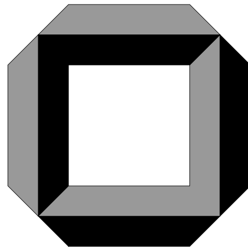
Proof Obligations for Correctness of Modifies Clauses

Ralf Sasse

October 23, 2004

Studienarbeit



Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme

Verantwortlicher Betreuer: Prof. Dr. Peter H. Schmitt
Betreuer: Dr. Bernhard Beckert

Danksagung

An dieser Stelle möchte ich mich bei meinen Betreuern Prof. Peter H. Schmitt und Dr. Bernhard Beckert bedanken, die mich bei der Erstellung dieser Studienarbeit sehr unterstützt haben.
Bedanken möchte ich mich auch bei Andreas Roth, der immer gerne bereit war, meine Fragen zu beantworten und mir einen besseren Einblick in das KeY-Projekt ermöglichte.

Hiermit versichere ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Ralf Sasse
Karlsruhe, October 23, 2004

Contents

1 Introduction **1**
1.1 KeY Project . . . 1
1.2 Goal of this work . . . 1
1.3 Structure . . . 1

2 Dynamic Logic DL_J **3**
2.1 Syntax . . . 3
2.2 Semantics . . . 4

3 Modifies Clauses in DL_J **5**
3.1 Modifies Clauses in general . . . 5
3.2 Modifies Clauses formally . . . 5
3.3 Special Cases . . . 7

4 Correctness of a Method with respect to Modifies Clauses **8**

5 Proof Obligations for Modifies Clauses **10**
5.1 Proof Obligation Creation . . . 10
5.2 Proof Obligation Correctness . . . 13
5.3 Change in the Proof Obligation for better machine usability . . . 21
5.4 Variation of the Formula Build-up . . . 23

6 Related Work **24**
6.1 Spoto and Poll . . . 24
6.2 Catano and Huisman . . . 24
6.3 General . . . 24

7 Implementation **26**

8 Summary **27**
8.1 Future Work . . . 27

1 Introduction

1.1 KeY Project

This Studienarbeit was done within the scope of the KeY project [ABBG+00]. The KeY project aims at integrating formal methods into the industrial software engineering process. A commercial CASE tool is used as starting point and tools for formal specification and verification are integrated into it. The final goal behind this is that programmers who have little or no experience in the use of formal methods can also profit from the advantages of formal methods. Therefore it is necessary to conceal the process of verification from the user as far as possible, i.e. to automate the verification.
Right now there is a working prototype of the KeY tool which uses TogetherCC from Borland as the CASE tool. The KeY tool aims at doing verification for object-oriented software programmed in JAVA. A first release will happen in the very near future.

1.2 Goal of this work

The logic which is used for verification in the KeY-project is the JAVA CARD Dynamic Logic [Beck01]. This logic is based on dynamic logic introduced by [Hare84]. A "modifies clause" is a list of program locations given with a method specification and it states which variables or class attributes may have been changed after the execution of the method. All variables or class attributes which are not part of the modifies clause have to be the same after the method execution as before for the method to be correct with respect to the modifies clause.
The goal of this "Studienarbeit" was to add the possibility to verify that a given modifies clause for a method is indeed a correct one using dynamic logic. This is important because modifies clauses allow speed-ups in the interactive theorem proving as can be seen in the paper [BeSc03] by Beckert and Schmitt. Also in the "Studienarbeit" of Bastian Katz [Katz03] we need the assumption that the modifies clause is correct and then using the method given by that work, i.e. replacing method calls by a rule, generated from the method specification with modifies clause, is correct as detailed there.
My approach uses the availability of the KeY tool to do the actual proof which happens mainly automated. Thus only a proof obligation has to be created and can then be discharged by the system.

1.3 Structure

Chapter 2 gives a short overview over the dynamic logic which is used for the rest of this work. Chapter 3 defines modifies clauses in the environment given

by Chapter 2. In Chapter 4 the "correctness" of a modifies clause is defined. Chapter 5 is the main part of this work where the proof obligation which has to be created to show the correctness as defined in Chapter 4 is developed and the correctness of the approach is shown. Then Chapter 6 points out what has been done by other approaches and what problems they encountered and why I could overcome them here. Then in Chapter 7 a few details and problems of the implementation are discussed. Chapter 8 finishes the work and summarizes what has been done and what would be nice to add.

2 Dynamic Logic DL_J

The logic which is used in all works related to the KeY tool is first order dynamic predicate logic for JAVA CARD, called DL_J. It is mainly defined in [Beck01] and the main points of its syntax and semantics will be presented here.

2.1 Syntax

Context and Signature Syntax and semantics is always defined with respect to the *context* which includes the class definitions of a program. From the context we can derive large parts of the *signature*, like for example functions which represent the fields from the class definition.

Variables In DL_J there are two sorts of variables: on the one hand there are program variables and on the other hand logical variables. Only logical variables are variables in the original meaning but they are not allowed to appear free in DL_J and they cannot be assigned to in programs. Logical variables will be presented as x, y, z. Program variables are only variables in the sense of the JAVA programming language. In the logic they are considered to be non-rigid 0-ary functions. They can be changed by the program like fields of objects (unary functions) and may self-evidently not be quantified. They are presented as **x, y, z**.

Functions Attributes and array elements are considered functions exactly like the program variables. For an attribute *attr* of an object which is referenced by a term t we also write $t.attr$ instead of $attr(t)$. Functions can be rigid, meaning they cannot be changed by the execution of the program. All functions defined by the JAVA semantics (like e.g. addition $+$) shall be treated like rigid functions in DL_J.

Terms The terms of DL_J are built over the logic variables and functions as usual. These include program variables as non-rigid 0-ary functions and constants as rigid 0-ary functions.

Programs All executable JAVA CARD programs are allowed as programs except the class declarations.

Formulas Formulas for a given signature consist of terms, predicates and logic junctors as usual. They are extended to dynamic logic by the definition that for every formula ϕ and for every program p the following are formulas too: $\langle \mathtt{p} \rangle \phi$ and $[\mathtt{p}]\, \phi$. With the usual semantics for dynamic logics holding true here.

Example 2.1. With formulas ϕ and ψ and a program p

$$\phi \quad \rightarrow \quad [\mathtt{p}]\,\psi$$

is an example of a Hoare Triple in JAVA CARD Dynamic Logic.

