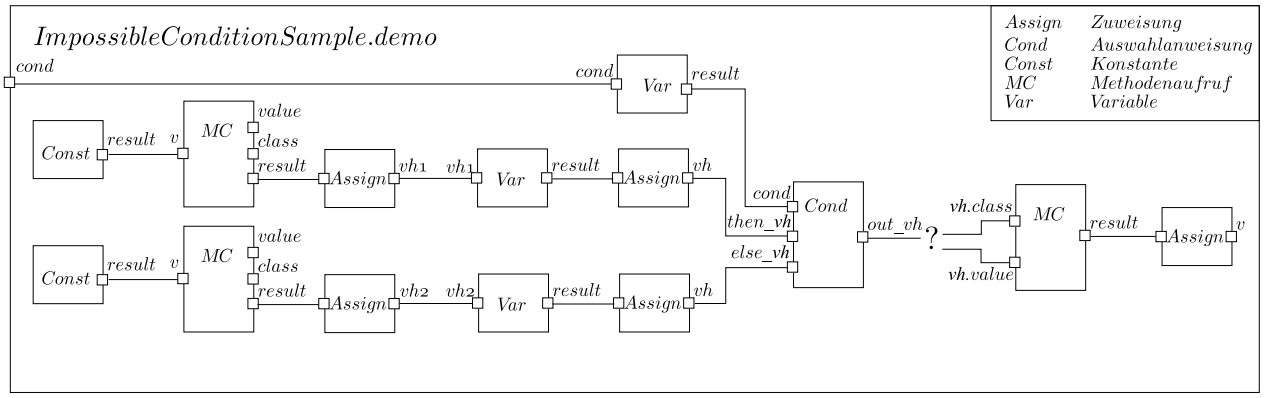

Abbildung 4.5: Partielles Modell des Programms aus Abbildung 4.4

Diese Lösung ist nur dann anwendbar, wenn die Anzahl der Objekte und deren Verhältnis zueinander (z.B. Baum, DAG, etc.) zur Zeit der Modellbildung bereits bekannt sind. Dies verbietet also weitgehend die Anwendung von dynamischen Datenstrukturen und schränkt die zu modellierenden Programme stark ein. Auch rekursive Methoden können auf diese Weise nicht modelliert werden, da die benötigten Input- und Output-Verbindungen nicht in jedem Fall bestimmt werden können (z.B. rekursive Datenstrukturen). Durch diese Methode werden sehr viele Ports und Verbindungen generiert, woraus sich viele Abhängigkeiten zwischen den Komponenten ergeben. Schließlich bleibt auch die Frage ungelöst, auf welche Weise die benötigten Informationen über die Objekte und ihre Verhältnisse zueinander in den Modellbildungsprozeß einfließen können. Hier müßten eventuell Anfragen an den Benutzer gestellt werden.

(b) Verbindungen könnten dynamisch hinzugefügt werden, sobald eine Komponente eine Verbindung für eine Variable erfordert, für die kein eindeutiger Port identifiziert werden kann. Es müßte eine neue Verbindung „von hinten nach vorne“ in die vorhergehenden Komponenten eingefügt werden, bis eindeutige Ports für alle benötigten Variablen gefunden werden können. Unglücklicherweise ist das nicht immer möglich (z.B. bei `while`-Schleifen). Auch diese Lösungsmöglichkeit kann nicht bei rekursiven Datenstrukturen und Methoden angewendet werden.

(c) Bei nicht eindeutig auflösbaren Input-Verbindungen für Variablen wird so weit im Referenzpfad zurückgegangen bis eine eindeutige Identifizierung der zugehörigen Verbindung möglich ist. Als nachteilig anzusehen ist die Tatsache, daß bei dieser Art der Verbindungsbildung das Verhalten der Komponenten komplexer ausfällt, da hier noch zusätzlich die für die Komponente relevante Variable extrahiert werden muß. Auch wird durch solche Verbindungen das Zurückpropagieren von Werten entlang der Verbindungen unmöglich. Wie die beiden vorhergehenden Lösungsansätze sind auch hier rekursive Methoden nicht möglich. Es kann aber unter Anwendung von k-Limiting (siehe unten) eine näherungsweise Lösung gefunden werden, was eine Einbuße an Qualität der Diagnosen zur Folge hat.

Ein weiteres Problem stellen mittelbar oder unmittelbar rekursive Methoden dar. Bei der Modellierung solcher Konstrukte stößt das Modell gleich zweifach an seine Grenzen. Einerseits können die benötigten Input- und Output-Verbindungen bei rekursiven Datenstrukturen nicht eindeutig bestimmt werden, andererseits kann das Verhalten des Modells in solchen Fällen nicht mit einer endlichen Menge von Komponenten beschrieben werden, da jede Komponente, welche einen rekursiven Methodenaufruf repräsentiert, wiederum einen Aufruf der Methode und damit wiederum eine Komponente enthält.

Das Problem der nicht eindeutig bestimmbaren Input- und Output-Verbindungen kann näherungsweise gelöst werden, indem die benötigten und veränderten Variablen durch ihren Referenzpfad identifiziert werden. Der Referenzpfad zu der Variablen wird nach einer bestimmten Pfadlänge abgebrochen, und es wird angenommen, daß sämtliche Variablen, welche über Referenzpfade der veränderten Variablen der letzten analysierten Länge erreichbar sind, durch den Methodenaufruf verändert werden. Dies entspricht im wesentlichen dem in der Literatur verbreiteten Ansatz des *k-Limiting* [Deu94]. Dieses im weiteren beschriebene Verfahren umgeht zugleich auch die Beschränkungen der zuvor behandelten Beispiele und Lösungsansätze.

Eine ähnliche Vorgehensweise ist bei den Input-Verbindungen möglich, wobei hier auch dann Referenz-Pfade auf Variablen eliminiert werden können, falls deren Pfad eine Erweiterung eines bereits in der Menge der mit Input-Verbindungen assoziierten Variablen enthaltenen Referenzpfades darstellt.

Durch diese Vorgehensweise verändert sich die Semantik der Input- und Output-Verbindungen. Ein Output-Referenzpfad, welcher durch k-Limiting abgebrochen wurde, hat die Semantik: „Die mit dem Pfad assoziierte Variable und alle Variablen, welche durch Erweiterung des Referenzpfades erreichbar sind, werden verändert.“ Für Input-Referenzpfade gilt ähnliches: „Es werden die durch den Referenzpfad gekennzeichnete Variable benötigt und alle durch Erweiterung des Referenzpfades ansprechbaren Variablen.“

Zusätzlich zu den durch das Modell der Methode bestimmten Input-Verbindungen sind in einigen Fällen durch die veränderte Semantik auch noch weitere Input-Verbindungen notwendig: zu den vom Modell des Methodenaufrufs benötigten Referenzpfaden sind all jene Referenzpfade als Input-Verbindungen hinzuzufügen, die durch Erweiterung eines der geforderten Referenzpfade gebildet werden können und deren zugehörige Variable im Kontext des Methodenaufrufs verändert wird. Dies gilt auch für `while`-Anweisungen, da diese Komponenten durch ihre hierarchische Struktur ähnlich zu Methodenaufrufen ebenfalls komplexere Abläufe zusammenfassen. Weiters müssen alle durch k-Limiting abgebrochenen Referenzpfade der mit Output-Verbindungen assoziierten Variablen eines Methodenaufrufs auch als Input-Referenzpfade gefordert werden. Andernfalls können Konflikte bei der Zusammenführung von zwei durch Methodenaufrufe veränderten Variablenmengen auftreten, da nicht festgestellt werden kann, welche Werte aktuell sind. Durch die zusätzlichen Input-Verbindungen wird eine Abhängigkeit zwischen den Methodenaufrufen erzwungen und damit das Problem der Zusammenführung vermieden.

Ein Spezialfall des hier beschriebenen Verfahrens ist, immer nur Referenzpfade der Länge eins zuzulassen. Dadurch wird in einigen Fällen die Qualität der Diagnosen herabgesetzt, da viele zusätzliche Abhängigkeiten zwischen den Komponenten eingeführt werden. Auch können entlang solcher Verbindungen nicht in jedem Fall Werte von den Output-Ports zu den Input-Ports zurückpropagiert werden. Dies verursacht in einigen Fällen unpräzise Diagnosen, die stark herabgesetzte Komplexität der Verfahrensweise rechtfertigt dies allerdings.

Beispiel 4.2 Das in Abbildung 4.4 dargestellte, im ursprünglichen direkten Modell nicht darstellbare (siehe Abbildung 4.5), Java-Programm kann unter Anwendung des erweiterten Verfahrens in das in Abbildung 4.6 dargestellte Modell konvertiert werden.

Da dieser Lösungsansatz die bisherigen Probleme weitgehend umgeht und die Komplexität der Berechnung relativ gering ist, wird dieses veränderte Modell im weiteren als das *erweiterte direkte Modell* betrachtet. Das ursprüngliche Modell ist aufgrund seiner Beschränkungen zwar für sehr einfache Programme geeignet, interessante Java-Programme – insbesondere solche mit objektorientierten Sprachelementen – können damit aber nicht analysiert werden.

Das bisher vorgestellte Modell und alle Varianten und Lösungsansätze setzen voraus, daß sämtliche Datenstrukturen azyklisch sind, da sonst nicht eindeutig festgestellt werden kann, welche Variablen durch eine Zuweisung verändert werden. Eben dieses Problem tritt dann

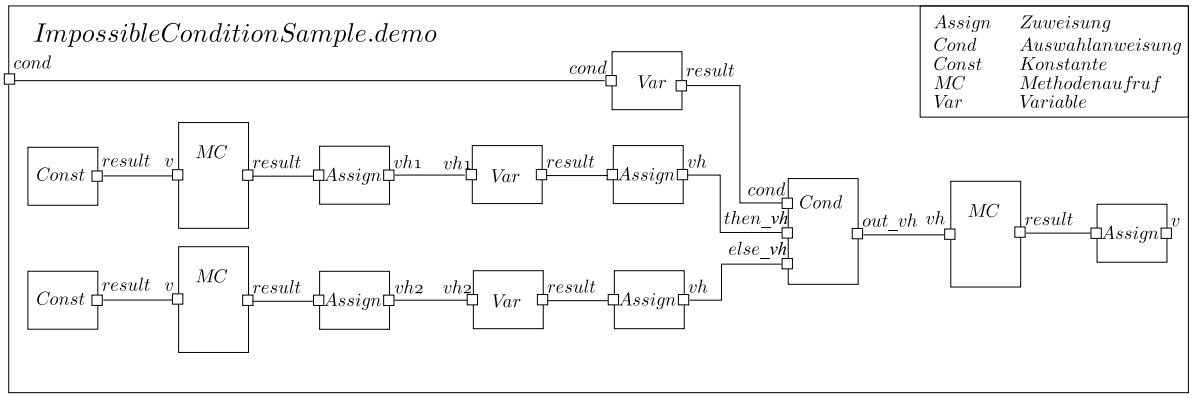

Abbildung 4.6: Erweitertes Modell des Programms aus Abbildung 4.4

auf, wenn zwei oder mehrere Referenzpfade auf ein und dieselbe Variable verweisen. Dies hat zur Folge, daß, sobald die Variable über einen dieser Referenzpfade verändert wird, auch alle anderen Referenzpfade als verändert angesehen werden müssen. Andernfalls entspricht das Modell nicht mehr der Semantik der Java-Sprache, und es können falsche Diagnosen entstehen oder offensichtliche Diagnosen nicht gefunden werden.

Beispiel 4.3 Als Beispiel für Aliasing zwischen zwei Referenzpfaden sei das in Abbildung 4.7 dargestellte Programm und sein Modell in Abbildung 4.8 angeführt.

```
1  class List {
2      int value;
3      List next;
4      List(List nxt) {
5          value = 1;
6          next = nxt;
7      }
8      int getValue() { return value; }
9      void setValue(int v) { value = v; }
10     List getNext() { return next; }
11 }
12
13 public class AliasingSample {
14     public static void demo() {
15         List l1 = new List(null);
16         List l = new List(l1);
17         int v;
18         while(l.getNext() != null) {
19             l = l.getNext();
20         }
21         l.setValue(2);
22         v = l1.getValue();
23     }
24 }
```

Abbildung 4.7: Programm mit Aliasingeffekten

Das Beispiel verdeutlicht einen Fall von Aliasing [Deu94, Ghi98] zwischen der Variablen `l` und `l1`. Betrachtet man das Programm anhand der Semantik der Java-Sprache, ist es ohne Belang ob die Instanzvariable des in Zeile 15 angelegten Objekts durch die Variable `l` oder `l1` verändert

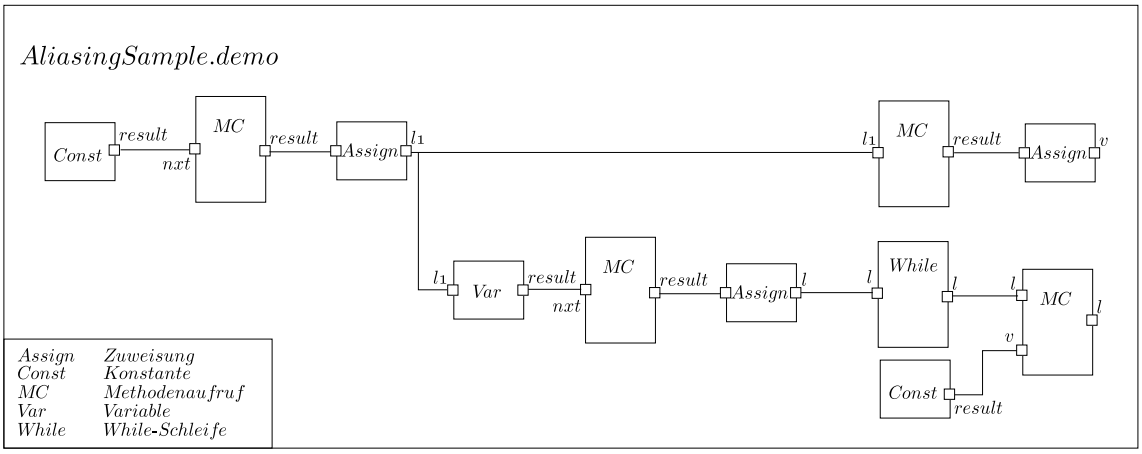

Abbildung 4.8: Falsches Modell des Programms aus Abbildung 4.7, verursacht durch Aliasing zwischen den Variablen `l` und `l1` ab Zeile 21

wird. Das Modell hingegen arbeitet zunächst rein syntaktisch und betrachtet daher `l` und `l1` als unterschiedliche Variablen. Unter dieser Annahme ergibt sich das in Abbildung 4.8 dargestellte Modell. Dieses korrespondiert aber nicht mit dem Programm, da im Modell die Abhängigkeit zwischen den Variablen `l` und `l1` nicht gegeben ist. Im Modell drückt sich dies durch die Tatsache aus, daß die Komponente, welche den Methodenaufruf von `setValue` repräsentiert, zwar die Variable `l` verändert, nicht jedoch `l1`. Folglich wird bei der Generierung des Modells für den Methodenaufruf von `getValue()` die falsche Verbindung für `l1` verwendet. Daraus ergibt sich die Diskrepanz zwischen Modell und Java-Semantik.

Um diese Probleme zu beseitigen, muß die Modellbildung von einer Aliasing-Analyse begleitet werden. Nach der Modellierung einer neuen Komponente muß jeweils geprüft werden, ob sich neben den durch das Modell berechneten Effekten der Komponenten noch weitere, zusätzliche Effekte durch das Vorhandensein von Aliasing zwischen Referenzpfaden ergeben. Da eine exakte Analyse, ob zwei Referenzpfade auf dieselbe Variable verweisen, im allgemeinen nicht berechenbar ist [Hor97, Lan92], muß eine näherungsweise Lösung für das Problem ermittelt werden. In der Literatur existieren eine Reihe von Analyseverfahren, welche dies bewerkstelligen. Eine Übersicht findet sich in [Ghi98].

Im folgenden wird die jeweilige Vorgehensweise im Detail behandelt. Nach dem Erzeugen einer Komponente muß für alle den Output-Ports der Komponente zugeordneten Referenz-Pfade geprüft werden, ob diese möglicherweise in einer Alias-Relation mit einem anderen Referenzpfad stehen [BC92]. Bei dieser Untersuchung sind drei mögliche Ergebnisse zu unterscheiden:

(a) *Zwei Referenzpfade verweisen mit Sicherheit auf dieselbe Variable.* Dies bewirkt, daß die beiden Referenzpfade in der Modellbildung als Synonyme betrachtet werden können: wird eine Variable über einen Referenzpfad verändert, muß auch der andere als verändert angesehen werden.

(b) *Zwei Referenzpfade verweisen möglicherweise auf dieselbe Variable.* In diesem Fall kann erst zur Laufzeit anhand konkreter Werte für die Objekte festgestellt werden, ob die beiden Referenzpfade auf dieselbe Variable verweisen. Folglich müssen bei der Modellbildung beide Möglichkeiten in Betracht gezogen werden.

Bilden die beiden Referenzpfade tatsächlich ein Alias-Paar, wird wie im vorhergehenden Fall verfahren, d.h. es werden beide Referenzen als verändert angesehen. Andernfalls verweisen die Referenzen auf unterschiedliche Variablen und können ihre Werte daher auch unabhängig

voneinander ändern. Daher wird in diesem Fall nur die wirklich veränderte Referenz als modifiziert angesehen. Die andere hingegen bleibt unverändert.

Da bei der Modellbildung angenommen werden muß, daß die beiden Referenzpfade ein Alias-Paar sind, müssen für ersteren Fall keine zusätzlichen Vorkehrungen getroffen werden. In letzterem Fall hingegen muß der Variablen ihr unveränderter Wert zugeordnet werden können. Um diesen zu erhalten, ist der Komponente die entsprechende Variable als zusätzliche Input-Variable zuzuordnen. Dadurch kann stets der aktuelle Wert der Variablen ermittelt werden und ggf. an den Ausgang weitergereicht werden. Dies erfolgt analog zu der Vorgehensweise bei `while`-Schleifen und Methodenaufrufen mit Vererbung und Polymorphismus.

(c) *Die beiden Referenzpfade verweisen keinesfalls auf dieselbe Variable.* In diesem Fall können die beiden Variablen als völlig unabhängig betrachtet werden. Es müssen daher keine zusätzlichen Vorgehensweisen in die Modellbildung einfließen.

Mit den soeben angeführten Überlegungen sei die Beschreibung der Bildung der Modellstruktur abgeschlossen. Im folgenden wird das Modell der einzelnen Komponententypen behandelt, d.h. die Verhaltensbeschreibung der einzelnen Komponenten näher betrachtet.

4.1.3 Verhaltensbeschreibung der Komponenten des direkten Modells

Um das in den bisherigen Abschnitten behandelte Modell zur modellbasierten Diagnose einsetzen zu können, muß neben der Struktur des Modells auch eine Beschreibung des korrekten Verhaltens (und ggf. auch der Effekte der Verhaltensmodi im Fehlerfall) der einzelnen Komponenten des Modells angegeben werden.

Im Gegensatz zur Struktur des Modells wird die Beschreibung des Verhaltens nicht durch das gegebene Programm, sondern durch die Semantik der Java-Sprache bestimmt, da die in diesem Modell verwendeten Komponenten jeweils einzelne Sprachelemente repräsentieren. Als Ausnahme sind hier Auswahlanweisungen, Methodenaufrufe und `while`-Anweisungen anzuführen, da die Verhaltensbeschreibung dieser Komponenten eine Zusammenfassung der Zweige der Auswahlanweisung, der einzelnen Elemente der aufgerufenen Methode bzw. des Schleifenrumpfs darstellt und daher nicht ausschließlich durch die Semantik der Sprache bestimmt ist. Daher kann das Verhalten der Modellkomponenten bereits beim Entwurf des Modells vollständig spezifiziert werden.

In diesem Abschnitt wird das korrekte Verhalten der Modellkomponenten, sowie das Verhalten derselben in bestimmten Fehlermodi behandelt. Die Beschreibung des Verhaltens erfolgt in Prädikatenlogik, wobei das Prädikat $ab(C)$ den Fall darstellt, daß sich die Komponente C nicht entsprechend der vom Programmierer intendierten Semantik verhält ('*ab*' steht für *abnormal*). Für den Debuggingprozeß bedeutet diese Interpretation, daß das durch die Komponente repräsentierte Programmelement als falsch anzusehen ist. Arbeitet die Komponente C entsprechend der vom Programmierer intendierten Semantik, kann dies durch $\neg ab(C)$ dargestellt werden. Weiters werden Funktionen definiert, mit deren Hilfe die an den Ports der Komponente anliegenden Werte bestimmt werden können. Der am Port p einer Komponente C anliegende Wert wird durch die Funktion $p(C)$ repräsentiert. Diese Repräsentation entspricht der in Kapitel 3 verwendeten.

- Das Verhalten von Komponenten, welche *Zuweisungs-Anweisungen* repräsentieren, ist durch folgenden logischen Ausdruck gegeben:

$$\neg ab(C) \Rightarrow \texttt{out}(C) = \texttt{in}(C),$$

wobei C eine Komponente darstellt, welche eine Zuweisung an eine Variable repräsentiert. Da der Input-Port `in` der Komponente mit dem auf der rechten Seite der Zuweisung stehenden Ausdruck verbunden ist und der Output-Port `out` mit der Variablen auf der linken Seite der Zuweisung (d.h. der Zielvariablen der Zuweisung) assoziiert ist, kann das Verhalten der Komponente auf folgende Weise interpretiert werden: Falls angenommen werden kann, daß die Komponente korrekt funktioniert, wird der Variablen der Wert des in der Zuweisung angegebenen Ausdrucks zugewiesen. In diesem Modell wird dies durch die von obigem Ausdruck geforderte Gleichheit der Werte zwischen Input- und Output-Port der Komponente ausgedrückt. Weiters ist anzumerken, daß bei Zuweisungen keine weiteren (Fehler-)Modi betrachtet werden, d.h. die Komponente verhält sich entweder entsprechend ihrer Spezifikation oder in einem unbekannten Fehlermodus, über den keine Aussagen getroffen werden können.

- Das Verhalten von Komponenten, welche *Auswahlanweisungen* repräsentieren, ist dem von Zuweisungsanweisungen ähnlich: Es werden die Werte zwischen den entsprechenden Input-Ports `then_v` bzw. `else_v` und den Output-Ports `out_v` weitergereicht, in Abhängigkeit des am Input-Port `cond` anliegenden Wertes. Ist der Wert wahr, d.h. die mit der Auswahlanweisung assoziierte Bedingung ist erfüllt, dann werden die Werte der Ports `then_v` und `out_v` gleichgesetzt. Ist die Bedingung nicht erfüllt, werden die Werte der Ports `else_v` und `out_v` gleichgesetzt. Ist nicht bekannt, zu welchem Wert die Auswertung der Bedingung führt, kann keine Aussage über die Gleichheit zwischen den Input- und Output-Ports getroffen werden. In Prädikatenlogik kann das Verhalten von Auswahlanweisungen also durch die beiden Implikationen

  $$\neg ab(C) \wedge \texttt{cond}(C) = true \Rightarrow \texttt{out\_v}(C) = \texttt{then\_v}(C)$$

  und

  $$\neg ab(C) \wedge \texttt{cond}(C) = false \Rightarrow \texttt{out\_v}(C) = \texttt{else\_v}(C)$$

  dargestellt werden, wobei v ein Platzhalter für alle den Ports `then_x`, `else_x` und `out_x` zugeordneten Variablen x ist. Eine Auswahlanweisung verhält sich also ähnlich einer Zuweisungsanweisung, durch den zusätzlichen Input-Port `cond` wird in diesem Fall eine der beiden Quellen für die „Zuweisung“ bestimmt. Wie im Fall der Zuweisungsanweisung werden auch hier keine weiteren Fehlermodi spezifiziert.

- Repräsentiert eine Komponente einen *Methodenaufruf*, sind zwei Fälle zu unterscheiden: Ein Methodenaufruf kann entweder auf das aktuelle Objekt oder durch eine Variable getätigt werden. Im ersten Fall besitzt der Aufruf die Form `methodId(ArgumentList)`, im zweiten hingegen die Form `variableId.methodId(ArgumentList)`. Der erste Fall kann als Spezialfall des zweiten Falls angesehen werden, da der Aufruf im ersten Fall implizit über die vom Übersetzer generierte Variable `this` erfolgt, welche jenem Objekt zugeordnet ist, auf dem die aktuelle Methode operiert.

  Um eine Verhaltensbeschreibung für den Methodenaufruf zu erhalten, sei angenommen, daß eine Beschreibung $B_{methodId}$ des Verhaltens für die entsprechende Methode bereits bestimmt ist. Das Verhalten des Methodenaufrufs ist dann durch eine Menge M von logischen Formeln gegeben, wobei diese Menge wie folgt bestimmt wird: Für jede Formel

  $$(A \wedge \neg ab(C_{B_{methodId}}) \Rightarrow B) \in B_{methodId}$$

  wird eine neue Formel

  $$\neg ab(C) \wedge A' \Rightarrow B'$$

in M generiert, wobei A' aus A durch Ersetzen der formalen Parameter der Methode durch die mit den Argumenten des Methodenaufrufs assoziierten Verbindungen erhalten wird. Auch alle Platzhalter für in der aufgerufenen Methode benötigten Instanzvariablen müssen durch die entsprechenden mit den Instanzvariablen assoziierten Verbindungen ersetzt werden. Analog zum Verfahren für A und A' wird bei der Konversion von B zu B' vorgegangen.

Zur Regelsubstitution sei angemerkt, daß durch die Elimination von $\neg ab(C_{B_{methodId}})$ im Antezedenten der ursprünglichen Regeln ein Modell für den Methodenaufruf entsteht, welches alle Komponenten der aufgerufenen Methode als korrekt ansieht. D.h. die den Aufruf modellierende Komponente repräsentiert das Verhalten der aufgerufenen Methode unter der Annahme, daß alle Komponenten der Methode entsprechend ihres korrekten Verhaltens arbeiten.

Weiters ist zu beachten, daß durch diese Konstruktion des Modells angenommen wird, daß das Modell der aufzurufenden Methode bereits verfügbar ist. Diese Voraussetzung kann bei rekursiven Methodenaufrufen aber nicht erfüllt werden, da das Modell der Methode wiederum einen Methodenaufruf enthält, und um diesen zu konstruieren, müßte das Modell der Methode bereits verfügbar sein. Daher können rekursive Methoden auf diese Weise nicht modelliert werden.

Als mögliche Lösung bietet sich an, rekursive Methodenaufrufe durch spezielle Methodenaufrufs-Komponenten zu modellieren, welche einen Evaluator auf der entsprechenden Methode anstoßen, um die Werte für die Output-Ports zu berechnen, anstatt das Verhalten durch logische Formeln nachzubilden. Nachteil dieser Vorgehensweise ist, daß ein Zurückpropagieren von Werten von den Output-Ports zurück zu den Input-Ports nicht möglich ist. Auch müssen in diesem Fall die Werte aller möglicherweise benötigten Variablen bekannt sein, um den Evaluator aufrufen zu können. Aussagen über eine Untermenge der Variablen, wie etwa bei Komponentenmodellen möglich, können hierbei nicht berechnet werden.

Einen weiteren wichtigen Punkt bei der Beschreibung des Verhaltens stellt die Auswahl der korrekten Methode im Fall von Vererbungshierarchien in Kombination mit polymorphen Methoden dar. Um dies zu ermöglichen, muß der Methodenaufrufs-Komponente die Klasse jenes Objekts mitgeteilt werden, auf der die Methode operiert. Anhand dieser Information kann die Komponente das Modell der aufzurufenden Methode aus der Menge der Modelle aller möglichen Methoden auswählen.

Dies erfolgt durch die vom Übersetzer generierte Instanzvariable `class`, welche beim Erzeugen des Objekts mit einem Identifier der Klasse des Objekts initialisiert wird. Der Identifier ermöglicht es, den dynamischen Typ des Objekts – und damit die abzuarbeitende Methode – zu ermitteln. Durch diesen Mechanismus kann Vererbung und Polymorphismus modelliert werden. Hier ist anzumerken, daß dieser Mechanismus bei Aufrufen von Klassenmethoden nicht notwendig ist, da in diesem Fall die Methode bereits bei der Bildung der Modellstruktur eindeutig bestimmt werden kann.

Weiters ist zu beachten, daß die Synthese der Verhaltensbeschreibung des Methodenaufrufs durch Ersetzen der Platzhalter für Instanzvariablen und der Parameter nur beim ursprünglichen direkten Modell ausreichend ist. Für das erweiterte Modell müssen noch weitere Regeln hinzugefügt werden, um die Werte der mit den Output-Ports assoziierten Referenzpfade auf Objekte vollständig bestimmen zu können, falls durch die aufgerufene Methode nur ein Teil der Struktur der mit den Output-Ports assoziierten Referenzpfade auf Objekte verändert wird. Für primitive Datentypen sind hingegen keine zusätzlichen Regeln notwendig.

Als Beispiel betrachte man eine Methode, welche nur einen bestimmten Wert an eine Instanzvariable eines Objekts zuweist. Alle anderen Instanzvariablen bleiben dadurch unverändert. Aufgrund der Modellkonstruktion ist der Output-Port mit einem Referenzpfad, welcher auf das gesamte Objekt verweist, assoziiert. Die durch die Methode nicht veränderten Instanzvariablen des Objekts müssen also von dem Input-Port (dieser ist ebenfalls mit einem Referenzpfad auf das gesamte Objekt assoziiert) weitergereicht werden.

Die dazu erforderlichen Regeln müssen ebenfalls dem Modell hinzugefügt werden:

$$\neg ab(C) \Rightarrow \texttt{out}_{\texttt{path}}(C) = \texttt{in}_{\texttt{path}}(C),$$

wobei `out` ein mit einem Referenzpfad auf ein Objekt assoziierter Output-Port ist und `in` der korrespondierende Input-Port. Aufgrund der Modellkonstruktion muß ein solcher immer existieren. Weiters ist $\texttt{out}_{\texttt{path}}(C)$ der Wert jener Instanzvariablen, welche ausgehend von dem mit `out` assoziierten Referenzpfad, erweitert um den Pfad `path`, angesprochen wird. Analoges gilt für $\texttt{in}_{\texttt{path}}(C)$: hier wird von dem mit dem Input-Port `in` assoziierten Referenzpfad ausgegangen. Dabei ist zu beachten, daß Regeln für genau jene Referenzpfade `path` hinzugefügt werden müssen, welche nicht mit durch die Methode veränderten Instanzvariablen assoziiert sind. Insbesondere sei angemerkt, daß an dieser Stelle auch die Alias-Informationen berücksichtigt werden müssen, um nicht für eine Instanzvariable mehrere Werte zu berechnen und damit bereits eine Inkonsistenz des Modells zu erzeugen.

Abschließend sei darauf hingewiesen, daß für Methodenaufrufe keine speziellen Fehlermodi spezifiziert sind und die entsprechenden Komponenten also entweder fehlerfrei oder in einem unbekannten Fehlermodus arbeiten.

- Komponenten, welche *`return`-Anweisungen* repräsentieren, weisen ein analoges Verhalten zu Zuweisungskomponenten auf. Die Komponenten besitzen einen Input-Port `in`, welcher mit dem zu retournierenden Ausdruck assoziiert ist und einen Output-Port `return`, welcher mit dem retournierten Wert assoziiert ist. Das korrekte Verhalten wird durch die logische Formel

  $$\neg ab(C) \Rightarrow \texttt{return}(C) = \texttt{in}(C)$$

  gegeben, wobei C die Komponente der `return`-Anweisung repräsentiert.

  Sind mehrere `return`-Anweisungen in einer Methode erlaubt, d.h. alle anderen Komponenten, welche Variablen verändern, weisen einen zusätzlichen Input-Port `return_in` auf, welcher der Hilfsvariablen `return` zugeordnet ist, muß das Verhalten aller Komponenten dahingehend verändert werden, daß das „normale“ Verhalten der Komponente nach dem Auftreten einer `return`-Anweisung unterdrückt wird. Dies kann durch Ersetzen aller Regeln der Form

  $$A \Rightarrow B$$

  der Verhaltensbeschreibung einer Komponente C durch Regeln der Form

  $$A \wedge \texttt{return\_in}(C) = \#\texttt{no\_value} \Rightarrow B$$

  erfolgen.[5]

  Bei `while`-Anweisungen muß zusätzlich die Schleifenbedingung dahingehend erweitert werden, daß, falls die Hilfsvariable `return` des Schleifenrumpfs einen Wert ungleich `#no_value` aufweist, der Schleifenrumpf nicht ausgeführt wird.

[5] Unter der Annahme, daß die Hilfsvariable `return` zu Beginn der Methode mit dem künstlichen Token `#no_value` initialisiert wurde

Weiters müssen – analog zu `while`-Schleifen und polymorphen Methodenaufrufen – zusätzliche Regeln hinzugefügt werden, welche das Weiterreichen der ursprünglichen Werte an die Output-Ports der Komponente modellieren:

$$\neg ab(C) \wedge \texttt{return\_in}(C) \neq \#\texttt{no\_value} \Rightarrow \texttt{out\_v}(C) = \texttt{in\_v}(C),$$

wobei v für alle mit Output-Ports assoziierten Variablen instanziert werden muß und $\texttt{in\_v}(C)$ bzw. $\texttt{out\_v}(C)$ die an den entsprechenden Input- bzw. Output-Ports anliegenden Werte bezeichnen.

Weiters ist zu beachten, daß bei `return`-Anweisungen der Wert des Input-Ports `return_in` an den Output-Port `return` weitergereicht werden muß (unabhängig vom Verhaltensmodus der Komponente):

$$\texttt{return\_in}(C) \neq \#\texttt{no\_value} \Rightarrow \texttt{return}(C) = \texttt{return\_in}(C).$$

Durch diese Modifikationen wird das ursprüngliche Verhalten der Komponenten nur dann anwendbar, wenn nicht zuvor eine `return`-Anweisung abgearbeitet wurde. Andernfalls werden alle Werte von Variablen unverändert weitergereicht.

- Repräsentiert eine Komponente einen *Ausdruck*, sind hier (wie bei der Bildung der Struktur des Modells) vier Fälle zu unterscheiden: *Konstanten*, *Variablen*, *Operatoren* und *Funktions-* bzw. *Konstruktoraufrufe*.

  - Das Verhalten von Komponenten, welche *Konstanten* repräsentieren, wird durch die Formel
    $$\neg ab(C) \Rightarrow \texttt{result}(C) = const$$
    bestimmt, wobei *const* den Wert der im Quelltext des Programms angegebenen Konstanten darstellt.
  - Das Verhalten von Komponenten, welche *Variablen* repräsentieren, wird analog zu jenem der Zuweisungsanweisung spezifiziert, wobei der Input-Port `in` in diesem Fall dem aktuellen Wert der Variablen zugeordnet ist. Das Verhalten wird also wiederum durch folgende logische Formel beschrieben:
    $$\neg ab(C) \Rightarrow \texttt{result}(C) = \texttt{in}(C),$$
    wobei C die Komponente der Konstanten bzw. der Variablen darstellt.
  - Das Verhalten von *Operator*-Komponenten kann ähnlich jenem der Konstanten oder der Variablen spezifiziert werden. Komponenten, welche unäre Operatoren repräsentieren, besitzen jeweils einen Input-Port `in`, welcher den vom Operator benötigten Ausdruck repräsentiert, und einen Output-Port `result`, der dem Ergebnis der Operation zugeordnet ist. Das korrekte Verhalten der Komponente kann durch folgende logische Formel ausgedrückt werden:
    $$\neg ab(C) \Rightarrow \texttt{result}(C) = op(\texttt{in}(C)),$$
    wobei $op(A)$ das Ergebnis der Anwendung des Operators op auf den Ausdruck A darstellt und C die den Operator modellierende Komponente repräsentiert.

    Binäre Operatoren werden auf ähnliche Weise modelliert: Jede solche Komponente C besitzt zwei Input-Ports `in1` und `in2`, welche den beiden Ausdrücken zugeordnet sind, auf die der Operator angewendet wird. Sie sind mit den Output-Ports result

der Komponenten C_1 und C_2 verbunden, welche die Unter-Ausdrücke des Ausdrucks repräsentieren. Weiters besitzt die Komponente einen Output-Port `result`, welcher dem Ergebnis der Operation zugeordnet ist. Das korrekte Verhalten der Operator-Komponente kann dann wie folgt beschrieben werden: Der durch die Komponente repräsentierte Operator wird auf die beiden Werte $\texttt{result}(C_1)$ und $\texttt{result}(C_2)$ der Unter-Ausdrücke angewendet und das auf diese Weise erhaltene Ergebnis muß mit dem Wert $\texttt{result}(C)$ der Komponente übereinstimmen. Formal kann das durch die logische Formel

$$\neg ab(C) \Rightarrow \texttt{result}(C) = op(\texttt{result}(C_1), \texttt{result}(C_2))$$

dargestellt werden, wobei $op(A, B)$ die Anwendung des Operators op auf die Ausdrücke A und B bezeichnet.

