Program Verification Using Change Information

Bernhard Beckert and Peter H. Schmitt
Universität Karlsruhe
Institute for Logic, Complexity, and Deduction Systems
D-76128 Karlsruhe, Germany
i12www.ira.uka.de/~key

Abstract

We propose an extension of the design-by-contract approach. In addition to preconditions, postconditions, and invariants as the basis for proving properties of a program, also information is provided on which parts of the state are not *changed by running the program. This is done in the form of* modifier sets. *We present a precise semantics of modifier sets and theorems on how to use them in program-correctness proofs. This technique has been implemented as part of the KeY system.*

1. Introduction

The work reported in this paper has been carried out as part of the KeY project. The goal of this project is to develop a tool supporting formal specification and verification of JAVA CARD programs within a commercial platform for UML based software development, see [1, 2] for details. This approach is based on the design-by-contract paradigm as pioneered by Bertrand Meyer [11]. Contracts are verified statically in KeY using a (partly automated) interactive theorem prover. Experiments with the KeY system suggest that the performance of the prover could be greatly enhanced if in addition to the terms of the contract, i.e., preconditions, postconditions, and invariants, additional information was available. We concentrate here on change information. More precisely, we associate with a JAVA program p (typically a method) a set Mod_p of terms (or expressions), called the modifier set (for p), with the understanding that Mod_p is part of the specification of p. Its semantics is that those parts of a program state that are *not* mentioned in Mod_p will never be changed by executing p (a state element is mentioned in Mod_p if there is a term in Mod_p referring to this element). We took the general idea for this set-up from the Java Modeling Language (JML) [8, 9]. JML is a behavioural interface specification language for JAVA, which allows to express change information via what in JML jargon is called *modifies clauses*.

A typical problem of program verification reads as follows: whenever program p is executed in a program state satisfying condition Γ then condition ϕ will be true after termination of p, where Γ and ϕ are arbitrary formulas of first-order logic. The KeY system uses Dynamic Logic (DL) as its underlying program logic. In DL the above statement is written as $\Gamma \to [p]\phi$. In Section 7 we will show a small example of a verification task that cannot be proved on the basis of the given pre- and postconditions alone. But, adding the modifier set a proof becomes possible.

In this paper, we concentrate on the theoretical underpinnings of modifier sets. Based on this theory, proof rules have been formulated and implemented within the KeY system. The details of that implementation will be described in [4].

Related work. The ESC/Java tool (Extended Static Checker for Java) [6] uses a subset of JML as assertion language; an extension of ESC/Java for checking JML *modifies clauses* is described in [5]. Despite the undisputed usefulness of this tool its results are still very preliminary: failing assertions of a rather simple kind go undetected and failures are reported, where in reality the assertion is correct. A survey of other JML-oriented tool projects is given in [10]. An extension of the expressivity of JML is investigated in [13]. In [12], a static analysis algorithm is proposed that checks modifies clauses for a simple object-oriented in vitro language. Correctness is proved via abstract interpretation over a trace semantics.

Plan of this paper. After reviewing the necessary prerequisites in Section 2, we define a precise semantics for modifier sets in Section 3. As a second contribution, we define a transformation on first-order formulas based on modifier sets (Section 4). In Section 5, we prove that validity of

$$\Gamma \to \phi_{\mathit{Mod}}$$

implies validity of

$$\Gamma \to [p]\phi \ ,$$

where ϕ_{Mod} is the transformation of ϕ using the modifier set *Mod* that is part of the specification of p. We show that, under certain additional assumptions, the reverse implication holds as well (Section 6). How to apply the transformation in software verification is discussed in Section 7. In Section 8 we discuss extensions and future work.

Acknowledgement. We gratefully acknowledge assistance from Bastian Katz, Rajeev Goré, and Kerry Trentelman, both in writing and in personal discussions. Also we thank Richard Bubel, Martin Giese, Andreas Roth, and Steffen Schlager for useful comments on earlier versions of this paper. Moreover, we thank an anonymous referee for his detailed review and valuable suggestions.

2. Program Logic

The KeY system uses the instance $\mathbf{DL}_J$ of Dynamic Logic as its program logic for JAVA CARD.[1] See e.g. [7] for a general exposition of Dynamic Logic and [3] for details on $\mathbf{DL}_J$. For our theoretical investigations in this paper, we also assume that the underlying programming language is JAVA CARD. The results being proved, however, hold true for all deterministic programming languages whose semantics can be described by Kripke structures in terms of Definition 1.

Besides concrete JAVA CARD programs, we allow *abstract* methods (resp. method calls) to occur, for which no implementation is given, but for which a specification exists that may be used to formally reason about their behaviour.[2]

Dynamic Logic is a modal logic with a modality $[p]$ for every program p (in $\mathbf{DL}_J$, for every sequence p of legal JAVA CARD statements). The formula $[p]\phi$ expresses that, if the program p terminates in a state s, then ϕ holds in s. A formula $\psi \to [p]\phi$ expresses that, for every state s_0 satisfying pre-condition ψ, if a run of the program p starting in s_0 terminates in s_1, then the post-condition ϕ holds in s_1. For deterministic programs, there is exactly one such world s_1 (if p terminates) or there is no such world (if p does not terminate). The formula $\psi \to [p]\phi$ is thus equivalent to the Hoare triple $\{\psi\}p\{\phi\}$. In contrast to Hoare logic, the set of formulas of DL is closed under the usual logical operators.