Sequents A *sequent* has the form $l_1, ..., l_n \vdash r_1, ..., r_m$ with formulas l_i and r_i. Its semantics is the same as that of $l_1 \wedge ... \wedge l_n \rightarrow r_1 \vee ... \vee r_m$.

2.2 Semantics

The semantics by which the formulas of DL_J are interpreted are Kripke structures $\mathcal{K} = (\mathcal{S}, \rho)$ where $\mathcal{S}$ is a set of states and ρ is a transition relation for the interpretation of programs.
The states $s \in \mathcal{S}$ are predicate logic structures $\mathcal{M}$ over the corresponding signature Σ. We limit the choice of $\mathcal{S}$ by requesting that

1. all states have the same universe. Therefore we speak of *the* universe of a DL_J-Kripke-structure.

2. $\mathcal{S}$ includes all structures of the predicate logic for the given signature and the universe of $\mathcal{K}$.

These two requirements lead to a uniquely defined set $\mathcal{S}$ for every choice of a signature and a universe.
For the transition relation depending on the program the following holds in general: for the set $\mathcal{P}$ of all valid JAVA CARD programs the transition relation ρ is:
$\rho : \mathcal{P} \rightarrow 2^{\mathcal{S} \times \mathcal{S}}$. JAVA CARD programs are deterministic and therefore for $p \in \mathcal{P}$ $\rho(p)$ is a partial function $\rho(p) : \mathcal{S} \rightarrow \mathcal{S}$. We consider concrete JAVA CARD programs and limit the choice of ρ like this: when interpreting the states of $\mathcal{S}$ as program states or states of the *Java Virtual Machine* $\rho(p)(s)$ is the state which is reached by execution of the program p in state s (or if p does not terminate then s does not have an image under $\rho(p)$). By that the choice of ρ is uniquely determined under the precondition that the semantics of JAVA CARD is unique.

3 Modifies Clauses in DL_J

This section is close to [Katz03]'s explanations on modifies clauses with only minor additions as needed for this work.

3.1 Modifies Clauses in general

A modifies clause is basically a list of arguments or fields from a given program or method call or global memory locations. In our dynamic logic DL_J we write the modifies clauses down the same way as we write down preconditions and postconditions of methods. Such a modifies clause belongs to a method specification. In our implementation that is given in the comments preceding the method using a "@modifies ..." line where the "..." are the actual elements of the modifies clause which are given comma separated.
It is implicit that all variables and attributes which are not listed in the modifies clause may not be changed by the execution of the method. Only those which are part of the modifies clause may be altered but they do not have to be changed.

3.2 Modifies Clauses formally

Now to make all this more precise we formally define:

Definition 3.1. [Modifies clauses of sets]
Let $\mathcal{K} = (\mathcal{S}, \rho)$ be a DL_J-Kripke-structure for a given signature Σ. A set M of ground terms, i.e. terms without logical variables, is called *modifies clause* for a pair of states $(s_1, s_2) \in \mathcal{S} \times \mathcal{S}$ if and only if for all functions $f \in \Sigma$ with arity n_f and all n_f-tupel $o_1, ..., o_{n_f}$ the following holds:

$$f^{s_1}(o_1, ..., o_{n_f}) \neq f^{s_2}(o_1, ..., o_{n_f})$$

only if there is a $t \in M$ of the form $t = f(t_1, ..., t_{n_f})$ [1] with $t_i^{s_1} = o_i$ for $(1 \leq i \leq n_f)$. Then we write

$$(s_1, s_2) \models M.$$

With this definition you have to note that the parameters t_i are evaluated in the prestate s_1. A term $t = f(t_1, ..., t_n)$ in a modifies clause does not state that maybe $t^{s_1} \neq t^{s_2}$ but it states this:

$$f^{s_1}(t_1^{s_1}, ..., t_n^{s_1}) \neq f^{s_2}(t_1^{s_1}, ..., t_n^{s_1})$$

This may not look intuitive but one has to note that it is not possible to provide all ground terms which might change especially as one can't know which are going

[1] We will write *t.attr* in addition to *attr*(t) for attributes of objects, and also $t^{s'}.attr^s$ for $attr^s(t^{s'})$, to stay with the usual way of writing for JAVA.

to be evaluated to be the same objects. The determination that the prestate is relevant may look arbitrary but it has clear advantages: Starting with the next definition we will consider state pairs which are in a relation $\rho(p)$. There will be references for object attributes which change during the execution of the program [2]. On the other hand it is not unusual that an object first gets changed in its' attributes but then all references to it are overwritten. Even though we will not be able to describe programs in the sense of this definition in all cases.
A case which is exceedingly bad is the declaration of local variables in a program p without limit on the scope of p, i.e. programs of the form $p \equiv$ ...int x; ... in contrast to $p \equiv$ {... int x; ...} (both are correct JAVA CARD programs). From now on we will only consider programs which do not have local variable declarations with unlimited scope.

Definition 3.2. [Modifies clauses of programs]
Let $\mathcal{K} = (\mathcal{S}, \rho)$ be a DL_J-Kripke-structure to a given signature Σ. A set M of ground terms is called *modifies clause* of a program p if

$$(s_1, s_2) \models M$$

for all $(s_1, s_2) \in \rho(p)$.

Therefore the modifies clause of a program p lists which program variables and object attributes may maximally be changed by the execution of p. It does not require them to change. Most useful are modifies clauses which are as small as possible for a program but on the other hand there might be programs which would require a modifies clause of infinite size. We are not calculating modifies clauses for given programs. We only check whether the given modifies clauses are correct, so we do not face the problem of an infinite modifies clause as we only have a finite input. [3]

Example 3.3. Let **C** be a class with the two nonstatic attributes **att1**, **att2** of type integer. With **a**, **b** program variables of type **C** and **p** $\equiv$ {**a** = **b**; **a.att1** = **4**;}. Then {**a**, **b.att1**} is a modifies clause of **p**.

Notice that in this example {**a**, **a.att1**} is not a modifies clause of **p** because if we assume **p** is started in state s where $\mathbf{a}^s \neq \mathbf{b}^s$ then **a** does not point to the object whose attribute **att1** changes.