– *Funktionsaufrufe* werden analog zu Methodenaufrufen modelliert, wobei der Output-Port `return` der Komponente durch den Port `result` ersetzt wird. Aufrufe von *Konstruktoren* durch `new`-Ausdrücke werden durch eine spezielle Komponente modelliert, welche das Anlegen eines neuen Objekts, das Initialisieren von dessen Instanzvariablen und den Aufruf des Konstruktors zusammenfaßt. Der Output-Port `class` der Komponente ist der Klasse des erzeugten Objekts zugeordnet:

$$\neg ab(C) \Rightarrow \texttt{class}(C) = Class,$$

wobei $Class$ die Klasse des erzeugten Objekts bezeichnet. Weiters werden vor Beginn der Abarbeitung des Modells des Konstruktors alle Instanzvariablen des erzeugten Objekts mit deren Default-Werten initialisiert. Die Werte jener Variablen, welche vom Modell des Konstruktors nicht verändert werden, werden an die Output-Ports der Komponente weitergeleitet. Die Werte der weiteren Variablen werden durch das Modell des Konstruktors bestimmt.

Allen Komponenten, welche Ausdrücke repräsentieren, ist gemeinsam, daß keine speziellen Verhaltensmuster für eventuelle Fehlermodi angegeben werden. Solche Komponenten funktionieren also entweder korrekt oder in einem unbekannten Fehlermodus.

- Komponenten, welche `while`-Anweisungen repräsentieren, weisen eine komplexere Verhaltensbeschreibung als einfache Anweisungen wie Zuweisungen oder Auswahlanweisungen auf, da diese Komponenten einen hierarchischen Aufbau besitzen und die Modelle für die Schleifenbedingung und den Schleifenrumpf eventuell mehrfach ausgewertet werden müssen. Das Verhalten der `while`-Komponente entspricht dem intuitiven Verhalten der Schleife: Falls die Schleifenbedingung erfüllt ist, d.h. die Auswertung der Bedingung ergibt `true`, wird der Schleifenrumpf ausgewertet, und es werden neue Werte für die Variablen berechnet. Anschließend wird eine neue Iteration begonnen und die Bedingung erneut ausgewertet. Dies erfolgt solange bis die Schleifenbedingung nicht länger erfüllt ist, d.h. bis der Wert `false` bei der Auswertung der Bedingung berechnet wird.

  Die Werte der Variablen vor den einzelnen Iterationen werden in Hilfsvariablen v_X^i gespeichert, wobei X eine Variable bezeichnet, welche entweder mit einem Input-Port `in_X` oder einem Output-Port `out_X` assoziiert ist und i die Anzahl der bisherigen Iterationen bezeichnet. Zu Beginn der Berechnungen werden die Werte aller Variablen, welche mit Input-Ports assoziiert sind, den entsprechenden Hilfsvariablen zugewiesen. Analog erhalten nach dem Abbruch der Schleife alle Variablen, welche mit Output-Ports assoziiert sind, ihre Werte durch Zuweisungen aus den entsprechenden Hilfsvariablen. Weiters muß

spezifiziert werden, daß die Werte aller Hilfsvariablen, welche nicht durch Auswerten des Schleifenrumpfs verändert werden, in der nächsten Iteration unverändert erhalten bleiben.

Um die Schleifenbedingung auszuwerten, müssen zunächst die Werte der darin benötigten Variablen aus den Hilfsvariablen der vorhergehenden Iteration ermittelt werden. Anschließend wird, unter der Annahme, daß alle Komponenten der Bedingung korrekt funktionieren, die Schleifenbedingung ausgewertet. Ergibt sich dabei, daß die Bedingung erfüllt ist, werden die Input-Verbindungen des Schleifenrumpfs mit den aktuellen Werten aus der vorangegangenen Iteration initialisiert und der Schleifenrumpf anschließend unter der Annahme ausgewertet, daß alle Komponenten des Schleifenrumpfs korrekt funktionieren. Daraus ergeben sich die Werte der durch den Schleifenrumpf veränderten Variablen für die nächste Iteration. Ist die Schleifenbedingung hingegen nicht erfüllt, d.h. es wird bei der Auswertung der Wert `false` ermittelt, bricht die Schleife ab. Zu beachten ist, daß bei diesem Modell davon ausgegangen wird, daß bei der Auswertung der Schleifenbedingung keine Nebeneffekte auftreten, d.h. es dürfen keine Funktionsaufrufe durchgeführt werden, welche Belegungen von Variablen oder Instanzvariablen modifizieren.

Diese Beschreibung läßt sich in eine logische Formel fassen:

$$
\begin{aligned}
&\neg ab(C) \Rightarrow \\
&\left(\begin{array}{l}
\forall_X \texttt{in\_X}(C) = v_X^0(C) \wedge \\
\forall_X \texttt{out\_X}(C) = v_X^{MAX}(C) \wedge \\
\forall_{X \in (VC \cup VBI) \setminus VBO} \forall_{i<MAX} v_X^{i+1}(C) = v_X^i(C) \wedge \\
\left(\begin{array}{l}
\left(\begin{array}{l} SD_C(C) \cup \{\neg ab(C_C) | C_C \in COMP_C(C)\} \cup \\ \left\{v_X^i(C) = \texttt{ci\_X}(C) | X \in VC\right\} \end{array}\right) \Rightarrow \\
\forall_{Y \in VBO} \left(\begin{array}{l} SD_B(C) \cup \{\neg ab(C_B) | C_B \in COMP_B(C)\} \cup \\ \left\{v_X^i(C) = \texttt{bi\_X}(C) | X \in VBI\right\} \end{array}\right) \vDash v_Y^{i+1}(C) = \texttt{bo\_Y}(C)
\end{array}\right) \wedge \\
\left(\left(\begin{array}{l}
\left(\begin{array}{l} SD_C(C) \cup \{\neg ab(C_C) | C_C \in COMP_C(C)\} \cup \\ \left\{v_X^i(C) = \texttt{ci\_X}(C) | X \in VC\right\} \end{array}\right) \wedge \\
\nexists j : j < i \wedge \\
\left(\begin{array}{l} SD_C(C) \cup \{\neg ab(C_C) | C_C \in COMP_C(C)\} \cup \\ \left\{v_X^j(C) = \texttt{ci\_X}(C) | X \in VC\right\} \end{array}\right)
\end{array}\right) \vDash \texttt{cond}(C) = false\right) \Rightarrow MAX = i\right)
\end{array}\right),
\end{aligned}
$$

wobei VC bzw. VBI die Menge der Variablen bezeichnet, welche in der Schleifenbedingung bzw. im Schleifenrumpf benötigt werden. VBO bezeichnet jene Variablen, welche bei der Abarbeitung des Schleifenrumpfs verändert werden. `bi_X` bzw. `bo_X` bezeichnen die Input- bzw. Output-Ports des Modells des Schleifenrumpfs, `ci_X` die Input-Ports des Modells der Schleifenbedingung, welche mit der Variablen X assoziiert sind, $COMP_B(C)$ und $COMP_C(C)$ bezeichnen die Komponenten der Modelle des Schleifenrumpfs und der Schleifenbedingung, mit den zugehörigen Systembeschreibungen $SD_B(C)$ und $SD_C(C)$. $\texttt{cond}(C)$ bezeichnet den dem Port `cond` des Modells der Schleifenbedingung zugeordneten Wert, welcher dem Ergebnis der Auswertung der Schleifenbedingung entspricht. MAX bezeichnet die Nummer der letzten Iteration der Schleife.

Zusätzlich zum korrekten Verhalten der `while`-Anweisung ist ein Fehlermodus $loop(C, MAX)$ definiert, welcher die maximale Anzahl der Iterationen auf MAX festlegt:

$$loop(C, MAX) \Rightarrow \left(\begin{array}{l} \forall_X \texttt{in\_X}(C) = v_X^0(C) \wedge \\ \forall_X \texttt{out\_X}(C) = v_X^{MAX}(C) \wedge \\ \forall_{X \in (VC \cup VBI) \setminus VBO} \forall_{i<MAX} v_X^{i+1}(C) = v_X^i(C) \wedge \\ \forall_{i<MAX} \forall_{Y \in VBO} \left(\begin{array}{l} SD_B(C) \cup \{\neg ab(C_B) | C_B \in COMP_B(C)\} \cup \\ \{v_X^i(C) = \texttt{bi\_X}(C) | X \in VBI\} \vDash v_Y^{i+1}(C) = \texttt{bo\_Y}(C) \end{array} \right) \end{array} \right).$$

Bei dem hier beschriebenen Verhaltensmodell für `while`-Schleifen ist zu beachten, daß dieses nur für das ursprüngliche Modell korrekt funktioniert. Für das erweiterte Modell müssen – analog zur Vorgehensweise bei Methodenaufrufen – zusätzliche Regeln in das Modell der Schleifenbedingung und des Schleifenrumpfs eingeführt werden, um nicht veränderte Variablen korrekt zwischen Input- und Output-Ports des Modells des Schleifenrumpfs weiterreichen zu können und die tatsächlich benötigten Variablen aus den möglicherweise allgemeineren, mit Objekten assoziierten Referenzpfaden herauszuprojizieren.

4.1.4 Komplexität des Modells

Abschließend bleibt die Frage zu behandeln, bis zu welcher Größe von Programmen das Modell sinnvoll angewendet werden kann. Um dies feststellen zu können, ist eine Abschätzung der oberen Schranke der Berechnungskomplexität beim Finden von Diagnosen notwendig.

Wie in Kapitel 3 angedeutet, ist der Aufwand beim Finden von Diagnosen vorwiegend von der Anzahl der Komponenten des Systems abhängig. Eine grobe Abschätzung für die maximale Anzahl von Komponenten eines Modells eines gegebenen Programms P bieten folgende Überlegungen:

Sei M die zu diagnostizierende Methode im Programm P. Sei c die maximale Anzahl der Aufrufe von Methoden und Konstruktoren in einer Methode in P, s die maximale Anzahl von Anweisungen in einer Methode in P und e die maximale Anzahl von Operatoren, Variablenreferenzen und Konstanten in einer Methode in P. Weiters sei d die maximale Aufruftiefe von Methoden, ausgehend von M.[6] Dann kann die maximale Anzahl der Komponenten des Modells von M durch $c^d \cdot (s + e)$ begrenzt werden. Diese Abschätzung gilt für die Menge *aller* erzeugten Komponenten, inklusive der für hierarchische Komponenten generierten Modelle des Verhaltens der Komponenten. Die Anzahl der Komponenten auf oberster Ebene – dies sind jene Komponenten, welche in einer Diagnose aufscheinen können – einer Methode kann durch $s + e$ beschränkt werden.

Unter Berücksichtigung der Tatsache, daß durch Einsatz moderner Diagnose-Algorithmen und Theorembeweiser Systeme mit bis zu einigen Tausend Komponenten in wenigen Sekunden diagnostiziert werden können, läßt sich die maximale Größe von mit diesem Modell handhabbaren Programmen auf einige k-Bytes festlegen.

Hier ist zu beachten, daß diese Abschätzung nur die Anzahl der Komponenten berücksichtigt, nicht jedoch den Berechnungsaufwand, welcher bei der Abarbeitung der Verhaltensbeschreibung der einzelnen Komponenten im Theorembeweiser anfällt. Dieser kann in einigen Fällen um Größenordnungen höher ausfallen als bei einfachen Komponenten (wie z.B. Komponenten, welche einen Additionsoperator modellieren). Beispiele hierfür wären etwa Schleifen oder (re-

[6] Rekursive Aufrufe von Methoden weisen die Tiefe 0 auf, da für das Verhalten der den Aufruf modellierenden Komponente nicht durch komponentenbasierte Repräsentationen dargestellt wird.

kursive) Methodenaufrufe[7]. Enthält ein zu analysierendes Programm überdurchschnittlich viele solcher Konstrukte, sinkt die noch sinnvoll handhabbare Größe dementsprechend.

Zu den hier angeführten Abschätzungen für den Berechnungsaufwand der Diagnosen ist noch der Aufwand bei der Modellbildung zu beachten. Der Aufwand bei der Modellierung von Methoden ist im wesentlichen linear abhängig von der Anzahl der Anweisungen und Ausdrücken der Methode. Gegenüber dem bei der Aliasing-Analyse benötigten Aufwand können diese Berechnungen jedoch vernachlässigt werden, da die Analyse von Aliasing-Relationen mit Referenzpfaden polynomieller Länge ein Problem darstellt, dessen Komplexität zumindest in der Komplexitätsklasse $\mathcal{NP}$ liegt [Lan92] und damit exponentiellen Aufwand benötigt.

4.2 Indirektes Modell

Das in den vorangegangenen Abschnitten behandelte direkte Modell erzeugt die Modellstruktur nur anhand des gegebenen Programms, unabhängig von den konkreten Beobachtungen. Insbesondere sind die Eingabewerte des Programms zu diesem Zeitpunkt nicht bekannt. Folglich muß bei der Modellbildung darauf geachtet werden, daß das Modell für *alle* möglichen Eingabewerte die Semantik des gegebenen Programms widerspiegelt. Dadurch wird die Modellbildung notwendigerweise stark verkompliziert. Dies zeigt sich insbesondere bei dem Problem der möglichen Alias-Relationen zwischen zwei oder mehreren Referenzpfaden, welche durch die Referenz-Semantik der Sprache von nicht-primitiven Datentypen herrührt. Da diese Analyse zur Übersetzungszeit sehr aufwendig ist und darüber hinaus nur näherungsweise berechnet werden kann, wird in diesem Abschnitt ein weiteres Modell vorgestellt, welches die Analyse vermeidet, aber dennoch die Semantik der Sprache korrekt nachbildet.

Die Vereinfachung, die das indirekte Modell bietet, beruht darauf, nicht – wie in dem vorangegangenen direkten Modell – die Referenz-Semantik direkt über die Referenzpfade auf lokale-, Instanz- und Klassenvariablen zu modellieren, sondern Zugriffe auf Objekte indirekt durch Identifier der Objekte zu repräsentieren. Weiters werden die lokalen Variablen und Klassenvariablen getrennt von den Instanzvariablen der Objekte repräsentiert.

Die Grundidee des im weiteren näher vorgestellten indirekten Modells beruht auf der Tatsache, daß bei der Abarbeitung der Anweisungen des Programms in sequentieller Reihenfolge vorgegangen wird und dabei durch die jeweiligen Anweisungen einige Variablen der aktuellen Variablen- oder Objektumgebung verändert werden. Diese veränderten Variablen, zusammen mit den unveränderten Variablen, werden als neue Umgebungen für die darauffolgende Anweisung verwendet, welche wiederum einige Variablen dieser Umgebung verändert. Diese Idee entspricht im wesentlichen jener der *Structural Operational Semantics* [NN92] aus der formalen Semantik, welche die sequentielle Abarbeitung und die damit verbundenen Änderungen der Bindungen der Variablen in den jeweiligen Zuständen vor und nach der Abarbeitung einer Anweisung modelliert.

Hier wird angenommen, daß jede Anweisung durch eine Funktion modelliert werden kann, welche als Eingabe eine Umgebung und eine Menge von Objekten entgegennimmt und als Ergebnis ebenfalls wieder ein Paar aus Umgebung und Menge von Objekten berechnet. Die Umgebung stellt wieder eine Funktion dar, welche jeder lokalen und globalen Variablen einen Wert zuweist. Die Menge der Objekte (der sogenannte *Objektraum*) enthält die Menge aller Objekte, welche in dem dem Objektraum zugeordneten Zustand der Programmabarbeitung ansprechbar sind, wobei jedes Objekt einen Identifier besitzt, mit dem es eindeutig identifiziert werden kann. Der Objektraum kann also auch als Funktion von Objekt-Identifiern in Objektstrukturen an-

[7] Darüber hinaus kann in diesen Fällen nicht immer garantiert werden, daß die Abarbeitung des Verhaltens dieser Komponenten überhaupt terminiert. Dies muß z.B. durch die Angabe einer maximalen Anzahl von Iterationen oder einer maximalen Rekursionstiefe sichergestellt werden.

gesehen werden. Das von der eine Anweisung modellierenden Funktion berechnete Ergebnis spiegelt die Effekte der modellierten Anweisung wieder, d.h. die berechnete Umgebung und der Objektraum werden entsprechend der Semantik der mit der Funktion modellierten Anweisung verändert. Die Abarbeitung einer Sequenz von Anweisungen kann also durch eine Folge von Paaren $(Env_n, ObjectSpace_n)$ dargestellt werden, wobei Env_n und $ObjectSpace_n$ die Umgebung und den Objektraum nach der Abarbeitung der n-ten Anweisung darstellen. Bei Abarbeitung der Anweisung S_{n+1} werden die durch die Abarbeitung der vorangegangenen Anweisung S_n berechnete Umgebung bzw. der Objektraum als Grundlage zur Berechnung der neuen Umgebung bzw. des neuen Objektraums verwendet. Dies ist in Abbildung 4.9 dargestellt.

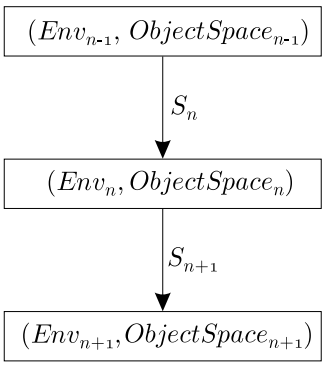

Abbildung 4.9: Modell von Anweisungen als Funktionen

Die Anwendung dieses Konzepts bei der Modellbildung von Java-Programmen führt zu dem im weiteren behandelten Modell, das stark an das erweiterte Modell des vorangegangenen Abschnittes angelehnt ist. Zunächst werden nur Referenzpfade der Länge eins betrachtet, d.h. es werden nur lokale Variablen und Klassenvariablen direkt als Variablen repräsentiert und damit auch nur für diese Variablen Input- und Output-Ports generiert. Alle längeren Referenzpfade müssen auf Instanzvariablen von Objekten zugreifen, da in Java komplexe Strukturen nur über Klassen mit Instanzvariablen modelliert werden können, und eine lokale oder globale Variable nicht auf eine andere lokale oder globale Variable verweisen kann. Dies folgt aus der Tatsache, daß Java keinen *address-of*-Operator (wie etwa den Operator '`&`' in C++) und keine Zeiger mit Arithmetik bereitstellt und daher keine Referenz auf lokale Variablen erhalten werden kann. Es ist natürlich möglich, daß zwei lokale oder globale Variablen auf dasselbe Objekt verweisen. Aus den Eigenschaften der Sprache folgt, daß lokale Variablen ausschließlich durch Anwendung von Zuweisungsoperatoren verändert werden können und nicht durch Methodenaufrufe. Dies gilt nicht für globale Variablen, da diese natürlich auch in der aufgerufenen Methode ansprechbar sind und damit durch Zuweisungen verändert werden können.

Aufgrund dieser Tatsache können die Veränderungen von lokalen und globalen Variablen durch Anweisungen bereits zur Übersetzungszeit eindeutig bestimmt werden und daher auf Basis dieser Informationen der Datenfluß dieser Variablen zwischen den einzelnen Anweisungen modelliert werden. Analog zu dem vorangegangenen Modell wird diese Art des Datenflusses durch Verbindungen zwischen Ports von Komponenten modelliert.

Alle weiteren Veränderungen, d.h. Veränderungen an Instanzvariablen von Objekten werden im Objektraum abgebildet. Der Objektraum enthält sämtliche zu diesem Zeitpunkt der Abarbeitung existierenden Objekte mit allen darin enthaltenen Instanzvariablen. Der Objektraum bildet also in gewisser Weise den dynamischen Speicher von Java nach. Einen augenfälligen Unterschied zwischen dem Objektraum und einem konventionellen dynamischen Speicher stellt die Tatsache dar, daß die Objekte im Objektraum durch eindeutige Identifier angesprochen werden anstelle eines Zeigers auf ihre Speicheradresse.

Mit Hilfe des Objektraums können in dem hier vorgestellten Modell auch indirekte Zugriffe auf Instanzvariablen von Objekten modelliert werden: Die lokalen und globalen Variablen sind im Gegensatz zum erweiterten Modell des letzten Abschnitts nicht mit Referenzpfaden auf Objekte assoziiert, sondern enthalten die Identifier der entsprechenden Objekte. Veränderungen an Instanzvariablen beeinflussen daher in keiner Weise die lokalen und globalen Variablen, da sich die Identifier der Objekte nicht verändern. Es wird also nur ein Eintrag im Objektraum verändert. Wird auf eine Instanzvariable in lesender Weise zugegriffen, muß wieder das zugehörige Objekt über den Objekt-Identifier ermittelt werden und dann der Wert der gewünschten Instanzvariablen aus dem Objektraum ausgelesen werden.

Die im vorangegangenen Modell durch Aliasing zwischen Referenzpfaden verursachten Probleme treten in dieser Repräsentation nicht auf, da die lokalen und globalen Variablen nur Objekt-Identifier enthalten und daher unabhängig voneinander verändert werden können. Die Probleme der statischen Alias-Analyse werden hier umgangen, da die Analyse nicht zur Übersetzungszeit durchgeführt wird, sondern durch die Zugriffe auf die Objekträume während der modellbasierten Diagnose erfolgt. Da zu diesem Zeitpunkt die konkreten Werte der Eingabevariablen bekannt sind, kann die Struktur der Verweise zwischen den Objekten im Objektraum nachgebildet werden. Somit ist es ausreichend, die Konsistenz der einzelnen Versionen des Objektraums zwischen den Anweisungen sicherzustellen. Dadurch kann die Analyse von möglichen Alias-Relationen vollständig vermieden werden. Dies wird jedoch durch einen erheblichen Mehraufwand an benötigtem Speicher und Rechenaufwand erkauft, da die Objekträume zwischen den einzelnen Anweisungen jeweils kopiert und gespeichert werden müssen.

Analog zum direkten Modell kann das indirekte Modell durch die Vorgehensweise bei der Generierung der Modellstruktur aus dem gegebenen Programm und der vom Programm unabhängigen Verhaltensbeschreibung der einzelnen Komponenten charakterisiert werden. Im weiteren wird zunächst die Bildung der Modellstruktur aus dem gegebenen Quelltext des zu debuggenden Programms behandelt.

4.2.1 Struktur des indirekten Modells

Die Abbildung von strukturellen Elementen der Programmiersprache auf Komponenten und Verbindungen des Modells erfolgt in ähnlicher Weise wie in Abschnitt 4.1.1 beschrieben. Die Menge der Komponenten wird aus einzelnen Anweisungen und deren Teilen (wie z.B. Operatoranwendungen, Funktionsaufrufen oder Variablenreferenzen) generiert. Ein wesentlicher Unterschied besteht jedoch darin, daß in dem Modell aus Abschnitt 4.1 alle Arten von Variablen – d.h. Klassen-, Instanz- und lokale Variablen – als Verbindungen repräsentiert werden. Im Gegensatz dazu werden jetzt nur lokale Variablen und Klassenvariablen als Verbindungen repräsentiert. Dies sind zugleich die Variablen aus der Umgebung Env. Die Instanzvariablen von Objekten werden hingegen im Objektraum als Teile der darin enthaltenen Objekte repräsentiert.

Diese Trennung von lokalen Variablen bzw. Klassenvariablen und Instanzvariablen vereinfacht zudem auch die Bildung der Modellstruktur, da nun nicht mehr zwischen Variablen von primitiven Datentypen und Variablen von nicht-primitiven Datentypen unterschieden werden muß. Dies wird möglich, da Variablen von nicht-primitiven Datentypen hier nur Verweise auf Objekte im Objektraum enthalten und die Verweise wie primitive Datentypen behandelt werden können. D.h. die Identifier lassen sich nicht weiter in kleinere Strukturen aufspalten und können wie primitive Datentypen kopiert und an Methoden übergeben werden.

Da Instanzvariablen nicht länger als Verbindungen modelliert werden, entfällt bei Methodenaufrufen, `while`-Schleifen und Auswahlanweisungen auch die Notwendigkeit, den entsprechenden Komponenten Output-Ports für alle Instanzvariablen neu angelegter Objekte hinzuzufügen. Auch werden Veränderungen an Instanzvariablen von Objekten hier im Objektraum abgebildet und daher müssen auch in diesem Fall keine weiteren Ports zu den Komponenten hinzugefügt

werden. Folglich werden auch alle damit verbundenen Probleme des vorherigen Modells – wie z.B. Probleme mit rekursiven Datenstrukturen oder Alias-Relationen zwischen Referenzpfaden – umgangen.

Die grundlegende Vorgehensweise bei der Generierung der Komponenten, Ports und Verbindungen für Programmelemente und Variablen erfolgt wie in Abschnitt 4.1.1 beschrieben. Da im indirekten Modell die für die einzelnen Komponenten zu erzeugenden Input- und Output-Ports aufgrund der Modifikationen des Modells von jenen aus Abschnitt 4.1.1 in einigen Fällen abweichen, seien die relevanten Informationen hier nochmals zusammengefaßt.

- *Zuweisungen* an lokale Variablen oder Klassenvariablen werden wie zuvor durch eine Komponente mit jeweils einem Input- und einem Output-Port repräsentiert. Der Input-Port ist mit dem Ergebnis der Auswertung des auf der rechten Seite der Zuweisung stehenden Ausdrucks assoziiert, der Output-Port mit der auf der linken Seite der Zuweisung stehenden lokalen Variablen oder Klassenvariablen.

  Bei Zuweisungen an Instanzvariablen ist zu beachten, daß in diesem Fall ein zusätzlicher Input-Port `objectspace_in` hinzugefügt werden muß, welcher mit dem Objektraum vor der Abarbeitung der Zuweisung assoziiert ist. Der Output-Port `objectspace_out` ist mit dem Objektraum nach der Ausführung der Anweisung assoziiert. Die modifizierte Variable wird also nicht durch eine eigene Verbindung repräsentiert, sondern es muß die entsprechende Variable von der Verhaltensbeschreibung der Komponente im Objektraum angesprochen werden. Um dies zu ermöglichen muß ein weiterer Input-Port `object` hinzugefügt werden, welcher das zur Instanzvariablen gehörende Objekt im Objektraum identifiziert. Im Fall einer Zuweisung der Form `v = expr`, wobei `v` ein einfacher Name und `expr` den zuzuweisenden Ausdruck darstellen, ist der Port mit der Hilfsvariablen `this` aus der Umgebung zu assoziieren, welche den Objekt-Identifier des aktuellen Objekts enthält. Wird hingegen eine Instanzvariable über einen Referenzpfad angesprochen, dessen Länge größer als eins ist, d.h. die Zuweisung ist von der Form `path.field = expr`, so wird der Port `object` mit jenem Port, welcher mit dem Objekt-Identifier des über den Pfad `path` angesprochenen Objekts assoziiert ist, verbunden.

- *Auswahlanweisungen* werden ebenfalls nahezu unverändert übernommen. Diese werden auch hier durch eine Komponente mit mehreren Input- und Output-Ports repräsentiert. Wie im ursprünglichen Modell sind die Input- und Output-Ports den in den beiden Zweigen veränderten Variablen zugeordnet. Im Gegensatz zu dem ursprünglichen Modell werden hier aber nur lokale Variablen und Klassenvariablen betrachtet. Instanzvariablen neu erzeugter Objekte sind ebenfalls zu ignorieren. Diese werden im Objektraum behandelt. Zusätzlich dazu existieren beim indirekten Modell auch Ports `objectspace_then`, `objectspace_else` und `objectspace_out`, welche den Objekträumen nach der Abarbeitung des `then-` und `else-`Zweiges bzw. der gesamten Auswahlanweisung zugeordnet sind. Diese Ports sind in analoger Weise wie die anderen Ports der Komponente zu behandeln. Schließlich wird natürlich ein Input-Port `cond` benötigt, welcher dem Ergebnis der Bedingung der Auswahlanweisung zugeordnet ist.

- *Aufrufe von Methoden* werden analog zu Abschnitt 4.1.1 als eine Komponente repräsentiert und können aus dem Modell der aufzurufenden Methode durch Ersetzen der Parameter durch die beim Aufruf verwendeten Argumente erhalten werden. Die Input-Ports entsprechen hier den in der Methode benötigten Parametern und Klassenvariablen. Analog dazu sind die Output-Ports den durch die Methode veränderten lokalen Variablen und Klassenvariablen zugeordnet. Wie zuvor bei den Auswahlanweisungen sind alle bei der Abarbeitung der Methode veränderten Instanzvariablen und Instanzvariablen neu erzeugter

Objekte zu ignorieren, da diese Veränderungen im Objektraum abgebildet werden. Zu diesem Zweck werden der Komponente zusätzliche Input- und Output-Ports `objectspace_in` und `objectspace_out` hinzugefügt, welche mit den Objekträumen vor bzw. nach der Abarbeitung des Methodenaufrufs assoziiert sind.

Bei Aufrufen von Instanzmethoden ist ein zusätzlicher Input-Port `object` hinzuzufügen, welcher dem Objekt-Identifier jenes Objekts zugeordnet ist, auf dem die aufgerufene Methode operieren soll. Im Fall eines Methodenaufrufs der Form `methodId(ParameterList)` oder `this.methodId(ParameterList)` oder `super.methodId(ParameterList)` ist der Port mit der vom Übersetzer generierten Variablen `this` assoziiert, da die Methode auf dem aktuellen Objekt operiert. Wird die Methode hingegen über eine andere Variable oder einen Referenzpfad aufgerufen, d.h. der Methodenaufruf hat die Form `path.methodId(ParameterList)`, wird der Port mit dem Output-Port `result` des Modells von `path` verbunden, da dieser mit dem Objekt-Identifier des gewünschten Objekts assoziiert ist.

Diese Vorgehensweise ist notwendig, um das Objekt (und damit dessen Typ und somit die entsprechende Methode) zu ermitteln, auf welchem die aufgerufene Methode operieren soll.

Durch die in diesem Modell getroffenen Vereinfachungen können auch rekursive Methoden modelliert werden, da die Mengen der benötigten und veränderten lokalen Variablen und Klassenvariablen aus dem Quelltext bestimmt werden können und diese – im Gegensatz zu veränderten Instanzvariablen – durch rekursive Aufrufe der Methode nicht beliebig erweitert werden können. Dadurch sind auch die Mengen der mit diesen Variablen assoziierten Input- und Output-Ports ohne großen Aufwand zu bestimmen.

Im Fall von Vererbungshierarchien mit polymorphen Methoden wird wie in Abschnitt 4.1.1 beschrieben vorgegangen: Es werden die Vereinigungsmengen der Mengen der Input- und Output-Ports aller möglicherweise aufgerufenen Methoden gebildet und diese als Input- bzw. Output-Ports der Methodenaufrufs-Komponente zugeordnet. Auch hier werden Instanzvariablen nicht betrachtet, sondern es werden eigene Input- und Output-Ports für die Objekträume eingeführt.

- *`return`-Anweisungen* werden durch eine Komponente mit jeweils einem Input- und einem Output-Port repräsentiert. Der Input-Port ist mit dem von der Methode zurückzuliefernden Ausdruck der `return`-Anweisung assoziiert, der Output-Port mit einer Hilfsvariablen `return`. Dies entspricht dem in Abschnitt 4.1.1 beschriebenen Modell. Daraus folgt, daß auch hier nur maximal eine `return`-Anweisung in jeder Methode modelliert werden kann. Es kann jedoch auch hier das für das Modell aus Abschnitt 4.1 beschriebene Verfahren angewendet werden, um durch Unterdrücken der Effekte aller darauffolgenden Komponenten mehrere `return`-Anweisungen zu ermöglichen.

- Das Modell von *Ausdrücken* entspricht weitgehend jenem aus Abschnitt 4.1.1.

  - Die Konversion von *Konstanten* und *Operatoren* in Komponenten erfolgt wie dort angegeben.
  - Im Fall von *Variablen* sind zwei Fälle zu unterscheiden: *lokale Variablen und Klassenvariablen* werden wie in Abschnitt 4.1.1 angegeben modelliert. Modelle von Zugriffen auf *Instanzvariablen* hingegen werden ähnlich einer Zuweisung an eine Instanzvariable modelliert: Die Komponenten besitzen zwei Input-Ports und einen Output-Port. Der erste Input-Port `objectspace_in` ist dem Objektraum vor der Abarbeitung des Ausdrucks zugeordnet. Dieser enthält den aktuellen Wert der Variablen, welcher das

Ergebnis des Ausdrucks darstellt. Der zweite Input-Port `object` ist dem Objekt-Identifier des anzusprechenden Objekts zugeordnet. Hier sind – analog zu Aufrufen von Instanzmethoden – zwei Fälle zu unterscheiden: Zugriffe der Form `variableId` und Zugriffe über einen Referenzpfad (`path.variableId`). Die Herstellung der Verbindung für den Input-Port `object` erfolgt analog zu der für Methodenaufrufe beschriebenen Vorgehensweise. Der Output-Port `result` ist mit dem aktuellen Wert der angesprochenen Instanzvariablen assoziiert und stellt das Ergebnis des Ausdrucks dar.

– Für *Funktionsaufrufe* gelten die oben getroffenen Aussagen für Methodenaufrufe. Wie im direkten Modell wird der Port `result` durch Umbenennen des Ports `return` aus dem Modell der Methode erhalten. Repräsentiert der Methodenaufruf einen Aufruf eines Konstruktors in einem `new`-Ausdruck, muß – entsprechend dem direkten Modell aus Abschnitt 4.1.1 – der Port `result` hinzugefügt werden, welcher dem Identifier des erzeugten Objekts zugeordnet ist.

  An dieser Stelle sei auf eine Besonderheit bei der Modellierung von Konstruktoren und `new`-Ausdrücken hingewiesen: Da Konstruktoren auf Instanzvariablen des neu erzeugten Objekts zugreifen können, werden Konstruktoren bei der Modellbildung wie Instanzmethoden behandelt und somit auch ein Input-Port `this` generiert, welcher dem Objekt-Identifier jenes Objekts zugeordnet ist, auf dem der Konstruktor operiert. Konstruktoren modellieren also nur die Initialisierung des neu angelegten Objekts, nicht jedoch die Generierung des Objekt-Identifiers. Dies erfolgt durch die Verhaltensbeschreibung der den `new`-Ausdruck repräsentierenden Komponente, welche den erzeugten Objekt-Identifier sowohl an den Output-Port `result` weiterreicht, als auch an den Input-Port `this` des Modells des Konstruktors. Weiters weist die den `new`-Ausdruck repräsentierende Komponente Input-Ports `objectspace_in` und `objectspace_out` auf, da sämtliche Instanzvariablen des neu erzeugten Objekts in den Objektraum eingetragen werden und mit deren Default-Werten initialisiert werden.

• *`while`-Anweisungen* werden analog zu dem Modell aus Abschnitt 4.1 durch eine hierarchische Komponente mit Sub-Komponenten für Schleifenbedingung und Schleifenrumpf modelliert, wobei sowohl bei der Komponente selbst, als auch bei ihren Sub-Komponenten keine Instanzvariablen bei den Input- und Output-Ports betrachtet werden. Diese werden wie zuvor durch zusätzliche Input- und Output-Ports modelliert, welche den Objekträumen vor bzw. nach der Abarbeitung der gesamten Schleife bzw. einer Iteration der Schleife zugeordnet sind.

Bei der soeben beschriebenen Vorgehensweise ist zu beachten, daß die Ports `objectspace_in` und `objectspace_out` nur bei Komponenten notwendig sind, welche tatsächlich Objekte aus dem Objektraum ansprechen bzw. welche tatsächlich Veränderungen im Objektraum durchführen. Durch Weglassen überflüssiger Ports kann die Anzahl der benötigten Objekträume – und damit der Aufwand bei der Propagierung der Werte entlang der Verbindungen – stark herabgesetzt werden. Der Output-Port kann bei Komponenten weggelassen werden, welche keine Instanzvariablen verändern. Dies sind z.B. Methodenaufrufe und Schleifen ohne Nebeneffekte in Instanzvariablen. In ähnlicher Weise können mit Objekträumen assoziierte Input-Ports weggelassen werden, falls diese nicht benötigt werden. Dies umfaßt z.B. Methodenaufrufe, Schleifen und Auswahlanweisungen, deren Modelle keine Referenzen auf Instanzvariablen enthalten.

Nach der Beschreibung des strukturellen Aufbaues des Modells bleibt noch die Beschreibung des Verhaltens der einzelnen Elemente des Modells. Im folgenden wird die Verhaltensbeschreibung der Komponenten erläutert.

4.2.2 Verhaltensbeschreibung der Komponenten des indirekten Modells

Im Gegensatz zu dem Modell aus den Abschnitt 4.1 fällt die Verhaltensbeschreibung für das hier behandelte Modell komplexer aus, da nicht nur Werte von den Input- zu den Output-Ports propagiert werden müssen, sondern auch die Werte der Instanzvariablen der Objekte in den Objekträumen vor bzw. nach der entsprechenden Anweisung in Betracht gezogen werden müssen. Andererseits können auf diese Weise viele durch rekursive Datenstrukturen, rekursive Methodenaufrufe oder Aliasing-Relationen bedingte Probleme des direkten Modells vermieden werden.