[1]The main restrictions of JAVA CARD w.r.t. full JAVA are (1) there are not threads (no concurrency), (2) no floating-point arithmetics, and (3) GUIs are not supported.

[2]Abstract methods are similar to *atomic programs*, i.e., programs that are just represented by a single symbol and do not have any internal structure. Atomic programs are the basic notion in propositional dynamic logic. An atomic program abstracts from concrete programs in the same way atoms of propositional logic abstract from first-order formulas.

The semantic domains in which $\mathbf{DL}_J$ formulas are interpreted are Kripke structures $\mathcal{K} = (S, \rho)$, where S is the set of states for $\mathcal{K}$ and ρ is the transition relation interpreting programs. Since we consider deterministic programs, ρ is a (partial) function, i.e., for every program p, $\rho(p) : S \to S$. The states $s \in S$ are typed first-order structures s, for some fixed signature Σ. We work under the constant domain assumption, i.e., for any two states $s_1, s_2 \in S$ the universes of s_1 and s_2 are the same set U. We sometimes refer to U as *the* universe of $\mathcal{K}$. Furthermore we assume that the set of states S of any Kripke structure $\mathcal{K}$ consists of *all* first-order structures with signature Σ over some fixed universe. Variations, in which some symbols of the signature are declared *rigid* and have a fixed interpretation for all $s \in S$, are possible and indeed for practical purposes essential. For example, addition $+$ on integers cannot be changed by executing a program and will therefore be declared *rigid*. For the sake of a clear presentation we do not consider *rigid* symbols in Σ. Their inclusion poses no difficulty.

We restrict attention to purely functional signatures Σ. The relation between JAVA constructs and signature elements is as follows: Classes give rise to types, (local) program variables occur as 0-ary function symbols (constants) in Σ, attributes occur as unary function symbols (where $\mathit{obj.attr}$ is the same as $\mathit{attr}(\mathit{obj})$), and n-dimensional arrays are represented by an $(n+1)$-ary function symbol (i.e., $\mathit{obj}[i_0, \ldots, i_{n-1}]$ is the same as $\mathit{arr}(\mathit{obj}, i_0, \ldots, i_{n-1})$). If all constructs of a program p occur in this sense in Σ, we call p a Σ-*program*. A signature Σ may, however, also contain symbols not occurring in programs such as, for example, user defined abstract data types. The interpretation of a function symbol f in a state s is denoted by f^s.

Logical variables, which are different from program variables, never occur in programs. They are rigid in the sense that if a value is assigned to a logical variable, it is the same for all states.

From what we have said it follows that once Σ and the universe U are fixed, the set S of states is also fixed. Thus, our Kripke structures will only differ in the state transition function ρ interpreting programs. In addition, when a programming language is chosen (in this case JAVA CARD), the possible choices for ρ have to be restricted as well, such that the constructs of the programming language are interpreted in the right way. If only concrete JAVA programs are allowed (i.e., *abstract* methods are not considered), then only one unique choice for ρ is possible. If abstract methods are used, on the other hand, their different possible implementations lead to a multitude of different possible interpretations, i.e., transition functions ρ. Note, that a program with different possible interpretations has an unknown *deterministic* behaviour (as opposed to a non-deterministic behaviour).

In the logics considered in [7], the only items that may change during program execution are program variables. In our logic $\mathbf{DL}_J$ we cannot maintain this restriction. Since we are dealing with an object-oriented programming language, we need to consider Kripke structures $\mathcal{K} = (S, \rho)$ and programs p such that states s_1, s_2 occur with $(s_1, s_2) \in \rho(p)$ and $f^{s_1} \neq f^{s_2}$ for some function symbol f. Because of this generality some of the familiar tautologies and proof rules are not available for our version of DL, e.g.,

$$[x = c;]\phi \ \leftrightarrow \ \phi[c/x] \ ,$$

where c is a constant. More precisely, for $\phi = [p]\psi$ the equivalence

$$[x = c;][p]\psi \ \leftrightarrow \ [p]\psi[c/x]$$

may not be true, since p could change the meaning of c.

Most familiar tautologies are, fortunately, still true, e.g.,

$$([p]\phi_1 \wedge [p](\phi_1 \to \phi_2)) \to [p]\phi_2 .$$

Since we deal only with deterministic programs, we also have

$$([p](\phi_1 \vee \phi_2)) \leftrightarrow ([p]\phi_1 \vee [p]\phi_2) \ .$$

From now on, we assume that a fixed set $\mathbf{K}_\Sigma$ of Kripke structures $\mathcal{K} = (S, \rho)$ is given that, as described above, depends (only) on the signature Σ, the universe U, and the restrictions on ρ, i.e., the semantics of JAVA CARD (resp. the chosen programming language). The set S of states is the same for all elements of $\mathbf{K}_\Sigma$.