[2] Attributes of objects created during p are a special case but DL_J's semantics allows us a way to formulate that too.
[3] see following chapter about correctness

3.3 Special Cases

Arrays require a little extra thought, as they do not have attributes, accessed with the "." operator, but elements, indexed by x with the "$[x]$" operator. It is thus necessary to make sure one checks all elements. It would be nice to have a construct with which one would indicate that all array elements may be changed. See what happens if one would confuse the array being changeable with all elements being allowed to change:

Example 3.4. Let **C** be a class with an attribute **arr** which is an array of int of size two. With **m()** $\equiv$ **arr**[**0**] = **1;**, we consider {**this.arr**[**0**]} as the correct modifies clause and {**this.arr**} is not a correct modifies clause because **this.arr**[**0**] is missing because giving the array itself in a modifies clause means the array object can change and not that all elements may change.

4 Correctness of a Method with respect to Modifies Clauses

A method call, with respect to a modifies clause, is correct if everything that is changed after the method call is part of the modifies clause. On the other hand it is not necessary that everything in the modifies clause really gets changed. In the case that something is not changed, it is not needed to be part of the modifies clause but if it is part of the modifies clause that does not affect whether the method call is correct w.r.t. the modifies clause or not. The modifies clause is not minimal then but that is not a point of concern for this work. For checking the "correctness of a modifies clause", meaning the correctness of a program or method w.r.t. a modifies clause, having a larger modifies clause does not make it any harder, it just generates a small extra term in the proof obligation formula. But for using the modifies clause it is a different thing as for example with the method in [Katz03] a larger modifies clause creates a much larger lemma which has to be proven later.

Example 4.1. Let **C** be a class with two nonstatic attributes **att1, att2** of type integer and a void method **m()**, with **m()** $\equiv$ **{this.att1 = 5;}**. Then **{this.att1}** is a correct modifies clause of **m()** as well as **{this.att1, this.att2}**.

Adding to a correct modifies clause certainly keeps the modifies clause correct but it slightly complicates the proof obligation. It is best to have a modifies clause that is as narrow as possible.

Definition 4.2. [Correct modifies clause of a method]
A method is correct w.r.t. a modifies clause, in a given framework of classes in which the method is invoked, iff everything which is changed after the execution of the method is a member of the modifies clause for all start states.

Let $\mathcal{K} = (\mathcal{S}, \rho)$ be a DL_J-Kripke-structure to a given signature Σ. A set M of ground terms is a "correct *modifies clause* of a method **m(...)**", meaning **m(...)** is correct w.r.t. the modifies clause, if

$$(s_1, s_2) \models M$$

for all $(s_1, s_2) \in \rho(m(...))$.

The correctness is non-modular, meaning it depends on the given framework whether or not the modifies clause of a method is correct. This is not surprising as the pre-/post-condition part of a specification is also non-modular. Inside that framework though the correctness of a method w.r.t. a modifies clause is

independent of the actual starting state of the method as all initial states are considered.

Example 4.3. Let **C** be a class with two nonstatic attributes **att1, att2** of type integer and a void method **m()**. With **m()** $\equiv$ **{this.att1 = 5;}** as in Example 4.1. If **this.att1** $= 5$ holds in the initial state one could consider the empty clause as a modifies clause but that is not a correct modifies clause as defined above because the modifies clause has to be independent of the start state and in any start state with **this.att1** $\neq 5$ the empty clause is not a correct modifies clause.

Why is the correctness non-modular and correctness depends on the given framework? That is easy to see with an example which is a little larger:

Example 4.4. Let **C** be a class with two nonstatic attributes **att1, att2** of type integer and a void method **m()**. Let **D** be a class with two static attributes **att3, att4** of type integer and a static void method **n()**. With **n()** $\equiv$ **{ att3 = 5; }** and **m()** $\equiv$ **{ D.n() }**. Now the modifies clause of **m()** is { **D.att3** } but it obviously depends on **n()** and so with another framework in which **n()** is changed the modifies clause would have to be different.

5 Proof Obligations for Modifies Clauses

5.1 Proof Obligation Creation

Verifying modifies clauses is no easy task and therefore it will be of advantage to make use of KeY's proving system to the extent possible. Therefore we need a way to create a proof obligation for modifies clauses as input to the KeY theorem prover.
I will use the short-hand notation $x \neq y$ instead of $!(x = y)$ from DL_J in the rest of this work for easier readability.
For a given method $m(...)$ and modifies set M we find all types, i.e. classes, appearing therein and being reachable from any of those types (also transitive over multiple types) and call those $class_1$, ..., $class_n$ where $classindex = \{1, ..., n\}$ is the index set. The type of the class which contains method $m(...)$ belongs to that list of types.
Now for each $class_i$, $i \in \{1, ..., n\}$, where $class_i$ is of non-array type, find all attributes of the class and enumerate them as $class_i.attribute_1$, ..., $class_i.attribute_{n_i}$, with the index set called $ind(i) = \{1, ..., n_i\}$ in all cases. For an array type no attributes need to be found as all elements, $class_i[\lambda]$, are indexed by an integer λ.
We give a formula in DL_J which states that only elements of the modifies clause may have been changed if the formula is valid. This formula will be created by smaller sub-formulas which are given first:

Definition 5.1. For a non-static $attribute_a$ of $class_i$:

- o' is a variable of type $class_i$
- o_{iak} is a term which is defined by
  - o_{iak} is of type $class_i$
  - $o_{iak}.attribute_a \in M$
  - $o_{iak}.attribute_a$ is the k-th appearance of a term of type $class_i$ with $o_{iak}.attribute_a \in M$
- x has the type of $class_i.attribute_a$

Then:

$$\varphi_{ia} =$$

$$[o' \neq o_{ia1} \wedge o' \neq o_{ia2} \wedge ... \wedge o' \neq o_{ian_{(ia)}}$$

$$\rightarrow \forall x(x = o'.attribute_a \rightarrow [\mathtt{m(...)}]\, o'.attribute_a = x)]$$

As we require that the complete formula F (which this is a small part of, see below for F) is valid for a modifies clause to be correct and all the connectives are conjunctions this partial formula has to hold. It says for the given i and a that, whenever the interpretation of o' is different from all objects e where the type of e is $class_i$ and, with l a term which is interpreted as e, $l.attribute_a$ is part of the modifies clause, the part after the implication has to hold true, otherwise the formula is true anyway. The part after the implication states that whatever value x the attribute $o'.attribute_a$ takes it has to be the same after the execution of $m(...)$.