Die Verhaltensbeschreibung der Komponenten bildet die Effekte der durch die Komponenten repräsentierten Anweisungen oder Anweisungsteile nach. Dabei wird jeweils von einer Umgebung und einem Objektraum ausgegangen, welche vor der Abarbeitung der der Komponente entsprechenden Anweisung gültig sind. Als Ergebnis werden eine Umgebung und ein Objektraum berechnet, welche nach der Abarbeitung der Anweisung gültig sind. Das Verhalten der Komponente bzw. ihrer entsprechenden Anweisung kann also als eine Funktion beschrieben werden, welche ein Paar $(Env, ObjectSpace)$ in ein Paar $(Env', ObjectSpace')$ transformiert, wobei Env eine Umgebung und $ObjectSpace$ einen Objektraum darstellen. Eine graphische Repräsentation ist in Abbildung 4.9 dargestellt.

Um die Semantik der Java-Sprache korrekt zu modellieren, muß das Verhalten der einer Anweisung entsprechenden Komponente so gewählt werden, daß die berechnete Umgebung Env' genau jene Effekte widerspiegelt, welche die Abarbeitung der Anweisung, ausgehend von der Umgebung Env, ergeben würde. Analoges gilt für den Objektraum $ObjectSpace'$. Das Verhalten einer Komponente spezifiziert also eine Abhängigkeit der Umgebungen bzw. der Objekträume vor und nach der Abarbeitung der mit der Komponente assoziierten Anweisung.

Im folgenden werden die Verhaltensmodelle der einzelnen Komponenten des Modells näher betrachtet:

- Das Verhaltensmodell von *Zuweisungen* gestaltet sich komplexer als das des direkten Modells in Abschnitt 4.1.3, da hier zwei Fälle zu unterscheiden sind: Die Variable auf der linken Seite der Zuweisung kann entweder eine lokale Variable oder eine Klassenvariable sein. Im ersten Fall ist das Verhalten mit jenem des direkten Modells äquivalent:

  $$\neg ab(C) \Rightarrow \texttt{in}(C) = \texttt{out}(C),$$

  wobei $ab(C)$, $\texttt{in}(C)$ und $\texttt{out}(C)$ wie in Abschnitt 4.1.3 definiert sind.

  Ist die Zielvariable der Zuweisung hingegen eine Instanzvariable, werden die Effekte der Zuweisung im Objektraum repräsentiert:

  $$\neg ab(C) \Rightarrow \texttt{objectspace\_out}(C) = objectspace',$$

  wobei $objectspace'$ aus $\texttt{objectspace\_in}(C)$ durch Ersetzen des Wertes der Instanzvariablen `variableId` des Objekts, welches durch den Objekt-Identifier `object(C)` identifiziert wird, durch den Wert von $\texttt{in}(C)$ erzeugt wird. Alle anderen Instanzvariablen im Objektraum werden unverändert übernommen:

  $$\begin{aligned} objectspace' = \texttt{objectspace\_in}(C) & \\ - \left\{ val(O,F,V) \;\middle|\; \begin{array}{l} val(O,F,V) \in \texttt{objectspace\_in}(C), \\ O = \texttt{object}(C), F = \texttt{variableId} \end{array} \right\} & \\ \cup \{ val(\texttt{object}(C), \texttt{variableId}, \texttt{in}(C)) \}. & \end{aligned}$$

`in`(C) bezeichnet den Wert des mit der Zuweisung assoziierten Ausdrucks und `object(C)` den am Input-Port `object` der Komponente C anliegenden Wert, welcher den Objekt-Identifier des angesprochenen Objekts darstellt. `variableId` stellt den *kanonischen Namen*[8] der angesprochenen Instanzvariablen dar. $val(O, F, V)$ bezeichnet, daß die Variable F des durch den Identifier O angesprochenen Objekts den Wert V aufweist. Hier ist zu beachten, daß für jede Kombination aus Objekt-Identifier und Instanzvariablen immer höchstens ein Atom der Form $val(O, F, V)$ im Objektraum enthalten ist.

Bei Zuweisungen an eine Variable, welche Objekt-Identifier enthält, kann der Fall eintreten, daß nach der Zuweisung Objekte im Objektraum weder ausgehend von lokalen Variablen, noch ausgehend von Klassenvariablen erreichbar sind. In diesem Fall können die Objekte vom Programm aus nicht mehr angesprochen werden und können daher verworfen werden. Dies entspricht dem Garbage-Kollektor des Java-Laufzeitsystems, welcher periodisch nicht mehr ansprechbare Objekte aus dem Speicher entfernt. Da die eingeschränkte Sprache keine Rücksicht auf `finalize`-Methoden nimmt, ist es nicht zwingend notwendig, diese Funktionalität nachzubilden, um die Korrektheit des Modells zu gewährleisten. Aufgrund des erhöhten Aufwands beim Kopieren des Objektraums kann es jedoch von Vorteil sein, diese Objekte zu entfernen. Dies kann z.B. durch eine untere Schranke für die Größe des Objektraums gesteuert werden.

- Die Funktionsweise von *Auswahlanweisungen* kann als eine Art Auswahlfunktion dargestellt werden, da in Abhängigkeit von dem Ergebnis der Auswertung der Bedingung die Effekte eines der beiden Zweige der Auswahlanweisung als Effekte der gesamten Anweisung „durchgeschaltet“ werden. Einerseits müssen die veränderten Umgebungen entsprechend dem Ergebnis der Auswertung der Auswahlbedingung ausgewählt werden, andererseits die entsprechende Version des Objektraums an den Ausgang weitergereicht werden. Die Veränderungen der Umgebung, d.h. die modifizierten lokalen Variablen und Klassenvariablen, werden durch Output-Ports von Komponenten in den jeweiligen Zweigen der Auswahlanweisung repräsentiert. Folglich ist dieser Teil der Verhaltensbeschreibung äquivalent zu jener aus Abschnitt 4.1.3:

$$\neg ab(C) \wedge \texttt{cond}(C) = true \Rightarrow \texttt{out\_v}(C) = \texttt{then\_v}(C)$$

bzw.

$$\neg ab(C) \wedge \texttt{cond}(C) = false \Rightarrow \texttt{out\_v}(C) = \texttt{else\_v}(C),$$

wobei v für jede mit den Ports `out_x(C)` assoziierte Variable x sowie den Objektraum (falls Ports dafür existieren) instanziert werden muß.

- Das Verhalten von *Methodenaufrufen* kann durch Erweiterung der Modelle der aufzurufenden Methoden erhalten werden. Zunächst ist die aufzurufende Methode zu ermitteln. Dies erfolgt durch Abfragen des Typs jenes Objekts im Objektraum, auf das der Identifier $object(C)$ verweist. Mit Hilfe dieses Typs kann anschließend die aufzurufende Methode identifiziert werden und das entsprechende Modell M ausgewählt werden. Das Modell für den Methodenaufruf ist dann wie folgt zu konstruieren: für jede Regel der Form

$$(A \wedge \neg ab(C_M) \Rightarrow B) \in M$$

wird eine Regel

$$\neg ab(C) \wedge A' \Rightarrow B'$$

[8] Der *kanonische Name* einer Instanzvariablen ist der voll qualifizierte Name der Klassen-Deklaration der die Variable deklarierenden Klasse, gefolgt von dem Namen der Variablen.

erstellt, wobei A' aus A und B' aus B durch Ersetzen der Platzhalter für Parameter und Klassenvariablen durch die Input- und Output-Ports der Methodenaufrufs-Komponente erhalten werden. Weiters müssen die Platzhalter für `this` in M durch $object(C)$ ersetzt werden. Auch sind die Platzhalter, welche mit den Objekträumen vor bzw. nach der Abarbeitung der Methode assoziiert sind, durch die entsprechenden Ports im Kontext des Aufrufs zu ersetzen.

Hier ist zu bemerken, daß obige Konstruktion das Modell M der aufzurufenden Methode benötigt, um das Modell für den Methodenaufruf zu generieren. Daraus folgt, daß auf diese Weise keine rekursiven Methodenaufrufe modelliert werden können. Diese Einschränkung kann etwa dadurch umgangen werden, daß ab einer gewissen Aufruftiefe eine spezielle Methodenaufrufs-Komponente verwendet wird, welche – anstatt ein komponentenbasiertes Modell der Methode zu verwenden – einen Evaluator verwendet, um den neuen Zustand $(Env_{out}(M), ObjectSpace_{out}(M))$ zu berechnen. $Env_{out}(M)$ repräsentiert die Veränderungen der Umgebung, d.h. $Env_{out}(M)$ enthält die Werte der den Output-Ports von M zugeordneten Variablen. $ObjectSpace_{out}(M)$ ist dem durch die Methode berechneten Objektraum – und somit dem Output-Port `objectspace_out` der Komponente – zugeordnet. Nachteil dieser Methode ist, daß ein Zurückpropagieren der Werte von den Ausgängen zu den Eingängen des Modells nicht möglich ist. Auch ist nicht bekannt, welche Objekte und Instanzvariablen im rekursiven Aufruf der Methode benötigt werden. Daher muß die konservative Annahme getroffen werden, daß alle Objekte und Instanzvariablen benötigt werden, welche ausgehend von Parametern und benötigten lokalen Variablen und Klassenvariablen angesprochen werden können. Daraus folgt, daß der Evaluator nur dann aufgerufen werden kann, wenn alle diese Variablen einen definierten Wert aufweisen. Andernfalls können keine Aussagen über die Effekte des Methodenaufrufs getroffen werden. Dies stellt gegenüber der Modellierung mittels Komponenten eine Einschränkung dar, da durch ein Komponentenmodell in einigen Fällen auch Aussagen über eine Untermenge der modifizierten Variablen getroffen werden können.

- Die Modellierung einer ***`return`-Anweisung*** erfolgt analog zu einer Zuweisung an eine lokale Hilfsvariable `return` und wird hier nicht weiter behandelt. Die Modifikation der Verhaltensbeschreibung im Fall mehrerer `return`-Anweisungen in einer Methode erfolgt analog zu der in Abschnitt 4.1.3 beschriebenen.

- Die Behandlung von *Ausdrücken* wird weitgehend unverändert aus Abschnitt 4.1.3 übernommen.

  - Im Fall von *Variablenreferenzen* sind hier jedoch zwei Fälle zu unterscheiden: Repräsentiert die Komponente eine lokale Variable oder eine Klassenvariable, kann das Modell unverändert übernommen werden. Wird hingegen eine Instanzvariable des aktuellen Objekts angesprochen, erfolgt der Zugriff auf die Variable durch den Objektraum:

    $$\neg ab(C) \Rightarrow \texttt{result}(C) = V$$

    mit

    $$val(\texttt{object}(C), \texttt{variableId}, V) \in \texttt{objectspace\_in}(C),$$

    wobei `variableId` den kanonischen Namen der angesprochenen Instanzvariablen und `object`(C) den Objekt-Identifier des angesprochenen Objekts bezeichnet. `objectspace_in`(C) repräsentiert den Objektraum, welcher vor Ausführung des Ausdrucks gültig ist. Der Wert von V ist eindeutig bestimmt, da im Objektraum für jede Kombination aus Objekt-Identifier und Instanzvariable höchstens ein Eintrag existieren kann.

– Repräsentiert eine Komponente einen *Konstruktor-Aufruf* durch einen `new`-Ausdruck, muß das dadurch neu angelegte Objekt mit einem eindeutigen Identifier assoziiert werden und in den Objektraum eingefügt werden. Zunächst muß ein neues Objekt des entsprechenden Typs generiert und in den Objektraum, welcher vor der Abarbeitung des Konstruktors gültig ist, eingefügt werden und dessen Variablen mit deren Default-Werten initialisiert werden:

$$\begin{aligned} objectspace' = \texttt{objectspace\_in}(C) \\ \cup \{val(newObjId, F, defaultValue(F)) \mid F \in InstVars\} \\ \cup \{type(newObjId, Type)\}, \end{aligned}$$

wobei $objectspace'$ den Objektraum nach dem Einfügen des Objekts darstellt und $newObjId$ den Objekt-Identifier des generierten Objekts repräsentiert. $defaultValue(F)$ stellt den Default-Wert der Instanzvariablen F dar. Für Variablen primitiver Datentypen ist der Default-Wert '`0`', für alle anderen Typen '`null`'. $InstVars$ bezeichnet die Menge aller Instanzvariablen der Klasse $Type$ des erzeugten Objekts. $type(newObjId, Type)$ stellt eine Assoziation zwischen dem Objekt-Identifier und dem Typ mit dem Namen $Type$ des erzeugten Objekts her. Diese Information wird bei Aufrufen von polymorphen Methoden benötigt, um anhand des Objekt-Identifiers das Modell der entsprechenden Methode auszuwählen.

Der auf diese Weise erhaltene Objektraum $objectspace'$ wird bei der Substitution der Platzhalter für den Objektraum im Modell des aufgerufenen Konstruktors anstelle des ursprünglichen Objektraums verwendet. Wird der Objektraum im Modell des Konstruktors nicht verändert, gilt $\texttt{objectspace\_out}(C) = objectspace'$. Andernfalls wird $\texttt{objectspace\_out}(C)$ der Wert der Verbindung `objectspace_out` des Modells des Konstruktors zugewiesen.

Als berechnetes Ergebnis der Komponente wird der Identifier des erzeugten Objekts zurückgeliefert: $\texttt{result}(C) = newObjId$.

- Das Verhalten von Komponenten, welche `while`-Schleifen repräsentieren, wird im wesentlichen analog zu dem in Abschnitt 4.1.3 beschriebenen Verhaltensmodell definiert. Hier ist zu beachten, daß – analog zu Auswahlanweisungen und zu Methodenaufrufen – keine Instanzvariablen von Objekten betrachtet werden. Zusätzlich zu den Hilfsvariablen v_X^i sind noch Variablen os^i notwendig, welche den Status des Objektraums vor jeder Iteration enthalten. Die Modellierung dieser Variablen erfolgt analog zu den Hilfsvariablen v_X^i.

Beispiel 4.4 Zur Illustration des indirekten Modells sei das Programm aus Abbildung 4.7 nochmals betrachtet. Eine Repräsentation des Programms mit dem in diesem Abschnitt vorgestellten indirekten Modell ist in Abbildung 4.10 dargestellt.

Das Modell ist jenem aus Abbildung 4.8 sehr ähnlich, es sind jedoch zusätzliche Ports und Verbindungen eingefügt, welche den Objekträumen vor bzw. nach der Abarbeitung der einzelnen Anweisungen zugeordnet sind. Dadurch werden zusätzliche Abhängigkeiten in das Modell eingefügt, welche den Datenfluß durch Instanzvariablen abbilden.

Zu jeder Verbindung ist der dafür berechnete Wert bzw. der damit assoziierte Objektraum dargestellt. Dabei wird davon ausgegangen, daß die Methode `demo()` mit einem leeren Objektraum als Input am Port `objectspace_in` aufgerufen wird. Es ist zu beobachten, wie sich durch den Aufruf der Methode `l.setValue(2)` zwar der Wert des durch den Objekt-Identifier `#obj1` identifizierten Objekts verändert – die Instanzvariable `value` wird auf den Wert `2` gesetzt – der zugehörige Identifier bleibt jedoch unverändert. Die im Modell aus Abbildung 4.8

fehlende Abhängigkeit zwischen den Anweisungen in Zeile 21 und 22 wird hier durch die zusätzliche Verbindung der beiden Komponenten modelliert, welche die Veränderung des Objektraums widerspiegelt.

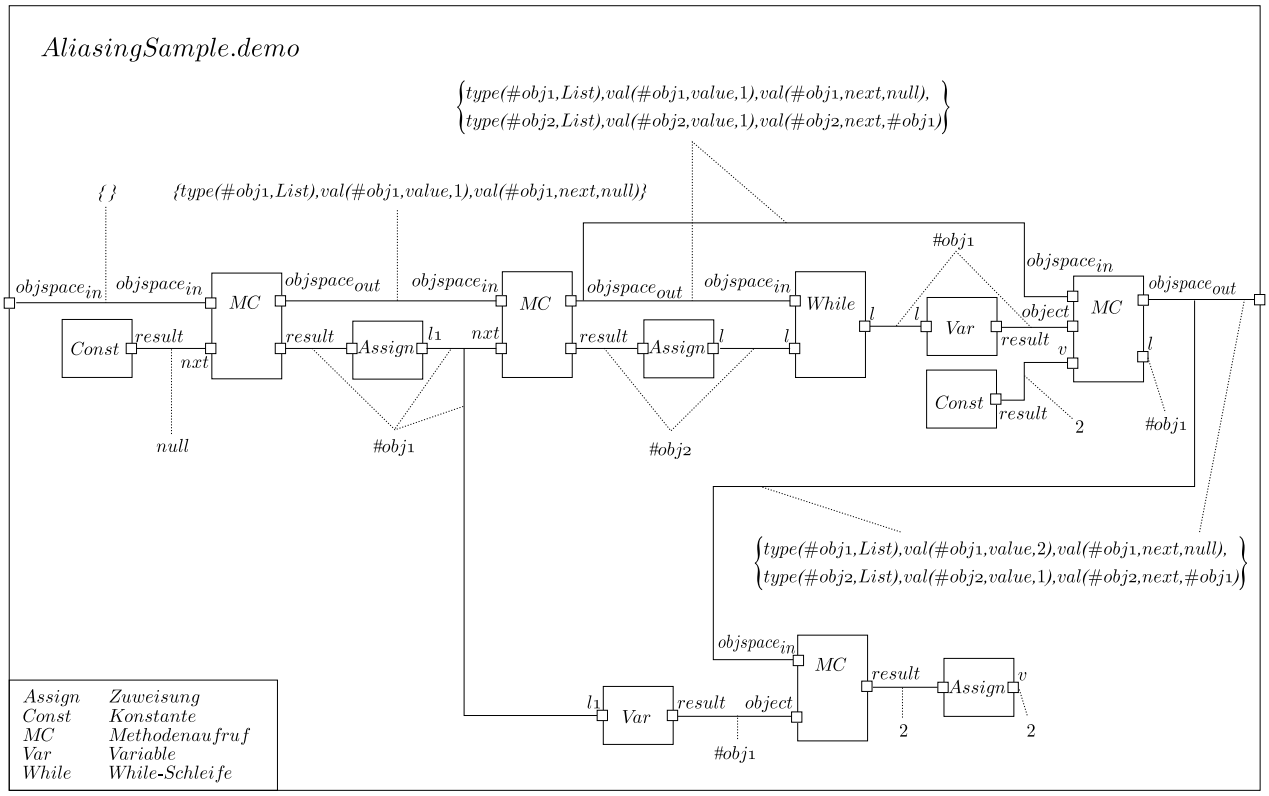

Abbildung 4.10: Indirektes Modell des Programms aus Abbildung 4.7

An dieser Stelle sei nochmals auf einige Besonderheiten des Modells hingewiesen:

- Bei den Komponenten, welche Zuweisungsanweisungen repräsentieren, ist anzumerken, daß diese den Objektraum nicht verändern, da die den Komponenten zugeordneten Variablen in diesem Beispiel lokale Variablen sind. Daher sind den Komponenten auch keine Input- und Output-Ports für die Objekträume beigefügt.
- Weiters fehlt der Port `object` bei den Konstruktoraufrufen und der gesamten Methode `demo`, da diese Methoden nach außen hin Klassenmethoden darstellen und der vom Modell des Konstruktors benötigte Input-Parameter `object` von der den Aufruf repräsentierenden Komponente generiert wird.
- Der Komponente, welche den Methodenaufruf von `l1.getValue()` repräsentiert, fehlt der Output-Port `objectspace_out`, da durch Aufrufen dieser Methode der Objektraum nicht verändert wird.

Um das soeben vorgestellte Modell zur modellbasierten Diagnose verwenden zu können, ist es erforderlich, einen Mechanismus zu schaffen, mit dessen Hilfe inkonsistente Teilmengen von Anweisungen ermittelt werden können. Im Fall der lokalen und globalen Variablen können Inkonsistenzen durch Propagieren der Werte entlang der Verbindungen zwischen den Komponenten aufgezeigt werden. Eine Inkonsistenz kann bestimmt werden, falls von zwei Komponenten unterschiedliche Werte für eine Verbindung bestimmt werden. Dies entspricht genau

der herkömmlichen Vorgehensweise des direkten Modells. Zusätzlich dazu ist es notwendig, Inkonsistenzen innerhalb der Objekträume festzustellen, d.h. falls für eine Instanzvariable zwei unterschiedliche Werte berechnet werden. Dies erfolgt durch Feststellen, ob eine Instanzvariable bereits einen Wert zugewiesen bekommen hat. Wird einer Instanzvariablen ein Wert zugewiesen und ist dieser Wert ungleich zu dem bereits in der Variablen gespeicherten Wert, ist eine Inkonsistenz vorhanden. Dies kann beim Zusammenführen zweier für eine Verbindung berechneter Objekträume erfolgen. Eine Inkonsistenz zwischen zwei Objekträumen OS_1 und OS_2 tritt auf, falls ein Objekt-Identifier O und eine Instanzvariable F existieren, für die gilt:

$$\exists val(O, F, V_1) \in OS_1$$

und

$$\exists val(O, F, V_2) \in OS_2$$

mit

$$V1 \neq V2.$$

Auf Basis der auf diese Weise gefundenen Inkonsistenzen können anschließend die daran beteiligten Komponenten berechnet werden und dem Diagnosealgorithmus aus Kapitel 3 übergeben werden. Somit kann eine Diagnose für das Modell – und damit auch für das dadurch abgebildete Programm – ermittelt werden.

Durch die Einführung von Objekträumen und Objekt-Identifiern wird es möglich, das Modell noch weiter zu verfeinern. Im folgenden Abschnitt werden zwei Erweiterungen des Modells vorgestellt, welche die Qualität der mit dem Modell berechenbaren Diagnosen verbessern bzw. den durch das Modell darstellbaren Sprachumfang erweitern.

4.2.3 Erweiterungen des indirekten Modells

In diesem Abschnitt werden zwei mögliche Erweiterungen des indirekten Modells behandelt, welche die Qualität des Modells bzw. dessen Ausdrucksstärke verbessern. Einerseits ist es möglich, die Objekträume in kleinere Einheiten zu unterteilen, was die Abhängigkeiten zwischen den einzelnen Anweisungen des zu untersuchenden Programms besser widerspiegelt. Andererseits kann das Modell dahingehend erweitert werden, daß auch Arrays und Strings modelliert werden können.

Unterteilen des Objektraums

Das in den beiden vorangegangenen Abschnitten behandelte indirekte Modell ist aufgrund seines Aufbaues mächtiger als das davor behandelte direkte Modell. Die Probleme des direkten Modells werden umgangen, indem alle Instanzvariablen von Objekten im Objektraum zusammengefaßt werden. Der Zugriff auf diese Variablen erfolgt dann über Identifier, welche das anzusprechende Objekt im Objektraum identifizieren. Die Semantik der Programmelemente kann als Menge von Funktionen angegeben werden, wobei die einzelnen Funktionen die Zusammenhänge zwischen Umgebungen und Objekträumen vor und nach der Abarbeitung des jeweiligen Programmelements darstellen.

Aufgrund der Modellkonstruktion werden die Objekträume als Verbindungen zwischen den die Programmelemente repräsentierenden Komponenten modelliert. Da diese Verbindungen nicht nur die tatsächlich benötigten Instanzvariablen, sondern alle Instanzvariablen aller im Programm bis dahin erzeugten Objekte repräsentieren, können sich dadurch viele – im ursprünglichen Programm nicht gegebene – Abhängigkeiten zwischen Programmelementen ergeben.

Diese zusätzlichen Abhängigkeiten können das Finden von Diagnosen im Modell stark erschweren, da bei Vorhandensein von zusätzlichen Abhängigkeiten eine größere Anzahl von Beobachtungen notwendig sein kann, um eine eindeutige Diagnose zu erhalten. Zunächst kann

die Anzahl der Verbindungen, welche Objekträume repräsentieren, vermindert werden, indem Komponenten nur dann mit Verbindungen für Objekträume ausgestattet werden, wenn diese tatsächlich benötigt werden. So benötigt z.B. eine Komponente, die einen lesenden Zugriff auf eine Instanzvariable modelliert, keine Output-Verbindung für den Objektraum, da dieser durch das Verhaltensmodell der Komponente nicht verändert wird. Diese Optimierungen wurden schon in den letzten beiden Abschnitten behandelt und ermöglichen es, die Anzahl der Verbindungen stark zu reduzieren.

Um die Anzahl der benötigten Verbindungen noch weiter herabzusetzen, kann ein weiterer Ansatz verfolgt werden: Der Objektraum selbst wird unterteilt und durch separate Verbindungen repräsentiert. Die feinste mögliche Unterteilung ist jene, bei der ein Teil des geteilten Objektraums genau einer Definition einer Instanzvariablen zugeordnet ist. D.h. ein Teil eines Objektraums enthält alle Werte aller im Programm erzeugten Objekte für eine Instanzvariable einer Klasse. Eine Unterteilung in kleinere Teilmengen ist nicht ohne weiteres möglich. Um dies zu erreichen, wäre eine detaillierte Analyse der Struktur der Verweise zwischen den einzelnen Objekten zur Übersetzungszeit notwendig. Da solche Verfahren gegenüber der Unterteilung auf Ebene der Instanzvariablen um einige Größenordnungen aufwendiger sind (diese Verfahren sind im wesentlichen äquivalent zu jenen der Aliasing-Analyse), kann die hier gewählte Unterteilung als geeigneter Kompromiß zwischen Effizienz und Exaktheit des Modells angesehen werden.

Durch das Unterteilen der Objekträume können künstliche Abhängigkeiten zwischen den Komponenten weitgehend vermieden werden, da die Komponenten nur mit jenen Verbindungen verbunden werden, welche tatsächlich benötigt bzw. verändert werden. Dadurch wird das Finden von Diagnosen vereinfacht, da die Menge der von einander abhängigen Komponenten – und damit die Anzahl der an Konflikten beteiligten Komponenten – in vielen Fällen stark herabgesetzt werden kann.

Durch diese Vorgehensweise wird zugleich auch der Aufwand beim Propagieren der Werte durch das Komponentenmodell vermindert, da bei unterteilten Objekträumen immer nur die einzelnen Teile der Objekträume kopiert und weiterpropagiert werden müssen, anstelle der gesamten Objekträume. Einen negativen Effekt dieses Ansatzes stellt die Tatsache dar, daß einige Komponenten eine große Anzahl von Input- und Output-Ports aufweisen. Dies kann bei naiver Vorgehensweise beim Propagieren der Werte zu einem erhöhten Aufwand führen, da das Verhalten einzelner Komponenten für jede dieser Verbindungen erneut abgearbeitet werden muß. Dieser Nachteil läßt sich durch eine geeignete Reihenfolge und einige Sorgfalt beim Propagieren der Werte weitgehend vermeiden.

Beispiel 4.5 Zur Illustration des Modells mit unterteilten Objekträumen betrachte man das Java-Programm in Abbildung 4.11. Die Methode `demo()` des Programms legt eine Instanz der Klasse `Pair` an und arbeitet eine Sequenz von Lese- und Schreiboperationen auf den beiden Instanzvariablen `value1` und `value2` des Objekts ab. Das Komponentenmodell der Methode `demo()` ist in Abbildung 4.12 zu betrachten. Die zusätzliche Abhängigkeit zwischen der Komponente $[\texttt{p.getValue1()}]_{29}$, welche das Auslesen der Instanzvariablen `value1` durch den Aufruf der Methode `Pair.getValue1()` in Zeile 29 repräsentiert und der Komponente $[\texttt{p.setValue2(3)}]_{27}$,, welche eine Veränderung der Instanzvariablen `value2` durch den Aufruf der Methode `setValue(3)` repräsentiert (in Zeile 27), ist klar ersichtlich.

Wird nun eine Beobachtung für die Variable `v1` nach der Zuweisung in Zeile 29 angegeben, welche dem berechneten Wert `1` von `v1` widerspricht, ergeben sich folgende Einfachfehlerdiagnosen[9]: $\{\{[\texttt{1}]_{25}\}, \{[\texttt{new Pair(1,2)}]_{25}\}, \{[\texttt{p=new Pair(1,2)}]_{25}\}, \{[\texttt{p}]_{27}\}, \{[\texttt{p.setValue2(3)}]_{27}\}, \{[\texttt{p}]_{29}\}, \{[\texttt{p.getValue1()}]_{29}\}, \{[\texttt{v1=p.getValue1()}]_{29}\}\}$.

[9]Die Komponenten werden durch deren zugeordnete Ausdrücke oder Anweisungen des Programms zusammen mit der Zeilennummer ihres Auftretens angegeben.

```
1  class Pair {
2      int value1;
3      int value2;
4
5      Pair(int value1, int value2) {
6          this.value1 = value1;
7          this.value2 = value2;
8      }
9      int getValue1() {
10         return value1;
11     }
12     void setValue1(int value) {
13         value1 = value;
14     }
15     int getValue2() {
16         return value2;
17     }
18     void setValue2(int value) {
19         value2 = value;
20     }
21 }
22
23 public class DependencySample {
24     public static void demo() {
25         Pair p = new Pair(1,2);
26
27         p.setValue2(3);
28
29         int v1 = p.getValue1();
30         int v2 = p.getValue2();
31     }
32 }
```

Abbildung 4.11: Programm zur Demonstration von künstlichen Abhängigkeiten

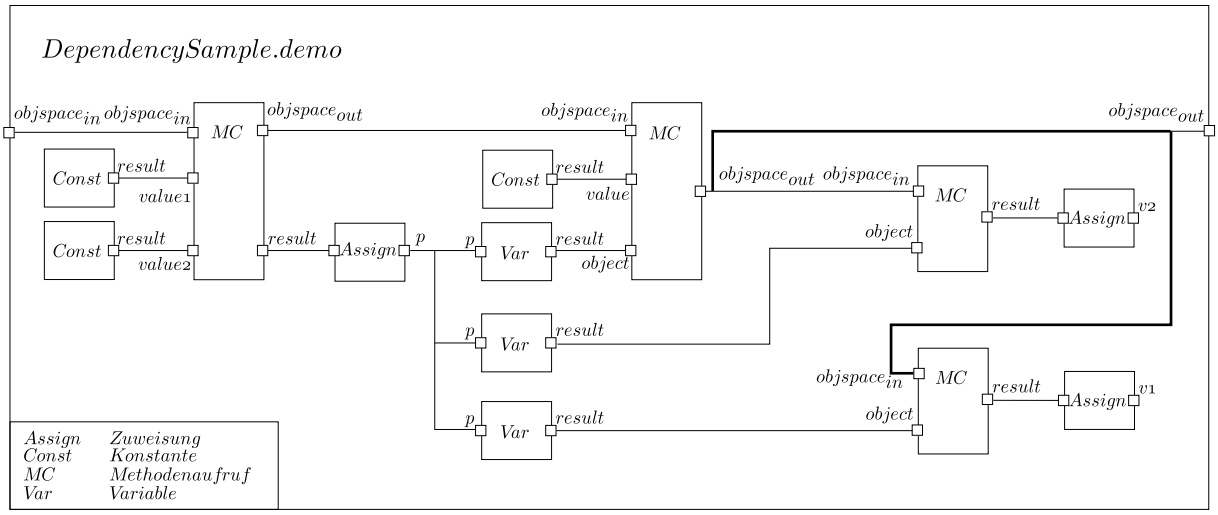

Abbildung 4.12: Indirektes Modell des Programms aus Abbildung 4.11. Die durch das Modell künstlich eingeführte Abhängigkeit ist hervorgehoben dargestellt

Die Diagnosen enthalten zwar alle erwarteten Einfachfehlerdiagnosen, zusätzlich dazu sind aber noch weitere Diagnosen enthalten, welche nur aufgrund des gegebenen Programms nicht zu erwarten sind. Dies sind die Diagnosen $\{[\texttt{p}]_{27}\}$ und $\{[\texttt{p.setValue2(3)}]_{27}\}$. Diese werden durch die zuvor genannte künstliche Abhängigkeit hervorgerufen, da durch das Zusammenfassen der beiden Instanzvariablen `value1` und `value2` in einem Objektraum nicht unterschieden werden kann, welche der beiden Variablen durch die Methode `setValue2(int)` verändert wird.

Durch Unterteilen des Objektraums in zwei Verbindungen, welche den Instanzvariablen `value1` und `value2` zugeordnet sind, wird diese zusätzliche Abhängigkeit vermieden. Das Modell mit geteilten Objekträumen ist in Abbildung 4.13 dargestellt. Man beachte, daß die Komponente, welche die Anweisung `p.getValue1()` in Zeile 29 repräsentiert, nicht mehr von der Anweisung `p.setValue2(2)` abhängig ist, sondern nur mehr direkt von der Anweisung `new Pair(1,2)`.

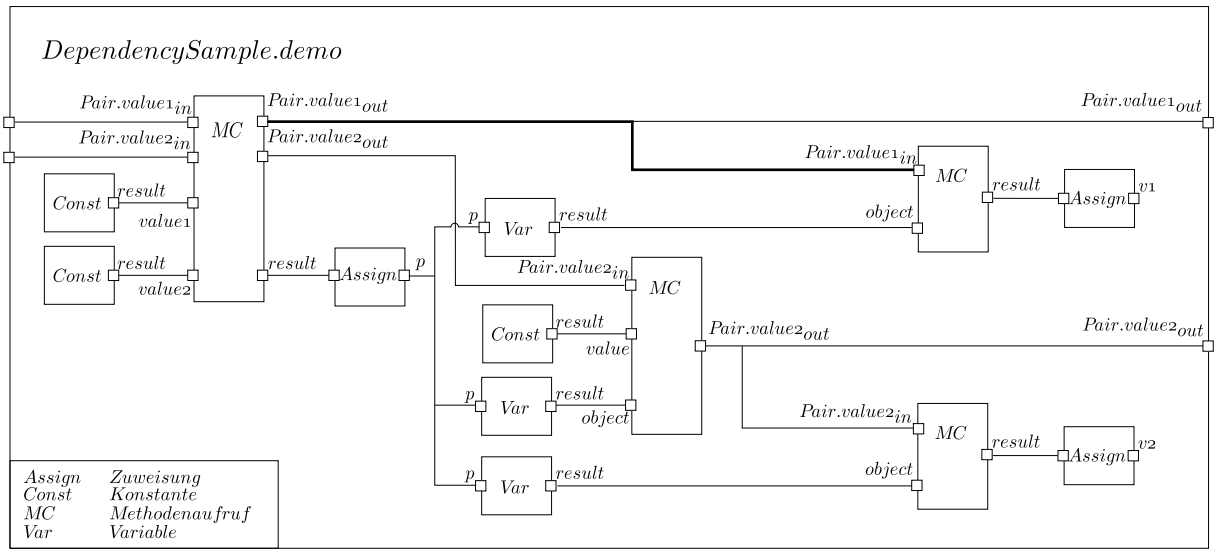

Abbildung 4.13: Modell mit unterteilten Objekträumen für das Programm aus Abbildung 4.11. Die durch das Teilen der Objekträume veränderte Abhängigkeit ist hervorgehoben dargestellt

Sei nun die Beobachtung aus dem Beispiel in dieses Modell übernommen. Dann ergeben sich die folgenden Diagnosen: $\{\{[\texttt{1}]_{25}\}, \{[\texttt{new Pair(1,2)}]_{25}\}, \{[\texttt{p=new Pair(1,2)}]_{25}\}, \{[\texttt{p}]_{29}\}, \{[\texttt{p.getValue1()}]_{29}\}, \{[\texttt{v1=p.getValue1()}]_{29}\}\}$. Wie erwartet enthält die Menge nur mehr die intuitiv erwarteten Diagnosen, da die beiden unerwünschten Diagnosen aufgrund der vermiedenen Abhängigkeit nicht mehr berechnet werden.

Modellierung von Arrays und Strings

Da in einer großen Anzahl von Java-Programmen Arrays und Strings verwendet werden, wird in der Folge eine Erweiterung des bisher beschriebenen, mit unterteilten Objekträumen operierenden Modells vorgestellt, welches auch die Modellierung von Arrays und Strings ermöglicht. In diesem Abschnitt wird vorwiegend auf die Modellierung von Arrays eingegangen, da die Implementierung der Java-Klasse `java.lang.String` im wesentlichen auf Arrays basiert und Strings daher unter Verwendung dieser Implementierung leicht modelliert werden können[10].

[10]Es ist eine zusätzliche Komponente notwendig, welche String-Literale modelliert. Aufgabe dieser Komponente ist es, für jedes im Quelltext des Programms angegebene String-Literal eine Instanz der Klasse `java.lang.String` zu erzeugen und diese in den entsprechenden Objektraum einzufügen.

Arrays werden in Java als nicht-primitive Datentypen angesehen, d.h. Instanzen werden immer über Verweise angesprochen. Für die Modellbildung bedeutet dies, daß Arrays stets im Objektraum modelliert werden. Dies entspricht der in den vorangegangenen Abschnitten besprochenen Vorgehensweise bei der Modellierung von Klassen.

Die für die Modellbildung relevanten Aspekte eines Arrays sind dessen Länge und Inhalt. Arrays können also in gewisser Weise als parametrisierte Klasse angesehen werden, welche zwei Instanzvariablen besitzt:

- Die Instanzvariable `data` enthält für jeden möglichen Index des Arrays den zugeordneten enthaltenen Wert. Dies ist im Fall von primitiven Datentypen der Wert selbst, im Fall von nicht-primitiven Datentypen ein Verweis auf das an dieser Stelle enthaltene Objekt (oder `null`, falls auf kein Objekt verwiesen wird).
- Die Instanzvariable `length` enthält die Länge des Arrays. Diese ist konstant und kann nach dem Erzeugen des Arrays nicht verändert werden.