Definition 1 *Let S be the set of all first-order structures over signature Σ with some fixed universe U. Then, the* semantics of the programming language *is given by a set $\mathbf{K}_\Sigma$ of Kripke structures that all share S as their set of states.*

Definition 2 *A Σ-formula ϕ is called* valid *if*

$$s, \beta \models \phi$$

for every state $s \in S$ of every Kripke structure $(S, \rho) \in \mathbf{K}_\Sigma$ and every variable assignment β (i.e., function from the set of logical variables to the fixed universe U). ∎

Note: In the rest of this paper, we suppress any mentioning of variable assignments β, but they should be considered to be implicitly present. Since logical variables are rigid for all programs, variable assignments never play a crucial role in the arguments of this paper. Showing them would only clutter the notation.

3. Modifier Sets

Below, we define the syntax of modifier sets and which transitions from a state s_1 to state s_2 *satisfy* a given modifier set *Mod*, i.e., are allowed by *Mod*.

Definition 3 *A* modifier set *Mod (over signature Σ) is a set of ground Σ-terms (i.e., terms without logical variables).*

Let S be the set of states of $\mathbf{K}_\Sigma$. A pair (s_1, s_2) of states from S satisfies *Mod, denoted by*

$$(s_1, s_2) \models \mathit{Mod} \ ,$$

iff, for

(a) all n-ary function symbols $f \in \Sigma$ ($n \geq 0$),

(b) all n-tuples $o_1, \ldots, o_n$ from the universe of $\mathbf{K}_\Sigma$,

the following holds:

$$f^{s_1}(o_1, \ldots, o_n) \neq f^{s_2}(o_1, \ldots, o_n)$$

implies that there is a term $t \in$ Mod of the form

$$t = f(t_1, \ldots, t_n)$$

with

$$o_i = t_i^{s_1} \ \ (1 \leq i \leq n) \ .$$

∎

Note, that the terms t in a modifier set are evaluated in the pre-state s_1. Consequently, if $\mathit{obj.attr} \notin \mathit{Mod}$ (where *obj* is a constant resp. a program variable),[3] then Definition 3 implies

$$\mathit{obj}^{s_1}.\mathit{attr}^{s_1} \ = \ \mathit{obj}^{s_1}.\mathit{attr}^{s_2} \ .$$

By contrast, it does *not* imply

$$(\mathit{obj.attr})^{s_1} \ = \ (\mathit{obj.attr})^{s_2} .$$

Now we proceed to define the relation between modifier sets and programs.

Definition 4 *Let Mod be a modifier set over signature Σ, let p be a Σ-program, and let $\mathcal{K} = (S, \rho) \in \mathbf{K}_\Sigma$ be a Kripke structure. Then,*

$$\mathcal{K} \models (\mathit{Mod}, p)$$

iff

$$(s_1, s_2) \models \mathit{Mod}$$

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

If $\mathcal{K} \models (\mathit{Mod}, p)$ for all $\mathcal{K} \in \mathbf{K}_\Sigma$, then Mod is called a modifier set for *p.* ∎

[3]Remember that $\mathit{obj.attr}$ is just a different notation for $\mathit{attr}(\mathit{obj})$.

p	*Mod*$_p$
$i = i + 1;$	$\{i\}$
$w.count = w.count + 1;$	$\{w.count\}$
$w = v; w.count = w.count + 1;$	$\{w, v.count\}$

Table 1. Example modifier sets.

Example 1 *Table 1 shows some simple* JAVA *programs p and modifier sets Mod$_p$ for these programs.*

Note that the program p_3 in the third row changes the value of $v.count$ and not that of $w.count$ (except if $w = v$). That is,

$$v^{s_1}.count^{s_1} \neq v^{s_1}.count^{s_2}$$

but

$$w^{s_1}.count^{s_1} = w^{s_1}.count^{s_2} \qquad (\text{if } w \neq v),$$

where s_1 is the intial state and s_2 is the state after running p_3. This explains why $w.count$ does not occur in the modifier set of p_3. ∎

To determine the smallest modifier set *Mod$_p$* for a JAVA program p is, of course, undecidable. Also, not all programs have a *finite* modifier set. Experience so far shows, however, that useful approximations can be obtained for many useful cases.

Similarly to pre- and postconditions, modifier sets can contain the $this$ reference and names of the arguments used in the method declaration. These have to be instantiated to construct the modifier set for a concrete method call.

The modifier set *Mod* that we assume to be part of the specification of program p restricts the possible transition functions ρ. This is reflected by the following definition introducing the notion of (Mod, p)-Kripke structures, which are Kripke structures satisfying the restriction imposed by *Mod* on the possible interpretations of p.

Definition 5 *Let Mod be a modifier set over signature Σ, and let p be a Σ-program. A Kripke structure $\mathcal{K} \in \mathbf{K}_\Sigma$ is a* (Mod, p)-Kripke structure *if $\mathcal{K} \models (Mod, p)$ (Def. 4).*

A Σ-formula ϕ is (Mod, p)-valid *if it is true in every state $s \in S$ of every (Mod, p)-Kripke structure $(S, \rho) \in \mathbf{K}_\Sigma$.* ∎

4. The Modifier Transformation

Before we come to the core of this section, we introduce *conditional terms* as an auxiliary syntactical construct.