Definition 5.2. Now for a static $attribute_a$ of $class_i$:

- The formula $\{class_i.attribute_a \notin M\}$ is a syntactic construct, it either evaluates to true or false and that evaluation is put into the actual formula.
- x has the type of $class_i.attribute_a$

Then:

$$\varphi_{ia} =$$
$$[\{class_i.attribute_a \notin M\}$$
$$\rightarrow \forall x(x = class_i.attribute_a \rightarrow [\texttt{m(...)}]\, class_i.attribute_a = x)]$$

If the formula $\{class_i.attribute_a \notin M\}$ evaluates to true the right-hand side of the implication has to hold because this element is not in the modifies clause, if it is false, i.e. it is in the modifies clause, nothing has to be shown as the left-hand side of the implication is wrong and thus the whole formula is true.

In the following I will use the construct $\forall_{0\leq\lambda\leq n}\lambda : \Phi$ instead of $\forall\lambda : 0 \leq \lambda \wedge \lambda \leq n \rightarrow \Phi$ for better readability.

Definition 5.3. Now for a $class_i$ which is of some array type:

- let $length(i)$ be the size of the array $class_i$
- thus $length(i) - 1$ indexes the last element of that array
- o' is a variable of type $class_i$
- o_{ik} and z_{ik} are defined by:
  - o_{ik} is of type $class_i$
  - $o_{ik}[\beta] \in M$ for some β, it is the k-th appearance of an array-dereferenced term of type $class_i$ in M

  - $z_{ik} = \beta$ for $o_{ik}[\beta] \in M$

- x has the type of $class_i[\beta]$
- λ is an integer

Then:

$$\begin{aligned}\varphi_i = \forall_{0 \leq \lambda \leq (length(i)-1)} \lambda \\ [((\lambda = z_{i1}) \rightarrow o' \neq o_{i1}) \\ \wedge((\lambda = z_{i2}) \rightarrow o' \neq o_{i2}) \\ \wedge \ldots \\ \wedge((\lambda = z_{in_i}) \rightarrow o' \neq o_{in_i}) \\ \rightarrow \forall x (x = o'[\lambda] \rightarrow [\mathtt{m(\ldots)}]\, o'[\lambda] = x)]\end{aligned}$$

There is no need here to separate static and non-static cases as an array element can never be static on the one hand. On the other hand if an instance of $class_i$ happens to be static (because it is itself a static attribute of some other class for example) that does not matter as in the formula there will be an appropriate $o' \neq o_{ij}$ and on the logic side a static element which is referenced via a dynamic object will be translated to its static reference and thus will match the o_{ij}. For example having a class $MyClass$ with a static attribute arr and it being called somewhere as $o.arr$ where o is some term of type $MyClass$ the $o.arr$ gets translated to $MyClass.arr$ and thus we need no distinction between static and non-static in the array case.

Definition 5.4. With $ind(i)$ being the index set of attributes for a non-array $class_i$ we define for each non-array $class_i$ the formula ψ_i:

$$\psi_i = \bigwedge_{a \in ind(i)} \varphi_{ia}$$

This is the formula for all attributes of a class $class_i$.

Definition 5.5. For an array $class_i$ the formula psi_i is defined by:

$$\psi_i = \varphi_i$$

This suffices as there is no explicit attribute listing, but all elements are already handled in the different build-up of φ_i for arrays.

Definition 5.6. Finally the whole formula F, with $classindex$ being the set of class indices, is defined by:

$$F = \bigwedge_{i \in classindex} \forall o' : class_i \;\; \psi_i$$

where o' is a variable. This means that for all classes $class_i$ the formula ψ_i has to hold for all objects $o' : class_i$.

5.2 Proof Obligation Correctness

Now my main theorem is:

Theorem 5.7. *For a given method m(...) and modifies set M, the method is correct w.r.t. M iff the formula F, with the notation as introduced above, is valid:*

$$F = \wedge_{i \in classindex} \forall o' : class_i \;\; \psi_i$$

Proof:
"$\Rightarrow$": We know that the modifies clause is correct and will use a proof by contradiction.
Let's assume that the formula is not valid, then there is a state for which it is false. From now on we only concentrate on this state and thus we can assume that the formula is false. By the construction of the formula that means that at least one ψ_i is false which in turn means at least one φ_{ia} is false if $class_i$ is a non-array type or φ_i is false if it is an array type. That is because the formula is built up by the conjunction of all the φ_{ia}'s iterated over the classes and attributes and the φ_i's iterated over the array types and for the formula to be false it is enough if one of these is false.
Without loss of generality this happens for $class_1$. If $class_1$ is a non-array type then it happens for $attribute_1$ (simply reorder the numbering accordingly). If $class_1$ is an array type we have to inspect the whole φ_1 formula.
Now with case distinction we have to look at the possible cases: $class_1$ being a non-array type then $class_1.attribute_1$ could be a non-static attribute or a static attribute or $class_1$ is an array type.

Case 1: non-static attribute of a non-array type.

$$\forall o' : class_1[o' \neq o_{111} \wedge o' \neq o_{112} \wedge ... \wedge o' \neq o_{11n(11)}$$

$$\rightarrow \forall x(x = o'.attribute_1 \rightarrow [\texttt{m(...)}]\, o'.attribute_1 = x)]$$

is false. Now there has to be at least one term of type $class_1$ for which this is false, let's call this term *obj*. Putting *obj* into the formula we get:

$$[obj \neq o_{111} \wedge obj \neq o_{112} \wedge ... \wedge obj \neq o_{11n(11)}$$

$$\rightarrow \forall x(x = obj.attribute_1 \rightarrow [\texttt{m(...)}]\, obj.attribute_1 = x)]$$

is false. From this we can conclude that $obj.attribute_1$ is not part of the modifies clause as otherwise the antecedent of the implication would be false (because if it would be part of M then a term $obj \neq obj$ would be in the antecedent yielding false) and with that the formula true which it is not by assumption. With this

we have that the antecedent is true and can look at the right-hand side of the implication:

$$\forall x(x = obj.attribute_1 \rightarrow [\texttt{m(...)}]\, obj.attribute_1 = x)$$

is false. There certainly is a x with the value of $obj.attribute_1$, call it val, then for this we have the following (while for all other values of x the implication is true as the left-hand side is false):

$$(val = obj.attribute_1 \rightarrow [\texttt{m(...)}]\, obj.attribute_1 = val)$$

which has to be false. In case this is false $obj.attribute_1$ would have to be in the modifies clause M which it is not as we have seen and this means the modifies clause is wrong in contradiction to our precondition of the modifies clause being correct. That means that the assumption is wrong, which was that the formula is wrong, and thus the formula is true.