Der Parameter der Klasse ist der Typ der Werte, welche im Array gespeichert werden können. Abbildung 4.14 zeigt eine mögliche Definition einer solchen `ArrayRepresentation`-Klasse.

```
class ArrayRepresentation<Type> {
    public Set<Index↦Type> data;
    public final int length;
}
```

Abbildung 4.14: Für die Modellbildung relevante Aspekte eines Arrays. `Type` gibt den Typ der im Array enthaltenen Elemente an

Die Modellbildung für Arrays erfolgt auf folgende Weise: ein Array vom Typ `Type` wird durch eine Instanz der in Abbildung 4.14 abgebildeten Klasse `ArrayRepresentation<Type>` dargestellt, wobei für jeden Typ `Type` eine eigene Klasse generiert wird. Abbildung 4.14 stellt also in gewisser Weise ein Schema für das Generieren von Klassen dar, vergleichbar mit Templates in C++. Darüber hinaus benötigt die Instanzvariable `data` eine spezielle Behandlung, da Zugriffe auf diese Variable durch die Verhaltensbeschreibung der Komponenten modelliert werden, welche Zuweisungen an Arrays (z.B. `array[index]=value`), Zugriffe auf Arrays (z.B. `array[index]`) und das Erzeugen neuer Arrays (z.B. `new int[size]`) repräsentieren.

Um die Repräsentation von Arrays im indirekten Modell abzuschließen, werden im folgenden Modelle für Zuweisungen an Arrays, Zugriffe auf Arrays und das Erzeugen neuer Arrays angegeben.

- *Zugriffe auf Arrays* werden ähnlich wie Zugriffe auf Instanzvariablen von Klassen repräsentiert. Es wird eine Komponente mit drei Input-Ports und einem Output-Port angelegt. Analog zu Zugriffen auf Instanzvariablen repräsentiert der Input-Port `object` den Objekt-Identifier des Arrays, auf welches der Zugriff erfolgt. Der Input-Port `objectspace_in` ist mit dem aktuellen Objektraum vor der Abarbeitung des Zugriffs (oder – im Fall des Modells mit unterteilten Objekträumen – mit der Instanzvariablen `data` der Klasse des Arrays) assoziiert. Die Klasse des Arrays wird durch Instanzieren des Schemas aus Abbildung 4.14 mit dem Typ der Elemente des Arrays gebildet. Schließlich ist der Input-Port `index` dem Index jenes Array-Elements zugeordnet, auf welches der Zugriff erfolgt. Der Output-Port `result` ist dem Ergebnis des Array-Zugriffs zugeordnet.

  Die Verhaltensbeschreibung der Komponente kann durch einige Modifikationen aus jener aus Abschnitt 4.2.2 übernommen werden:

$$\neg ab(C) \Rightarrow \texttt{result}(C) = Value$$

mit

$$val(\texttt{object}(C), \texttt{index}(C), Value) \in \texttt{objectspace\_in}(C),$$

wobei C die Komponente angibt, welche den Zugriff modelliert. $\texttt{objectspace\_in}(C)$ bezeichnet den Objektraum vor der Abarbeitung des Zugriffs und $\texttt{object}(C)$ den Objekt-Identifier des Arrays. Die einzelnen Elemente des Arrays werden durch künstliche Instanzvariablen der Klasse `ArrayRepresentation<Type>` dargestellt, wobei der Name der Instanzvariablen jeweils dem Index im Array entspricht. Diese Variablen sind jedoch nur eine andere Form der Repräsentation des Inhalts der Variablen `ArrayRepresentation<Type>.data` und müssen daher bei Verwendung von unterteilten Objekträumen zu einer Verbindung (für die Variable `ArrayRepresentation<Type>.data`) zusammengefaßt werden.

- Für *Zuweisungen an Arrays* gilt ähnliches wie für lesende Zugriffe auf Arrays. Auch hier wird ein gegenüber dem Modell aus den Abschnitten 4.2.1 und 4.2.2 leicht verändertes Modell verwendet. Wie im Fall eines lesenden Zugriffs wird die Komponente um einen zusätzlichen Input-Port `index` erweitert, welcher dem Index des Zugriffs zugeordnet ist. Der Input-Port `objectspace_in` und der Output-Port `objectspace_out` sind wiederum dem Objektraum (bzw. der Variablen `ArrayRepresentation<Type>.data`) vor bzw. nach der Abarbeitung der Zuweisung zugeordnet. Alle weiteren Ports werden unverändert aus dem bisherigen Modell übernommen.

  Die Verhaltensbeschreibung der Komponente kann durch geeignete Modifikationen aus jener in Abschnitt 4.2.2 übernommen werden. Der modifizierte Objektraum $objectspace'$ wird hier durch die Formel

  $$\begin{aligned} objectspace' = {} & \texttt{objectspace\_in}(C) \\ & - \left\{ val(O, I, V) \;\middle|\; \begin{array}{l} val(O, I, V) \in \texttt{objectspace\_in}(C), \\ O = \texttt{object}(C), I = \texttt{index}(C) \end{array} \right\} \\ & \cup \{ val(\texttt{object}(C), \texttt{index}(C), \texttt{in}(C)) \}. \end{aligned}$$

  bestimmt, wobei $\texttt{objectspace\_in}(C)$ den Objektraum vor der Abarbeitung der Zuweisung und $\texttt{index}(C)$ den bei der Zuweisung verwendeten Index im Array repräsentiert. $\texttt{in}(C)$ stellt den zuzuweisenden Wert dar. Alle anderen Ausdrücke bleiben gegenüber der bisherigen Verhaltensbeschreibung unverändert.

- Bei Komponenten, welche das *Erzeugen neuer Arrays mittels* `new`*-Ausdrücken* modellieren, sind umfangreichere Veränderungen notwendig, da in diesem Fall auf die Anzahl der Array-Dimensionen des Typs des erzeugten Arrays Rücksicht genommen werden muß.

  Für die weitere Beschreibung des Modells sei angenommen, daß das erzeugte Array m Dimensionen aufweist und die ersten n $(0 < n \leq m)$ Dimensionen im `new`-Ausdruck spezifiziert sind.

  Es wird eine Komponente mit einer variablen Anzahl von Input- und Output-Ports generiert. Zunächst sind n Ports `dim_i` $(i = 1, \ldots, n)$ notwendig, welche der Anzahl der Elemente in der jeweiligen Dimension (aufsteigend numeriert von links nach rechts im `new`-Ausdruck) zugeordnet sind. Weiters sind Input- und Output-Ports für die Objekträume vor bzw. nach der Abarbeitung des `new`-Ausdrucks hinzuzufügen. Wird das Modell mit unterteilten Objekträumen verwendet, sind stattdessen Ports für alle Instanzvariablen der Klassen `ArrayRepresentation<B`$[]^i$`>`[11] $(i = m - n + 1, \ldots, m)$ hinzuzufügen, wobei B den

[11] Hier wird $B[]^i$ als Kurzform für $\underbrace{B[]\cdots[]}_{i \text{ mal}}$ verwendet.

Basistyp[12] des Arrays darstellt. Dies ist notwendig, da durch den `new`-Ausdruck Instanzen dieser Klassen bzw. Arrays erzeugt werden und daher die Objekträume um die neu angelegten Objekte erweitert werden müssen. Schließlich ist noch der Output-Port `result` hinzuzufügen, welcher wie in Abschnitt 4.2.1 dem Objekt-Identifier des erzeugten Arrays zugeordnet ist.

Die Verhaltensbeschreibung der Komponente C kann im Fall des Modells ohne unterteilte Objekträume in ähnlicher Form wie in Abschnitt 4.2.2 angegeben werden. Der Wert von $\texttt{objectspace\_out}(C)$ wird dann nach folgendem Verfahren bestimmt:

$$defaultValue = \begin{cases} \texttt{(B)0} & \text{falls B primitiver Datentyp,} \\ \texttt{null} & \text{sonst} \end{cases}$$

$$element_{d_1,\ldots,d_l} = \begin{cases} unique\_oid(d_1,\ldots,d_l) & \text{falls } l < n, \\ \texttt{null} & \text{falls } l = n < m \\ defaultValue & \text{falls } l = m \end{cases}$$

$$fields_{d_1,\ldots,d_l} = \bigcup_{k=0}^{dim_{l+1}(C)-1} \{val(element_{d_1,\ldots,d_l}, k, element_{d_1,\ldots,d_l,k})\}$$

für alle l mit $0 \leq l < n$. Für $l = 0$ sei $d_1,\ldots,d_l = \epsilon$. $unique\_oid(d_1,\ldots,d_l)$ ist eine Funktion, welche für gegebene Indizes $d_1,\ldots,d_l$ einen eindeutigen Objekt-Identifier für das an dieser Stelle im Array enthaltene Objekt generiert.

Aufbauend auf obige Abkürzungen wird der durch die Komponente berechnete Objektraum durch

$$\neg ab(C) \Rightarrow \texttt{objectspace\_out}(C) = objectspace'$$

mit

$$objectspace' = \texttt{objectspace\_in}(C) \cup \bigcup_{\substack{d_1,\ldots,d_l \\ l<n}} \{val(element_{d_1,\ldots,d_l}, \texttt{ArrayRepresentation<B[]}^{m-l}\texttt{>.length}, \texttt{dim\_l}(C))\} \cup \bigcup_{\substack{d_1,\ldots,d_l \\ l<n}} fields_{d_1,\ldots,d_l}$$

angegeben. Im Fall des Modells mit unterteilten Objekträumen sind die Instanzvariablen `length` und `data` der Klassen $\texttt{ArrayRepresentation<B[]}^{m-l}\texttt{>}$ jeweils getrennt zu behandeln.

Das Ergebnis der Komponente ist das Array, welches dem Objekt-Identifier $element_\epsilon$ zugeordnet ist:

$$\neg ab(C) \Rightarrow \texttt{result}(C) = element_\epsilon.$$

- Zugriffe der Form `array.length` können anhand der Repräsentation des Arrays durch die Klasse `ArrayRepresentation<Type>` analog zu „normalen“ Zugriffen auf Instanzvariablen modelliert werden.

[12] Der Basistyp eines Arrays des Typs $\texttt{T[]}\cdots[]$ ist der Typ `T`, wobei `T` kein Array-Typ ist.

4.2.4 Komplexität des Modells

Da die Struktur des indirekten Modells jener des Modells aus Abschnitt 4.1 sehr ähnlich ist, gelten auch hier die Überlegungen zur Komplexität des Modells aus Abschnitt 4.1.4.

Es bleibt anzumerken, daß der durch das Propagieren von Objekträumen verursachte Aufwand den erforderlichen Berechnungsaufwand zum Finden von Diagnosen noch weiter erhöht. Dies gilt insbesondere für Programme, welche eine große Anzahl von Objekten generieren. Dadurch liegt die maximale noch mit dem Modell handhabbare Größe etwas unter jener für das Modell aus Abschnitt 4.1.

Im nächsten Kapitel wird die Implementierung des indirekten Modells mit unterteilten Objekträumen vorgestellt. Es werden die wichtigsten Algorithmen und Klassen der Implementierung betrachtet und die Beschränkungen der Implementierung aufgezeigt. Schließlich wird eine einfache Benutzeroberfläche präsentiert, welche das Debugging von Java-Programmen ermöglicht.

Kapitel 5

Implementierung

Einen weiteren Schwerpunkt dieser Arbeit stellt die Implementierung des im letzten Kapitel beschriebenen indirekten Modells dar. Es wurde das in Abschnitt 4.2 vorgestellte Modell mit der Erweiterung auf unterteilte Objekträume (siehe Abschnitt 4.2.3) implementiert. Die Implementierung der Verhaltensbeschreibung der Komponenten erfolgt im Gegensatz zu Kapitel 4 nicht in Form logischer Formeln, sondern als Constraint-System. Dies hat eine verminderte Geschwindigkeit des Modells zur Folge, vereinfacht die Implementierung hingegen beträchtlich. Auch kann auf diese Weise die Struktur des Modells abgebildet werden, ohne alle Verhaltensbeschreibungen in eine einheitliche Form überführen zu müssen. Zusätzlich wird die Repräsentation von Objekträumen stark vereinfacht.

In diesem Kapitel werden die wesentlichen Teile der Implementierung behandelt und eine graphische Benutzeroberfläche vorgestellt, die es ermöglicht, aufgrund von Beobachtungen an Variablen fehlerhafte Anweisungen oder Anweisungsteile in Methoden von Java-Programmen zu lokalisieren. Weiters werden einige Probleme beim Spezifizieren von Beobachtungen zusammen mit einer möglichen Lösung und deren Implementierung behandelt.

Zur Implementierung wurde die Sprache *Smalltalk* im Zusammenspiel mit der Entwicklungsumgebung *Visual Works* verwendet, da die Sprache ein sehr leistungsfähiges, objektorientiertes Rahmenwerk bietet, mit welchem komplexe Abläufe leicht realisiert werden können. Weiters baut die vorliegende Implementierung auf einigen bereits existierenden Bibliotheken auf, welche ebenfalls in dieser Sprache realisiert sind. Dies sind eine Bibliothek zur modellbasierten Diagnose und ein Parser für Java-Programme. Einen weiteren Vorteil stellt die Tatsache dar, daß in Smalltalk geschriebene Programme unverändert auf unterschiedlichen Plattformen zum Ablauf gebracht werden können.

5.1 Architektur der Implementierung

Wie schon bei der Beschreibung der Modelle in Kapitel 4 angedeutet, setzen sich die in dieser Arbeit verwendeten Modelle aus zwei, weitgehend voneinander unabhängigen Teilen zusammen. Dies sind der *strukturelle Aufbau* des Modells und die *Verhaltensbeschreibung* der einzelnen Komponenten. Der strukturelle Aufbau des Modells spiegelt die Abhängigkeiten zwischen den einzelnen Anweisungen und Ausdrücken des Programms wieder und wird durch die Anweisungen bzw. Ausdrücke und deren explizit oder implizit benutzten oder veränderten Variablen bestimmt. Das Verhalten der erzeugten Komponenten ist davon weitgehend unabhängig, da nur die Zusammenhänge zwischen den Input- und Output-Ports der Komponenten spezifiziert werden. Das Verhalten der Komponenten ist in einigen Fällen von der Struktur des Modells abhängig, da einige Komponenten für jede benutzte oder veränderte Variable eigene Input- und Output-Ports aufweisen und daher die Anzahl der Ports dieser Komponenten nicht konstant ist.

Die genaue Anzahl wird in diesen Fällen beim Aufbau der Modellstruktur bestimmt.

Die Implementierung des indirekten Modells beruht auf den soeben genannten Eigenschaften des Modells und ist ebenfalls in zwei voneinander weitgehend unabhängige Teilbereiche gegliedert:

(a) Die Implementierung des *Aufbaues der Modellstruktur*. Dies umfaßt das Erzeugen von Modellen für alle Konstruktoren und Methoden aller Klassen des Programms. In dieser Phase der Modellbildung wird für jeden Konstruktor und für jede Methode eine Menge von Komponenten und deren Verbindungen untereinander berechnet.

(b) Die Implementierung der *Verhaltensbeschreibung* der in Schritt (a) erzeugten Komponenten. Dies beinhaltet auch die Implementierung der Erweiterungen des Constraint-Netzwerks, welche durch die gegenüber einfachen Werten veränderte Semantik der Objekträume notwendig wird.

In den folgenden Abschnitten wird zunächst die Implementierung des Aufbaues der Modellstruktur näher betrachtet. Anschließend wird die Implementierung der Verhaltensbeschreibung der einzelnen Modellkomponenten sowie der Erweiterungen des Constraint-Netzwerks behandelt.

5.2 Implementierung der strukturellen Modellbildung

Um einen ersten Überblick über die Implementierung zu erhalten, wird zunächst die Struktur von Java-Programmen und deren Parse-Bäumen betrachtet, da die Implementierung auf den Parse-Bäumen aufbaut und daher in einigen Teilen eng an die Struktur der Bäume angelehnt ist. Aufgrund der Beschränkungen bezüglich des Umfangs dieser Arbeit werden hier nur die wichtigsten Strukturen und deren Eigenschaften aufgelistet. Für Details sei auf die Spezifikation der Sprache Java [GJS96] und auf die Implementierung des Java-Parsers verwiesen.

Um Namenskonflikte zu vermeiden und Programme besser strukturieren zu können, werden Java-Programme zunächst in *Packages* organisiert. Packages sind hierarchisch aufgebaut und können weitere Sub-Packages enthalten. Innerhalb eines Packages sind jene Teile eines Programms zusammengefaßt, welche aufgrund ihrer Implementierung oder ihrer Funktionsweise als zusammengehörige Einheit angesehen werden können. Darüber hinaus erhalten die innerhalb eines Packages definierten Klassen und Interfaces weitgehend uneingeschränkten Zugriff auf die Variablen und Methoden aller in demselben Package definierten Klassen und Methoden.

Elemente der Packages sind eine Menge aus Sub-Packages und eine Menge von sog. *Compilation Units*. Diese stellen die eigentlichen Dateien dar, welche den Quelltext der Klassen und Interfaces des Programms enthalten. Zumeist werden in einer Compilation Unit nur einzelne Klassen oder Interfaces definiert, bei Verwendung von *inner classes* oder ähnlichen Konstrukten kann die Anzahl der in einer Compilation Unit definierten Typen auf mehr als einen Typ ansteigen. Auch bilden die Compilation Units die kleinste Einheit, welche vom Übersetzer separat behandelt werden kann.

Die in *Klassen* und *Interfaces* enthaltenen Elemente sind Definitionen von Variablen, Methoden und Konstruktoren. Auch Vererbung zwischen Klassen oder Interfaces ist möglich. Hier ist zu beachten, daß nur einfache Vererbung zwischen Klassen erlaubt wird, hingegen eine beliebige Anzahl an Interfaces von einer Klasse implementiert werden kann. Die Menge der in einer Klasse definierten Variablen kann in zwei Kategorien eingeteilt werden: *Instanzvariablen* von Objekten und *Klassenvariablen*. Instanzvariablen werden beim Erzeugen neuer Instanzen der Klasse angelegt, Klassenvariablen hingegen nur einmal beim Initialisieren der Klasse. Im Gegensatz dazu können Interfaces nur Klassenvariablen enthalten, welche Konstanten repräsentieren. Ähnliches gilt für die definierten Methoden: während Interfaces nur abstrakte Methoden (d.h. Methoden ohne Implementierung) enthalten dürfen, gilt diese Einschränkung für Klassen nicht.

```
1  package Demo;
2
3  public class ParseTreeDemo {
4      public static void demo(int from, int to)
5      {
6          int start, stop, i;
7          if (from < to) {
8              start = from;
9              stop = to;
10         }
11         else {
12             start = to;
13             stop = from;
14         }
15         i = start;
16         while (i <= stop ) {
17             // do something
18             ++i;
19         }
20     }
21 }
```

Abbildung 5.1: Programm zur Demonstration der Struktur eines Parse-Baumes

Methoden weisen eine innerhalb ihres definierenden Typs eindeutige Signatur[1] auf und setzen sich aus einer Folge von Anweisungen zusammen, welche das Verhalten der Methoden spezifizieren.

Anweisungen und *Ausdrücke* stellen in gewisser Weise die kleinsten Bausteine eines Programms dar, da Anweisungen und Ausdrücke nicht weiter unterteilt werden können (mit Ausnahme von Auswahlanweisungen und Schleifen, welche weitere Anweisungen und Ausdrücke enthalten können). Hierzu zählen etwa unäre und binäre Operatoren, Zuweisungsanweisungen, Variablenreferenzen, Auswahlanweisungen, Methodenaufrufe, Schleifen, etc.

Die in der vorliegenden Arbeit verwendete Implementierung des Java-Parsers orientiert sich weitgehend an der bisherigen Beschreibung des strukturellen Aufbaues eines Java-Programms. Es wird für jedes Element des Programms (wie etwa Package, Compilation Unit, Klassendeklaration, Methodendeklaration, Anweisung bzw. Ausdruck, etc.) ein eigener Knoten im Parse-Baum generiert, wobei für jedes repräsentierte Sprachelement eine eigene Klasse verwendet wird. Die Instanzen dieser Klassen stellen die einzelnen Knoten im Parse-Baum dar. Die Wurzel des Parse-Baumes bildet eine Instanz der Klasse *Host Environment*, welche das gesamte Laufzeitsystem von Java repräsentiert und alle Packages des Programms beinhaltet. Die Nachfolger eines Knotens stellen Unterelemente des repräsentierten Programmelements dar. Weiters weist jeder Knoten des Parse-Baums spezifische Attribute auf, welche die relevanten Eigenschaften des repräsentierten Programmelements widerspiegeln. Dies umfaßt z.B. im Fall einer Variablendeklaration den Typ der deklarierten Variablen.

Beispiel 5.1 Abbildung 5.2 zeigt den Parse-Baum des Programms aus Abbildung 5.1.

Einen groben Überblick über die Anbindung der Implementierung an den Java-Parser bietet das in Abbildung 5.3 dargestellte Diagramm. Die nach außen hin sichtbaren Schnittstellen der Implementierung stellen die drei Klassen `ValueBasedProgramModel`, `ValueBasedClassModel` und `ValueBasedMethodModel` dar, welche jeweils Modelle des gesamten Programms, einer Klasse

[1] Die Signatur einer Methode setzt sich aus deren Namen und den Typen ihrer formalen Parameter zusammen.

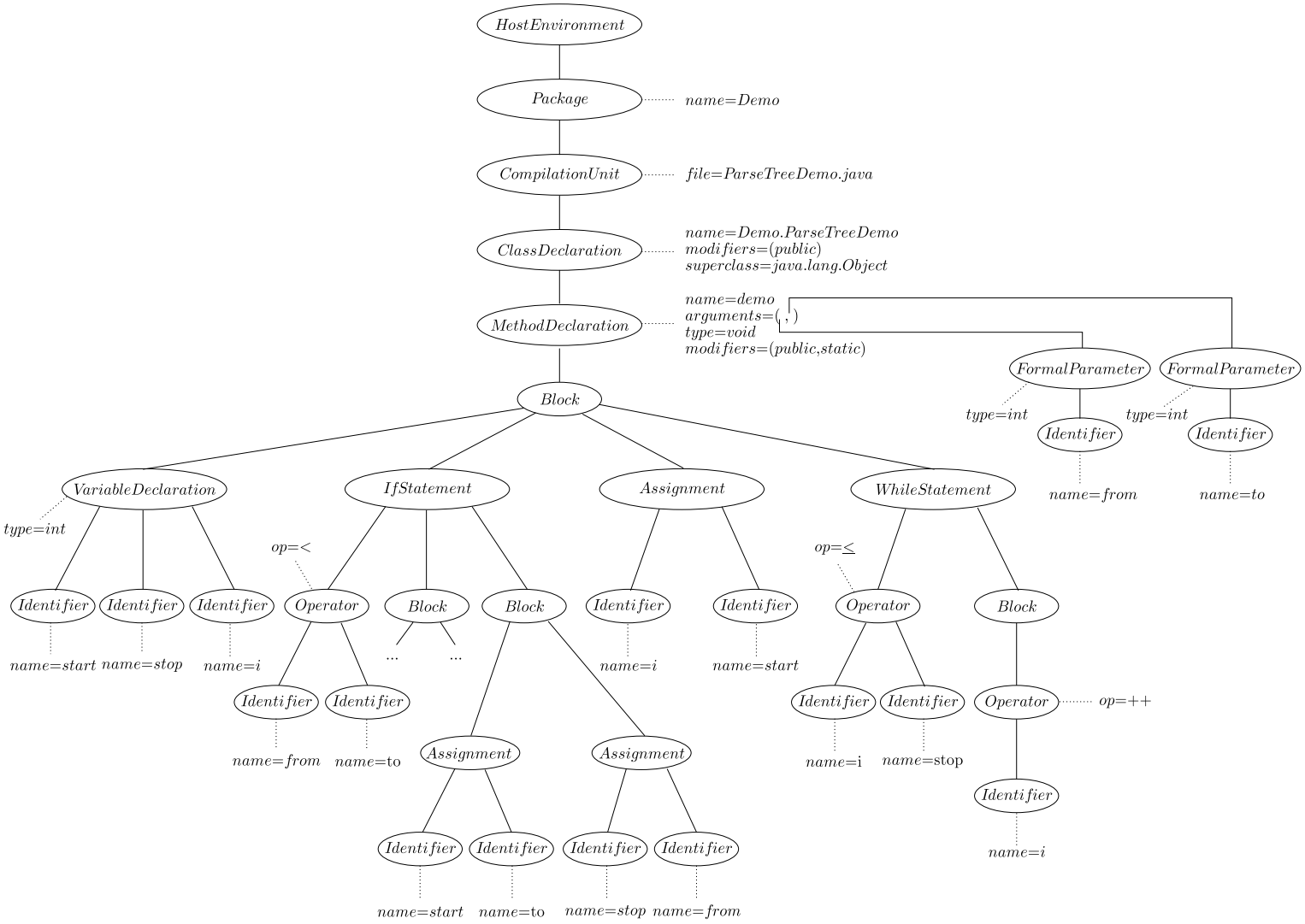

Abbildung 5.2: Parse-Baum des Programms aus Abbildung 5.1. Der Unterbaum des `then`-Zweiges der Auswahlanweisung ist aufgrund von Platzbeschränkungen nicht vollständig dargestellt. Die Attribute der Knoten sind in der Form `<AttributName>=<Wert>` angegeben

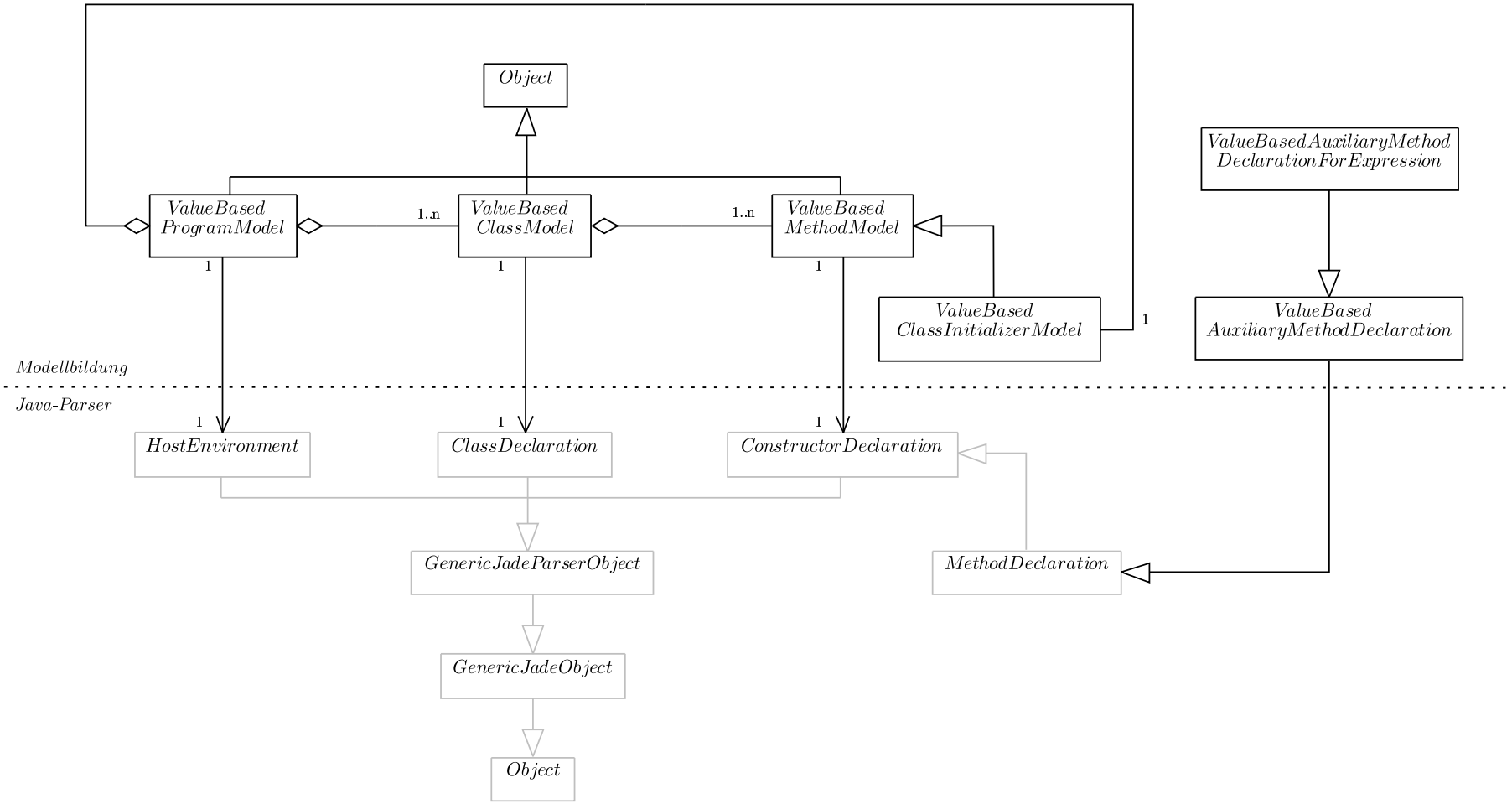

Abbildung 5.3: Anbindung der Implementierung der Modellbildung an den Java-Parser. Klassen des verwendeten Java-Parsers sind heller dargestellt

oder einer Methode repräsentieren. Jede dieser Klassen korrespondiert mit einer entsprechenden Klasse des Parse-Baumes, welche das modellierte Programmelement darstellt. Die Aufgabe der Klassen besteht im wesentlichen darin, den Modellbildungsprozeß zu steuern und die generierten Modelle der einzelnen Methoden des Programms zur weiteren Verwendung (z.B. in einem Debugger) zur Verfügung zu stellen. Im folgenden werden die einzelnen Klassen genauer vorgestellt und die Aufgaben und Verwendung der wichtigsten Methoden behandelt. Für weitere Details sei auf die Implementierung verwiesen.

5.2.1 ValueBasedProgramModel

Die Klasse `ValueBasedProgramModel` bildet die eigentliche Schnittstelle der Modellbildung gegenüber dem Debugger. Aufgabe der Klasse ist es, aus einem gegebenen Java-Programm (genauer: dessen Parse-Baum) Modelle aller in diesem Programm definierten Klassen und Methoden zu erstellen und diese dem Debugger zugänglich zu machen.

`initialize:`

Zentrale Methode dieser Klasse ist die Methode `initialize:`, welche als Argument den Parse-Baum des Programms erwartet und die Modellbildung steuert. Die Vorgehensweise hierbei kann wie folgt zusammengefaßt werden:

1. Es wird für jede im Programm definierte Klasse eine Instanz der Klasse `ValueBasedClassModel` generiert, welche die Modelle der Klasse und aller darin definierten Methoden repräsentiert.

2. Die Abhängigkeiten zwischen den einzelnen Methoden des Programms werden berechnet[2] und die Methoden aufgrund dieser Abhängigkeiten in Gruppen eingeteilt. Eine Methode m ist abhängig von einer Methode m', falls die Methode m einen Aufruf der Methode m' enthält. Eine Gruppe von Methoden enthält alle jene Methoden, welche entweder direkt oder mittelbar rekursive Aufrufe von sich selbst enthalten. Der Abhängigkeitsgraph einer Gruppe ist also stets zyklisch oder enthält nur einen einzigen Knoten.

   Die auf diese Weise entstandenen Gruppen werden bei der Modellbildung jeweils gesondert betrachtet. Diese Vorgehensweise ist notwendig, um auch rekursive Methodenaufrufe korrekt modellieren zu können (siehe `modelMethodsFrom:`).

3. Für jede der in Schritt 2 berechneten Gruppen von Methoden werden Modelle gebildet und in die entsprechenden Instanzen der Klassen `ValueBasedClassModel` eingetragen. Ein Modell eines Konstruktors oder einer Methode wird durch eine Instanz der Klasse `ValueBasedMethodModel` repräsentiert.

`modelMethodsFrom:`

Eine weitere zentrale Methode der Implementierung ist die Methode `modelMethodsFrom:`, welche von der Methode `initialize:` benötigt wird. Aufgabe dieser Methode ist es, für eine gegebene Gruppe von rekursiven Methoden Modelle der Methoden zu berechnen. Hier ist zu beachten, daß der Modellbildungsprozeß im Fall von rekursiven Methoden mehrmals ausgeführt werden muß, um korrekte Modelle zu erhalten. Aus diesem Grund wird in dieser Methode eine iterative Fixpunkt-Berechnung durchgeführt, welche abbricht, sobald die berechneten Modelle aller Methoden stabil sind. Ein Modell einer Methode wird als stabil angesehen, falls das Modell

[2] Die Berechnung des Abhängigkeitsgraphen und der Gruppen von rekursiven Methoden erfolgt durch die Klasse `JavaDependencyGraph`, welche ebenfalls in der verwendeten Bibliothek des Java-Parsers enthalten ist.

äquivalent zu dem in der vorangegangenen Iteration berechneten Modell der Methode ist. Da die relevanten Aspekte des indirekten Modells die Input- und Output-Verbindungen des Modells sind, ist es ausreichend, die Mengen der Input- und Output-Verbindungen zu betrachten. D.h. ein Modell ist stabil genau dann, wenn die Input- und Output-Verbindungen des Modells der aktuellen Iteration mit jenen der vorangegangenen Iteration übereinstimmen.

Einen Spezialfall stellt eine Methode ohne rekursive Aufrufe dar. In diesem Fall enthält die zu modellierende Gruppe nur diese eine Methode. Dann kann die Fixpunkt-Berechnung für diese Gruppe vermieden werden, da die Input- und Output-Verbindungen der Methode bereits nach einem Durchgang korrekt ermittelt werden können.

Der in der Methode `modelMethodsFrom:` verwendete Algorithmus ist in Abbildung 5.4 dargestellt. Hier ist anzumerken, daß der Test in Zeile 1 notwendig ist, um Termination des Algorithmus sicherzustellen, da dieser während der Bildung der Modelle der Methoden rekursiv aufgerufen wird.

```
ModelMethodsFrom(group)
 1  if group already modeled
 2     then  do nothing
 3     else  mark group modeled
 4           for  each m in group
 5           do  setup empty model m_e for method m
 6               make m_e the current model for method m
 7           repeat
 8                   stable ← true
 9                   for  each m in group
10                   do m_old ←  current model of method m
11                      m_new ←  compute model of method m
12                      make m_new the current model for method m
13                      if m_old ≠ m_new
14                        then stable ← false
15             until stable ∨ |group| = 1
```

Abbildung 5.4: Algorithmus zur Berechnung von Modellen einer Gruppe von (rekursiven) Methoden

`methodModel:`

Eine weitere bei der Modellbildung benötigte Methode ist die Methode `methodModel:`. Die Methode erwartet die Deklaration eines Konstruktors oder einer Methode des Programms in Form einer Instanz der Klasse `ConstructorDeclaration` bzw. `MethodDeclaration` und gibt das Modell des Konstruktors bzw. der Methode zurück. Hier ist zu beachten, daß von dieser Methode stets das *aktuell gültige* Modell der Methode zurückgegeben wird. Dieses kann bei der Modellierung einer Gruppe rekursiver Methoden vom endgültigen Modell abweichen.[3]

Wurde noch kein Modell für den angegebenen Konstruktor bzw. die angegebene Methode bestimmt, wird die Gruppe der Methode bzw. des Konstruktors ermittelt und die Berechnung der Modelle aller darin enthaltenen Konstruktoren und Methoden durchgeführt. Anschließend wird das Modell des gewünschten Konstruktors bzw. der gewünschten Methode zurückgegeben.

[3]Es können gegenüber dem endgültigen Modell einige Input- oder Output-Verbindungen fehlen.

Diese Methode wird während der Modellbildung verwendet, um bei Methodenaufrufen die Modelle der aufgerufenen Methoden zu bestimmen. Auch der Debugger benötigt diese Methode, um das Modell der zu debuggenden Methode zu erhalten.

`buildClassInitializerModelForMethod:`

Die Methode `buildClassInitializerModelForMethod:` generiert ein Modell (eine Instanz der Klasse `ValueBasedClassInitializerModel`) für alle von einer gegebenen Methode benötigten Klassenvariablen. Auch alle statischen Initializer-Blöcke der benötigten Klassen werden in das Modell integriert. Im Gegensatz zu der Methode `modelMethodsFrom:` ist hier keine Fixpunktberechnung notwendig, da das generierte Modell von keiner anderen Methode aufgerufen werden kann. Es wird ausschließlich vom Java-Debugger benötigt, um die benötigten Klassenvariablen der zu debuggenden Methode zu initialisieren.

5.2.2 ValueBasedClassModel

Die Klasse `ValueBasedClassModel` dient vorwiegend als Container für die Modelle der in der repräsentierten Klasse definierten Methoden und stellt Assoziationen zwischen den Modellen der Methoden und deren Deklarationen bereit. Während der Initialisierung von `ValueBasedProgramModel` werden Instanzen dieser Klasse für jede im Programm definierte Klasse generiert und bei der Instanz von `ValueBasedProgramModel` registriert.

