An Improved Rule for While Loops in Deductive Program Verification

Bernhard Beckert[1], Steffen Schlager[2], and Peter H. Schmitt[2]

[1] University of Koblenz-Landau, Institute for Computer Science,
D-56072 Koblenz, Germany
beckert@uni-koblenz.de

[2] Universität Karlsruhe, Institute for Theoretical Computer Science,
D-76128 Karlsruhe, Germany
{schlager, pschmitt}@ira.uka.de

Abstract. Performance and usability of deductive program verification systems can be enhanced if specifications not only consist of pre-/post-condition pairs and invariants but also include information on which memory locations are modified by the program. This allows to separate the aspects of (a) which locations change and (b) how they change, state the change information in a compact way, and make the proof process more efficient. In this paper, we extend this idea from *method specifications* to *loop invariants*; and we define a proof rule for while loops that makes use of the change information associated with the loop body. It has been implemented and is successfully used in the KeY software verification system.

1 Introduction

The Idea of Specifying Change Information and a Motivating Example. Deductive program verification systems are mostly based on program logics, such as dynamic logic [11,13,12] and Hoare logic [3]. Their performance and usability can be greatly enhanced if specifications of programs not only consist of the usual pre-/post-condition pairs and invariants but also include additional information, such as knowledge about which memory locations are changed by a program. More precisely, we associate with a program p a set Mod_p of 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* referenced by an expression in Mod_p will never be changed by executing p.

As a motivating example, consider the following program $p_{\min}$ that computes the minimum of an array a of integers:

```
$m := a[0];$ $i := 1;$
while $(i < \mathit{length}(a))$ do
    if $(a[i] < m)$ then $m := a[i];$ fi
    $i := i + 1;$
od
```

K.-K. Lau and R. Banach (Eds.): ICFEM 2005, LNCS 3785, pp. 315–329, 2005.

A correct (though incomplete) post-condition for this program is

$$\phi_{\min} \;=\; (\forall x)(0 \leq x < \mathit{length}(a) \rightarrow a[x] \leq m)$$

stating that, after running $p_{\min}$, the variable m indeed contains the minimum of a. However, a specification that just consists of $\phi_{\min}$ is rather weak. The problem is that $\phi_{\min}$ can also be established using, for example, a program that sets m as well as all elements of a to 0, which of course is not the *intended* behaviour. To exclude such programs, the specification must also state what the program does modify (the variables i and m) and does not modify (the array a and its elements). One way of doing this is to extend the post-condition with an additional part

$$\phi_{inv} \;=\; (\forall x)(0 \leq x < \mathit{length}(a) \rightarrow a[x] = a'[x])$$

where a' is a new array variable (not allowed to occur in the program) that contains the "old" values of the array elements. To make sure a' has the same elements as a, the formula ϕ_{inv} must also be used as a pre-condition and, thus, be turned into an invariant. In Dynamic Logic, this specification of $p_{\min}$ is written as $\phi_{inv} \rightarrow [p_{\min}](\phi_{\min} \wedge \phi_{inv})$.

But, then, ϕ_{inv} also has to be made part to the loop invariant

$$\phi_{loopinv} \;=\; \phi_{inv} \;\wedge\; 0 \leq i \leq \mathit{length}(a) \wedge (\forall x)(0 \leq x < i \rightarrow a[x] \leq m)$$

that is used during the proof that $p_{\min}$ indeed satisfies its specification, making that proof more complex and proof construction more difficult and less efficient.

In general, loop invariants are "polluted" by formulas stating what the loop does *not* do. All relevant properties of the pre-state that need to be preserved have to be encoded into the invariant, even if they are in no way affected by the loop. Thus, two aspects are intermingled:

- Information about what intended effects the loop *does* have.
- Information about what non-intended effects the loop *does not* have.

This problem can be avoided by encoding the second aspect (i.e., the change information) with a modifier set instead of adding it to the invariant. The two aspects then get separated both in the specification and in the correctness proof, as the (sub-)proofs that a program (a) satisfies its post-condition and (b) satisfies its modifier set are also separated as well.

For our program $p_{\min}$, an appropriate modifier set is

$$\mathit{Mod}_{\min} = \{i, m\} \;.$$

It states in a very compact and simple way that $p_{\min}$ only changes i and m and, in particular, does *not* change the array a.