Case 2: static attribute of a non-array type.

$$\varphi_{ia} = [\{class_1.attribute_1 \notin M\}$$

$$\rightarrow \forall x(x = class_1.attribute_1 \rightarrow [\texttt{m(...)}]\, class_1.attribute_1 = x)]$$

is false. For that $class_1.attribute_1$ must not be part of M as otherwise the antecedent is wrong and the implication is true. That gives us:

$$\forall x(x = class_1.attribute_1 \rightarrow [\texttt{m(...)}]\, class_1.attribute_1 = x)$$

is false which means it is false for some x, where the x has to have the value of $class_1.attribute_1$ which we call val.

$$(val = class_1.attribute_1 \rightarrow [\texttt{m(...)}]\, class_1.attribute_1 = val)$$

is false. That is only possible if $class_1.attribute_1$ may change, i.e. is in the modifies clause M, which it is not as seen above and so we have a contradiction and that means the assumption was wrong and thus the formula is true.

Case 3: array type.

$$\forall o' : class_1$$

$$\big[\forall_{0 \leq \lambda \leq (length(1)-1)} \lambda$$

$$[((\lambda = z_{11}) \rightarrow o' \neq o_{11})$$

$$\wedge((\lambda = z_{12}) \rightarrow o' \neq o_{12})$$

$$\wedge \ldots$$

$$\wedge((\lambda = z_{1n_1}) \rightarrow o' \neq o_{1n_1})$$

$$\rightarrow \forall x(x = o'[\lambda] \rightarrow [\mathtt{m(\ldots)}]\, o'[\lambda] = x)]]$$

is false. For this to be false there only has to be one index α for which the formula doesn't hold:

$$\forall o' : class_1$$

$$[[((\alpha = z_{11}) \rightarrow o' \neq o_{11})$$

$$\wedge((\alpha = z_{12}) \rightarrow o' \neq o_{12})$$

$$\wedge \ldots$$

$$\wedge((\alpha = z_{1n_1}) \rightarrow o' \neq o_{1n_1})$$

$$\rightarrow \forall x(x = o'[\alpha] \rightarrow [\mathtt{m(\ldots)}]\, o'[\alpha] = x)]]$$

is false. WLOG α equals the z_{1y} from z_{11} through z_{1k} (simply reorder them). Using that we can boil the formula down to:

$$\forall o' : class_1$$

$$[o' \neq o_{11} \wedge o' \neq o_{12} \wedge \ldots \wedge o' \neq o_{1k}$$

$$\rightarrow \forall x(x = o'[\alpha] \rightarrow [\mathtt{m(\ldots)}]\, o'[\alpha] = x)]$$

is false. Now there has to be at least one term of type $class_1$ for which this is false, let's call this term obj. Putting obj into the formula we get:

$$[obj \neq o_{11} \wedge obj \neq o_{12} \wedge \ldots \wedge obj \neq o_{1k})$$

$$\rightarrow \forall x(x = obj[\alpha] \rightarrow [\mathtt{m(\ldots)}]\, obj[\alpha] = x)]$$

which is false. From this we can conclude that $obj[\alpha]$ is not part of the modifies clause (and also for no term t with $t = obj$ we have $t[\alpha] \in M$) as otherwise the antecedent of the implication would be false (because if it would be part of M then a term $obj \neq obj$ would be in the antecedent yielding false (or a term $t = obj$ would be there yielding false), especially if there were no terms on the left-hand side of the outer implication it could not be in the modifies clause!) and with that the formula true which it is not by assumption. With this we have that the antecedent is true and can look at the right-hand side of the implication:

$$\forall x(x = obj[\alpha] \rightarrow [\mathtt{m(\ldots)}]\, obj[\alpha] = x)$$

is false. There certainly is a x with the value of $obj[\alpha]$, call it val, then for this we have the following (while for all other values of x the implication is true as the left-hand side is false):

$$(val = obj[\alpha] \rightarrow [\mathtt{m(...)}]\, obj[\alpha] = val)$$

which has to be false. In case this is false $obj[\alpha]$ would have to be in the modifies clause M which it is not as we have seen and this means the modifies clause is wrong in contradiction to our precondition of the modifies clause being correct. That means that the assumption is wrong, which was that the formula is wrong, and thus the formula is true.

As all 3 cases are proven this direction of the proof is complete.

"$\Leftarrow$": We know that the formula is valid.
Assume that the modifies clause is not correct, i.e. there is something that got changed and is not in M for some initial state on which we work from now on. WLOG that happens for $class_1$. If $class_1$ is not of an array type it happens for $attribute_1$. So we have one of the following cases:
case 1: for an object obj of non-array type $class_1$ for which $attribute_1$ is a non-static attribute $obj.attribute_1$ got changed and is not in M.
case 2: $class_1.attribute_1$ is a static attribute of a non-array type which got changed and which is not in M.
case 3: for an object obj of array type $class_1$ for an α the array element $obj[\alpha]$ is changed and is not in M.

Now let's take a closer look at each of the cases from above:
Case 1: $obj.attribute_1$ has been changed but is not a member of M means that the following formula is false:

$$\forall x(x = obj.attribute_1 \rightarrow [\mathtt{m(...)}]\, obj.attribute_1 = x)$$

That is false because there is at least one value of x such that the left-hand side holds and from above it is required that the attribute gets changed by the method $m(...)$ and so it cannot be the same after the method execution.
Now the following subformula is part of the large formula for which we know it is true by precondition and as it is a conjunctive part this has to be true too:

$$\forall o' : class_1(o' \neq o_{111} \wedge o' \neq o_{112} \wedge ... \wedge o' \neq o_{11n(11)}$$

$$\rightarrow \forall x(x = o'.attribute_1 \rightarrow [\mathtt{m(...)}]\, o'.attribute_1 = x)$$

As this holds for all objects and terms of type $class_1$ it has to hold for the special case of obj where the formula looks like this:

$$(obj \neq o_{111} \wedge obj \neq o_{112} \wedge ... \wedge obj \neq o_{11n(11)}$$

$$\rightarrow \forall x(x = obj.attribute_1 \rightarrow [\texttt{m(...)}]\, obj.attribute_1 = x)$$

By our assumption $obj.attribute_1 \notin M$ and therefore there is no o_{11j} such that $o_{11j} = obj$ and therefore the conjunction in the left-hand side of the implication holds true and the right-hand side has be true too to make the whole formula true which gives us:

$$\forall x(x = obj.attribute_1 \rightarrow [\texttt{m(...)}]\, obj.attribute_1 = x)$$

is true. That is in direct contradiction to the fact that we have seen this same formula to be false above and by that contradiction the assumption is false which means that the modifies clause M is indeed correct.