Die Schnittstellen und Methoden dieser Klasse beschränken sich auf das Hinzufügen und Testen von Modellen für Methoden und leisten keinen signifikanten Beitrag zur Implementierung der Modellbildung. Sie werden daher an dieser Stelle nicht weiter betrachtet.

5.2.3 ValueBasedMethodModel

Die Klasse `ValueBasedMethodModel` implementiert die eigentliche Funktionalität der Modellbildung für einen Konstruktor oder eine Methode. Aufgabe dieser Klasse ist es, Assoziationen zwischen Variablen des Programms und den Verbindungen des Modells herzustellen, welche mit den Variablen assoziiert sind. Darüber hinaus werden Mengen der vom Modell benötigten und modifizierten Variablen und deren Verbindungen berechnet. Weiters wird eine Menge von Komponenten und Verbindungen zwischen diesen generiert, welche das Modell des Konstruktors oder der Methode repräsentieren.

Ablauf des Modellbildungsprozesses

Um das Modell eines Konstruktors oder einer Methode zu bestimmen, wird zunächst eine Instanz der Klasse `ValueBasedMethodModel` angelegt und mit der Konstruktor- oder Methodendeklaration assoziiert. Anschließend werden Verbindungen für alle externen[4] Variablen generiert und dem Modell hinzugefügt. Dies umfaßt alle Instanzvariablen und Klassenvariablen aller im Programm definierten Klassen sowie alle in Interfaces deklarierten Konstanten. Weiters sind Verbindungen für die Parameter des Konstruktors bzw. der Methode hinzuzufügen. Auch werden Verbindungen für implizite Parameter, wie z.B. für die Variable `this` bei Instanzmethoden, hinzugefügt.

Als nächster Schritt werden alle Anweisungen im Rumpf des Konstruktors oder der Methode in der Reihenfolge ihrer Abarbeitung (entsprechend der Java-Semantik) behandelt. Für jede der Anweisungen und ihrer Unter-Anweisungen bzw. -Ausdrücke wird – abhängig von der Art des

[4] *Externe Variablen* sind Variablen, deren Gültigkeitsbereich die Konstruktor- oder Methodendeklaration übersteigt.

zu modellierenden Programmelements – eine Komponente im Modell generiert und deren Input-Ports mit den aktuell gültigen Verbindungen für die damit assoziierten Variablen verbunden. Die aktuell gültige Verbindung für eine Variable wird mit Hilfe der zu Beginn der Modellbildung generierten Instanz von `ValueBasedMethodModel` bestimmt. Zugleich werden diese Verbindungen als „benutzt“ markiert, um nach vollendeter Modellierung des Konstruktors bzw. der Methode die vom Modell benötigten Verbindungen ermitteln zu können.

Eine ähnliche Vorgehensweise ergibt sich für die Output-Ports der Komponente: Für jeden Output-Port der Komponente wird eine neue Verbindung im Modell generiert und mit der mit dem Port assoziierten Variablen assoziiert. Diese Verbindung ersetzt die bisher aktuelle Verbindung für die Variable und stellt für die weitere Modellbildung die aktuelle Verbindung dar. Weiters wird die Verbindung als „modifiziert“ markiert, um die von dem Modell veränderten Verbindungen berechnen zu können.

Nachdem die Modellierung der Anweisungen des Konstruktors bzw. der Methode abgeschlossen ist, werden alle unbenutzten Verbindungen entfernt, wobei eine Verbindung dann als unbenutzt angesehen wird, falls diese mit keiner Komponente verbunden ist. Dieser Schritt ist notwendig, da zu Beginn der Modellbildung Verbindungen für alle externen Variablen hinzugefügt wurden, von denen das Modell jedoch meist nur einige wenige benötigt. Ähnliches gilt für deklarierte, aber unbenutzte lokale Variablen und Parameter. Alternativ dazu wäre eine leicht veränderte Vorgehensweise möglich: Verbindungen für externe Variablen werden erst bei deren erster Verwendung angelegt. Dadurch kann das zuvor genannte Entfernen der überflüssigen Verbindungen entfallen. Um die Modellbildung nicht unnötig zu verkomplizieren, wird diese Optimierung in der vorliegenden Implementierung nicht durchgeführt.

Zuordnung von Variablen zu Verbindungen

Dieser Abschnitt betrachtet die Implementierung der Zuordnung zwischen Variablen des Programms und Verbindungen des Modells. Um die Implementierung des Managements von Variablen und Verbindungen zu vereinfachen, werden in `ValueBasedMethodModel` nicht direkt Assoziationen zwischen den Variablen und den zugehörigen Verbindungen gespeichert, sondern es wird eine zusätzliche Schicht eingeführt, welche von der Art der Variablen abstrahiert. Dadurch wird der Zugriff auf Verbindungen des Modells und das Einfügen von Verbindungen in das Modell vereinheitlicht.

Die zu diesem Zweck eingeführte Schicht wird von der Klasse `ConnectionName` implementiert. Instanzen dieser Klasse stellen einen eindeutigen Namen dar, welcher eine Variable – und damit eine Verbindung im Modell – repräsentiert. Die Variable wird durch ihren Namen zusammen mit der ihre Definition enthaltenden Klassen- oder Interface-Deklaration eindeutig bestimmt. Lokale Variablen werden durch ihren Namen ohne umgebende Deklaration repräsentiert.

Die Zuordnung von Variablen zu Verbindungen erfolgt in einem zweistufigen Prozeß: In einem ersten Schritt wird eine Instanz der Klasse `ConnectionName` ermittelt, welche die Variable repräsentiert. Diese Instanz wird im darauffolgenden Schritt verwendet, um die Zuordnung zwischen der mit der repräsentierten Variablen assoziierten Verbindung des Modells und dem `ConnectionName` herzustellen. Der Zugriff auf bereits bestehende Verbindungen im Modell erfolgt auf analoge Weise.

Im folgenden werden die wichtigsten Attribute der Klasse `ValueBasedMethodModel` vorgestellt, welche die Zuordnung der Variablen (bzw. `ConnectionName`s) zu den Verbindungen enthalten. Weiters wird auf die Berechnung der verwendeten und modifizierten Variablen und Verbindungen eingegangen.

- `currentAssocs`

  Dieses Attribut repräsentiert die aktuellen Zuordnungen zwischen den Variablen des Pro-

gramms und den Verbindungen des Modells. In dieser Instanzvariablen sind zu jedem Zeitpunkt der Modellbildung die aktuellen Zuordnungen zwischen den die Variablen repräsentierenden `ConnectionName`s und den Verbindungen enthalten. Diese werden durch das Hinzufügen neuer Komponenten und Verbindungen zu dem Modell laufend verändert.

- `usedConnections`

  Dieses Attribut enthält nach Beendigung der Modellbildung die Menge aller von dem Modell benötigten externen Variablen und Verbindungen. Es werden Assoziationen zwischen den `ConnectionName`s, welche die benötigten Variablen repräsentieren, und den zugehörigen Verbindungen gespeichert. Hier ist zu beachten, daß dieses Attribut erst nach abgeschlossener Modellbildung gültig ist, da dessen Werte – analog zu jenen des Attributs `currentAssocs` – während der Modellbildung verändert werden.

  Die Berechnung dieses Attributs erfolgt während der Modellbildung in iterativer Weise: Wird eine bereits bestehende Verbindung für einen `ConnectionName` angefordert, wird zunächst ermittelt, ob diese Verbindung bereits durch eine Komponente des Modells berechnet wurde. In diesem Fall enthält das Attribut `modifiedConnections` bereits einen Eintrag für diese Verbindung. Andernfalls bezeichnet die Verbindung eine durch das Modell bisher nicht veränderte externe Variable (bzw. Parameter-Variable) und muß zu der Menge der benötigten Verbindungen hinzugefügt werden. Hier ist zu beachten, daß dies nur erfolgen darf, falls die Verbindung nicht bereits in der Menge enthalten ist, da stets die *erste* Verwendung jeder Verbindung ermittelt wird.

- `modifiedConnections`

  Dieses Attribut enthält die Menge aller durch das Modell veränderten Variablen und Verbindungen. Es werden Assoziationen zwischen `ConnectionName`s, welche die veränderten Variablen repräsentieren, und den zugehörigen Verbindungen gespeichert. Hier ist zu beachten, daß dieses Attribut erst nach abgeschlossener Modellbildung gültig ist, da auch dessen Werte während der Modellbildung verändert werden.

  Die Berechnung dieses Attributs erfolgt in ähnlicher Weise wie beim Attribut `usedConnections`: Wird eine Verbindung mit einer Variablen (genauer: mit einem `ConnectionName`) assoziiert, wird diese Verbindung diesem Attribut hinzugefügt. Hier sind keine weiteren einschränkenden Bedingungen notwendig, da in diesem Fall immer die *zuletzt* gültige Verbindung für eine Variable ermittelt werden soll.

Modellbildung der Programmelemente

Die Methode `buildModel` stellt die zentrale Methode des Modellbildungsprozesses dar. Durch Aufrufen dieser Methode wird die Modellbildung angestoßen. Diese Methode führt nur einen Aufruf der Methode `buildValue:` der Klasse `ConstructorDeclaration` durch, welche die eigentliche Modellierung des Konstruktors bzw. der Methode durchführt.

Die bei der Modellbildung generierten Komponenten und Verbindungen werden dem Attribut `constraintPropagator` hinzugefügt, welches das generierte Modell in Form eines Constraint-Systems enthält. Eine detailliertere Beschreibung des Constraint-Systems und der verwendeten Komponenten und Verbindungen findet sich in Abschnitt 5.3 und in [Wot97]. Die Implementierung des Constraint-Systems selbst ist nicht Teil der vorliegenden Arbeit, sondern diese wurde aus der bereits bestehenden Bibliothek zur modellbasierten Diagnose entnommen.

Zusätzlich zu der Methode `buildModel` existieren noch zwei weitere Methoden, welche für die Modellbildung essentielle Bedeutung aufweisen: Das sind die Methoden `asConstraintModel` und `asRecursiveModel`. Aufgabe dieser Methoden ist es, das berechnete Modell der Methode

in eine geeignete Repräsentation zu überführen, welche bei der Implementierung von hierarchischen Verhaltensbeschreibungen als inneres Modell verwendet werden kann. Die Methode `asConstraintModel` wird verwendet, falls es sich um einen nicht-rekursiven Aufruf handelt. In diesem Fall wird eine Repräsentation des Modells als Constraint-Netzwerk geliefert. Im rekursiven Fall wird die Methode `asRecursiveModel` angewendet, welche eine auf einem Evaluator beruhende Repräsentation der Methode zurückliefert.

5.2.4 ValueBasedClassInitializerModel

Die Klasse `ValueBasedClassInitializerModel` ist eine Erweiterung der Klasse `ValueBasedMethodModel` und wird verwendet, um Initializer von Klassenvariablen sowie statische Initializer-Blöcke zu modellieren. Sie wird von der Klasse `ValueBasedProgramModel` benötigt, um ein Modell von allen Klassenvariablen jener Klassen zu berechnen, von welchen eine gegebene Methode Input-Verbindungen zu Variablen aufweist (siehe auch die Beschreibung der Methode `buildClassInitializerModelForMethod:` der Klasse `ValueBasedProgramModel`).

Die Funktionsweise der Klasse ist im wesentlichen zu jener der Klasse `ValueBasedMethodModel` äquivalent, wobei an einigen Stellen leichte Erweiterungen vorgenommen wurden, um die korrekte Modellierung der Klassenvariablen der zu modellierenden Klassen zu gewährleisten. Die Erweiterungen stellen sicher, daß vor dem ersten lesenden Zugriff auf jede Verbindung, welche eine Klassenvariable repräsentiert, alle Klassenvariablen und statischen Initializer-Blöcke der die Klassenvariable deklarierenden Klasse modelliert werden. Stellt die angesprochene Klassenvariable eine sog. *Compile-Time-Constant*[5] dar, ist eine leicht abweichende Vorgehensweise notwendig: Es muß nur das Modell der angesprochenen Konstante gebildet werden. Alle Klassenvariablen, welche keine Compile-Time-Konstanten deklarieren, müssen in diesem Fall ignoriert werden. Auch die Initialisierung der Klassenvariablen mit deren Default-Wert kann in diesem Fall übergangen werden.

Diese Vorgehensweise bildet in weiten Teilen die in der Java-Spezifikation angegebene Reihenfolge bei der Initialisierung nach. Es sind aber auch einige Abweichungen zum Java-Standard zu bemerken:

- Da die Modellierung der Initializer nicht während der Modellbildung der Methoden berechnet wird, sondern erst nachdem das Modell der Methode vollständig bestimmt wurde, sind zwar die benötigten externen Variablen – und damit die verwendeten Klassen – bekannt, nicht jedoch die Reihenfolge des Zugriffs auf die Variablen. Daraus ergibt sich, daß die Reihenfolge der Initialisierung der Klassen nicht in jedem Fall der durch die Java-Spezifikation geforderten entspricht.

- Auch ist der Zeitpunkt der Initialisierung vor den Beginn der Methode vorverlegt, was Abweichungen von Werten von Klassenvariablen hervorrufen kann, falls die Initializer Nebeneffekte in Klassenvariablen anderer Klassen verursachen.

- Weiters werden in der vorliegenden Implementierung nur jene Klassen initialisiert, von welchen Klassenvariablen als Input-Variablen benötigt werden. Zuweisungen an Klassenvariablen bedingen *keine* Initialisierung der zugehörigen Klasse.

Die hier genannten Abweichungen gegenüber der Java-Spezifikation stellen keine strengen Einschränkungen für die zu modellierenden Programme dar, da Programme, deren Korrektheit von der Reihenfolge der Initialisierung der Klassen abhängig ist bzw. welche Initializer mit Nebeneffekten in Variablen anderer Klassen aufweisen, in der Praxis relativ selten sind.

[5] Eine Compile-Time-Konstante ist ein Ausdruck, dessen Wert bereits zur Übersetzungszeit vom Compiler berechnet werden kann. Deklarationen von Compile-Time-Konstanten sind Klassenvariablen, welche mit den Modifiers `public` und `final` deklariert wurden und deren Initializer eine Compile-Time-Konstante darstellen.

5.2.5 Modellierung der Programmelemente

In diesem Abschnitt werden die wichtigsten Methoden und Vorgehensweisen bei der Modellierung der Programmelemente näher betrachtet. Die Modellierung erfolgt durch die Methode `buildValueBasedModel:`, welche in den Klassen der zu modellierenden Programmelemente des vom Java-Parser erzeugten Parse-Baumes implementiert sind. Die Methode erwartet eine Instanz der Klasse `ValueBasedMethodModel` als Argument, welche das Modell der Methode darstellt. Die Modellierung eines Konstruktors oder einer Methode wird durch den Aufruf dieser Methode durch die Klasse `ValueBasedMethodModel` angestoßen.

Sämtliche erzeugten Komponenten sind von der Klasse `ValueBasedDiagnosisComponent` abgeleitet, welche die gemeinsame Funktionalität aller Komponenten implementiert. Dies sind das Speichern einer Referenz auf die repräsentierte Anweisung bzw. den repräsentierten Ausdruck sowie einige häufig verwendete Funktionen wie z.B. das Generieren einer textuellen Repräsentation der Komponente oder das Finden eines Portnamens einer gegebenen Verbindung.

- Die Modellierung eines *Konstruktors* oder einer *Methode* ist in der Klasse `Constructor-Declaration` implementiert, wobei einige Methoden in der Klasse `MethodDeclaration` überschrieben werden. Die Modellierung erfolgt in drei Schritten:

  - Zu Beginn werden neue Verbindungen für die Parameter des Konstruktors oder der Methode dem Modell hinzugefügt. Weiters werden Verbindungen für eventuelle implizite Parameter angelegt. Ist das zu modellierende Programmelement ein Konstruktor, werden implizite Verbindungen für die Variable `this` und eine Verbindung für die vom Übersetzer angelegte Variable `callingContext`[6] generiert. Im Fall einer Methode wird zusätzlich dazu eine Verbindung für die Variable `return` angelegt, welche dem von der Methode berechneten Wert zugeordnet ist.

    An dieser Stelle sei angemerkt, daß die hier generierten Verbindungen für die (impliziten) Parameter nicht in jedem Fall vom Modell benötigt werden. So kann z.B. der implizite Parameter `this` bei Klassenmethoden weggelassen werden. Da unbenutzte Variablen ohnedies nach Beendigung der Modellbildung entfernt werden, wird auf die zusätzlichen Fallunterscheidungen an dieser Stelle verzichtet.

  - Nach dem Hinzufügen der Verbindungen für die Parameter zu dem Modell werden die Anweisungen des Konstruktors bzw. der Methode modelliert. Dies erfolgt durch den Aufruf der Methode `buildValueBasedModel:` der Klasse `Block`, welche den Rumpf der Deklaration darstellt.

    Ist das zu modellierende Programmelement ein Konstruktor, wird zuvor noch sichergestellt, daß als erste Anweisung des Konstruktors ein Aufruf eines Konstruktors der Basisklasse oder eines anderen Konstruktors derselben Klasse durchgeführt wird. Weiters werden alle Initializer von Instanzvariablen der Klasse modelliert.

  - Abschließend werden die zu Beginn eingeführten Verbindungen für (implizite) Parameter wieder aus den Attributen `currentAssocs` und `modifiedConnections` des Modells entfernt. Dies ist notwendig, da diese Verbindungen nur lokal in dem Modell sichtbar sind und daher nicht in den Output-Verbindungen aufscheinen dürfen.

- Die Modellierung eines *Blocks* von Anweisungen erfolgt in der Art, daß die Anweisungen in der Reihenfolge ihrer Abarbeitung (entsprechend der Java-Spezifikation) einzeln modelliert werden.

[6] Die Variable `callingContext` wird in `new`-Ausdrücken zum Generieren von Objekt-Identifiern benötigt. Eine detailliertere Beschreibung dieser Variablen findet sich bei der Beschreibung des Verhaltens von `new`-Ausdrücken in Abschnitt 5.3.

Danach werden als abschließender Schritt alle in dem Block deklarierten lokalen Variablen aus den Attributen `currentAssocs`, `usedConnections` und `modifiedConnections` des Modells entfernt. Dies ist notwendig, da diese Verbindungen nur innerhalb des Blocks sichtbar sind und daher nicht weiter im Modell aufscheinen dürfen.

- Ist das zu modellierende Programmelement eine *Variablendeklaration*, sind zwei Fälle zu unterscheiden:
  - Die Variable wird nur deklariert, d.h. es ist kein Initializer-Ausdruck angegeben. In diesem Fall wird eine neue Verbindung für diese Variable dem Modell hinzugefügt.
  - Die Variable weist einen Initializer-Ausdruck auf. In diesem Fall wird der Initializer-Ausdruck modelliert, welcher als Zuweisung repräsentiert ist. Es muß keine zusätzliche Verbindung hinzugefügt werden, da dies bereits bei der Modellierung der Zuweisung erfolgt.
- Die Modellierung von *Zuweisungen* gestaltet sich komplexer als die der bisher betrachteten Programmelemente, da hier mehrere Fälle zu unterscheiden sind. Einerseits muß bei der Modellierung zwischen Variablen der Umgebung und Variablen des Objektraums unterschieden werden, andererseits sind sowohl einfache als auch zusammengesetzte Zuweisungsoperatoren (wie z.B. '`+=`') zu beachten.

  Zunächst sei die Vorgehensweise bei der Modellbildung für einfache Zuweisungen betrachtet: Dem Modell wird eine neue Komponente hinzugefügt, welche die Zuweisung repräsentiert. Deren Ports werden entsprechend der in Kapitel 4 angegebenen Vorgehensweise verbunden und schließlich wird eine neue Verbindung für die Variable erzeugt. Die Art der Komponente ist von der Art des Ausdrucks auf der linken Seite der Zuweisung abhängig: bezeichnet der Ausdruck eine Variable der aktuellen Umgebung, d.h. die Variable stellt eine lokale-, Parameter- oder Klassenvariable dar, wird eine Instanz der Klasse `ValueBasedDCVariableAssignment` generiert. Andernfalls stellt der Ausdruck eine Zuweisung an eine Instanzvariable im Objektraum dar, und es wird eine Instanz der Klasse `ValueBasedDCFieldAssignment` generiert.

  Im Fall von zusammengesetzten Zuweisungsoperatoren werden Komponenten der Klassen `ValueBasedDCCompoundVariableAssignment` bzw. `ValueBasedDCCompoundFieldAssignment` angelegt.

  Bei der Modellierung von Zuweisungen an Instanzvariablen ist zu beachten, daß die erzeugten Komponenten gegenüber Kapitel 4 einen zusätzlichen Output-Port `result` aufweisen, welcher dem zugewiesenen Wert zugeordnet ist. Dies ist notwendig, da – entsprechend der Java-Spezifikation – Zuweisungen auch als Ausdrücke angesehen werden können und daher einen Port `result` aufweisen müssen.

  Analog zu allen bisher betrachteten Programmelementen ist auch hier bei der Modellierung die Reihenfolge der Ausführung der Unter-Anweisungen entsprechend der Java-Spezifikation zu beachten. So muß z.B. die linke Seite der Zuweisung vor der rechten Seite modelliert werden, um Nebeneffekte in korrekter Weise zu berücksichtigen.

  An dieser Stelle ist eine Optimierung möglich, falls die Zuweisung an eine Variable der aktuellen Umgebung erfolgt und der Ausdruck auf der linken Seite von der Form `path.variableId` ist. In diesem Fall muß kein vollständiges Modell für den Unter-Ausdruck `path` erstellt werden, da dessen Wert ohnedies nicht benötigt wird. Es ist ausreichend, ein Modell zu berechnen, welches nur die Nebeneffekte von `path` enthält. Die soeben beschriebene Optimierung wird durch die Methode `buildSideEffectModel:` implementiert, welche in allen Klassen des Parse-Baumes implementiert ist, deren Instanzen in einer linken Seite einer Zuweisung auftreten können.

- Die Modellierung von *Konstruktor-* und *Methodenaufrufen*, sowie von *`new`-Ausdrücken* wird durch die Methode `buildValueBasedModel:` in der Klasse `ConstructorInvocation` des Parse-Baumes implementiert, wobei einige Methoden in den Klassen `JavaMethodCall` bzw. `JavaClassInstanceCreation` überschrieben sind.

  Die Modellierung erfolgt weitgehend wie in Kapitel 4 beschrieben. Abhängig von der Art der aufgerufenen Methode werden unterschiedliche Komponenten erzeugt: für Aufrufe von Klassenmethoden oder Konstruktoren werden Instanzen der Klasse `ValueBasedDCStaticMethodInvocation` angelegt, für Aufrufe von dynamischen Methoden Instanzen der Klasse `ValueBasedDCDynamicMethodInvocation` und für `new`-Ausdrücke Instanzen von `ValueBasedDCClassInstanceCreation`.

  Bei der Modellierung von Konstruktoren ist zu beachten, daß diese wie dynamische Methoden (allerdings ohne Miteinbeziehen von überschriebenen Methoden) behandelt werden, da Konstruktoren auf die Instanzvariablen des erzeugten Objekts zugreifen können. Das eigentliche Generieren und Einfügen der Objekte in den Objektraum wird von der Klasse `ValueBasedDCClassInstanceCreation` modelliert.

  Die generierten Komponenten sind von hierarchischem Aufbau, d.h. sie enthalten wiederum Modelle, welche das Verhalten der Komponenten implementieren. Die Menge der Input- und Output-Verbindungen entspricht der Vereinigungsmenge der Input- bzw. Output-Ports der inneren Modelle. In diesem Fall sind die enthaltenen Modelle Repräsentationen der aufgerufenen Methode, welche aus den Modellen der aufgerufenen Methoden ermittelt werden. Dies erfolgt durch Aufrufen der Methode `asConstraintModel`, welche das berechnete Modell in ein Constraint-Netzwerk überführt. Im Fall von rekursiven Aufrufen wird die Methode `asRecursiveModel` angewendet, welche eine auf einem Evaluator basierende Repräsentation zurückliefert.

  Im Fall von Konstruktoraufrufen, Aufrufen von Klassenmethoden oder Aufrufen der Form `super.methodId(...)` ist dies stets das einzige enthaltene Modell. Erfolgt der Aufruf der Methode hingegen dynamisch, müssen zusätzlich alle Modelle von überschreibenden Methoden in Subklassen hinzugefügt werden. Die Implementierung des Verhaltens der Komponenten muß dann – abhängig von dem am Input-Port `object` anliegenden Wert – eines dieser Modelle auswählen. Die genaue Vorgehensweise sowie die restliche Verhaltensbeschreibung wird in Abschnitt 5.3 näher betrachtet.

- *`return`-Anweisungen* werden als Zuweisungen an die implizite lokale Variable `return` modelliert und daher durch Komponenten der Klasse `ValueBasedDCVariableAssignment` repräsentiert.

  Da die in Kapitel 4 beschriebene Erweiterung des Modells auf mehrere `return`-Anweisungen in einer Methode nicht implementiert ist, wird in dem Modell vermerkt, daß eine `return`-Anweisung modelliert wurde. Dies hat zur Folge, daß jeder weitere Versuch, eine Anweisung dem Modell hinzuzufügen, mit einem Fehler abbricht. Auf diese Weise wird sichergestellt, daß `return`-Anweisungen stets als letzte Anweisungen in Methoden auftreten.

- Die Modellierung von *Auswahlanweisungen* erfolgt wie in Kapitel 4 beschrieben. Die bei der Modellierung generierte Komponente ist eine Instanz der Klasse `ValueBasedDCSelectionStatement`. Im weiteren werden einige Details der Implementierung näher betrachtet.

  Die Modellierung der Anweisungen der `then-` und `else-`Zweige erfolgt nicht in dem als Argument zu `buildValueBasedModel:` angegebenen Modell, sondern es wird jeweils ein neues Modell generiert, welches alle Verbindungen des angegebenen Modells enthält. Die

Modellierung der Zweige erfolgt in dem für den jeweiligen Zweig generierten Modell. Anschließend werden alle Komponenten und Verbindungen der Modelle der Zweige in das ursprüngliche Modell übernommen und die Input- und Output-Ports der die Auswahlanweisung repräsentierenden Komponente entsprechend verbunden. Durch diese Vorgehensweise können alle in einem der beiden Zweige veränderten Verbindungen ermittelt werden. Dies ist notwendig, um die Menge der Input- und Output-Ports der die Auswahlanweisung repräsentierenden Komponente zu bestimmen.

- Die Modellierung von *Ausdrücken* bietet wenige Besonderheiten und erfolgt wie bei der Beschreibung des indirekten Modells angegeben. Nachfolgend werden die generierten Komponenten sowie die Implementierung einiger in Abschnitt 4 nicht behandelter Ausdrücke aufgelistet.

  - *Literale* werden durch Komponenten der Klasse `ValueBasedDCLiteralExpression` repräsentiert.
  - Unäre und binäre *Operatoren* werden durch die Klassen `ValueBasedDCUnaryOperator` bzw. `ValueBasedDCBinaryOperator` nachgebildet. Eine Ausnahme stellen die Operatoren '`++`' und '`--`' dar, welche durch hierarchische Komponenten implementiert werden. Abhängig davon, ob die betroffenen Ausdrücke Variablen in der Umgebung oder im Objektraum ansprechen, werden die Operatoren durch die Komponenten `ValueBasedDCVariable*Operator` bzw. `ValueBasedDCField*Operator` modelliert. Stellt der Operator einen Prefix-Operator dar, ist „`*`“ durch „`Pre`“ zu ersetzen. Andernfalls durch „`Post`“.
  - Auswahlausdrücke (`?:`) werden analog zu Auswahlanweisungen implementiert, wobei für jeden Zweig des Ausdrucks sowie zu den Output-Ports der die Auswahl repräsentierenden Komponente ein Port `result` hinzugefügt wird, welcher das Ergebnis des jeweiligen Ausdrucks repräsentiert. Weitere Details der Implementierung sind bei der Beschreibung der Auswahlanweisung nachzulesen.
  - Die Implementierung von *Funktionsaufrufen* und ***`new`-Ausdrücken*** erfolgt wie bereits bei der Beschreibung von Konstruktor- und Methodenaufrufen behandelt.
  - *Typumwandlungen* werden durch Komponenten der Klassen `ValueBasedDCTypeConversion` repräsentiert. Die Komponenten sind ähnlich zu unären Operatoren aufgebaut und weisen jeweils einen Input- und einen Output-Port auf, welcher mit dem Ausdruck bzw. dem Ergebnis der Umwandlung assoziiert ist.

    Hier ist anzumerken, daß nur Typumwandlungen zwischen primitiven Datentypen explizit im Modell repräsentiert werden, da Konversionen zwischen nicht-primitiven Datentypen auf das Modell keine Auswirkungen haben und daher ignoriert werden können.
  - Bei *Zugriffen auf Variablen* ist – analog zu Zuweisungen – zu unterscheiden, ob der Zugriff auf die Umgebung oder den Objektraum erfolgt. Die generierten Komponenten sind Instanzen der Klassen `ValueBasedDCVariableAccess` bzw. `ValueBasedDCFieldAccess`.

    Die für Zuweisungen beschriebene Optimierung bei Zuweisungen an Klassenvariablen ist auch in diesem Fall anwendbar und in analoger Weise implementiert.

- Die von dem Modell unterstützten Arten von *Schleifen* sind `while`-Anweisungen, `do`-Anweisungen, sowie `for`-Anweisungen. Die für die jeweiligen Anweisungen generierten Komponenten sowie nennenswerte Details der Implementierung werden im weiteren behandelt.

- *`while`-Anweisungen* werden durch Instanzen der Klasse `ValueBasedDCWhileStatement` modelliert.
- *`do`-Anweisungen* werden durch Instanzen der Klasse `ValueBasedDCDoStatement` modelliert.
- *`for`-Anweisungen* werden in der vorliegenden Implementierung nicht durch spezielle Komponenten repräsentiert, sondern auf eine `while`-Anweisung zurückgeführt.

  Dies erfolgt in ähnlicher Weise wie die Modellierung der Zweige von Auswahlanweisungen: Zunächst wird ein künstlicher Block erzeugt, welcher die Anweisungen des Initialisierungs-Teils der `for`-Anweisung, gefolgt vom Rest der Schleife, repräsentiert als `while`-Anweisung, enthält. Dieser Block wird dann analog zu einem der Zweige einer Auswahlanweisung modelliert. Die Bedingung der generierten `while`-Anweisung wird durch die bei der `for`-Anweisung angegebene Bedingung bestimmt. Der Rumpf der Schleife bildet sich aus dem Rumpf der `for`-Anweisung, gefolgt von den Anweisungen des Update-Teils der Schleife.

  Diese Vorgehensweise bei der Modellierung ist möglich, da das Modell keine `break`- oder `continue`-Anweisungen unterstützt und die Update-Anweisungen der Schleife in jedem Fall nach der Abarbeitung des Schleifenrumpfs abgearbeitet werden.

  Bei der Modellierung von `for`-Anweisungen durch `while`-Anweisungen ist zu beachten, daß im Initialisierungs-Teil der Schleife deklarierte Variablen in den Output-Verbindungen des Modells der `while`-Anweisung aufscheinen, da der Gültigkeitsbereich der Deklaration der bei der Modellbildung erzeugte Block ist und die Variablen somit auch außerhalb der `while`-Anweisung sichtbar sind. Dies steht im Gegensatz zu einer Modellierung der gesamten `for`-Anweisung durch eine separate Komponente. Die zusätzlichen Output-Ports stellen jedoch eine Möglichkeit bereit, Beobachtungen für diese Variablen zu spezifizieren.

Modelle der Schleifen werden als hierarchische Komponenten implementiert, welche die Modelle der Schleifenbedingung und des Schleifenrumpfs enthalten.

Die Bildung der Modelle für die Schleifenbedingung und den Schleifenrumpf erfolgt unter Einsatz zweier Hilfsklassen, welche zu einem gegebenen Block bzw. einem gegebenen Ausdruck temporäre Methodendeklarationen erzeugen, mit deren Hilfe ein Modell für den Block bzw. den Ausdruck berechnet werden kann. Die beiden Klassen `ValueBasedAuxiliaryMethodDeclaration` bzw. `ValueBasedAuxiliaryMethodDeclarationForExpression` sind von der Klasse `MethodDeclaration` abgeleitet. Diese Klassen werden zusammen mit einem neu erzeugten, leeren Modell verwendet, um Modelle für den Block bzw. die Anweisung zu berechnen. Ihre Aufgabe besteht darin, bei der Modellbildung Verbindungen für alle Variablen des aktuellen Modells zu generieren sowie das Ergebnis des Ausdrucks als Output-Verbindung `return` zur Verfügung zu stellen.

Die unter Einsatz der soeben beschriebenen Hilfsklassen berechneten Modelle für die Schleifenbedingung bzw. den Schleifenrumpf werden durch Anwenden der Methode `asConstraintModel` des Modells in Constraint-Systeme überführt und als hierarchische Komponenten in `ValueBasedDCWhileStatement` bzw. `ValueBasedDCDoStatement` eingesetzt.

Damit ist die Beschreibung der wichtigsten Algorithmen und Klassen der strukturellen Modellbildung abgeschlossen. Im nächsten Abschnitt wird die Implementierung der Verhaltensbeschreibung der einzelnen Komponenten des Modells näher betrachtet.

5.3 Implementierung des Verhaltens der Komponenten

Im letzten Abschnitt wurde die Bildung der Modellstruktur aus einem gegebenen Java-Programm behandelt. Einzelne Programmelemente, wie Anweisungen und Ausdrücke wurden als Komponenten repräsentiert, welche mit Verbindungen untereinander vernetzt sind, wobei die Verbindungen den benutzten oder veränderten Variablen des Programms entsprechen. Die Komponenten modellieren die durch das von ihnen repräsentierte Programmelement verursachten Veränderungen der Werte der Variablen, indem sie Zusammenhänge zwischen den Input- und Output-Ports der Komponenten herstellen.

Die Menge der Komponenten und Verbindungen kann als Constraint-System aufgefaßt werden, wobei die Verbindungen den Constraint-Variablen entsprechen und die Komponenten den Constraints. Das Berechnen von Werten für Verbindungen des Systems erfolgt durch Propagieren der durch Beobachtungen bekannten Werte über Komponenten und Verbindungen hinweg. Dieser Prozeß wird solange fortgesetzt, bis keine neuen Werte für Verbindungen mehr berechnet werden können, oder eine Inkonsistenz des Systems festgestellt wird. Ein System wird als inkonsistent betrachtet, sobald für eine Verbindung zwei unterschiedliche Werte berechnet werden. Hier ist zu beachten, daß dies nur für Verbindungen gilt, welche einfache Werte repräsentieren. Für Verbindungen, welche einem Objektraum zugeordnet sind, ist eine abgeänderte Vorgehensweise notwendig, da diese Verbindungen eine Menge von Variablen repräsentieren. Die hierfür notwendigen Erweiterungen des Constraintsystems werden in Abschnitt 5.4 betrachtet. Eine detailliertere Beschreibung der in dieser Arbeit verwendeten Bibliothek zur modellbasierten Diagnose und des dabei eingesetzten Constraint-Systems ist in [Wot97] nachzulesen.

Im folgenden wird die Implementierung der Verhaltensbeschreibung der Komponenten behandelt. Es werden die wichtigsten Klassen und deren Aufgaben sowie einige essentielle Details der Implementierung angeführt. Eine Übersicht über die Abhängigkeiten der Klassen untereinander ist in den Abbildungen 5.5 und 5.6 dargestellt.

- Die Klasse `ValueBasedDiagnosisComponent` repräsentiert die Basisklasse für alle Komponenten des Modells. Es werden einige Methoden bereitgestellt, welche bei der Implementierung der Subklassen verwendet werden. Dies umfaßt z.B. das Ermitteln von Portnamen für einen gegebenen Namen einer Verbindung oder das Initialisieren der Fehlermodelle der Komponente.

  Weiters implementiert diese Komponente die Schnittstelle zum Constraint-Netzwerk der modellbasierten Diagnose. Die Schnittstelle der Klasse zum Constraint-Propagator wird durch die Superklasse `SCConstraintComponent` implementiert, welche aus der bereits bestehenden Bibliothek zur modellbasierten Diagnose entnommen ist.

  Schließlich wird durch diese Klasse eine Schnittstelle zum Debugger bereitgestellt, welche eine Zuordnung zwischen den Komponenten des Modells und den Programmelementen ermöglicht. Dies erfolgt durch eine Instanz der Klasse `StatementReference`, welche einen Verweis auf das repräsentierte Programmelement enthält.

- Die Klasse `ValueBasedCachingDiagnosisComponent` erweitert die Klasse `ValueBasedDiagnosisComponent` in der Weise, daß für jede Input- und Output-Verbindung festgestellt werden kann, ob deren Wert seit der letzten Abarbeitung des Constraints verändert wurde. Diese Funktionalität wird von den Subklassen verwendet, um bei Komponenten mit vielen Ports nur jene erneut zu berechnen, welche tatsächlich verändert wurden. Dies ermöglicht in einigen Fällen eine beträchtliche Steigerung der Geschwindigkeit bei der Suche nach Conflict-Sets.