Definition 6 *If t_1, t_2 are terms and ψ is a formula, then*

$$\text{if } \psi \text{ then } t_1 \text{ else } t_2$$

is a conditional term. *Its in a structure s is defined by:*

$$(\text{if } \psi \text{ then } t_1 \text{ else } t_2)^s = \begin{cases} t_1^s & \text{if } \psi^s = true \\ t_2^s & \text{otherwise} \end{cases}$$

∎

Conditional terms are a device for writing formulas more succinctly. In first-order formulas they can be eliminated:

Lemma 1 *For every first-order formula ϕ there is a logically equivalent formula ϕ^* without conditional terms.*

Proof: For an atomic formula ϕ and an occurrence occ of the conditional term if ψ then t_1 else t_2 in ϕ, let ϕ_1 be obtained from ϕ by replacing occ by t_1 and ϕ_2 by replacing occ by t_2. Then ϕ is equivalent to

$$(\psi \wedge \phi_1) \vee (\neg\psi \wedge \phi_2) \ .$$

Applying this observation repeatedly we obtain ϕ^*. For arbitrary non-atomic ϕ we obtain ϕ^* by an easy induction on the complexity of ϕ. ∎

To motivate the following definitions, consider a specification for a program p consisting of a precondition Γ, a postcondition ϕ, and a modifier set *Mod*. The transformed formula ϕ_{Mod}, to be defined below, is intended to have the same value in the initial state that ϕ has in the final state after execution of p. Roughly speaking, symbols not in *Mod* remain unchanged and symbols f in *Mod* are replaced by new corresponding symbols c_f (i.e., c_f represents the value of f after running p). Since the interpretation of the new symbol c_f is unknown, the impact of c_f in logical deductions mimics the impact of f after an unknown change by program p. The general idea seems very intuitive, but the details are surprisingly subtle.

Definition 7 *For any signature Σ let*

$$\Sigma_{mod} = \Sigma \cup \{c_f \mid f \in \Sigma\} \ ,$$

where all c_f (1) have the same signature as f, (2) are not in Σ, and (3) are different from each other.

Definition 8 *Let Mod be a modifier set. Then, the* modifier transformation *is defined as follows.*

For terms t, we define the transformed term t_{Mod} inductively:

1. *If t is a logical variable, then*

$$t_{Mod} = t \ .$$

2. *Otherwise, if $t = f(t^1, \dots, t^n)$, let*

$$f(s_1^1, \dots, s_1^n),\ \dots,\ f(s_k^1, \dots, s_k^n)$$

be all terms in Mod with leading function symbol f. Then,

$$\begin{aligned} t_{Mod} \quad = \quad & \mathsf{if}\ \bigvee_{i=1}^{k} \bigwedge_{j=1}^{n} (t_{Mod}^j = s_i^j) \\ & \mathsf{then}\ c_f(t_{Mod}^1, \dots, t_{Mod}^n) \\ & \mathsf{else}\ f(t_{Mod}^1, \dots, t_{Mod}^n) \end{aligned}$$

where $c_f \in \Sigma_{mod}$ (Def. 7).

For first-order formulas ϕ (not containing any program), the transformed formula ϕ_{Mod} is obtained by replacing all occurrences of atomic subformulas $p(t^1, \dots, t^r)$ in ϕ by $p(t_{Mod}^1, \dots, t_{Mod}^r)$. ▪

Example 2 *Let v be a constant. If $v \in Mod$, then*

$$v_{Mod} \quad = \quad \mathsf{if}\ true\ \mathsf{then}\ c_v\ \mathsf{else}\ v$$

(note that the empty conjunction is identical to $true$), which can be simplified to c_v.

If $v \notin Mod$, then

$$v_{Mod} \quad = \quad \mathsf{if}\ false\ \mathsf{then}\ c_v\ \mathsf{else}\ v$$

(the empty disjunction is identical to $false$), which can be simplified to v.

If $Mod = \{v.a\}$, then $(v.a)_{Mod}$ is

$$\mathsf{if}\ v_{Mod} = v\ \mathsf{then}\ v_{Mod}.c_a\ \mathsf{else}\ v_{Mod}.a\ . \qquad (1)$$

Since $v \notin Mod$, one can reduce v_{Mod} to v. Thus, term (1) can be reduced to $v.c_a$.

If $Mod = \{v, w.a\}$, then $(v.a)_{Mod}$ is

$$\mathsf{if}\ v_{Mod} = w\ \mathsf{then}\ v_{Mod}.c_a\ \mathsf{else}\ v_{Mod}.a\ . \qquad (2)$$

Since, in this case, $v \in Mod$, one can reduce v_{Mod} to c_v. Thus, term (2) can be reduced to

$$\mathsf{if}\ c_v = w\ \mathsf{then}\ c_v.c_a\ \mathsf{else}\ c_v.a\ .$$

A further reduction is not immediately possible as the value c_v may or may not be equal to the value of w. ▪

5. Correctness of the Transformation

Our strategy to prove the main theorem (Theorem 1 below) resembles in spirit what is usually called the *substitution lemma*: Take a syntactical item t, perform some syntactic change to obtain $t_{changed}$, and evaluate $t_{changed}$ in an appropriate structure s, i.e., compute $t^s_{changed}$. Now, look for a change to s to obtain $s_{changed}$ such that evaluating the unchanged syntactical item in the changed structure yields the same result, i.e., $t^s_{changed} = t^{s_{changed}}$.