Case 2: $class_1.attribute_1$ has been changed but is not a member of M means that the following formula is false:

$$\forall x(x = class_1.attribute_1 \rightarrow [\texttt{m(...)}]\, class_1.attribute_1 = x)$$

That is false because there is at least one value of x such that the left-hand side holds and from above it is required that the attribute gets changed by the method $m(...)$ and so it cannot be the same after the method execution.
Now the following subformula is part of the large formula for which we know it is true by precondition and as it is a conjunctive part this has to be true too:

$$[\{class_1.attribute_1 \notin M\}$$

$$\rightarrow \forall x(x = class_1.attribute_1 \rightarrow [\texttt{m(...)}]\, class_1.attribute_1 = x)]$$

By our assumption $class_1.attribute_1 \notin M$ so the left-hand side of the implication is true and as the whole implication has to be true the right-hand side needs to be true, too, which is:

$$\forall x(x = class_1.attribute_1 \rightarrow [\texttt{m(...)}]\, class_1.attribute_1 = x)$$

needs to be true. That is a contradiction to above where we found out that this formula is false. This allows us to conclude that the assumption was wrong and thus the modifies clause M is indeed correct.

Case 3: In this case for an α the array element $obj[\alpha]$ has changed and is not in M. That means the following formula is false:

$$\begin{gathered}[((\alpha = z_{11}) \rightarrow obj \neq o_{11})\\ \wedge((\alpha = z_{12}) \rightarrow obj \neq o_{12})\\ \wedge\ldots\\ \wedge((\alpha = z_{1n_1}) \rightarrow obj \neq o_{1n_1})\\ \rightarrow \forall x(x = obj[\alpha] \rightarrow [\mathtt{m(\ldots)}]\, obj[\alpha] = x)]\end{gathered}$$

That is false because the left-hand side of the outer implication is true as $obj[\alpha] \notin M$ means there is no term o_{1i} with $obj = o_{1i}$ and therefore all the inequalities hold. As these are the right-hand sides of the upper inner implications those all hold true and their conjunction is true. For the right-hand side of the outer implication there is at least one value of x such that the left-hand side of the lower inner implication holds and from above it is required that the element gets changed by the method $m(...)$ and so it cannot be the same after the method execution and thus the right-hand side of the lower inner implication is false which makes the lower inner implication false which in turn makes the outer implication and by that the formula false.
Now the following subformula is part of the large formula for which we know that it is true by precondition and as it is a conjunctive part this has to be true too:

$$\begin{gathered}\forall_{0 \leq \lambda \leq (length(1)-1)} \lambda\\ [((\lambda = z_{11}) \rightarrow obj \neq o_{11})\\ \wedge((\lambda = z_{12}) \rightarrow obj \neq o_{12})\\ \wedge\ldots\\ \wedge((\lambda = z_{1n_1}) \rightarrow obj \neq o_{1n_1})\\ \rightarrow \forall x(x = obj[\lambda] \rightarrow [\mathtt{m(\ldots)}]\, obj[\lambda] = x)]\end{gathered}$$

One part of this is, with the same index α as above:

$$\begin{gathered}[((\alpha = z_{11}) \rightarrow obj \neq o_{11})\\ \wedge((\alpha = z_{12}) \rightarrow obj \neq o_{12})\\ \wedge\ldots\\ \wedge((\alpha = z_{1n_1}) \rightarrow obj \neq o_{1n_1})\\ \rightarrow \forall x(x = obj[\alpha] \rightarrow [\mathtt{m(\ldots)}]\, obj[\alpha] = x)]\end{gathered}$$

which has to hold true in contradiction to the first formula in this case which is the same and is false. Therefore the assumption was wrong and the modifies clause M is thus correct.

As all 3 cases are proven this direction of the proof is complete too.

That concludes the proof of the main theorem. □

Writing the formula in detail as a whole gives a better feeling of what it looks like. I have done this here in the simple special case with only non-static non-array attributes:

$$\forall o' : class_1$$

$$($$

$$[o' \neq o_{111} \land o' \neq o_{112} \land ... \land o' \neq o_{11n_{(11)}}$$

$$\rightarrow \forall x(x = o'.attribute_1 \rightarrow [\mathtt{m(...)}]\, o'.attribute_1 = x)]$$

$$\land[o' \neq o_{121} \land o' \neq o_{122} \land ... \land o' \neq o_{12n_{(12)}}$$

$$\rightarrow \forall x(x = o'.attribute_2 \rightarrow [\mathtt{m(...)}]\, o'.attribute_2 = x)]$$

$$\land ...$$

$$\land[o' \neq o_{1n_1 1} \land o' \neq o_{1n_1 2} \land ... \land o' \neq o_{1n_1 n_{(1n_1)}}$$

$$\rightarrow \forall x(x = o'.attribute_{n_1} \rightarrow [\mathtt{m(...)}]\, o'.attribute_{n_1} = x)]$$

$$)$$

$$\land \forall o' : class_2$$

$$($$

$$[o' \neq o_{211} \land o' \neq o_{212} \land ... \land o' \neq o_{21n_{(21)}}$$

$$\rightarrow \forall x(x = o'.attribute_1 \rightarrow [\mathtt{m(...)}]\, o'.attribute_1 = x)]$$

$$\land[o' \neq o_{221} \land o' \neq o_{222} \land ... \land o' \neq o_{22n_{(22)}}$$

$$\rightarrow \forall x(x = o'.attribute_2 \rightarrow [\mathtt{m(...)}]\, o'.attribute_2 = x)]$$

$$\land ...$$

$$\land[o' \neq o_{2n_2 1} \land o' \neq o_{2n_2 2} \land ... \land o' \neq o_{2n_2 n_{(2n_2)}}$$

$$\rightarrow \forall x(x = o'.attribute_{n_2} \rightarrow [\texttt{m(...)}]\, o'.attribute_{n_2} = x)]$$

$$)$$

$$\wedge ...$$

$$\wedge \forall o' : class_n(...)$$

Above using the requirements given in the detailed build-up of the formula is necessary for it to be meaningful.