- Die Klasse `ValueBasedDCLiteralExpression` implementiert das Verhalten von Komponenten, welche Konstanten des Programms repräsentieren. Die repräsentierte Konstante

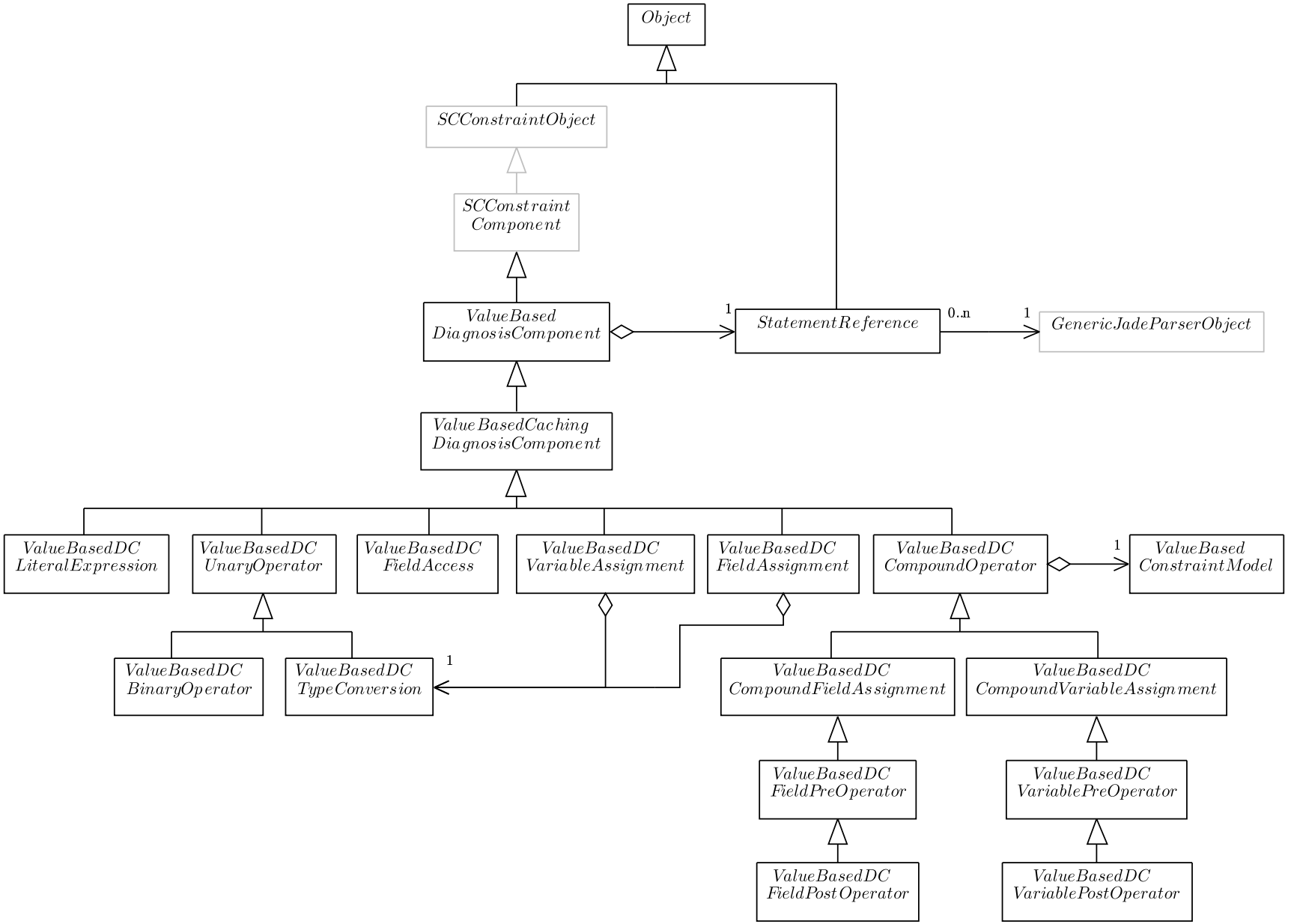

Abbildung 5.5: Klassenhierarchie der Modellkomponenten für Ausdrücke. Klassen der Bibliothek zur modellbasierten Diagnose und des Java-Parsers sind heller dargestellt

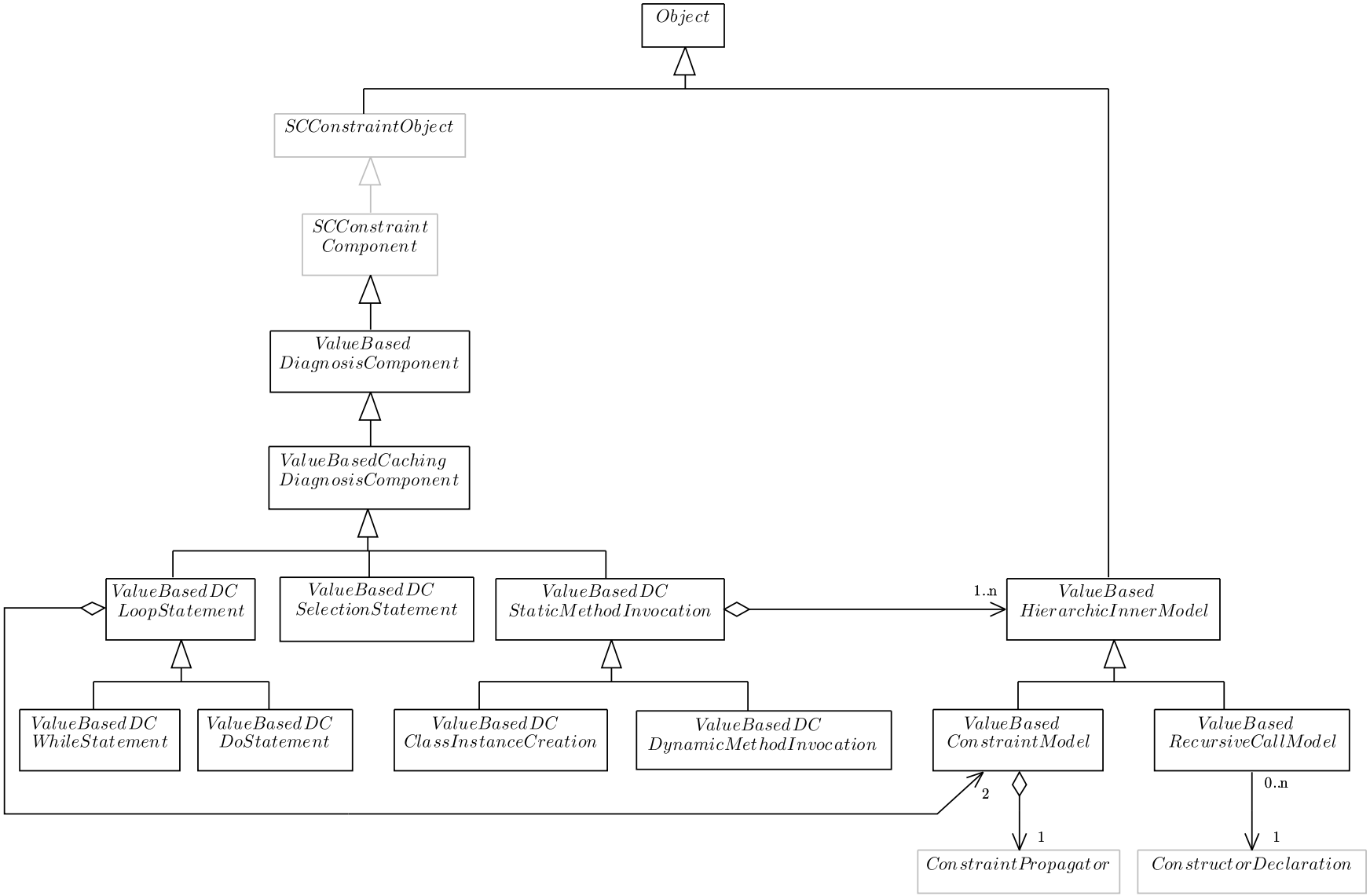

Abbildung 5.6: Klassenhierarchie der Modellkomponenten für Anweisungen. Klassen der Bibliothek zur modellbasierten Diagnose und des Java-Parsers sind heller dargestellt

ist in der Instanzvariablen `value` enthalten und wird an den Output-Port der Komponente propagiert.

- Die Klasse `ValueBasedDCUnaryOperator` implementiert das Verhalten von Komponenten, welche unäre Operatoren oder Zugriffe auf Variablen einer Umgebung repräsentieren. Die Komponente beinhaltet eine Instanzvariable, welche einen Selektor in Form eines `ByteSymbol`s enthält, welches die repräsentierte Operation auswählt. Der Selektor wird sowohl beim Propagieren von den Input-Ports zu den Output-Ports verwendet, als auch beim Propagieren in umgekehrter Richtung. Dies ist möglich, da für alle unären Operatoren gilt: $op(A) = op^*(A)$, wobei $op(A)$ die Anwendung des Operators op auf den Ausdruck A darstellt und op^* den zu op korrespondierenden inversen Operator.

  Hier ist zu beachten, daß sämtliche Operationen in Smalltalk-Arithmetik ausgeführt werden. Dies hat zur Folge, daß Berechnungen, welche Überläufe von Wertebereichen verursachen, in dem Modell unterschiedliche Ergebnisse gegenüber der Java-Spezifikation aufweisen können. Auf eine Nachbildung der Java-Semantik wurde in der vorliegenden Implementierung verzichtet, da dadurch das Propagieren von Werten von den Output-Ports zu den Input-Ports in vielen Fällen nicht möglich wäre.

- Die Klasse `ValueBasedDCBinaryOperator` ist eine Subklasse von `ValueBasedDCUnaryOperator` und implementiert das Verhalten von Komponenten, welche binäre Operatoren darstellen. Analog zu unären Operatoren wird die repräsentierte Operation durch einen Selektor ausgewählt. Um die Implementierung des Verhaltens der Operatoren übersichtlicher zu gestalten, wurden für jeden Operator drei Methoden implementiert, welche zusammen das vollständige Verhalten des Operators spezifizieren. Die bei der Abarbeitung des Constraints aufgerufene Methode wird durch den Selektor und durch die Menge der Ports, welche tatsächlich Werte aufweisen, bestimmt.

  Die bei der Beschreibung der Klasse `ValueBasedDCUnaryOperator` angegebenen Abweichungen gegenüber der Java-Spezifikation sind auch hier zu beobachten.

  Weiters werden Ausnahmesituationen bei arithmetischen Operationen bei vorliegender Implementierung unterschiedlich behandelt, als in der Java-Spezifikation angegeben: Eine Division durch Null bedingt keine Exception, sondern es wird ein ausgezeichneter Wert an den Output-Port propagiert, welcher den Fehlerfall repräsentiert.

- Die Klasse `ValueBasedDCTypeConversion` implementiert das Verhalten von Komponenten, welche Umwandlungen zwischen Datentypen modellieren. Diese Klasse ist ebenfalls von der Klasse `ValueBasedDCUnaryOperator` abgeleitet, da auf diese Weise die Methoden zur Verwaltung der Verbindungen wiederverwendet und zum Auswählen der gewünschten Operation durch einen Selektor herangezogen werden können. Zur Implementierung der Komponente ist anzumerken, daß bei Anwendung von verlustbehafteten Konversionen, wie z.B. `float` zu `int`, nicht von den Output-Ports zu den Input-Ports propagiert werden kann.

- Die Klasse `ValueBasedDCFieldAccess` implementiert das Verhalten von Komponenten, welche Zugriffe auf Instanzvariablen von Objekten repräsentieren. Hier ist zu beachten, daß im Fall eines Zugriffs über einen `null`-Verweis im Gegensatz zur Java-Spezifikation keine Exception ausgelöst, sondern das Verhalten der Komponente auf die Null-Operation reduziert wird.

- Die Klasse `ValueBasedDCVariableAssignment` stellt die Implementierung von Komponenten dar, welche Zuweisungen an Variable einer Umgebung modellieren. Die Komponente enthält eine Instanz der Klasse `ValueBasedDCTypeConversion`, mit deren Hilfe

implizite Konversionen bei Zuweisungen implementiert werden. Hier ist zu beachten, daß bei Anwendung von verlustbehafteten Konversionen nicht von den Output-Ports zu den Input-Ports propagiert werden kann.

- Die Klasse `ValueBasedDCFieldAssignment` implementiert Komponenten, welche Zuweisungen an Instanzvariablen von Objekten repräsentieren. Analog zu Zuweisungen werden auch bei diesen Komponenten Instanzen von `ValueBasedDCTypeConversion` verwendet, um implizite Konversionen zu modellieren. Wie schon bei Zugriffen auf Instanzvariablen behandelt, werden bei Zugriffen über `null`-Verweise keine Exceptions ausgelöst.

- Die Klasse `ValueBasedDCCompoundOperator` implementiert die gemeinsame Funktionalität aller zusammengesetzten Zuweisungsoperatoren sowie der Prefix- und Postfix-Inkrement- bzw. Dekrement-Operatoren. Diese Operatoren werden als hierarchische Komponenten implementiert, welche eine Instanz der Klasse `ValueBasedConstraintModel` enthalten, welche die eigentliche Implementierung der Verhaltensbeschreibung in Form eines Constraint-Netzwerks enthält. Das Generieren der Ports sowie des Constraint-Netzwerks erfolgt durch die Implementierung der Subklassen.

- Die Klasse `ValueBasedDCCompoundFieldAssignment` implementiert das Verhalten von Zuweisungen an Instanzvariablen von Objekten, wobei die Zuweisungen von der Form $var \oplus= expr$ mit $\oplus \in \{+, -, *, /, \&, |, \hat{}, \%, <<, >>, >>>\}$ sind. Die Implementierung erfolgt durch ein Constraint-Netzwerk, welches den Ausdruck $var = var \oplus expr$ nachbildet.

- Die Klasse `ValueBasedDCFieldPreOperator` implementiert das Verhalten von Komponenten, welche die Prefix-Operatoren '`++`' bzw. '`--`' – angewendet auf Instanzvariablen von Objekten – repräsentieren. Die Implementierung ist durch Wiederverwenden der Implementierung von zusammengesetzten Zuweisungen realisiert, wobei der Input-Port des Constraint-Netzwerks, welches *expr* repräsentiert, konstant auf '1' gesetzt wird.

- Die Klasse `ValueBasedDCFieldPostOperator` implementiert Komponenten, welche die Postfix-Operatoren '`++`' bzw. '`--`' – angewendet auf Instanzvariablen von Objekten – darstellen. Die Implementierung erfolgt durch Wiederverwendung der Implementierung der Prefix-Operatoren, wobei das Constraint-Netzwerk dahingehend abgeändert wird, daß am Output-Port `result` nicht der inkrementierte bzw. dekrementierte Wert, sondern der ursprüngliche Wert der Variablen ausgegeben wird.

- Die Klasse `ValueBasedDCCompoundVariableAssignment` implementiert das Verhalten von Komponenten, welche Zuweisungen an Variablen einer Umgebung modellieren. Die Implementierung erfolgt analog zu der Klasse `ValueBasedDCCompoundFieldAssignment`, wobei die Anzahl der benötigten Input- und Output-Ports abweichend ist. Dies wird durch die unterschiedliche Anzahl von Ports bei Zugriffen und Zuweisungen an Variablen eines Objektraums bzw. einer Umgebung verursacht.

- Die Klasse `ValueBasedDCVariablePreOperator` implementiert das Verhalten von Komponenten, welche die Prefix-Operatoren '`++`' und '`--`' – angewendet auf Variablen einer Umgebung – repräsentieren. Die Implementierung erfolgt analog zu jener von `ValueBasedDCFieldPreOperator`.

- Die Klasse `ValueBasedDCVariablePostOperator` implementiert das Verhalten von Komponenten, welche die Postfix-Operatoren '`++`' und '`--`' – angewendet auf Variablen einer Umgebung – repräsentieren. Die Implementierung erfolgt analog zu jener von `ValueBasedDCFieldPostOperator`.

- Die Klasse `ValueBasedDCLoopStatement` implementiert das gemeinsame Verhalten von `while`- und `do`-Anweisungen. Hierzu zählt die Initialisierung der Fehlermodelle sowie das Propagieren der Werte der Input- und Output-Ports zu den Modellen der Bedingung und des Schleifenrumpfs. Die Modelle der Bedingung und des Schleifenrumpfs werden durch Instanzen der Klasse `ValueBasedConstraintModel` repräsentiert.

  Die Komponente implementiert neben den üblichen Fehlermodellen $ab(C)$ und $\neg ab(C)$ ein weiteres Fehlermodell $loop(C, max)$. Dieses setzt die Anzahl der Iterationen der Schleife auf max, unabhängig von der Auswertung der Schleifenbedingung. Mögliche Werte für die Konstante max sind bei vorliegender Implementierung $n-1$ und $n+1$, wobei n die Anzahl der Iterationen im gegebenen Programm angibt. Um n zu bestimmen, ist es notwendig, allen Input-Verbindungen des Modells der Methode einen definierten Wert zuzuweisen und anschließend einen Programmablauf mit dem Modell zu simulieren. Auf diese Weise kann die Anzahl der Iterationen im ursprünglichen Programm und damit die möglichen Fehlermodelle bestimmt werden.

  Die Implementierung des Verhaltens der Komponente im Fehlermodell $\neg ab(C)$ ist durch die Subklassen zu implementieren.

- Die Klasse `ValueBasedDCWhileStatement` implementiert Komponenten, welche `while`-Anweisungen modellieren. Es wird die Implementierung der Klasse `ValueBasedDCLoopStatement` um die Implementierung des Verhaltens für das Fehlermodell $\neg ab(C)$ erweitert.

- Die Klasse `ValueBasedDCDoStatement` implementiert das Verhalten von Komponenten, welche `do`-Anweisungen darstellen. Die Implementierung erfolgt in analoger Weise wie bei `while`-Anweisungen.

- Die Klasse `ValueBasedDCSelectionStatement` implementiert das Verhalten von Komponenten, welche Auswahlanweisungen nachbilden. Entsprechend dem am Input-Port `cond` anliegenden Wert werden die Werte der Verbindungen vom entsprechenden Zweig der Auswahlanweisung zu den Output-Ports propagiert. Ist der Wert des Ports `cond` undefiniert, werden die am Ausgang anliegenden Werte auf Konsistenz mit mindestens einem Zweig überprüft. Weiters können in diesem Fall alle Werte von Verbindungen an die Output-Ports propagiert werden, deren Werte in beiden Zweigen der Anweisung identisch sind.

- Die Klasse `ValueBasedDCStaticMethodInvocation` implementiert das Verhalten von Komponenten, welche Modelle von Aufrufen von Klassenmethoden und Konstruktoren darstellen. Weiters implementiert die Klasse das gemeinsame Verhalten von Klassenmethoden-, Instanzmethoden-Aufrufen und `new`-Ausdrücken.

  Die Klasse enthält eine Menge von `ValueBasedHierarchicInnerModel`s, welche Modelle der möglicherweise aufgerufenen Methoden darstellen. Im Fall von Aufrufen von Klassenmethoden, Konstruktoren oder `new`-Ausdrücken enthält die Menge nur ein einziges Modell, da die aufgerufene Methode bereits zur Übersetzungszeit eindeutig bestimmt werden kann. Die Implementierung wurde allgemeiner gehalten, um eine leichtere Wiederverwendbarkeit bei der Implementierung von dynamischen Methodenaufrufen zu ermöglichen.

  Die Implementierung der Komponente umfaßt das Propagieren der Werte zwischen dem Modell der Methode und den Ports der Komponente. Auch werden Methoden bereitgestellt, welche es den Subklassen ermöglichen, Werte von Verbindungen vor dem Propagieren zu verändern bzw. das Propagieren der Werte zu unterdrücken.

- Die Klasse `ValueBasedDCClassInstanceCreation` implementiert das Verhalten von `new`-Ausdrücken. Die Komponente simuliert das Erzeugen eines neuen Objekts, das Initia-

lisieren der Instanzvariablen mit Default-Werten und das anschließende Abarbeiten der Initializer der Variablen und des Konstruktors.

Es wird die Implementierung der Klasse `ValueBasedDCStaticMethodInvocation` weitgehend wiederverwendet. Es sind jedoch einige Erweiterungen notwendig, um das Erzeugen neuer Objekte und das Initialisieren der Instanzvariablen zu simulieren.

Zunächst wird ein eindeutiger Objekt-Identifier für das neu erzeugte Objekt generiert. Dieser wird zum Output-Port `result` propagiert. Anschließend werden die Instanzvariablen des Objekts angelegt und mit deren Default-Werten initialisiert. Zu diesem Zweck weist die Komponente Input- und Output-Ports für *alle* Instanzvariablen der im Programm angegebenen Klasse auf, unabhängig davon, ob diese im Konstruktor der Methode benötigt oder verändert werden. Anschließend wird das Modell des Konstruktors abgearbeitet.

Aufgrund der Modellierung der Programmelemente als Constraint-Netzwerk muß beim Generieren der Objekt-Identifier mit einiger Sorgfalt vorgegangen werden. Die erzeugten Identifier müssen auf eine Weise generiert werden, welche die Eindeutigkeit der Identifier und die Termination des Propagierungsalgorithmus sicherstellt. Dies wird durch einen zusätzlichen Input-Port `callingContext` sichergestellt, welcher bei allen `new`-Ausdrücke repräsentierenden Komponenten hinzugefügt wird. Dieser Port ist mit einer vom Übersetzer generierten lokalen Variablen `callingContext` assoziiert, welche einen eindeutigen Kontext des Aufrufs darstellt.

Der Kontext wird ähnlich wie ein Call-Stack gebildet, wobei hier nicht Aufrufe von Methoden berücksichtigt werden, sondern Iterationen von Schleifen. Der Kontext wird also aus einer Folge von $(i_1, \ldots, i_n)$ gebildet, wobei n die Verschachtelungstiefe von Schleifen und i_k $(1 \leq k \leq n)$ die Nummer der jeweiligen Iteration der Schleife auf Tiefe k angibt. Der Kontext wird bei hierarchischen Komponenten an die inneren Modelle weitergegeben. Im Fall von Schleifen wird der Kontext zuvor um die Nummer der aktuellen Iteration erweitert. Der initiale Kontext, welcher als impliziter Parameter an die zu debuggende Methode übergeben wird, ist der leere Kontext.

Da die Constraint-Netzwerke in hierarchischen Komponenten niemals von mehreren Komponenten verwendet werden, können eindeutige Objekt-Identifier durch den Kontext in Verbindung mit der erzeugenden Komponente generiert werden.

Es ist zu beachten, daß die auf diese Weise generierten Identifier nur innerhalb eines Aufrufs des Constraint-Propagators gültig sind, da durch Verändern von Fehlermodellen von Komponenten jeweils unterschiedliche Identifier generiert werden können. Dies erschwert die Handhabung von Beobachtungen, da aus diesem Grund die Objekt-Identifier nicht explizit angegeben werden können (siehe auch Abschnitt 5.5).

Im Fall von rekursiven Aufrufen muß der Evaluator dahingehend modifiziert werden, daß den erzeugten Objekten auf ähnliche Weise eindeutige, reproduzierbare Identifier zugeordnet werden.

- Die Klasse `ValueBasedDCDynamicMethodInvocation` implementiert das Verhalten von Komponenten, welche dynamische Aufrufe von Methoden repräsentieren. Die Implementierung beruht zu großen Teilen auf jener der Klasse `ValueBasedDCStaticMethodInvocation`. Im Gegensatz zu statischen Aufrufen kann die tatsächlich aufgerufene Methode und ihr Modell erst bei Abarbeitung des Verhaltens der Komponente ermittelt werden. Dies erfolgt aufgrund des Typs des Objekts, auf welchem die Methode operiert. Dieser wird aus dem am Port `object` anliegenden Objekt-Identifier ermittelt.

- Die Klasse `ValueBasedHierarchicInnerModel` implementiert eine Schnittstelle, welche hierarchischen Komponenten den Zugriff auf Verbindungen und Komponenten des inneren

Modells gewährt. Instanzen von Subklassen dieser Klasse werden als innere Modelle bei der Modellierung von hierarchischen Komponenten verwendet.

Die Schnittstelle abstrahiert von der Art des Modells: es sind sowohl Modelle in Form von Constraint-Netzwerken, als auch auf einem Evaluator basierende Modelle möglich. Die Implementierung der Schnittstelle erfolgt in den Subklassen.

- Die Klasse `ValueBasedConstraintModel` implementiert die von der Klasse `ValueBasedHierarchicInnerModel` spezifizierte Schnittstelle in Form eines Constraint-Netzwerks.

- Die Klasse `ValueBasedRecursiveModel` stellt die Implementierung der von der Klasse `ValueBasedHierarchicInnerModel` spezifizierten Schnittstelle basierend auf einem Evaluator dar. Aufgabe dieser Klasse ist es, zwischen der Repräsentation der Variablen im Constraint-Netzwerk und jener des Evaluators umzuwandeln und den Evaluator zum Ablauf zu bringen.

  Da zum Zeitpunkt der Erstellung dieser Arbeit kein geeigneter Java-Evaluator zur Verfügung stand und die Entwicklung eines eigenen Evaluators den Rahmen der vorliegenden Arbeit gesprengt hätte, ist diese Klasse nicht vollständig implementiert. Insbesondere ist das Auswerten des Modells nicht möglich. Daraus ergibt sich, daß rekursive Aufrufe von Methoden zwar auf struktureller Ebene modelliert werden können, das Modell aber nicht zum Ablauf gebracht werden kann.

5.4 Erweiterungen des Constraint-Netzwerks

Um die Implementierung des Modells zu vervollständigen, bleiben noch die Behandlung von Objekten und Objekträumen während des Propagierens im Constraint-Netzwerk und die damit verbundenen Erweiterungen der verwendeten Bibliothek zu behandeln. Eine Übersicht über die Erweiterungen ist in Abbildung 5.7 dargestellt.

Das Finden von Conflict-Sets wird durch den in der vorliegenden Arbeit verwendeten Constraint-Propagator auf das Erkennen von Inkonsistenzen in einem Constraint-Netzwerk zurückgeführt. Eine Inkonsistenz in einem Constraint-Netzwerk liegt vor, sobald für eine Verbindung zwei widersprüchliche Werte berechnet werden. Repräsentiert eine Verbindung einen Objektraum, gestaltet sich die Entdeckung von Inkonsistenzen aufwendiger, da Objekträume eine Menge von Assoziationen zwischen Objekt-Identifiern und Werten einer Instanzvariablen des Objekts mit dem jeweiligen Identifier repräsentieren. Eine Inkonsistenz zwischen Objekträumen liegt dann vor, falls für eine in beiden Objekträumen enthaltene Instanzvariable eines Objekts zwei widersprüchliche Werte berechnet werden.

Die im folgenden behandelten Erweiterungen der Bibliothek zur modellbasierten Diagnose implementieren Verbindungen, welche das Propagieren von Objekträumen unterstützen. Weiters werden einige Klassen vorgestellt, welche Objekt-Identifier bzw. Objekträume repräsentieren.

- Die Klasse `ValueBasedModelConstraintVariable` repräsentiert eine Verbindung, welche einen einzelnen Wert propagieren kann und erweitert die in der verwendeten Bibliothek zur modellbasierten Diagnose enthaltene Klasse `SCConstraintVariable` dahingehend, daß zusätzliche, für den Debugger notwendige Informationen zu jeder Verbindung gespeichert werden. Dies sind u.a. der Name der Verbindung und der Java-Typ des propagierten Wertes.

- Die Klasse `ValueBasedObjectSpaceConstraintVariable` erweitert die Klasse `ValueBasedModelConstraintVariable` dahingehend, daß Objekträume propagiert werden

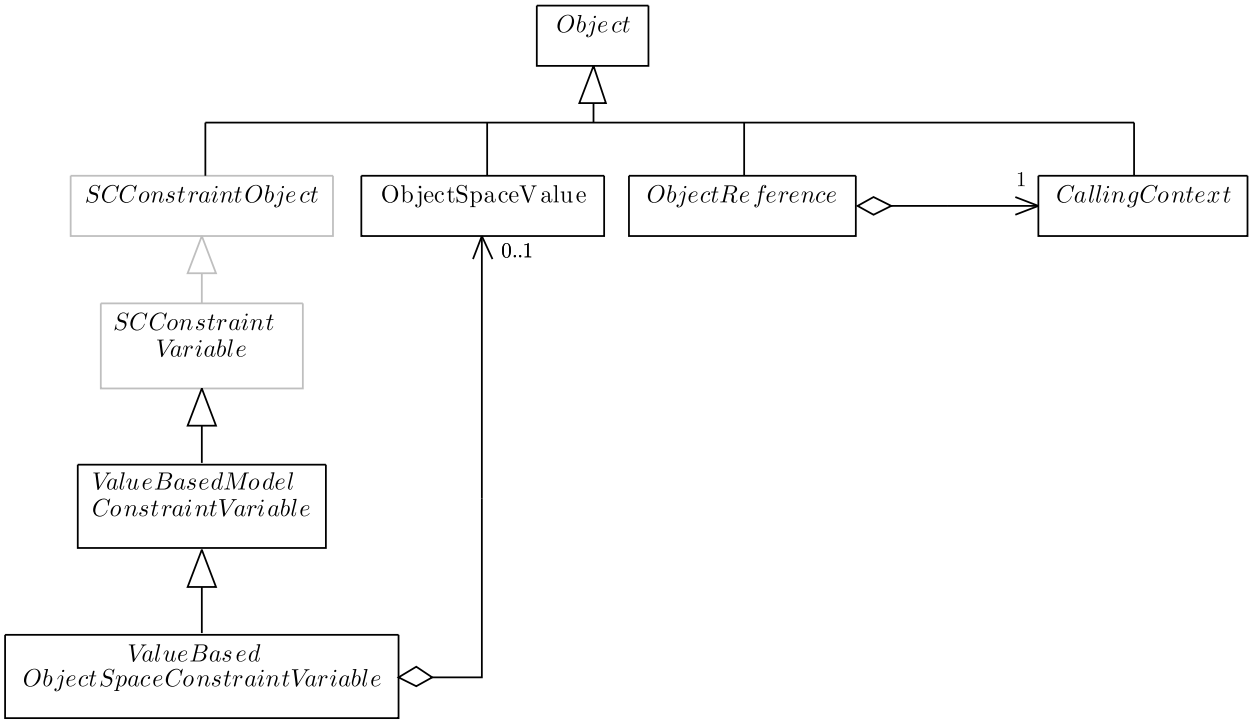

Abbildung 5.7: Erweiterungen des Constraint-Netzwerks. Klassen der verwendeten Bibliothek zur modellbasierten Diagnose sind heller dargestellt

können. Objekträume werden durch Instanzen der Klasse `ObjectSpaceValue` repräsentiert.

Wird ein Objektraum für eine solche Verbindung berechnet, wird dieser mit dem bereits in der Verbindung gespeicherten Objektraum zusammengeführt. Der auf diese Weise berechnete Wert stellt den neuen Wert der Verbindung dar. Ist das Zusammenführen der Objekträume auf konsistente Weise nicht möglich, wird dem Constraint-Propagator eine Inkonsistenz signalisiert.

Da Objekträume Werte von Instanzvariablen mehrerer Objekte zusammenfassen, ist es nicht ausreichend, die Zustände der Objekträume entsprechend der Abarbeitungsreihenfolge der Anweisungen zu propagieren. Zusätzlich dazu ist es erforderlich, die Objekträume in entgegengesetzter Richtung zu propagieren, um durch Rückwärtspropagieren ermittelte Werte von Instanzvariablen korrekt zu behandeln.

- Instanzen der Klasse `ObjectSpaceValue` repräsentieren Objekträume, eingeschränkt auf eine Variable. Ein Objektraum besteht aus einer Menge von Assoziationen zwischen Objekt-Identifiern und den Werten der repräsentierten Variablen. Es werden Methoden bereitgestellt, um Werte von Instanzvariablen für Objekte zu setzen bzw. zu ermitteln und um Objekträume zusammenzuführen.

- Instanzen der Klasse `ObjectReference` stellen Objekt-Identifier dar, welche – wie in Abschnitt 5.3 angegeben – mittels Kontext-Informationen berechnet werden. Zusätzlich dazu wird der Typ des generierten Objekts gespeichert, um das Bestimmen des Objekt-Typs bei dynamischen Methodenaufrufen zu vereinfachen.

- Die Klasse `CallingContext` repräsentiert den Kontext eines `new`-Ausdrucks und wird verwendet, um eindeutige Objekt-Identifier zu generieren. Details zur Berechnung sind bei der Beschreibung von `new`-Ausdrücken in Abschnitt 5.3 angegeben.

5.5 Beobachtungen

5.5.1 Beobachtungen und Objekt-Identifier

Um das Modell zur modellbasierten Diagnose einsetzen zu können, ist es erforderlich, Beobachtungen für Verbindungen anzugeben. Für Verbindungen, welche Variablen der Umgebung repräsentieren und deren Typ ein primitiver Datentyp ist, können Werte vor Abarbeiten des Constraint-Netzwerks in die jeweiligen Verbindungen eingetragen werden. Bei Beobachtungen, welche von Objekt-Identifiern abhängig sind, kann dies nicht in dieser Weise erfolgen, da die Objekt-Identifier aufgrund ihrer Konstruktion nur jeweils während eines Aufrufs des Constraint-Propagators gültig sind und daher nicht in Beobachtungen verwendet werden können. Dies kann anhand eines Beispiels demonstriert werden:

Beispiel 5.2 Angenommen, das in Abbildung 5.8 dargestellte Programm sei auf Korrektheit zu überprüfen. Zunächst wird das Modell der Methode `demo()` ohne Beobachtungen abgearbeitet.

```
1  package Demo;
2
3  class ValueHolder {
4      int value;
5      ValueHolder(int value) {
6          this.value = value;
7      }
8  }
9
10 public class OIDObservationDemo {
11     public static void demo()
12     {
13         int counter = 1;
14         ValueHolder object = null;
15         for(int n = 0; n < counter; n++){
16             object = new ValueHolder(n);
17         }
18     }
19 }
```

Abbildung 5.8: Programm zur Demonstration falscher Beobachtungen

Nach Zeile 17 ergeben sich die folgenden Werte für die einzelnen Variablen ($(0, new_{16})$ bezeichnet den Objekt-Identifier, welcher dem in Zeile 16 während der ersten Iteration der Schleife erzeugten Objekt zugeordnet ist):

Variable	berechnet	erwartet
`counter`	1	2
`object`	$(0, new_{16})$	–
`object.value`	0	1

Da die berechneten Werte von den erwarteten Werten abweichen, werden zwei Beobachtungen hinzugefügt: die Verbindung für die Variable `counter` wird auf den Wert `2` gesetzt. Weiters wird eine Beobachtung für den Wert von `object.value` in Zeile 17 hinzugefügt. Da die Variable im Objektraum enthalten ist, muß die Beobachtung ebenfalls als Objektraum dargestellt werden, d.h. es muß ein Objektraum erzeugt werden, welcher eine Assoziation zwischen dem Objekt-Identifier von `object` und dem erwarteten Wert von `value` beinhaltet: $\{val((0, new_{16}), \texttt{ValueHolder.value}, 1)\}$. Angenommen dieser Objektraum wird als Beobachtung für die Verbindung, welche `ValueHolder.value` repräsentiert, hinzugefügt.

Nach dem erneuten Anstoßen des Diagnosealgorithmus sind keine Einfachfehlerdiagnosen in der Menge der Diagnosen enthalten. Dies widerspricht den intuitiven Diagnosen $\{[1]_{13}\}$ und $\{[counter = 1]_{13}\}$, welche das erwartete Verhalten erklären würden.

Die Ursache hierfür liegt in der Art und Weise, wie die Beobachtung für die Variable `object.value` angegeben wurde. Dabei wurde der Objekt-Identifier explizit spezifiziert. Dies verletzt die in Abschnitt 5.3 aufgeführten Einschränkungen bezüglich der Gültigkeit der Objekt-Identifier.

Die Ursache für das Fehlen der Diagnosen findet sich bei näherer Betrachtung in der Art und Weise, wie Objekt-Identifier generiert werden: Durch die Beobachtung der Variable `counter` wird das Verhalten der Schleife verändert. Diese wird nun zweimal durchlaufen und dabei jeweils ein Objekt generiert. Im ersten Durchlauf wird ein Objekt mit Identifier $(0, new_{16})$ erzeugt, im zweiten Durchlauf eines mit Identifier $(1, new_{16})$. Folglich beinhaltet der mit Verbindung `ValueHolder.value` nach Abarbeitung der Schleife assoziierte Objektraum die Zuordnungen $\{val((0, new_{16}), 0), val((1, new_{16}), 1)\}$. Dadurch wird die Beobachtung dem „falschen“ Objekt zugeordnet. Anstelle des Objekts, welches durch die Variable `object` referenziert wird, wird die Beobachtung jenem Objekt zugeordnet, welches während der *ersten* Iteration der Schleife erzeugt wurde. Dieses weist aber einen widersprüchlichen Wert auf und verhindert somit die Berechnung von Einfachfehlerdiagnosen.