In our present context, t_{Mod} defined in Definition 8 will play the role of $t_{changed}$. The role of $s_{changed}$ will be filled by s_{Mod} introduced in the next definition.

Definition 9 *Let s be a Σ_{mod}-structure, and let Mod be a modifier set over Σ.*

Then, s_{Mod} is the structure arising from s by the following changes. For every function symbol $f \in \Sigma$, let

$$f(s_i^1, \dots, s_i^n) \qquad (1 \le i \le k)$$

be all terms in Mod with leading function symbol f. We define for every n-tuple $o_1, \dots, o_n$ of objects in the universe of s:

$$f^{s_{Mod}}(o_1, \dots, o_n) \quad = \\ \begin{cases} c_f^s(o_1, \dots, o_n) & \text{if there is an } i \text{ with} \\ & o_j = (s_i^j)^s\ (1 \le j \le n) \\ f^s(o_1, \dots, o_n) & \text{otherwise} \end{cases}$$

▪

Lemma 2 *For all Σ_{mod}-structures s, all modifier sets Mod over Σ, all Σ-terms t, and all first-order Σ-formulas ψ, the following are true:*

1. *$(t_{Mod})^s = t^{s_{Mod}}$*

2. *$s \models \psi_{Mod}$ iff $s_{Mod} \models \psi$*

Proof:
1. By structural induction on t.

1.0. Base case: $t = v$ with v a logical variable is trivially true, since $v_{Mod} = v$ and the transition from s to s_{Mod} does not effect variable assignments.

1.1. Simple step case: $t = a$ with a a constant symbol. This is the special instance of the next case for $n = 0$. It might however help the reader to see it spelled out separately.

If $a \in Mod$ then a_{Mod} reduces to c_a, otherwise a_{Mod} reduces to a. On the other hand, $a^{s_{Mod}} = c_a^s$ if $a \in Mod$ and $a^{s_{Mod}} = a^s$ otherwise.

1.2. Step case: $t = f(t^1, \dots, t^n)$ for some n-ary functions symbol $f \in \Sigma$. Let, as above, $f(s_i^1, \dots, s_i^n)$ $(1 \le i \le k)$ be all terms in *Mod* with leading function symbol f. We organise the proof into two cases:

1.2.1. Case A: There exists an i such that

$$(t_{Mod}^j)^s = (s_i^j)^s \qquad (1 \le j \le n)\ .$$

Then:

$$\begin{aligned}
(t_{Mod})^s &\overset{\text{def. of } t_{Mod}}{=} (\text{ if } \textstyle\bigvee_{i=1}^{k} \bigwedge_{j=1}^{n} (t^j_{Mod} = s^j_i) \\
&\qquad \text{then } c_f(t^1_{Mod}, \dots, t^n_{Mod}) \\
&\qquad \text{else } f(t^1_{Mod}, \dots, t^n_{Mod})\ )^s \\
&\overset{\text{case assumption}}{=} (c_f(t^1_{Mod}, \dots, t^n_{Mod}))^s \\
&\overset{\text{standard}^4}{=} c_f^s((t^1_{Mod})^s, \dots, (t^n_{Mod})^s) \\
&\overset{\text{def. of } s_{Mod}}{=} f^{s_{Mod}}((t^1_{Mod})^s, \dots, (t^n_{Mod})^s) \\
&\overset{\text{ind. hyp.}}{=} f^{s_{Mod}}((t^1)^{s_{Mod}}, \dots, (t^n)^{s_{Mod}}) \\
&\overset{\text{standard}}{=} (f(t^1, \dots, t^n))^{s_{Mod}} \\
&\overset{\text{def. of } t}{=} t^{s_{Mod}}
\end{aligned}$$

1.2.2. Case B: Negation of Case A:

$$\begin{aligned}
t^s_{Mod} &\overset{\text{def. of } t_{Mod}}{=} (\text{if } \textstyle\bigvee_{i=1}^{k} \bigwedge_{j=1}^{n} (t^j_{Mod} = s^j_i) \\
&\qquad \text{then } c_f(t^1_{Mod}, \dots, t^n_{Mod}) \\
&\qquad \text{else } f(t^1_{Mod}, \dots, t^n_{Mod}))^s \\
&\overset{\text{case assumption}}{=} (f(t^1_{Mod}, \dots, t^n_{Mod}))^s \\
&\overset{\text{standard}}{=} (f^s(t^1_{Mod})^s, \dots, (t^n_{Mod})^s \\
&\overset{\text{def. of } s_{Mod}}{=} (f^{s_{Mod}}(t^1_{Mod})^s, \dots, (t^n_{Mod})^s \\
&\overset{\text{ind. hyp.}}{=} (f^{s_{Mod}}((t^1)^{s_{Mod}}, \dots, (t^n)^{s_{Mod}}) \\
&\overset{\text{standard}}{=} (f(t^1, \dots, t^n))^{s_{Mod}} \\
&\overset{\text{def. of } t}{=} t^{s_{Mod}}
\end{aligned}$$

2. Follows immediately from part 1. ■

Theorem 1 *Let Mod be a modifier set over signature Σ, and let p be a Σ-program. Moreover, let Γ and ϕ be first-order formulas over signature Σ.*

Then, validity of the Σ_{mod}-formula

$$\Gamma \to \phi_{Mod}$$

implies (Mod, p)-validity of the Σ-formula

$$\Gamma \to [p]\phi \ .$$

(The reverse implication does not hold in general.)