Besides the separation of the two different aspects, modifier sets have the advantage that they encode what is changed, while invariants must encode all locations that are *not* changed, which for non-trivial programs are many more.

Extension to Loops. Modifier sets that are part of method or function specifications have been investigated before (see the section on related work). Now, in this paper, we extend the idea of modifier sets from *method specifications* to *loop invariants*. Here, as well, modifier sets allow

- to separate the aspects of which locations change and how they change,
- state the change information in a compact way
- make the proof process more efficient.

To achieve the latter point, we define a new Dynamic Logic proof rule for while loops that makes use of the information contained in a modifier set for the loop *body* (as is also described in the following, the rule can easily be adapted to other program logics, such as Hoare logic).

Loops in general can—and in practice often will—change a finite but *unknown* number of memory locations (though in our simple motivating example p_{min} the number of changed locations is known to be 2). A loop may, for example, change all elements in a list whose length is not known at proof time but only at run time. Therefore, to handle loops, we use an extended version of modifier sets that can describe location sets of unknown size (the modifier sets for methods described in [6] cannot do that).

Related Work. The Java Modeling Language (JML) [14,15] allows to express change information for Java methods via what in JML jargon is called *assignable clauses*.

The ESC/Java tool (Extended Static Checker for Java) [9] uses a subset of JML as assertion language; an extension of ESC/Java for checking JML *assignable clauses* is described in [8]. 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. In [22], a static analysis algorithm is proposed that checks assignable clauses for a simple object-oriented in vitro language. Correctness is proved via abstract interpretation over a trace semantics.

Daikon [18,10] is a heuristic approach to automatic detection of likely invariants by analysing program runs with concrete input values.

In [6], we have defined a precise semantics for method modifier sets and defined a transformation on first-order formulas based on modifier sets such that $\Gamma \to \phi_{Mod}$ 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 method p. This transformation can be used to employ modifier sets for proving the correctness of methods. However, it is restricted to modifier sets describing sets of memory location of fixed size, and it cannot easily be adapted to loop invariants—though the basic idea is similar to the new loop rule we present here.

Further related work is the Hoare calculus for a variant of C that is developed within the Verisoft project [21]. It allows to add simple modifier sets to procedure specifications. In [7], a method is presented that does not use explicit modifier sets but assumes that only what is mentioned in the pre- and post-condition may be changed.

Implementation in the KeY System. The work reported in this paper has been carried out as part of the KeY project [1,2]. 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.

Both the modifier set technique for methods from [6] and the rule for handling rules presented in this paper have been implemented in KeY. Experiments show that the performance of the prover is greatly enhanced using these extensions. KeY also contains functionality for verifying correctness of modifier sets [20].

Plan of This Paper. After reviewing the necessary pre-requisites in Section 2, we define our extended version of modifier sets in Section 3, which allows to describe location sets of unknown size. In Section 4, we introduce the notion of *quantified updates*. These updates, that are used in our verification rules, can be seen as a form of generalised substitutions. The new loop rule that makes use of modifier sets for loop bodies is introduced in Section 5. The implementation of the rule is described in Section 6. In Section 7, we give an extended example for its application. And, finally, in Section 8 we draw some conclusions.

2 Program Logic

To keep things simple in the paper, we consider as a programming language a simple deterministic while-language with assignments, if-then-else, while-loops, and arrays (due to lack of space we refrain from a formal definition of syntax and semantics). However, our approach applies to all deterministic programming languages whose semantics can be described by Kripke structures in terms of Def. 1. In the KeY tool we have implemented the invariant rule for the real object-oriented language Java Card taking all the difficulties likes aliasing and abrupt termination into account (see Sect. 6).

The program logic we consider in this paper is an instance of Dynamic Logic (DL) which is a multi-modal logic with a modality $[p]$ for every program p of the considered programming language. The formula $[p]\phi$ expresses that, if the program p terminates in a state s, then ϕ holds in s. A formula $\psi \rightarrow [p]\phi$ expresses that, for every state s_1 satisfying pre-condition ψ, if a run of the program p starting in s_1 terminates in s_2, then the post-condition ϕ holds in s_2. For deterministic programs, there is exactly one such world s_2 (if p terminates) or there is no such world (if p does not terminate). The formula $\psi \rightarrow [p]\phi$ 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.