Example 5.8. Let **C** be a class with two nonstatic attributes **att1, att2** of type integer and a void method **m()**. With $\mathbf{m()} \equiv \mathbf{\{this.att1 = 5\}}$. Then $\mathbf{\{this.att1\}}$ is a modifies clause of **m()** as we have seen in example 4.1. The proof obligation generated by the above rule would be:

$$\forall o' : C$$

$$($$

$$[o' \neq this$$

$$\rightarrow \forall x(x = o'.att1 \ \rightarrow [\texttt{m(...)}]\, o'.att1 = x)]$$

$$\wedge [true$$

$$\rightarrow \forall x(x = o'.att2 \ \rightarrow [\texttt{m(...)}]\, o'.att2 = x)]$$

$$)$$

Looking at this formula we see that if the above formula can be proven valid all objects of type C will have the following property: if m() is called either on the object O itself or another object of type C the attribute $O.att2$ will be the same because of the second implication which guarantees that the value of $O.att2$ stays the same after the method execution. Now the first implication guarantees that $O.att1$ doesn't change unless $O \equiv this$, that is the method is called on O and then $O.att1$ may change as it is part of the modifies clause.
Obviously it is very easy to proof this formula true meaning the given modifies clause is correct which was already obvious from looking at the modifies clause and the method specification.

5.3 Change in the Proof Obligation for better machine usability

For easier handling in the KeY prover this formula is in part replaced by an equivalent one after the following rule from [TBSc01] :

$$\forall x(x = o'.attribute_i \rightarrow [\texttt{m(...)}]\, o'.attribute_i = x)$$

is equivalent to

$$\exists x(x = o'.attribute_i \wedge [\texttt{m(...)}]\, o'.attribute_i = x)$$

This only works because there can not be multiple different x which make $x = o'.attribute_i$ true on the one hand and there certainly is one x which makes that true on the other hand.
With that replacement in place the smallest part of the build-up as described above looks like this in the non-static attribute case:

$$\varphi_{ia} = [o' \neq o_{ia1} \wedge o' \neq o_{ia2} \wedge ... \wedge o' \neq o_{ian_{(ia)}}$$
$$\rightarrow \exists x(x = o'.attribute_a \wedge [\texttt{m(...)}]\, o'.attribute_a = x)]$$

with the other definitions of φ_{ia} and φ_i changed accordingly and the rest of the formulas is not changed (at least not in their short-hand writing but if these parts are substituted into them, which happens repeatedly, then the changes take effect there too obviously).

Overall the whole formula now looks like this (again only for the non-static attribute case):

$$\forall o' : class_1$$
$$($$
$$[o' \neq o_{111} \wedge o' \neq o_{112} \wedge ... \wedge o' \neq o_{11n_{(11)}}$$
$$\rightarrow \exists x(x = o'.attribute_1 \wedge [\texttt{m(...)}]\, o'.attribute_1 = x)]$$
$$\wedge[o' \neq o_{121} \wedge o' \neq o_{122} \wedge ... \wedge o' \neq o_{12n_{(12)}}$$
$$\rightarrow \exists x(x = o'.attribute_2 \wedge [\texttt{m(...)}]\, o'.attribute_2 = x)]$$
$$\wedge ...$$

$$\wedge[o' \neq o_{1n_11} \wedge o' \neq o_{1n_12} \wedge ... \wedge o' \neq o_{1n_1n_{(1n_1)}}$$
$$\rightarrow \exists x(x = o'.attribute_{n_1} \wedge [\texttt{m(...)}]\, o'.attribute_{n_1} = x)]$$

$$)$$

$$\land \forall o' : class_2(...)$$

$$\land ...$$

$$\land \forall o' : class_n(...)$$

with the same o_{ijk} as described for the first formula.

Example 5.9. This is the changed formula for better machine usability for Example 5.8. Now the proof obligation is:

$$\forall o' : C$$

$$($$

$$[o' \neq this$$

$$\rightarrow \exists x(x = o'.att1 \ \land [\texttt{m(...)}]\, o'.att1 = x)]$$

$$\land[true$$

$$\rightarrow \exists x(x = o'.att2 \ \land [\texttt{m(...)}]\, o'.att2 = x)]$$

$$)$$

Example 5.10. Let's take a look at a somewhat larger example: we have three classes A, B and C. Class A has three attributes, two integer attributes a and b and an attribute of type C called c. Class B has two integer attributes p and q and a static integer attribute r. Class C has no attributes. Class A has a void method $m()$ which takes no arguments with $m() \equiv \mathbf{a} = \mathbf{a} + \mathbf{1}; \mathbf{B}.\mathbf{r} = \mathbf{5}$; and the modifies set $M = \{this.a, this.b, B.r\}$. Then the proof obligation is:

$$\forall o' : A$$

$$($$

$$[o' \neq this \rightarrow \exists x(x = o'.a \ \land [\texttt{m(...)}]\, o'.a = x)]$$

$$\land[o' \neq this \rightarrow \exists x(x = o'.b \ \land [\texttt{m(...)}]\, o'.b = x)]$$

$$\land[true \rightarrow \exists x(x = o'.c \ \land [\texttt{m(...)}]\, o'.c = x)]$$

$$)$$

$$
\begin{gathered}
\wedge \forall o' : B \\
(\\
[true \rightarrow \exists x(x = o'.p \ \wedge [\mathtt{m(\ldots)}]\, o'.p = x)] \\
\wedge [true \rightarrow \exists x(x = o'.q \ \wedge [\mathtt{m(\ldots)}]\, o'.q = x)] \\
\wedge [false \rightarrow \exists x(x = B.r \ \wedge [\mathtt{m(\ldots)}]\, B.r = x)] \\
) \\
\wedge \forall o' : C \ \ true
\end{gathered}
$$

So in general when the formula F is valid the modifies clause given is correct. If the modifies clause is correct on the other hand the formula is valid. Thus for an incorrect modifies clause the formula can't be proven valid and from the remaining part of it in the KeY system one can usually see where problems, i.e. missing elements, are.