Proof: Let $\mathcal{K} = (S, \rho) \in \mathbf{K}_\Sigma$ be a (Mod, p)-Kripke structure, let $s \in S$ be a state with $s \models \Gamma$, and let (s, s') be a state pair in $\rho(p)$. It remains to be shown that $s' \models \phi$ is true.

Let s_1 be the Σ_{mod}-expansion that coincides with s except for the interpretation of the new symbols:

$$c_f^{s_1}(o_1, \dots, o_n) = f^{s'}(o_1, \dots, o_n) \ .$$

Since s and s_1 coincide on the symbols of Γ, we get $s_1 \models \Gamma$. The validity of $\Gamma \to \phi_{Mod}$ yields $s_1 \models \phi_{Mod}$. By Lemma 2 this implies $(s_1)_{Mod} \models \phi$. We will argue that $(s_1)_{Mod} =_\Sigma s'$, i.e., that both structures agree on the symbols from Σ. Since ϕ is a Σ-sentence we will then get $s' \models \phi$ as desired.

We now set out to prove $(s_1)_{Mod} =_\Sigma s'$. According to Definition 9, $f^{(s_1)_{Mod}}(o_1, \dots, o_n)$ is defined by the following case distinction.

Case A: There exists an i such that

$$o_j = (s^j_i)^{s_1} \qquad (1 \le j \le n) \ ,$$

where $f(s^1_i, \dots, s^n_i)$ $(1 \le i \le k)$ are all terms in *Mod* with leading function symbol f. Then:

$$\begin{aligned}
f^{(s_1)_{Mod}}(o_1, \dots, o_n) &\overset{\text{def. of } (s_1)_{Mod}}{=} c_f^{s_1}(o_1, \dots, o_n) \\
&\overset{\text{def. of } s_1}{=} f^{s'}(o_1, \dots, o_n)
\end{aligned}$$

Case B: Negation of Case A. There is no i such that

$$o_j = (s^j_i)^{s_1} \qquad (1 \le j \le n) \ .$$

As all s^j_i are Σ-terms there is also no i such that $o_j = (s^j_i)^s$ $(1 \le j \le n)$. As $\mathcal{K} = (S, \rho)$ is a (Mod, p)-Kripke structure and $(s, s') \in \rho(p)$, this implies

$$f^s(o_1, \dots, o_n) = f^{s'}(o_1, \dots, o_n) \ .$$

Thus:

$$\begin{aligned}
f^{(s_1)_{Mod}}(o_1, \dots, o_n) &\overset{\text{def. of } (s_1)_{Mod}}{=} f^{s_1}(o_1, \dots, o_n) \\
&\overset{\text{def. of } s_1}{=} f^s(o_1, \dots, o_n) \\
&\overset{\text{as observed}}{=} f^{s'}(o_1, \dots, o_n)
\end{aligned}$$

■

In the following examples, we list a modifier set *Mod* for a program p, the formula $\Gamma \to [p]\phi$, and the formula $\Gamma \to \phi_{Mod}$ (we use simplifications of t_{Mod}, such as those detailed in Example 2, without mentioning). Also, we discuss the consequences of Theorem 1.

Example 3

$$\begin{aligned}
Mod &\equiv \{i\} \\
\Gamma \to [p]\phi &\equiv i = 0 \to [i = i + 1;](i = 0) \\
\Gamma \to \phi_{Mod} &\equiv i = 0 \to c_i = 0
\end{aligned}$$

The formula $\Gamma \to \phi_{Mod}$ is certainly not valid. No claim on the validity of $\Gamma \to [p]\phi$ can be derived. As it should be. ■

[4] We use the justification *standard* for steps that only involve basic definitions of predicate logic.

Example 4

$$\begin{aligned} \mathit{Mod} &\equiv \{i\} \\ \Gamma \rightarrow [p]\phi &\equiv (j > 0 \land i = 0) \rightarrow [i = i + j;](j > 0) \\ \Gamma \rightarrow \phi_{\mathit{Mod}} &\equiv (j > 0 \land i = 0) \rightarrow j > 0 \end{aligned}$$

Now the formula $\Gamma \rightarrow \phi_{\mathit{Mod}}$ is valid, and Theorem 1 implies validity of $\Gamma \rightarrow [p]\phi$, which is again correct. ▪

Example 5

$$\begin{aligned} \mathit{Mod} &\equiv \{w, v.n\} \\ \Gamma \rightarrow [p]\phi &\equiv v.n = 5 \rightarrow \\ & \quad [w = v; w.n = w.n + 1](v.n = 5) \\ \Gamma \rightarrow \phi_{\mathit{Mod}} &\equiv v.n = 5 \rightarrow v.c_n = 5 \end{aligned}$$

The formula $\Gamma \rightarrow \phi_{\mathit{Mod}}$ is not valid. This conforms to our intentions since $v.n = 5 \rightarrow [p](v.n = 5)$ is not valid. ▪