Darüber hinaus widerspricht die in dem Beispiel angegebene Diagnose in gewisser Weise der intuitiven Semantik der Beobachtung. Die in diesem Fall vom Benutzer intendierte Semantik der Beobachtung kann in etwa durch *„Der Wert der Instanzvariablen `value` jenes Objekts, auf das die Variable `object` verweist, soll den Wert `1` aufweisen.“* ausgedrückt werden. Dies wird durch die angegebene Beobachtung aber nicht spezifiziert, da diese den veränderten Wert von `object` nicht berücksichtigt.

Die in der vorliegenden Arbeit gewählte Lösung des im obigen Beispiel demonstrierten Problems besteht darin, bei Beobachtungen an Objekt-Identifiern und Variablen im Objektraum den Objekt-Identifier nicht explizit anzugeben, sondern stattdessen eine Verbindung auszuwählen, von welcher der Objekt-Identifier entnommen wird. Die Beobachtung wird dann durch ein zusätzliches Constraint zwischen der gewählten Verbindung und der beobachteten Verbindung spezifiziert. Um zu verhindern, daß diese Constraints in Diagnosen aufscheinen, werden diese Constraints durch Fokussieren auf die Komponenten des Modells des Programms ausgeschlossen.

An dieser Stelle sei darauf hingewiesen, daß diese Vorgehensweise beim Spezifizieren von Beobachtungen nur für jene Objekt-Identifier notwendig ist, welche innerhalb von Schleifen oder in Aufrufen rekursiver Methoden generiert werden. In allen anderen Fällen sind die Objekt-Identifier auch über mehrere Aufrufe des Constraint-Propagators hinweg eindeutig und können daher wie primitive Datentypen direkt angegeben werden.

Vorteil dieser Vorgehensweise ist, daß die Beobachtung auch bei veränderlichen Objekt-Identifiern die beabsichtigte Semantik aufweist. Als nachteilig muß angesehen werden, daß die Beobachtungen nur dann zur Diagnose beitragen können, falls die ausgewählte Verbindung tatsächlich einen Objekt-Identifier aufweist. Insbesondere bei Anwendung von $ab(C)$-Fehlermodellen können für eine große Anzahl von Verbindungen oftmals keine konkreten Werte berechnet werden. Daraus folgt auch, daß der Diagnoseprozeß nicht in jedem Fall fortgeführt werden kann, bis eine einzige Diagnose als mögliche Fehlerursache isoliert ist.

Die vorliegende Implementierung weist eine weitere Einschränkung auf. Die Menge der beobachtbaren Instanzvariablen wird in der Weise eingeschränkt, daß nur solche Variablen beobachtet werden können, welche Instanzvariablen von Objekten darstellen, für welche eine Verbindung des aktuellen Modells existiert, die den Objekt-Identifier des Objekts beinhaltet. So können z.B. keine Instanzvariablen von Objekten in einer verketteten Liste beobachtet werden, auf welche keine Verbindungen des Modells verweisen. Diese Einschränkungen stellen keine prinzipiellen

Grenzen des Verfahrens dar. Dieses könnte auf ähnliche Weise wie Zugriffe auf Instanzvariablen durch Field-Access-Ausdrücke erweitert werden.

Eine mögliche Verallgemeinerung besteht darin, beliebige in Java oder auf andere Weise spezifizierte Zusicherungen in das Programm einzufügen, mit deren Hilfe auch komplexere Beobachtungen angegeben werden können. Die Zusicherungen werden wie Teile von Java-Programmen in Verbindung mit Beobachtungen primitiver Datentypen modelliert und dem Modell hinzugefügt, jedoch durch Fokussieren aus der Menge der zu diagnostizierenden Komponenten ausgeschlossen. Auf diese Weise können dann auch z.B. Relationen zwischen zwei Variablen modelliert werden.

Als Beispiel sei die Beobachtung *„der Wert der Variablen* `var1` *muß gleich dem Wert der Variablen* `var2` *sein, wobei der exakte Wert nicht relevant ist“* angenommen. Diese Art von Beobachtungen kann bei herkömmlicher Vorgehensweise nicht spezifiziert werden. Bei Anwendung von Zusicherungen hingegen kann die Beobachtung durch ein Modell des Ausdrucks `var1==var2` in Verbindung mit einer herkömmlichen Beobachtung des Wertes `true` am Output-Port `result` des Operators ‘`==`’ dargestellt werden.

5.5.2 Selektion von Beobachtungspunkten

Ein weiterer Effekt der in unterschiedlichen Diagnosen abweichenden Objekt-Identifier stellt die Tatsache dar, daß die Bestimmung eines nächsten Beobachtungspunktes bei Verwendung der Entropie-basierten Methoden (siehe Abschnitt 3.3) nur mit stark herabgesetzter Qualität durchgeführt werden kann, da bei unterschiedlichen Objekt-Identifiern für Objekte Instanzvariablen ein und desselben Objekts nicht als solche identifiziert werden können. Auch werden von Diagnosen u.U. unterschiedlich viele Objekte erzeugt. Folglich wird das berechnete Entropie-Maß stark verzerrt.

Das in dieser Arbeit verwendete Verfahren zur Selektion eines nächsten Beobachtungspunktes basiert auf dem in Abschnitt 3.3 beschriebenen Verfahren, wobei die Menge der in die Berechnung einbezogenen Diagnosekandidaten auf die tatsächlich berechneten Diagnosen eingeschränkt wird. Es werden also keine Obermengen betrachtet. Weiters werden alle für einen Objektraum einer Verbindung berechneten Objekte als einzelne Verbindungen angesehen. Dies ermöglicht eine feinere Selektion, weist jedoch die durch unterschiedliche Objekt-Identifier verursachten Probleme auf. Ebendiese Probleme werden durch Objekte einer Verbindung, welche nicht von allen Diagnosen erzeugt werden, verursacht.

5.5.3 Implementierung von Beobachtungen

Dieser Abschnitt beschreibt die Implementierung des in Abschnitt 5.5.1 betrachteten Verfahrens zur Spezifikation von Beobachtungen als zusätzliche Constraints. Für jede Beobachtung einer Variablen wird ein die Beobachtung repräsentierendes Constraint dem Modell hinzugefügt. Durch Fokussieren werden diese Constraints als korrekt angesehen und von Conflict-Sets – und damit von Diagnosen – ausgeschlossen. Die Implementierung erfolgt durch vier Klassen, welche die zusätzlichen Constraints implementieren. Die Hierarchie der Klassen ist in Abbildung 5.9 dargestellt.

Beim Spezifizieren von Beobachtungen sind vier Fälle zu unterscheiden, wobei in einigen Fällen zusätzliche Unterscheidungen bezüglich der Stabilität[7] der Objekt-Identifier getroffen werden müssen:

[7]Ein Objekt-Identifier wird in diesem Zusammenhang als *stabil* angesehen, falls der Identifier über mehrere Aufrufe des Constraint-Propagators gültig ist. Dazu zählen alle Objekt-Identifier, welche außerhalb von Schleifen und rekursiven Methodenaufrufen erzeugt werden.

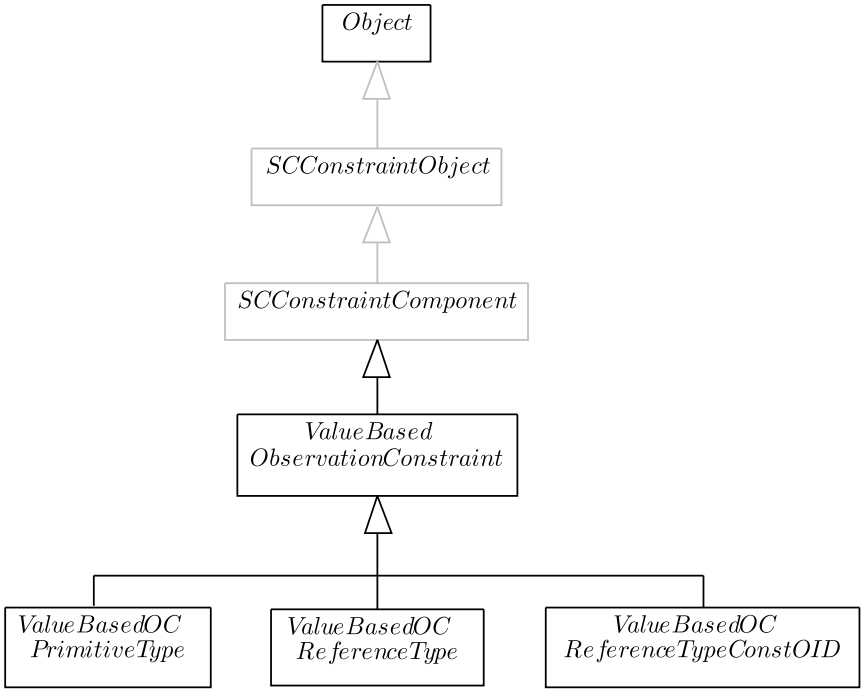

Abbildung 5.9: Constraint-Komponenten für Beobachtungen. Klassen der Bibliothek zur modellbasierten Diagnose sind heller dargestellt

1. Beobachtungen an *Variablen der Umgebung*, welche Werte *primitiver Datentypen* enthalten.

   Die Beobachtungen sind unabhängig von Objekt-Identifiern und Objekträumen. Daher können diese Beobachtungen auf herkömmliche Weise durch Setzen der Werte von Verbindungen angegeben werden.

2. Beobachtungen an *Variablen der Umgebung*, welche *Objekt-Identifier* enthalten.

   Eine Beobachtung wird durch eine Instanz der Klasse `ValueBasedObservationConstraint` dargestellt. Diese stellt ein Constraint mit einem Input-Port `oid` und einem Output-Port `observed` dar. Der Input-Port ist der Verbindung zugeordnet, welche den beobachteten Input-Port beinhaltet. Der Output-Port ist mit der beobachteten Verbindung assoziiert. Das Constraint stellt eine Beziehung zwischen den beiden Verbindungen her, wobei die Werte der beiden Verbindungen äquivalent sein müssen, um das Constraint zu erfüllen.

   Einen Spezialfall stellen Beobachtungen des Wertes `null` bzw. Beobachtungen von stabilen Objekt-Identifiern dar. Diese können – im Gegensatz zu den vom Modell generierten Objekt-Identifiern – wie Beobachtungen an primitiven Datentypen gehandhabt werden.

3. Beobachtungen an *Instanzvariablen* von Objekten, welche Werte *primitiver Datentypen* enthalten.

   Diese Beobachtungen werden durch die Klasse `ValueBasedOCPrimitiveType` implementiert. Die Klasse stellt eine Erweiterung der Klasse `ValueBasedObservationConstraint` dar, wobei eine zusätzliche Instanzvariable `observedValue` den beobachteten Wert aufnimmt. Die Implementierung des Constraints propagiert einen Objektraum, welcher eine Assoziation zwischen dem Objekt-Identifier und dem beobachteten Wert beinhaltet, zu der zu beobachtenden Verbindung, sobald ein Objekt-Identifier für die Input-Verbindung berechnet wurde.

Auch bei dieser Art von Beobachtung ist eine Optimierung für stabile Objekt-Identifier möglich. Die Beobachtung kann in diesem Fall direkt in den Objektraum der beobachteten Verbindung eingetragen werden, da der Objekt-Identifier bereits bekannt ist. Die Vorgehensweise entspricht weitgehend jener von Punkt 1. Es ist daher kein Constraint hinzuzufügen.

4. Beobachtungen an *Instanzvariablen* von Objekten, welche *Objekt-Identifier* enthalten.

   Diese Form von Beobachtungen erfordert eine Fallunterscheidung bezüglich der Stabilität der Objekt-Identifier:

   (a) Sind *beide* Objekt-Identifier *stabil*, wird die Beobachtung direkt in den Objektraum eingefügt. Dies erfolgt analog zu einer Beobachtung einer Variablen, welche Werte primitiver Datentypen beinhaltet.

   (b) Ist der *beobachtete* Objekt-Identifier *stabil*, der das *Objekt* repräsentierende Objekt-Identifier hingegen *nicht stabil*, ist analog zu Punkt 3 vorzugehen, da der beobachtete Identifier als Konstante angesehen – und somit analog zu primitiven Datentypen behandelt – werden kann.

   (c) Ist der *beobachtete* Objekt-Identifier *nicht stabil*, jener des *Objekts* hingegen *schon*, muß ein Constraint hinzugefügt werden. Das Constraint wird durch eine Instanz der Klasse `ValueBasedOCReferenceTypeConstOID` repräsentiert, welche analog zur Klasse `ValueBasedOCPrimitiveType` implementiert ist. Das Constraint weist einen Input-Port `value` auf, welcher dem beobachteten Objekt-Identifier zugeordnet ist, sowie einen Output-Port `observed`, welcher die beobachtete Verbindung repräsentiert. Die Instanzvariable `objectId` nimmt den Objekt-Identifier des beobachteten Objekts auf. Sobald am Input-Port ein konkreter Wert für den beobachteten Objekt-Identifier berechnet wurde, wird durch das Constraint eine Beobachtung des Objekts mit dem in `objectId` enthaltenen Identifier und dem beobachteten Wert in den Objektraum der zu beobachtenden Verbindung eingefügt.

   (d) Schließlich existiert der Fall einer Beobachtung, wo *keiner der beiden* Objekt-Identifier *stabil* ist. Die Implementierung von Constraints für diese Art von Beobachtung erfolgt durch die Klasse `ValueBasedOCReferenceType`. Die Klasse erweitert `ValueBasedObservationConstraint` um einen zusätzlichen Input-Port `value`, welcher dem beobachteten Objekt-Identifier zugeordnet ist. Der Port wird – analog zu dem Input-Port `oid` – mit einer Verbindung verbunden, welche den beobachteten Objekt-Identifier beinhaltet. Die Implementierung des Verhaltens des Constraints propagiert einen Objektraum, welcher eine Assoziation zwischen den von den Verbindungen `oid` und `value` berechneten Objekt-Identifiern beinhaltet, zu der mit dem Port `observed` verbundenen Verbindung.

   Auch hier sind Beobachtungen der Konstante `null` gesondert zu behandeln. In diesem Fall ist wie für stabile beobachtete Objekt-Identifier angegeben zu verfahren.

5.6 Einschränkungen der Implementierung

Abschließend werden Einschränkungen der vorliegenden Implementierung gegenüber der Java-Spezifikation aufgezeigt. Wird bei der Modellbildung ein Sprachelement angetroffen, welches durch das Modell oder die Implementierung nicht unterstützt wird, erfolgt eine Fehlermeldung und die Modellierung wird abgebrochen.

- *Strings* und *Arrays* können wie in Abschnitt 4.2.3 beschrieben implementiert werden. Wegen des hohen Aufwands bei der Implementierung und der gegenüber anderen Sprachelementen eher geringen Priorität wurden diese Sprachelemente nicht in die Implementierung aufgenommen. Darüber hinaus können Arrays und Strings durch dynamische Datenstrukturen (wie z.B. Listen) nachgebildet werden.

- Aufgrund des Aufbaues des Modells können keine *booleschen Ausdrücke mit Nebeneffekten* modelliert werden, da boolesche Ausdrücke nur soweit abgearbeitet werden, bis das Ergebnis des gesamten Ausdrucks eindeutig bestimmt ist. Im Modell werden die Ausdrücke hingegen immer komplett abgearbeitet, was Abweichungen gegenüber der Java-Semantik verursachen kann, falls die Ausdrücke Nebeneffekte aufweisen.

- Ebenfalls nicht unterstützt werden *mehrfache* `return`*-Anweisungen*. Da jede Methode mit mehreren `return`-Anweisungen in eine Methode mit einer einzelnen, abschließenden `return`-Anweisung umgewandelt werden kann, wurde auf eine Implementierung hier verzichtet. Auch würde die Qualität der Diagnosen durch die zusätzlichen Verbindungen stark herabgesetzt.

- `break`*- und* `continue`*-Anweisungen* werden vom der Implementierung zugrundeliegenden Modell nicht unterstützt und sind aus diesem Grund nicht implementiert. Da `switch`*-Anweisungen* in den meisten Fällen `break`-Anweisungen benötigen, wurde auf eine Implementierung derselben ebenfalls verzichtet.

- *Exceptions* sind durch das Modell ebenfalls nicht darzustellen. In der vorliegenden Implementierung können `throw`-Anweisungen nicht modelliert werden. Bei der Modellierung wird ein Fehler signalisiert. `try`- und `catch`-Anweisungen werden hingegen stillschweigend ignoriert, da diese ohne Auftreten von `throw`-Anweisungen ohnedies keine Effekte aufweisen.[8]

- *Inner classes* und *anonymous classes* werden ebenfalls nicht unterstützt. Eine Implementierung wäre zweifellos durch Modifikationen des Modells durchzuführen.

- Alle unter dem Begriff *Reflection* zusammengefaßten Eigenschaften der Sprache Java können ebenfalls nicht modelliert werden. Dies umfaßt sowohl das dynamische Laden von Klassen, als auch Methoden wie `.class()` etc.

- Der *Operator '*`instanceof`*'* ist in der vorliegenden Arbeit nicht implementiert.

- *Überläufe bei arithmetischen Operationen* sind nicht implementiert, da Programme diese relativ selten ausnützen. Darüber hinaus wäre die Leistungsfähigkeit des Constraint-Netzwerks stark beeinträchtigt, da dann bei diesen Komponenten kein Propagieren von den Output-Ports zu den Input-Ports möglich wäre.

- `synchronized`*-Anweisungen und Methoden* werden ebenfalls nur teilweise unterstützt. Da Threads vom Modell nicht unterstützt werden, kann das `synchronized`-Schlüsselwort weitgehend ignoriert werden, da ohnedies keine gleichzeitigen Zugriffe auf Methoden und Variablen stattfinden können. Aus diesem Grund werden `synchronized`-Anweisungen ignoriert und die zugehörigen Anweisungen in sequentieller Reihenfolge modelliert.

[8] Es wird angenommen, daß keines der Programmelemente `RunTimeException`s bedingt.

5.7 Benutzeroberfläche

Neben der in den bisherigen Abschnitten des Kapitels behandelten Implementierung des Modells und der Beobachtungen wurde eine bereits bestehende Benutzeroberfläche eines Java-Debuggers modifiziert, um das Debugging mit dem wertbasierten Modell zu ermöglichen.

Die Benutzeroberfläche stellt nur die wesentlichsten Funktionen bereit, welche zum Debugging von Methoden benötigt werden. Neben dem Laden und Parsen von Quelltexten ist das Bilden eines Modells für das Programm sowie das Spezifizieren von Argumenten für die aktuelle Methode möglich. Beim Debugging einer Methode sind zwei Vorgehensweisen möglich. (1) Einmalig Beobachtungen spezifizieren und anschließend alle berechneten Diagnosen betrachten. (2) Alternativ dazu kann das Verfahren solange iteriert werden, bis entweder eine eindeutige Diagnose als Fehlerursache identifiziert werden kann oder überhaupt keine Diagnose die Beobachtungen erklärt. Schließlich können noch einige Parameter des Modells und des Diagnosealgorithmus angegeben werden. Dies umfaßt die maximale Anzahl von Iterationen bei Schleifen, sowie die maximale Anzahl und Größe von Diagnosen. Die Benutzeroberfläche des Debuggers ist in Abbildung 5.10 dargestellt.

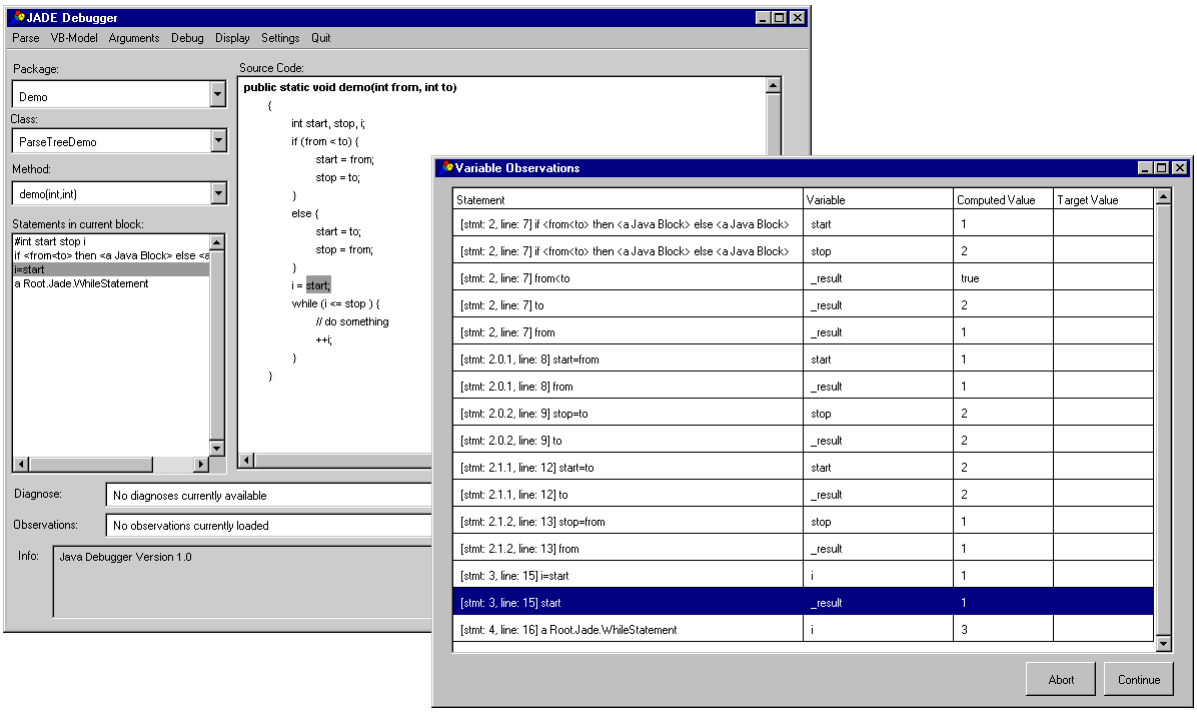

Abbildung 5.10: Benutzeroberfläche des Debuggers

Die Benutzeroberfläche ist in zwei Fenster unterteilt, wobei ein Fenster die Betrachtung des Programms, der berechneten Diagnose und der angegebenen Beobachtungen ermöglicht. Weiters enthält dieses Fenster eine Menüleiste, durch welche die Steuerung des Debuggers erfolgt. Das andere Fenster wird während des Debuggings zur Spezifikation von Beobachtungen verwendet. Es enthält eine Liste aller Verbindungen und Variablen des Modells, zusammen mit dem berechneten Wert. Um die Identifizierung der Variablen zu erleichtern, wird zusätzlich die Position des die Variable berechnenden Programmelements im Quelltext des Programms angezeigt.

Im nächsten Kapitel wird das Debugging mit dem Modell anhand einiger Beispiele betrachtet.

Kapitel 6

Debugging von Beispielen

Um die Anwendbarkeit des in dieser Arbeit vorgestellten Modells zur Diagnose von Java-Programmen zu zeigen, wird die Funktionsweise des Modells anhand einiger Beispiele demonstriert. Auch werden die durch das Modell berechneten Ergebnisse mit anderen Methoden zum Debugging von Programmen, wie etwa Slicing [Wei81, Wei82] oder mit auf funktionalen Abhängigkeiten beruhenden Modellen [MSW99] verglichen.

Wie in den vorangegangenen Kapiteln werden Komponenten des Modells durch die dadurch repräsentierten Programmelemente, zusammen mit der Zeilennummer ihres Auftretens im Programm, angegeben.

6.1 Vergleich mit weiteren Verfahren

Beispiel 6.1 Als Beispiel sei das in Abbildung 5.1 dargestellte Programm gegeben. Um die Methode `demo(int,int)` mit dem Modell analysieren zu können, sind zunächst die beiden Parameter `from` und `to` mit konkreten Werten zu initialisieren. Für das Beispiel seien dies die Werte `1` bzw. `2`. Weiters ist eine Beobachtung für die Variable `i` nach Beendigung der Abarbeitung der Methode gegeben. Der für die Variable erwartete Wert sei `2`. Nach Abarbeiten der Methode werden für die einzelnen Variablen der Methode folgende Werte erhalten:

Variable	berechnet	erwartet
`i`	3	2
`start`	1	1
`stop`	2	2

Es ist klar ersichtlich, daß der für die Variable `i` berechnete Wert im Widerspruch zu dem erwarteten Wert steht. Durch Einsatz von verschiedenen Verfahren und Modellen sollen mögliche Ursachen des Fehlverhaltens des Programms ermittelt werden. Im folgenden werden drei Verfahren vorgestellt und die damit erzielten Ergebnisse verglichen.

Zunächst wird das Programm mit dem in [MSW99] vorgestellten Modell analysiert. Dieses Modell berücksichtigt nur Variablen und deren Abhängigkeiten untereinander, ohne auf konkrete Werte Rücksicht zu nehmen. Aus dem Programm ist klar ersichtlich, daß die Variable `i` von den beiden Variablen `start` und `stop` abhängig ist. Da diese beiden Variablen korrekte Werte aufweisen und nach der Initialisierung von `i` in Zeile 15 nicht verändert werden, können die vorangehende Auswahlanweisung und deren Unter-Anweisungen als Fehlerursachen ausgeschlossen werden. Als mögliche Fehlerursachen verbleiben die Zuweisung an die Variable `i` in Zeile 15 und die `while`-Anweisung. Aufgrund des hohen Abstraktionsgrades des Modells können ohne zusätzliche Beobachtungen keine weiteren Aussagen über mögliche Fehlerursachen getroffen werden.

Ein weiteres Verfahren, welches bei der Identifizierung von Fehlerursachen eingesetzt werden kann, ist statisches Slicing [Wei81, Wei82]. Dieses Verfahren schränkt das gegebene Programm auf jene Anweisungen ein, welche zur Berechnung einer gegebenen Variable an einer bestimmten Programmposition relevant sind. In diesem Beispiel wird zunächst ein Slice für die Variable `i` in Zeile 20 bestimmt. Diese umfaßt die Anweisungen der Zeilen 7 bis 18, was der gesamten Methode entspricht. Die Menge der möglicherweise fehlerhaften Anweisungen kann durch Betrachten der verbleibenden Beobachtungen reduziert werden: Die Slices für die Variablen `start` und `stop` für Zeile 20 beinhalten die Anweisungen der Zeilen 7 bis 13. Da die beiden Variablen korrekte Werte aufweisen, kann angenommen werden, daß die Anweisungen dieser Slices keine Fehlerquellen darstellen. Folglich können diese Anweisungen aus dem Slice für die Variable `i` eliminiert werden. Es verbleiben die Anweisungen der Zeilen 15 bis 18 als mögliche Fehlerursachen. Das mit diesem Verfahren ermittelte Ergebnis entspricht jenem, welches durch Einsatz des abhängigkeitsbasierten Modells erhalten wurde.

Die Analyse der Methode mit dem in dieser Arbeit vorgestellten indirekten Modell gestaltet sich ähnlich wie beim zuvor behandelten, nur mit Abhängigkeiten arbeitenden Modell. Unter Benutzung identischer Werte für Parameter und Beobachtungen, werden durch das Modell folgende minimale Diagnosen ermittelt:

Diagnosen
$\{ab([\texttt{start}]_{15})\}$
$\{ab([\texttt{i} = \texttt{start}]_{15})\}$
$\{ab([\texttt{while}]_{16})\}$
$\{loop([\texttt{while}]_{16}, 1)\}$

Es werden also die Anweisungen und Ausdrücke der Zeilen 15 und 16 als mögliche Fehlerursachen identifiziert. Dies entspricht den mit den beiden vorhergehenden Verfahren erhaltenen Ergebnissen. Die Diagnose $\{loop([\texttt{while}]_{16}, 1)\}$ enthält jedoch mehr Information, als durch die anderen Modelle berechnet wurde. Die Diagnose drückt aus, daß, falls die Schleife anstatt zwei Iterationen nur eine durchläuft, das berechnete Ergebnis dem erwarteten Ergebnis entspricht. Da diese Diagnose spezifischere Informationen bereitstellt als die restlichen Diagnosen, sollte dieser Diagnose mehr Aufmerksamkeit geschenkt werden. Daraus folgt, daß die Diagnose $\{loop([\texttt{while}]_{16}, 1)\}$ als wahrscheinlichste Erklärung für das Fehlverhalten des Programms herangezogen wird. Dies erscheint sinnvoll, da spezifischere Diagnosen gegenüber allgemeineren und daher weniger aussagekräftigen Diagnosen bessere Hinweise zur Korrektur des Programms bieten.

6.2 Debugging mit Objekten

Das im soeben behandelten Beispiel analysierte Programm beschränkt sich auf die Verwendung von lokalen Variablen und Parameter-Variablen, welche im Modell als Variablen der Umgebung dargestellt werden. Weiters enthält das Programm keine relevanten objektorientierten Elemente. Um die Anwendbarkeit des Modells auf objektorientierte Programme zu zeigen, wird im folgenden Beispiel die Vorgehensweise bei Programmen mit objektorientierten Sprachelementen analysiert.

Beispiel 6.2 Zu analysieren sei das in Abbildung 4.11 dargestellte Programm. Nach Abarbeitung der Methode `demo()` sind die Variablen des Programms wie folgt belegt: Die Variable `p` verweist auf eine Instanz der Klasse `Pair`, welche zwei Instanzvariablen `value1` und `value2` enthält. Die Werte der beiden Variablen sind `1` resp. `3`. Weiters existieren zwei lokale Variablen `v1` und `v2`, welche die Werte `1` und `3` enthalten.

Im Gegensatz zum vorangegangenen Beispiel werden die Beobachtungen nicht allesamt zu Beginn des Debugging-Prozesses spezifiziert, sondern es wird in iterativer Weise versucht, das fehlerhafte Programmelement zu isolieren.

Zunächst sei eine Beobachtung für die Variable `v2` dem Modell der Methode hinzugefügt. Die Beobachtung enthalte den Wert `4`, welcher einen Widerspruch bedingt. Anschließend wird der Diagnosealgorithmus angestoßen, welcher die Diagnosen

Diagnosen
$\{ab([\texttt{new Pair(1,2)}]_{25})\}$
$\{ab([\texttt{p = new Pair(1,2)}]_{25})\}$
$\{ab([\texttt{3}]_{27})\}$
$\{ab([\texttt{p}]_{27})\}$
$\{ab([\texttt{p.setValue2(3)}]_{27})\}$
$\{ab([\texttt{p}]_{30})\}$
$\{ab([\texttt{p.getValue2()}]_{30})\}$
$\{ab([\texttt{v2 = p.getValue2()}]_{30})\}$

als mögliche Erklärungen berechnet. Als nächste zu beobachtende Variable berechnet der Algorithmus die Instanzvariable `value2` des erzeugten Objekts. Nach dem Hinzufügen einer Beobachtung, welche den Wert `4` fordert und erneutem Anstoßen des Diagnosealgorithmus verbleiben die Einfachfehlerdiagnosen

Diagnosen
$\{ab([\texttt{new Pair(1,2)}]_{25})\}$
$\{ab([\texttt{p = new Pair(1,2)}]_{25})\}$
$\{ab([\texttt{3}]_{27})\}$
$\{ab([\texttt{p}]_{27})\}$
$\{ab([\texttt{p.setValue2(3)}]_{27})\}$

Die Diagnosen, welche Fehler in der Anweisung `value2=p.getValue2()` repräsentieren, wurden durch die zusätzliche Beobachtung ausgeschlossen. Dies entspricht der erwarteten Vorgehensweise, da der Wert der Instanzvariablen `value2` bereits zuvor einen fehlerhaften Wert aufweist. Die nächsten spezifizierten Beobachtungen betreffen die Variable `p` jeweils in den Zeilen 25 und 27. Die Beobachtungen bestätigen die vom Programm berechneten Werte und eliminieren daher die Diagnosen $\{ab([\texttt{new Pair(1,2)}]_{25})\}$, $\{ab([\texttt{p = new Pair(1,2)}]_{25})\}$ und $\{ab([\texttt{p}]_{27})\}$. Es verbleiben zwei Diagnosen, welche auf einen Fehler im Aufruf der Methode `setValue2(int)` in Zeile 27 hinweisen:

Diagnosen
$\{ab([\texttt{3}]_{27})\}$
$\{ab([\texttt{p.setValue2(3)}]_{27})\}$

Bei genauerer Betrachtung (bzw. nach dem Hinzufügen einer zusätzlichen Beobachtung des Wertes `4` für die Output-Verbindung `result` der Komponente $[\texttt{3}]_{27}$), verbleibt als einzige mögliche Fehlerursache die Konstante `3`.

Obiges Beispiel veranschaulicht nicht nur die Anwendbarkeit des Modells auf Programme mit objektorientierten Sprachelementen, sondern demonstriert zugleich auch die Möglichkeit, die Analyse in iterativer Weise durchzuführen und vom Algorithmus Vorschläge unterbreiten zu lassen, welche Variablen als nächste zu beobachten sind, um mit möglichst wenigen Beobachtungen eine eindeutige Diagnose zu erhalten.

Beim Spezifizieren von Beobachtungen sind zwei Fälle zu unterscheiden (Details dazu sind in Abschnitt 5.5 näher beschrieben):

1. Die Beobachtung kann beim Spezifizieren vollständig angegeben werden. In diesem Fall kann der beobachtete Wert direkt in die Verbindung eingetragen werden.

2. Die Beobachtung benötigt zusätzliche, zum Zeitpunkt der Beobachtung nicht verfügbare, Informationen. Dieser Fall tritt dann ein, wenn Beobachtungen an Objekten vorgenommen werden, welche in Schleifen oder rekursiven Methodenaufrufen erzeugt werden. Hier kann der das Objekt repräsentierende Objekt-Identifier nicht angegeben werden, da dieser in verschiedenen Diagnosen unterschiedlich ausfallen kann. Stattdessen muß die Beobachtung in Form eines zusätzlichen Constraints spezifiziert werden.

Das folgende Beispiel zeigt ein durch Beobachtungs-Constraints verursachtes Problem bei der Berechnung von Diagnosen auf.

Beispiel 6.3 Dieses Beispiel demonstriert eine nachteilige Eigenschaft bei der Repräsentation von Beobachtungen durch Constraints. Die im weiteren behandelte Eigenschaft stellt die Tatsache dar, daß durch die Spezifikation von Beobachtungen durch Constraints nicht in jedem Fall eindeutige Diagnosen berechnet werden können. Zur Demonstration sei die Methode `demo()` des Programms in Abbildung 6.1 betrachtet. Die Methode erzeugt eine Instanz der Klasse `Inverter`

```
1  package Demo;
2
3  public class Inverter {
4      boolean value = false;
5
6      public Inverter() {
7      }
8
9      public void neg() {
10         value = !value;
11     }
12
13     public static void demo() {
14         Inverter i = null;
15         do {
16             i = new Inverter();
17         } while(false);
18
19         i.neg();
20         i.neg();
21         i.neg();
22     }
23 }
```

Abbildung 6.1: Programm ohne eindeutige Diagnose

und speichert eine Referenz auf das erzeugte Objekt in der lokalen Variablen `i`. Das erzeugte Objekt weist eine Instanzvariable `value` auf, welche mit dem Wert `false` initialisiert wird. Anschließend wird die Methode `neg`, welche den Wert von `value` negiert, einige Male auf das Objekt angewendet und auf diese Weise der Wert der Instanzvariablen auf `true` gesetzt. Zu beachten ist, daß die `do`-Anweisung in Zeile 15 benötigt wird, um zu erzwingen, daß Beobachtungen an der Variablen `i` oder an der Instanzvariablen des Objekts als Constraint repräsentiert werden. Andernfalls tritt der in diesem Beispiel demonstrierte Effekt nicht auf.

Als Beobachtungen seien für alle Verbindungen bzw. Variablen des Programms deren berechnete Werte spezifiziert, mit Ausnahme des letzten Wertes der Instanzvariablen `value` nach

Zeile 21. Für diese Verbindung sei der Wert `false` beobachtet, welcher einen Widerspruch zum berechneten Wert darstellt.

Da für alle Variablen und Verbindungen die berechneten Werte identisch mit den beobachteten Werten sind (mit Ausnahme der letzten Verbindung für `Inverter.value`), ist zu erwarten, daß der Diagnosealgorithmus die Einfachfehlerdiagnose $\{ab([\texttt{i.neg}]_{21})\}$ als eindeutige Fehlerursache berechnet. Die Erwartung einer einzelnen, das Fehlverhalten des Programms erklärenden Diagnose wird durch die Tatsache verstärkt, daß für *alle* Variablen und Verbindungen des Modells Beobachtungen spezifiziert sind.