5.4 Variation of the Formula Build-up

A different way to construct the formula F looks also interesting. It is informally given by example below with a short discussion on its advantages and disadvantages. This refers to Example 5.9 and this is the alternate formula whose form is defined by:

$$
\begin{gathered}
\forall o' : C \\
\exists x_1 \exists x_2 (x_1 = o'.att_1 \wedge x_2 = o'.att_2 \wedge \\
[\mathtt{m(\ldots)}]\, [(o' \neq this \rightarrow o'.att1 = x_1) \wedge (true \rightarrow o'.att_2 = x_2)])
\end{gathered}
$$

This construction has the advantage that the method $m(...)$ has to be expanded only once which could speed up the proof process in KeY but in case there is a loop in $m(...)$ finding a loop-invariant which allows proving all the requirements could get much harder than being able to unwind the loop separately for each requirement. This could be a starting point for future work on this topic.

6 Related Work

6.1 Spoto and Poll

The term *modifies clause*, as we use it here, is also known as "assignable clause" in JML notation. There is a paper by Spoto and Poll [SpPo03] about static analysis for the assignable clauses. The algorithm checks JML assignable clauses but they have only given a pseudo-algorithmic definition for their method and no implementation was done. Their algorithm uses abstract interpretation over a trace semantics for a simple object-oriented language to check the assignable clause by static analysis and they require some extra information by the user or specifier about what the things that are changed may be changed to.
This might have been useful if an implementation were available and a transfer from OCL input to JML could be done in a way so that most of the necessary extra information is generated automatically. But as there is no implementation we cannot make use of this approach.

6.2 Catano and Huisman

There is another paper about assignable clauses in JML by Catano and Huisman [CaHu03]. They describe a static checker which is neither sound nor complete but in practice can be used in quite a few instances they claim. Therefore it could be used for a quick check to see whether there are obvious things missing in the modifies clause but later on a real check would have to be done again. Thus that would not win us anything with the problem at hand as we want to have a guarantee that the modifies clause is correct and not only get told that we need to add something after the first try.
The checker is neither sound nor complete because it only does a static analysis of the program on a syntactic level and cannot account for aliasing because of that.

6.3 General

Both of the above approaches are not enough for us to ensure that in KeY the modifies clauses can be checked and guaranteed to be correct (in the sense we defined above). This is because of the problems described above.

The trouble appears in general as soon as aliasing happens because the static checking used cannot cope with aliasing whereas we do not have any problems with that because of using dynamic logic where the method in the diamond modality is symbolically executed and not only statically analysed. Thus we are easily beating the other approaches at the expense of needing the KeY tool which is available for

free. We are beating them in the sense that we have a working implementation while they do not have one for different reasons.

7 Implementation

The proof obligation generation as described in the previos chapters has already been implemented as part of the KeY project and works with all JAVA CARD constructs but does not allow things like $a[*]$ in the modifies clause to reference all array elements which would be a nice extension.

In the GUI of the case tool (when using KeY) there is the functionality to "Check-Modifies" which generates the proof obligation (as seen in chapter 5) in a KeY-prover window where it can be dismissed and by that we see whether the modifies clause is correct. If one can not close the proof, i.e. not show that the modifies clause is correct one can usually see from the remaining parts what might need to be added to the modifies clause to make it correct.

The performance for small case studies is very good as there the formulas stay pretty small and are easy to proof. Most of the proofs can even be completely done in automatic mode.

Real-world case studies have not yet been looked into.

8 Summary

This work presents a new way of checking modifies clauses for their correctness and is the first correct approach to this topic with a working implementation. You have seen in what environment my work is developed, what modifies clauses exactly are and what it means for them to be correct in detail. You have then seen that I check the modifies clauses with the help of the KeY tool by creating a proof obligation. You have seen a proof for the correctness of the proof obligation creation and some examples. Then I have shown in more detail what other approaches have done and what their shortcomings were and a short chapter on the implementation.
Once again I need to stress that the use of the KeY tool has made it possible that the simple proof obligation generation can be used and the created proof obligation can be discharged with little user interaction for a correct modifies clause.

8.1 Future Work

As seen in the implementation section on the array example one could take a look at different ways to write down larger groups of elements that are allowed to be modified to further facilitate the use of this checker.

One could also look into what happens with the variation of the formula from Chapter 5.4.

References

[ABBG+00] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel und Peter H. Schmitt. The KeY Approach: Integrating Object Oriented Design and Formal Verification. In Gerhard Brewka und Luís Moniz Pereira (Hrsg.), *Proc. 8th European Workshop on Logics in AI (JELIA)*, LNCS. Springer-Verlag, Oktober 2000. URL: `ftp://ftp.cs.chalmers.se/pub/users/reiner/jelia.ps.gz`.

[Beck01] Bernhard Beckert. A Dynamic Logic for the Formal Verification of Java Card Programs. In I. Attali und T. Jensen (Hrsg.), *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041. Springer, 2001, S. 6–24.

[BeSc03] Bernhard Beckert und Peter H. Schmitt. Program Verification Using Change Information. In *Proceedings, Software Engineering and Formal Methods (SEFM), Brisbane, Australia*. IEEE Press, 2003, S. 91–99.

[CaHu03] N. Cataño und M. Huisman. Chase: A Static Checker for JML's Assignable Clause. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi und Supratik Mukhopadhyay (Hrsg.), *VMCAI: Verification, Model Checking and Abstract Interpretation*, Band 2575 der *Lecture Notes in Computer Science*, New York, NY, USA, January 9-11 2003. Springer, S. 26–40.

[Hare84] David Harel. Dynamic Logic. In D. Gabbay und F. Guenthner (Hrsg.), *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, Kapitel II.10, S. 497–604. Dordrecht, 1984.

[Katz03] B. Katz. Studienarbeit Universität Karlsruhe "Eine Modifies-Klausel in KeY", 2003. URL: `http://i12www.ira.uka.de/~key/publicat.htm`.

[SpPo03] F. Spoto und E. Poll. Static Analysis for JML's assignable Clauses. In G. Ghelli (Hrsg.), *Proc. of FOOL-10, the 10th ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, New Orleans, Louisiana, USA, January 2003. ACM Press. Available at `www.sci.univr.it/~spoto/papers.html`.

[TBSc01] Bernhard Beckert Thomas Baar und Peter H. Schmitt. An Extension of Dynamic Logic for Modelling OCL's @pre Operator. In *Proceedings, Fourth Andrei Ershov International Conference, Perspectives of System Informatics.*, 2001.