Example 6

$$\begin{aligned} \mathit{Mod} &\equiv \{w.n\} \\ \Gamma \rightarrow [p]\phi &\equiv v.n = 5 \rightarrow \\ & \quad [w.n = w.n + 1](v.n = 5) \\ \Gamma \rightarrow \phi_{\mathit{Mod}} &\equiv v.n = 5 \rightarrow \\ & \quad (\mathsf{if}\ v = w\ \mathsf{then}\ v.c_n\ \mathsf{else}\ v.n) = 5 \end{aligned}$$

Here, the formula $\Gamma \rightarrow \phi_{\mathit{Mod}}$ is true provided that we have $v \neq w$ as an additional premiss. Otherwise, again nothing can be derived. ▪

6. Completeness of the Transformation

The modifier transformation is not only correct as shown in the previous section but also complete (i.e., the reverse of the implication in Theorem 1 holds), provided that the modifier set *Mod fully specifies* the program p. That is, if some state pair (s_1, s_2) satisfies *Mod*, then it must be possible according to JAVA's semantics that p causes a transition from s_1 to s_2.

To see that this is not always the case even if *Mod* is a modifier set for p (Def. 4), consider the following example: If *Mod* contains a local program variable c, a state pair (s_1, s_2) satisfying *Mod* may differ in the interpretation of c. But if, moreover, p is a method call, then p cannot change local variables (according to JAVA's semantics)—whatever its (unknown) implementation may be. In that case, one could argue that a too large modifier set *Mod* has been chosen and that c should be omitted from *Mod*, which solves the problem. Using a minimal modifier set is indeed a solution to this problem whenever p is an abstract method (call). Consider, however, the program $c = 0;$. A modifier set *Mod* for this program must contain c. But then, any state pair (s_1, s_2) with $c^{s_2} \neq 0$ and $t^{s_1} = t^{s_2}$ for $t \neq c$ satisfies *Mod* although $c = 0$; obviously cannot cause a transation to a state s_2 in which $c \neq 0$.

Definition 10 *A modifier set Mod (over signature Σ)* fully specifies *a Σ-program p if, for every set $\rho_p \subset S \times S$ of state pairs with*

1. *all $(s_1, s_2) \in \rho_p$ satisfy Mod,*
2. *$\{s_1 \mid \exists s_2\ (s_1, s_2) \in \rho_p\} = S$ (i.e., the domain of the partial function ρ_p is the whole state space S)*

there is a Kripke structure $(S, \rho) \in \mathbf{K}_\Sigma$ with $\rho(p) = \rho_p$.

This definition allows us to formulate the following completeness theorem, which reverses the implications of Theorem 1.

Theorem 2 *Let p be a Σ-program, and let Mod be a modifier set (over signature Σ) that fully specifies p w.r.t. $\mathbf{K}_{\Sigma_{mod}}$ (Def. 10). Moreover, let Γ and ϕ be first-order Σ-formulas.*

Then, (Mod, p)-Validity of the Σ_{mod}-formula

$$\Gamma \rightarrow [p]\phi$$

implies validity of the Σ_{mod}-formula

$$\Gamma \rightarrow \phi_{\mathit{Mod}} \quad .$$

Proof: We first observe that $\Gamma \rightarrow \phi_{\mathit{Mod}}$ is a statement about first-order logical consequence of Σ_{mod}-formulas only.

Let the set ρ_p of state pairs be defined by

$$\rho_p = \{(s, s_{\mathit{Mod}}) \mid s \in S\}$$

where S is the state space of $\mathbf{K}_{\Sigma_{mod}}$. By definition of s_{Mod} it is clear that all state pairs in ρ_p satisfy *Mod*. Thus, since *Mod* fully specifies p (Def. 10), there is a Kripke structure $\mathcal{K} = (S, \rho) \in \mathbf{K}_{\Sigma_{mod}}$ with $\rho(p) = \rho_p$. By construction of ρ_p, $\mathcal{K}$ is a (Mod, p)-Kripke structure.

Now, we consider an arbitrary Σ_{mod}-structure s satisfying $s \models \Gamma$ with the aim to show $s \models \phi_{\mathit{Mod}}$. By assumption, $\Gamma \rightarrow [p]\phi$ is true in all (Mod, p)-Kripke structures in $\mathbf{K}_{\Sigma_{mod}}$, thus this formula holds in particular in the state s of structure $\mathcal{K}$. That implies $s_{\mathit{Mod}} \models \phi$. And now Lemma 2 yields $s \models \phi_{\mathit{Mod}}$. ▪

In practice, we have fully specifying modifier sets only for abstract methods with unknown implementation (and not for arbitrary programs as the examples at the beginning of this section demonstrate). However, in [4] we show how the completeness theorem can be extended for the case where a modifier clause fully specifies a program in combination with a pre-/postcondition pair, which can be achieved for a much wider range of JAVA programs.

7. Applying the Transformation

Theorem 1 allows to derive $\Gamma \rightarrow [p]\phi$ from $\Gamma \rightarrow \phi_{Mod}$. That is, we can use the rule

$$\frac{\Gamma \rightarrow \phi_{Mod}}{\Gamma \rightarrow [p]\phi}$$

to prove a property of program p without analysing p—or even without knowing the implementation of p (in case p is a method call). This will in general not be very useful, since we need to take alse pre- and post-conditions of p into account.