Durch die Repräsentation von Beobachtungen als Constraints kann die Menge der Diagnosen nicht auf eine einzelne Diagnose reduziert werden. Die tatsächlich berechneten Diagnosen bilden eine Obermenge der erwarteten Diagnosen, nämlich

Diagnosen
$\{ab([\texttt{do}]_{15})\}$
$\{ab([\texttt{i.neg}]_{21})\}$

Diese Abweichung wird durch die Repräsentation von Beobachtungen als Constraints verursacht, da bei dieser Repräsentation die Beobachtung der Verbindung `Inverter.value` in Zeile 21 vom in Zeile 16 erzeugten Objekt-Identifier – und damit von der Output-Verbindung `i` der Komponente $[\texttt{do}]_{15}$ – abhängig ist. Um einen Konflikt im Constraint-Netzwerk nachzuweisen, sind daher immer *zwei* Komponenten (und das Beobachtungs-Constraint) notwendig: Dies sind die Komponenten $[\texttt{do}]_{15}$ und $[\texttt{i.neg}]_{21}$. Da der Diagnosealgorithmus auf Conflict-Sets aufbaut und in diesem Beispiel keine weiteren minimalen Conflict-Sets existieren, stellen diese beiden Komponenten stets Einfachfehlerdiagnosen dar. Die minimale Anzahl an Diagnosen erhöht sich somit auf zwei.

6.3 Qualitätseinbuße der Diagnosen bei Schleifen

Das Auftreten von Schleifen und rekursiven Methodenaufrufen in Programmen kann ebenfalls eine verminderte Qualität der berechneten Diagnosen verursachen. Das folgende Beispiel veranschaulicht dies anhand eines kurzen Programms.

Beispiel 6.4 Man betrachte das Programm im Abbildung 6.2. Die Funktionalität der Methode `demo(int,int)` des Programms entspricht jener des in Abbildung 3.2 dargestellten Schaltkreises. Aufgrund der Modellkonstruktion enthält das Modell der Methode eine größere Anzahl an Komponenten, da in diesem Modell auch Referenzen auf und Zuweisungen an lokale Variable als Komponenten repräsentiert werden.

Für dieses Beispiel seien die Werte `2` bzw. `3` an die Parameter `a` und `b` der Methode übergeben. Nach Abarbeitung der Methode ergeben sich für die Variablen folgende Werte:

Variable	berechnet	erwartet
`a`	2	2
`b`	3	3
`s0`	6	–
`s1`	6	–
`s2`	6	–
`x`	12	10
`y`	12	12

Wie der Tabelle zu entnehmen ist, wird für die Variable `x` ein gegenüber dem berechneten Wert abweichendes Ergebnis erwartet, der Wert der Variablen `y` wird hingegen bestätigt.

```
1  package Demo;
2
3  public class D74 {
4      static int add(int a, int b) {
5          while(b>0) {
6              ++a;
7              --b;
8          }
9          return a;
10     }
11
12     static int mult(int a, int b) {
13         return a * b;
14     }
15
16     public static void demo(int a, int b) {
17         int s0 = mult(a,b);
18         int s1 = mult(a,b);
19         int s2 = mult(a,b);
20         int x = add(s0,s1);
21         int y = add(s1,s2);
22     }
23 }
```

Abbildung 6.2: Programm des Schaltkreises aus Abbildung 3.2

Nach Hinzufügen der beiden Beobachtungen für `x` und `y` kann der Diagnoseprozeß gestartet werden. Als Ergebnis wird eine Menge von 12 Einfachfehlerdiagnosen berechnet:

Diagnosen
$\{ab([\texttt{a}]_{17})\}$
$\{ab([\texttt{b}]_{17})\}$
$\{ab([\texttt{mult(a, b)}]_{17})\}$
$\{ab([\texttt{s0} = \texttt{mult(a, b)}]_{17})\}$
$\{ab([\texttt{a}]_{18})\}$
$\{ab([\texttt{b}]_{18})\}$
$\{ab([\texttt{mult(a, b)}]_{18})\}$
$\{ab([\texttt{s1} = \texttt{mult(a, b)}]_{18})\}$
$\{ab([\texttt{s0}]_{20})\}$
$\{ab([\texttt{s1}]_{20})\}$
$\{ab([\texttt{add(s0, s1)}]_{20})\}$
$\{ab([\texttt{x} = \texttt{add(s0, s1)}]_{20})\}$

Bei genauerer Betrachtung fällt auf, daß gegenüber dem Beispiel 3.6 einige zusätzliche Diagnosen berechnet wurden (abgesehen von den Diagnosen, welche Variablenreferenzen oder Zuweisungen repräsentieren). Neben den erwarteten Diagnosen $\{ab([\texttt{mult(a, b)}]_{17})\}$ und $\{ab([\texttt{add(s0, s1)}]_{20})\}$ wird auch die Diagnose $\{ab([\texttt{mult(a, b)}]_{18})\}$ als mögliche Erklärung für das Fehlverhalten geliefert. Dies entspricht nicht dem erwarteten Ergebnis, da eine fehlerhafte Berechnung in Zeile 18, bei Korrektheit aller anderen Komponenten, ebenfalls ein fehlerhaftes Ergebnis für die Variable `y` erwarten läßt.

Bei genauerer Betrachtung des Diagnosevorgangs wird die Ursache für die Berechnung der zusätzlichen Diagnosen offensichtlich: Bei dem hier gewählten Beispiel ist es nicht ausreichend, die Werte entsprechend der Abarbeitungsreihenfolge der Anweisungen durch das Constraint-Netzwerk zu propagieren. Vielmehr hängt die Diagnose des Beispiels davon ab, daß die für die

Variablen `x` und `y` beobachteten Werte in entgegengesetzter Richtung zu den Input-Verbindungen der Komponenten $[\texttt{add(s1, s2)}]_{20}$ und $[\texttt{add(s1, s2)}]_{21}$ propagiert werden. Auf diese Weise können die Diagnosen $\{ab([\texttt{a}]_{18})\}$, $\{ab([\texttt{b}]_{18})\}$, $\{ab([\texttt{mult(a, b)}]_{18})\}$ und $\{ab([\texttt{s1 = mult(a, b)}]_{18})\}$ als Einfachfehlerdiagnosen ausgeschlossen werden, da durch Rückwärtspropagieren des für die Variable `x` beobachteten Wertes der Wert `4` für die Variable `s1` berechnet wird und dies im weiteren einen Widerspruch zu dem für `y` beobachteten Wert bedingt.

Die Implementierung der Methode `add(int,int)` des Programms in Abbildung 6.2 ist so gestaltet, daß kein Rückwärtspropagieren der Werte von den Output-Ports zu den Input-Ports des Modells möglich ist. Diese Eigenschaft wird durch die `while`-Anweisung verursacht, für welche im allgemeinen Fall kein Verfahren existiert, welches das Rückwärtspropagieren von Werten ermöglicht. Wird die Implementierung der Methode hingegen analog zu jener der Methode `mult(int,int)` gestaltet, d.h. etwa durch folgende Implementierung ersetzt,

```
1  static int add(int a, int b) {
2      return a + b;
3  }
```

wird die Beschränkung umgangen und die zusätzlichen Diagnosen eliminiert.

Zusammenfassend kann geschlossen werden, daß durch Auftreten von Schleifen oder rekursiven Methodenaufrufen in einem Programm aufgrund der Unmöglichkeit des Rückwärtspropagierens von Werten die Qualität der erhaltenen Diagnosen herabgesetzt werden kann. Dadurch können jedoch keine falschen Diagnosen verursacht werden. Durch Hinzufügen zusätzlicher Beobachtungen können die unerwünschten Diagnosen in jedem Fall ausgeschlossen werden.

6.4 Selektion von Beobachtungspunkten

Die Repräsentation von Instanzvariablen als Objekträume ermöglicht es, große Teile des Sprachumfangs von Java zu modellieren. Neben dem erhöhten Rechenaufwand bei der Diagnose werden auch weitere nachteilige Effekte verursacht. Insbesondere die Berechnung des besten nächsten Beobachtungspunktes kann bei Verwendung von Objekträumen beeinträchtigt werden.

Beispiel 6.5 Gegeben sei das Programm in Abbildung 6.3, wobei `demo()` die zu debuggende Methode sei. Der Ablauf des Programms gestaltet sich wie folgt: Zunächst wird die Klassenvariable `count` der Klasse `MeasurementSelection` initialisiert. Anschließend wird deren Wert in eine lokale Variable `countCopy` kopiert. Schließlich wird eine Methode `generate()` aufgerufen, welche eine Anzahl von Instanzen der Klasse `ValueHolder` erzeugt, welche jedoch nie benötigt werden. Die Anzahl der generierten Objekte wird ebenfalls durch die Klassenvariable `count` bestimmt.

Als Beobachtung sei der Wert `10` für die Variable `countCopy` in Zeile 19 angegeben. Dies steht klarerweise im Widerspruch zu dem im Initializer-Ausdruck der Variablen `count` angegebenen Konstanten `1`. Der Vollständigkeit halber sei die Menge der mit den bisherigen Information berechneten Diagnosen angegeben:

Diagnosen
$\{ab([\texttt{1}]_{11})\}$
$\{ab([\texttt{count = 1}]_{11})\}$
$\{ab([\texttt{count}]_{19})\}$
$\{ab([\texttt{countCopy = count}]_{19})\}$

Die Menge wird aber nicht weiter benötigt.

```
1  package Demo;
2
3  class ValueHolder {
4      int value;
5      ValueHolder(int value) {
6          this.value = value;
7      }
8  }
9
10 public class MeasurementSelection {
11     static int count = 1;
12     static void generate() {
13         for(int i = 0; i < count; i++) {
14             new ValueHolder(i);
15         }
16     }
17
18     public static void demo() {
19         int countCopy = count;
20         generate();
21     }
22 }
```

Abbildung 6.3: Programm mit verfälschter Selektion von Beobachtungspunkten

Um eine eindeutige Diagnose zu erhalten, müssen weitere Beobachtungen hinzugefügt werden, wobei deren Anzahl möglichst minimal sein soll. Bei der Auswahl des nächsten Beobachtungspunktes muß einige Sorgfalt angewendet werden. Betrachtet man Objekträume mit Variablen mehrerer Objekte als einen einzigen Wert, werden die Ergebnisse der Entropie-basierten Auswahlmethode schnell unbrauchbar, da diese Methode Verbindungen mit vielen Werten bevorzugt. Bei Vorhandensein von Objekträumen mit vielen enthaltenen Variablen kann schnell eine große Anzahl an unterschiedlichen Objekträumen für eine Verbindung entstehen.

Aus diesem Grund wird in der vorliegenden Implementierung jedes Objekt separat betrachtet. Dies liefert in vielen Fällen gute Ergebnisse, in einigen Fällen werden jedoch auch Verbindungen ausgewählt, welche keine Information zur Diagnose beitragen. Dies tritt auch in diesem Beispiel auf. Obwohl die Instanzen der Klasse `ValueHolder` erst nach dem initialen Beobachtungspunkt generiert werden, und diese darüber hinaus in keiner Weise benötigt werden, wird den einzelnen Instanzvariablen der erzeugten Objekte die gleiche Relevanz wie dem Wert der Variablen `count` nach Zeile 11 beigemessen. Dies kann zu einer großen Anzahl von Beobachtungen führen, welche von Benutzer angegeben werden müssen, aber keinen Einfluß auf die Diagnosekandidaten besitzen.

Anhand dieses Beispiels kann ein weiterer Effekt demonstriert werden: Es werden Instanzvariablen als Beobachtungspunkte vorgeschlagen, wobei die Instanzvariablen Teile von Objekten darstellen, welche im ursprünglichen Programm nicht erzeugt wurden. Die Spezifikation von korrekten Werten für solche Variablen ist für den Benutzer nur mit großem Aufwand möglich, da von den jeweiligen Diagnosen für viele Variablen gegenüber dem ursprünglichen Programm abweichende Werte vorhergesagt werden. Daher sind auch die Berechnungsgrundlagen für die Instanzvariablen möglicherweise verändert. Darüber hinaus können in manchen Programmen identische Objekt-Identifier in verschiedenen Diagnosen gänzlich verschiedene Objekte bezeichnen.

6.5 Strukturelle Fehler

Das in dieser Arbeit behandelte Modell ist vorwiegend dafür geeignet, funktionale Fehler in Programmen zu entdecken. Andere Fehlerarten, wie z.B. Synchronisationsfehler, falsche Algorithmen oder Fehler in der Struktur der Programme können nicht bestimmt werden.

Beispiel 6.6 Das Programm in Abbildung 6.4 implementiert die Addition eines positiven ganzzahligen Wertes zu einem weiteren ganzzahligen Wert. Dies erfolgt durch eine `while`-Anweisung,

```
1  package Demo;
2
3  public class StructuralFault {
4      public static int increment(int value, int amount) {
5          int result = value;
6          while(amount > 0) {
7              value++;          // this should be 'result++;'
8              amount--;
9          }
10         return result;
11     }
12 }
```

Abbildung 6.4: Programm mit strukturellem Fehler

welche das gewünschte Ergebnis durch inkrementieren des Ausgangswertes berechnet. Hier ist zu beachten, daß das Programm einen strukturellen Fehler in Zeile 7 enthält: Anstatt der Variablen `value` sollte die Variable `result` inkrementiert werden.

Für die Argumente der Methode seien die Werte `0` bzw. `5` angenommen. Dann ist der erwartete Wert für die Variable `result` in Zeile 10 gleich `5`, was einen Widerspruch zu den vom Programm berechneten Wert `0` darstellt. Nach dem Hinzufügen des erwarteten Wertes als Beobachtung und Anstoßen des Diagnoseprozesses ergeben sich die Einfachfehlerdiagnosen

Diagnosen
$\{[\texttt{value}]_5\}$
$\{[\texttt{result} = \texttt{value}]_5\}$
$\{[\texttt{result}]_{10}\}$

Es ist klar zu erkennen, daß die `while`-Anweisung, welche die fehlerhafte Anweisung enthält, nicht in der Menge der Diagnosen enthalten ist. Daraus folgt, daß das Modell zur Diagnose von Fehlern in der Struktur von Programmen nicht geeignet ist.

6.6 Geschwindigkeit

Ein weiterer kritischer Punkt ist die Geschwindigkeit des Systems. Insbesondere bei Schleifen und rekursiven Aufrufen, verbunden mit Auswahlanweisungen kann der Berechnungsaufwand für ein Programm schnell beträchtliche Ausmaße annehmen.

Beispiel 6.7 Das in Abbildung 6.5 dargestellte Programm weist einen – durch die Struktur der Modelle von Auswahlanweisungen bedingten – hohen Berechnungsaufwand auf. Dies wird durch die Tatsache hervorgerufen, daß die Anweisungen der beiden Zweige von Auswahlanweisungen ausgewertet werden, unabhängig von der Auswahlbedingung. Die dadurch verursachten Geschwindigkeitseinbußen können beträchtlich sein: Das Constraint-Netzwerk des in Abbildung 6.5

```
1  package Demo;
2
3  public class PerformanceSample {
4      public static void demo() {
5          if (false) {
6              for(int i=0; i<100; i++) {
7                  for(int j=0; j<100; j++) {
8                      // do something
9                  }
10             }
11         }
12         int value = 1;
13     }
14 }
```

Abbildung 6.5: Programm mit hohem Berechnungsaufwand beim Debugging

dargestellten Programms benötigt zur Abarbeitung durchschnittlich 61,3 Sekunden, während ein dazu äquivalentes Programm, welches anstatt der Auswahlanweisung eine `while`-Anweisung beinhaltet, nach 0,0004 Sekunden abgearbeitet ist.[1] Die beobachtete Steigerung der Geschwindigkeit wird erreicht, da bei `while`-Anweisungen der Schleifenrumpf nur dann abgearbeitet wird, falls die Schleifenbedingung erfüllt ist. Aus diesen Ergebnissen folgt, daß die vorliegende Implementierung für Programmkonstrukte mit hohem Berechnungsaufwand – wie verschachtelte Schleifen oder rekursive Methodenaufrufe – innerhalb von Auswahlanweisungen nur begrenzt geeignet ist.

Abschließend wird das Debugging mit dem Modell anhand eines „real-world“- Beispiels gezeigt.

Beispiel 6.8 Das betrachtete Programm ist in Abbildung 6.6 dargestellt. Die Methode `demo()` der Klasse `Rotate2d` erzeugt einen Vektor und eine Matrix und wendet die Matrix wiederholt auf den Vektor an. Die im Vektor gespeicherten Werte können als homogene Koordinaten eines Punktes im zweidimensionalen Raum angesehen werden. Die Matrix definiert eine Rotation um 90° um den Koordinatenursprung. Hier ist zu beachten, daß die im Programm angegebene Matrix einen Fehler enthält. Die korrekten Argumente beim Aufruf von `Matrix3x3(...)` in Zeile 46 wären: `0,-1,0,1,0,0,0,0,1`.

Nach Abarbeitung des gegebenen Programms ergeben sich für die Instanzvariablen der erzeugten Objekte folgende Werte:

Variable	berechnet	erwartet
`point.v0`	2	2
`point.v1`	3	3
`point.v2`	1	1
`operation.e00`	0	–
`operation.e01`	1	–
`operation.e02`	0	–
`operation.e10`	1	–
`operation.e11`	0	–
`operation.e12`	0	–

[1] Die Messung der Zeit erfolgte unter Windows NT 4.0 (Service Pack 5) auf einem Intel Pentium II mit 300MHz und 128Mb Speicher. Die verwendete Smalltalk-Version war Visual Works NC 5i.2.

Variable	berechnet	erwartet
operation.e20	0	–
operation.e21	0	–
operation.e22	1	–
rot90.v0	3	-3
rot90.v1	2	2
rot90.v2	1	1
rot180.v0	2	-2
rot180.v1	3	-3
rot180.v2	1	1
rot270.v0	3	3
rot270.v1	2	-2
rot270.v2	1	1
rot360.v0	2	2
rot360.v1	3	3
rot360.v2	1	1

Wie aus der Tabelle zu entnehmen ist, entsprechen die berechneten Werte für die Instanzvariablen der durch `rot90`, `rot180` und `rot270` angesprochenen Objekte nicht den erwarteten Werten. Nach Hinzufügen der Beobachtungen für die in der Tabelle angegebenen Variablen und Anstoßen des Diagnoseprozesses ergeben sich 16 Diagnosen als mögliche Fehlerursachen:

Diagnosen
$\{[\texttt{new Vector}(2,3,1)]_{45}\}$
$\{[\texttt{point} = \texttt{newVector}(2,3,1)]_{45}\}$
$\{[\texttt{new Matrix3x3}(\ldots)]_{46}\}$
$\{[\texttt{operation} = \texttt{new Matrix3x3}(\ldots)]_{46}\}$
$\{[\texttt{point}]_{48}\}$
$\{[\texttt{point.transform}(\texttt{operation})]_{48}\}$
$\{[\texttt{rot90} = \texttt{point.transform}(\texttt{operation})]_{48}\}$
$\{[\texttt{point}]_{49}\}$
$\{[\texttt{point.transform}(\texttt{operation})]_{49}\}$
$\{[\texttt{rot180} = \texttt{point.transform}(\texttt{operation})]_{49}\}$
$\{[\texttt{point}]_{50}\}$
$\{[\texttt{point.transform}(\texttt{operation})]_{50}\}$
$\{[\texttt{rot270} = \texttt{point.transform}(\texttt{operation})]_{50}\}$
$\{[\texttt{point}]_{51}\}$
$\{[\texttt{point.transform}(\texttt{operation})]_{51}\}$
$\{[\texttt{rot360} = \texttt{point.transform}(\texttt{operation})]_{51}\}$

Die große Anzahl an Diagnosen ist nicht weiter überraschend, da die Instanzvariablen aller Objekte in einer Verbindung zusammengefaßt sind. Es kann daher nicht unterschieden werden, welche Variablen von welchen Komponenten des Modells verändert werden.

Durch zusätzliches Wissen über die Implementierung der Methode `transform(Matrix3x3)` kann die Diagnose stark verbessert werden. Anstelle alle Beobachtungen nach Abarbeitung der gesamten Methode anzugeben, kann dies gleich nach dem Erzeugen der Objekte erfolgen, da die Instanzvariablen später nicht weiter verändert werden. Die Beobachtungen an den Instanzvariablen der Objekte, welche durch die Variablen `point`, `rot90`, `rot180`, `rot270` und `rot360` referenziert werden, können mit

```
1  package Demo;
2
3  class Vector3 {
4      int v0;
5      int v1;
6      int v2;
7
8      Vector3(int v0, int v1, int v2) {
9          this.v0 = v0;
10         this.v1 = v1;
11         this.v2 = v2;
12     }
13
14     Vector3 transform(Matrix3x3 m) {
15         int a0 = m.e00 * v0 + m.e01 * v1 + m.e02 * v2;
16         int a1 = m.e10 * v0 + m.e11 * v1 + m.e12 * v2;
17         int a2 = m.e20 * v0 + m.e21 * v1 + m.e22 * v2;
18         return new Vector3(a0, a1, a2);
19     }
20
21 }
22
23 class Matrix3x3 {
24     int e00,e01,e02,
25         e10,e11,e12,
26         e20,e21,e22;
27
28     Matrix3x3(int e00, int e01, int e02,
29             int e10, int e11, int e12,
30             int e20, int e21, int e22) {
31         this.e00 = e00;
32         this.e01 = e01;
33         this.e02 = e02;
34         this.e10 = e10;
35         this.e11 = e11;
36         this.e12 = e12;
37         this.e20 = e20;
38         this.e21 = e21;
39         this.e22 = e22;
40     }
41 }
42
43 public class Rotate2d {
44     public static void demo() {
45         Vector3 point = new Vector3(2,3,1);
46         Matrix3x3 operation = new Matrix3x3(0,1,0, 1,0,0, 0,0,1);
47
48         Vector3 rot90  = point.transform(operation);
49         Vector3 rot180 = rot90.transform(operation);
50         Vector3 rot270 = rot180.transform(operation);
51         Vector3 rot360 = rot270.transform(operation);
52     }
53 }
```

Abbildung 6.6: Programm zur Berechnung von Koordinaten

diesem Wissen zu den Output-Verbindungen der erzeugenden Komponenten vorverlegt werden. In diesem Beispiel sind dies die Komponenten $[\texttt{new Vector3}(2,3,1)]_{45}$, $[\texttt{new Matrix3x3}(\ldots)]_{46}$, $[\texttt{point.transform}(\texttt{operation})]_{48}$, $[\texttt{point.transform}(\texttt{operation})]_{49}$, $[\texttt{point.transform}(\texttt{operation})]_{50}$ und $[\texttt{point.transform}(\texttt{operation})]_{51}$. Damit reduziert sich die Menge der Diagnosen auf 7 Einfachfehlerdiagnosen:

Diagnosen
$\{ab([\texttt{new Vector3}(2,3,1)]_{45})\}$
$\{ab([\texttt{point} = \texttt{new Vector3}(2,3,1)]_{45})\}$
$\{ab([1]_{46})\}$
$\{ab([\texttt{new Matrix3x3}(\ldots)]_{46})\}$
$\{ab([\texttt{matrix} = \texttt{new Matrix3x3}(\ldots)]_{46})\}$
$\{ab([\texttt{point}]_{48})\}$
$\{ab([\texttt{point.transform}(\texttt{operation})]_{48})\}$

Unter der Annahme, daß das Programm keine falschen Variablenreferenzen und keine falschen Funktionsaufrufe enthält[2], kann die Diagnose

Diagnosen
$\{ab([1]_{46})\}$

als wahrscheinlichste Fehlerursache isoliert werden. Dieses Beispiel demonstriert, daß bei einer Sequenz von mit Objekträumen assoziierten Verbindungen und deren Komponenten die Beobachtungen möglichst nahe dem Beginn der Sequenz spezifiziert werden sollten, da auf diese Weise die Anzahl der Diagnosen stark herabgesetzt werden kann.

Abschließend bleibt die Geschwindigkeit des Modells zu untersuchen. Es wird die benötigte Rechenzeit beim Debugging der Methode `demo()` des vorliegenden Beispiels untersucht. Die Messungen wurden unter Windows NT 4.0 (Service Pack 5) auf einem Intel Pentium II mit 300MHz und 128Mb Speicher durchgeführt.

Das Modell der Methode `demo()` enthält 6 Anweisungen, welche in ein Modell mit 32 Komponenten übersetzt werden. Zur Bildung des Modells des gesamten Programms (inklusive der Klassen `Vector3` und `Matrix3x3`) werden durchschnittlich 1,21 Sekunden benötigt. Das Initialisieren des Modells der zu debuggenden Methode und das initiale Propagieren der Werte durch das Modell erfordert durchschnittlich 1,15 Sekunden.

Werden alle Beobachtungen nach Abarbeitung der Methode angegeben, werden die in folgender Tabelle dargestellten Werte erhalten:

Beschreibung	Laufzeit [s]
Berechnen der Diagnosen	22,89
Selektion eines Beobachtungspunktes	3,64

Werden die Beobachtungen sobald als möglich spezifiziert, ergibt sich folgendes Bild:

Beschreibung	Laufzeit [s]
Berechnen der Diagnosen	14,69
Selektion eines Beobachtungspunktes	1,25

Die erhaltenen Werte veranschaulichen, daß der Aufwand bei der Berechnung von Diagnosen in hohem Maße von der Struktur des Programms und den beobachteten Verbindungen abhängig ist. Die in diesem Beispiel beobachtete Beschleunigung der Berechnungen ergibt sich einerseits durch einen verminderten Aufwand beim Propagieren der Werte (Inkonsistenzen können aufgrund der bekannten Werte früher erkannt werden), andererseits aus der verminderten Anzahl von Diagnosen.

[2]Diese Annahme kann getroffen werden, da bei Ersetzen einer Variablen bzw. Methode durch eine andere jeweils Typ-Fehler auftreten würden. Eine Diagnose in diese Richtung scheint daher nicht sinnvoll.

Kapitel 7

Zusammenfassung und Ausblick

In der vorliegenden Arbeit werden zwei Modelle für Java-Programme vorgestellt, welche das Finden von funktionalen Fehlern in Programmen mittels modellbasierter Diagnose ermöglichen. Dabei wird insbesondere auf die objektorientierten Sprachelemente von Java Rücksicht genommen. Es ist anzumerken, daß bei der Modellierung der Programme nicht der gesamte Sprachumfang von Java unterstützt wird. In den Modellen können Exceptions, Threads, `break`-, `continue`- und `switch`-Anweisungen, Reflection, sowie boolesche Ausdrücke mit Nebeneffekten nicht dargestellt werden.

Der Aufbau des zu Beginn behandelten direkten Modells erfolgt mittels Komponenten, welche die im Programm verwendeten Sprachkonstrukte nachbilden. Der Datenfluß zwischen den einzelnen Programmelementen wird durch Verbindungen zwischen den Komponenten dargestellt, wobei eine Verbindung einer benötigten oder veränderten lokalen-, Instanz- oder Klassenvariablen entspricht.

Weiters werden Einschränkungen des Modells aufgezeigt: Aufgrund der Modellstruktur können rekursive Methodenaufrufe nicht modelliert werden. Auch beim Erzeugen oder Manipulieren von Objekten mit Instanzvariablen kann in einigen Fällen kein Modell gebildet werden. Schließlich werden durch Aliasing zwischen Variablen Probleme hervorgerufen. Als eine Lösung für die hier genannten Probleme wird eine Erweiterung des Modells von Variablen auf Referenzpfade in Verbindung mit k-Limiting vorgestellt. Als Abhilfe für das Problem von Aliasing zwischen Variablen bzw. Referenzpfaden ist der Einsatz einer Aliasing-Analyse notwendig.

Aufgrund der Beschränkungen und der hohen Komplexität der Erweiterungen des oben genannten direkten Modells wird ein weiteres Modell (das indirekte Modell) entwickelt, welches die Probleme des ursprünglichen Modells weitgehend vermeidet. Dieses Modell basiert ebenfalls auf einer komponentenorientierten Repräsentation des Programms. Im Gegensatz zum ursprünglichen Modell werden nur lokale Variablen und Klassenvariablen als Verbindungen repräsentiert. Alle Instanzvariablen werden durch eine einzelne Verbindung dargestellt, welche den Zustand aller Objekte (den sog. Objektraum) repräsentiert. Der Zugriff auf die Objekte und deren Instanzvariablen erfolgt durch eindeutige Identifier, welche beim Erzeugen von Objekten generiert werden. Schließlich werden zwei Erweiterungen des Modells vorgestellt, welche die Modellierung von Arrays und Strings bzw. eine verbesserte Qualität der Diagnosen ermöglichen.

Dieses Modell vermeidet die Einschränkungen des ursprünglichen Modells, das Generieren von Objekt-Identifiern während der Abarbeitung des Programms kann bei Beobachtungen an Instanzvariablen jedoch einige Probleme verursachen. Als Abhilfe wird ein Verfahren vorgestellt, welches Beobachtungen an Objekt-Identifiern und Instanzvariablen durch zusätzliche Komponenten repräsentiert.

Abschließend wird die Leistungsfähigkeit und die Grenzen des indirekten Modells anhand einiger Beispiele demonstriert. Es wird gezeigt, daß mit Hilfe des Modells eine große Anzahl

von Fehlern gefunden werden kann. Für Programme, welche Fehler in der Struktur aufweisen (z.B. fehlende Anweisungen oder Zuweisungen an falsche Variablen), ist das Modell jedoch nicht geeignet. Weiters muß die Abarbeitungsreihenfolge der Anweisungen des Programms bereits während der Modellbildung bekannt sein. Daraus ergibt sich, daß Programme mit mehreren Threads nicht untersucht werden können. Auch Exceptions und Sprünge können nicht behandelt werden.

Im weiteren werden einige offene Forschungsbereiche und mögliche Erweiterungen des Modells näher betrachtet:

- Die Verwendung von *Gleitkommazahlen* in Programmen kann bei der Diagnose unerwünschte Effekte verursachen, da aufgrund der beschränkten Genauigkeit der Darstellung bei der Berechnung der Diagnosen Rundungsfehler o.ä. auftreten können. Daraus ergibt sich, daß Werte, welche eigentlich äquivalent sein sollten, während des Diagnoseprozesses als unterschiedlich angesehen werden. Dies hat zur Folge, daß falsche Diagnosen berechnet werden. Als Abhilfe wäre etwa das Einführen von Toleranzgrenzen bei Vergleichen von Gleitkommazahlen möglich.

- Basierend auf einer Einfachfehlerdiagnose für eine *hierarchische Komponente* kann versucht werden, eine Diagnose für die durch die Komponente repräsentierten Programmelemente zu berechnen. Die beobachteten Werte für die inneren Modelle ergeben sich dabei aus den für die Input- und Output-Ports der hierarchischen Komponente berechneten Werten. So könnten z.B. im Fall einer `while`-Schleife Diagnosen für die Modelle der Schleifenbedingung und des Schleifenrumpfs ermittelt werden. Details dazu finden sich in [MSW00].

- Die Suche nach *Replacements* stellt eine weitere Möglichkeit dar, genauere Hinweise auf eventuelle Fehlerursachen zu erhalten. Diese Methode beruht darauf, nicht nur möglicherweise fehlerhafte Programmelemente zu identifizieren, sondern es werden auch mögliche Korrekturvorschläge berechnet. So könnten etwa Korrekturvorschläge für den Ausdruck `a + b` wie folgt aussehen: `a - b`, `1 + b`, `a + a`, etc. Eine genauere Abhandlung dieses Verfahrens ist in [SW99] nachzulesen.

- Durch Anwendung von *mehreren Testfällen* kann die Anzahl der berechneten Diagnosen herabgesetzt werden, da die Diagnosen nicht nur das Fehlverhalten gegenüber einem Testfall erklären müssen, sondern auch die Werte der übrigen Testfälle. Auf diese Weise wird das Ausfiltern jener Diagnosen erreicht, welche keine zulässige Erklärung für die Werte aller Testfälle bieten. Details zur Vorgehensweise dieses Verfahrens sind in [SW99] behandelt.

- Die *Kombination des Modells mit weiteren Modellen* stellt eine sinnvolle Erweiterung dar, um auch größere Programme mit dem Modell effizient analysieren zu können. Da diese wertbasierten Modelle nur für relativ kleine Programme sinnvoll anwendbar sind, muß zunächst durch andere Techniken versucht werden, die Fehlerursachen möglichst einzugrenzen. Die genauere Analyse mit einem wertbasierten Modell erfolgt dann auf dem eingeschränkten Programmbereich. Die Einschränkung des Programms auf einen relevanten Teilbereich kann etwa durch auf Abhängigkeiten basierenden Verfahren wie Slicing [Wei81] oder durch das in [MSW99] behandelte Modell erfolgen.

- Auch die Entwicklung von *weiteren Fehlermodellen*, welche unterschiedliche Fehlerarten in Programmen modellieren, ist ein offener Forschungsbereich. So könnte z.B. ein zusätzliches Fehlermodell für Auswahlanweisungen implementiert werden, welches den Fall einer negierten Auswahlbedingung darstellt. Auf diese Weise könnten bei der Diagnose detailliertere Hinweise auf die Art des Fehlers – und damit auf dessen Beseitigung – gegeben werden.

- Eine *intuitive Benutzeroberfläche* stellt einen essentiellen Teil eines modellbasierten Debuggers dar, da es dem Benutzer auf einfache Weise ermöglicht werden muß, das korrekte Verhalten des Programms in Form von Beobachtungen zu spezifizieren. Die Entwicklung einer solchen Benutzerschnittstelle stellt daher einen wichtigen Schritt in Richtung eines modellbasierten Debuggers dar. Hier ist insbesondere auf Beobachtungen an Objekten und deren Instanzvariablen zu achten, da diese in verschiedenen Diagnosen in unterschiedlicher Anzahl auftreten können und daher das Spezifizieren von Beobachtungen erschweren.

Zusammenfassend kann geschlossen werden, daß durch den Einsatz von modellbasierter Diagnose zum Debugging von Java-Programmen durchaus vielversprechende Ergebnisse erzielt werden können.

Literaturverzeichnis

[BC92] Michael Burke and Jong-Deok Choi. Precise and efficient integration of interprocedural alias information into data-flow analysis. *ACM Letters on Programming Languages and Systems*, 1(1):14–21, March 1992.

[BH95] Lisa Burnell and Eric Horwitz. Structure and Chance: Melding Logic and Probability for Software Debugging. *Communications of the ACM*, 38:31–57, 1995.

[Deu94] A. Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting. In *SIGPLAN'94 Conf. on Programming Language Design and Implementation*, pages 230–241, Orlando (Florida, USA), June 1994. ACM. SIGPLAN Notices, 29(6).

[dKW87] J. de Kleer and B. C. Williams. Diagnosing Multiple Faults. *Artificial Intelligence*, 32:97–130, 1987.

[dKW89] J. de Kleer and B. C. Williams. Diagnosis with Behavioral Modes. In *Proc. of the 11th IJCAI*, pages 1324–1330, Detroit, MI, 1989.

[Duc93] Mireille Ducassé. A pragmatic survey of automated debugging. In *Automated and Algorithmic Debugging. First International Workshop, AADEBUG'93, Linköping, Sweden*, May 1993.

[Ghi98] Rakesh Ghiya. *Putting Pointer Analysis to Work*. PhD thesis, School of Computer Science, McGill University, Montreál, Québec, Canada, 1998.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[GSW89] 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, November 1989.

[Hor97] Susan Horwitz. Precise Flow-Insensitive May-Alias Analysis is NP-Hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.

[Jac95] Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *IEEE Transactions on Software Engineering*, 4(2):109–145, 1995.

[Lan92] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[MSW99] Christinel Mateis, Markus Stumptner, and Franz Wotawa. Debugging of Java Programs using a Model-based Approach. In *International Workshop on Diagnosis (DX-99), Loch Awe, Scotland*, 1999.

[MSW00] Christinel Mateis, Markus Stumptner, and Franz Wotawa. A Value-Based Diagnosis Model for Java Programs. In *Eleventh International Workshop on Principles of Diagnosis (DX), Morelia, Mexico*, 2000.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, 1992.

[Paw96] Thomas Pawlin. Implementierung eines C-Übersetzers und Anwendung der modellbasierten Diagnose zum Software-Debugging von C-Programmen, 1996.

[Poo89] David Poole. Normality and Faults in Logic-Based Diagnosis. In N. S. Sridharan, editor, *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 1304–1310, Detroit, MI, USA, August 1989. Morgan Kaufmann.

[Rei87] R. Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32:57–95, 1987.

[SBL96] Steve Simkin, Neil Bartlett, and Alex Leslie. *The Java Programming Explorer*. Coriolis Group Books, Scottsdale, AZ, USA, March 1996.

[SD89] Peter Struss and Oscar Dressler. Physical Negation – Integrating Fault Models into the General Diagnostic Engine. In *International Joint Conference on Artificial Intelligence IJCAÍ89*, pages 1218–1323, 1989.

[Str92] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, Reading, MA, USA, 2^{nd} edition, 1992.

[SW99] Markus Stumptner and Franz Wotawa. Debugging Functional Programs. In *International Joint Conference on AI (IJCAI-99), Stockholm*, 1999.

[Ung91] Roland Unger. Ein Modellbasiertes Diagnosesystem – Erweiterung und Implementierung des Ansatzes nach Reiter, 1991.

[Wei81] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, March 1981.

[Wei82] Mark Weiser. Programmers Use Slicing When Debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[Wot96] Franz Wotawa. *Applying Model-Based Diagnosis to Software Debugging of Concurrent and Sequential Imperative Programming Languages*. PhD thesis, Technische Universität Wien, 1996.

[Wot97] Franz Wotawa. Dike: Diagnosekern – Verwendung und Implementierung. Technical report, Technische Universität Wien, Institut für Informationssysteme, September 1997.