Consider, as an example, a program p with pre- and post-conditions

$$\begin{array}{lcl} \phi_{pre} & \equiv & i > 0 \wedge j > 0 \\ \phi_{post} & \equiv & z = i + i \wedge i > 0 \ . \end{array}$$

In DL this is expressed as $\phi_{pre} \rightarrow [p]\phi_{post}$. Furthermore let $Mod_p = \{z\}$. Now assume that we want to show

$$(i = 5 \wedge j = 1) \rightarrow [p]z + j \geq 2 \ .$$

The usual technique (without modifier sets) is to apply the rule

$$\frac{\Gamma \rightarrow \phi_{pre} \qquad \phi_{post} \rightarrow \phi}{\Gamma \rightarrow [p]\phi}$$

i.e., to prove

1. $(i = 5 \wedge j = 1) \rightarrow \phi_{pre}$, and
2. $\phi_{post} \rightarrow z + j \geq 2$.

The first implication is certainly true, but the second cannot be proved.

Let us now try a different line of attack, making use of the modifier set. Since we aleady know

$$(i = 5 \wedge j = 1) \rightarrow [p]\phi_{post} \ ,$$

we could try to prove

$$(i = 5 \wedge j = 1) \rightarrow [p](\phi_{post} \rightarrow z + j \geq 2) \ .$$

Since

$$((C \rightarrow [p]A) \wedge (C \rightarrow [p](A \rightarrow B))) \rightarrow (C \rightarrow [p]B)$$

is a DL tautology, this would indeed prove our goal. By Theorem 1 the validity of

$$(i = 5 \wedge j = 1) \rightarrow [p](\phi_{post} \rightarrow z + j \geq 2)$$

is equivalent to the validity of

$$(i = 5 \wedge j = 1) \rightarrow (\phi_{post} \rightarrow z + j \geq 2)_{Mod} \ .$$

This formula can be simplified to

$$\begin{array}{l} (i = 5 \wedge j = 1) \rightarrow \\ ((c_z = i + i \wedge i > 0) \rightarrow c_z + j \geq 2) \ , \end{array}$$

and this is obviously a valid formula.

Generalising from the example, we can use the rule

$$\frac{\Gamma \rightarrow \phi_{pre} \qquad \Gamma \rightarrow (\phi_{post} \rightarrow \phi)_{Mod}}{\Gamma \rightarrow [p]\phi}$$

to prove properties of p making use of all parts of its specification. The right premiss of the above rule can be simplified to $(\Gamma \wedge (\phi_{post})_{Mod}) \rightarrow \phi_{Mod}$. Note that both the post-condition *and* that part of Γ which remains unchanged according to the modifier set are available to prove ϕ_{Mod}.

8. Extensions and Future Work

There are two obvious extensions to the work presented in this paper. The formula ϕ to which the modifier transformation is applied should be allowed to include the $@pre$ construct known, for example, from OCL (see e.g. [14]).

Second, the restriction on ϕ to be a first-order formula should be lifted. That is important because, currently, to prove, e.g., $\Gamma \rightarrow [p][q]\phi$, one has to find a formula ψ such that (a) $\Gamma \rightarrow [p]\psi$ and (b) $\psi \rightarrow [q]\phi$ hold, which then allows to apply the transformation and prove formulas (a) and (b) by showing $\Gamma \rightarrow \psi_{Mod}$ and $\psi \rightarrow \phi_{Mod}$ to be valid. In practice it can be difficult to come up with an appropriate formula ψ.

In both cases we know in which directions to look for a solution, but the details might be thorny.

Moreover, an algorithm to compute (approximations for) a modifier set for a given JAVA CARD program p should be defined and implemented.

The KeY system uses UML as its specification language. It is intended to define an extension of UML's constraint language OCL with mechanisms to express change information.

Finally, it may be worthwhile to extend the expressiveness of modifier sets such that programs for which currently no (finite) modifier set exists can also be handled.

References

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. Technical Report in Computing Science No. 2003-5, Department of Computing Science, Chalmers University and Göteborg University, Göteborg, Sweden, Feb. 2003. Available at `http://i12www.ira.uka.de/˜beckert/pub/key03.pdf`.

[2] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. d. Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer, 2000.

[3] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[4] B. Beckert and B. Katz. Using change information in the verification of Java programs with OCL specifications, 2003. Forthcoming.

[5] N. Cataño and M. Huisman. Chase: A static checker for JML's assignable clause. In *Proceedings, Verification, Model Checking and Abstract Interpretation (VMCAI)*, LNCS 2575, pages 26–40. Springer, 2003.

[6] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.

[7] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.

[8] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer Academic Publisher, 1999.

[9] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. TR 98-06t, Department of Computer Science, Iowa State University, 2002.

[10] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. TR 00-15, Department of Computer Science, Iowa State University, 2000.

[11] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[12] F. Spoto and E. Poll. Static analysis for JML's assignable clauses. In *Proceedings, Foundations of Object-Oriented Languages (FOOL10)*, 2003.

[13] K. Trentelman and M. Huisman. Extending JML specifications with temporal logic. In *Proceedings, Algebraic Methodology and Software Technology (AMAST)*, LNCS 2422, pages 334–348. Springer, 2002.

[14] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.