The semantic domains used to interpret DL formulas 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, $\rho(p) : S \rightarrow S$. The states $s \in S$ are typed first-order structures s, for some fixed signature Σ. We restrict attention to purely functional signatures Σ and 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. Some symbols of the signature are declared *rigid* and have a fixed interpretation for all $s \in S$. E.g., addition $+$ on integers cannot be changed by executing a program and will therefore be declared *rigid*. In contrast, the interpretation of *non-rigid* function symbols may differ from state to state. E.g., program variables occur as non-rigid 0-ary function symbols (constants) in Σ, and n-dimensional arrays are represented by non-rigid n-ary function symbols (i.e., $a[i_1, ..., i_n]$ is the same as $a(i_1, \ldots, i_n)$ (similarly, object attributes in an object-oriented language can be represented by unary function symbols). 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.

Once the signature Σ and the universe U are fixed, the set S of states is also fixed and our Kripke structures will only differ in the state transition function ρ interpreting programs. When a programming language is chosen (in this case a while-language), 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.

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 our while-language with arrays. 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 (mapping logical variables to elements of the universe U).

3 Modifier Sets

A *modifier set Mod_p* for a program p is a set of ground terms denoting locations (i.e., the terms must not contain logical variables but they can contain program variables, which are constants in the logic). In contrast to [6] where modifier sets are written as lists of ground terms of fixed length, we consider in this paper modifier sets describing location sets of unknown size, since while loops in general may modify an unknown number of locations that depends on the state in which the loop is started. Of course, such modifier sets can no longer be represented as simple enumerations of ground terms. Rather, we use formulas to define the set of ground terms that may change.

Definition 3. *Let χ^j be a Dynamic Logic formula over Σ, $f^j \in \Sigma$ a non-rigid function symbol, and $t_1^j, \dots, t_{n_j}^j$ terms ($j \geq 1$). Then, the set*

$$\{ \langle \chi^1, f^1(t_1^1 \dots, t_{n_1}^1) \rangle, \dots, \langle \chi^k, f^k(t_1^k \dots, t_{n_k}^k) \rangle \}$$

of pairs is a modifier set.

Intuitively, a location $f(s_1, \dots, s_n)$ may be changed by a program p when started in a state s if the modifier set for p contains an element $\langle \chi, f(t_1, \dots, t_n) \rangle$ and there is variable assignment β such that the following conditions hold:

1. $s, \beta \models t_i \doteq s_i$ for $1 \leq i \leq n$, i.e. β assigns the free logical variables occurring in t_i values such that t_i coincides with s_i.
2. $s, \beta \models \chi$, i.e. the characteristic formula χ holds for the variable assignment β.

A modifier set *Mod* is said to be correct for a program p if p at most changes the value of locations mentioned in *Mod.*

Definition 4. *Let Mod be a modifier set and let S be the set of states.*

A pair $(s_1, s_2) \in S \times S$ 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 $o_1, \dots, o_n$ from the universe U,

the following condition holds:

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

implies that there is a pair $\langle \chi, f(t_1, \dots, t_n) \rangle \in \mathit{Mod}$ and a variable assignment β such that

$$o_i = t_i^{s_1, \beta} \ (1 \leq i \leq n) \ \text{ and } \ s_1, \beta \models \chi \ .$$

The modifier set Mod is correct *for a program p, if*

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

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

Example 1. Consider the following program, where a is a one-dimensional array of integers.

$$i \ := 0; \ j \ := 0; \ \texttt{while} \ (i < \mathit{length}(a)) \ \texttt{do} \ a[i] \ := a[i] * 2; \ i \ := i + 1; \ \texttt{od}$$

We assume that the size $s = \mathit{length}(a)$ of the array is not fixed in advance but unknown. Thus, for giving a correct modifier set, it is not possible to enumerate the locations $a[0], a[1], \dots, a[s]$ as s is not known.

However, a correct modifier set for the above program can be written as

$$\{ \langle 0 \leq x < \mathit{length}(a) \rangle, a[x] \rangle, \langle \mathit{true}, i \rangle, \langle \mathit{true}, j \rangle \} \ .$$

illustrating that modifier sets are not necessarily minimal (j is not modified).

The modifier set $\{ \langle 0 \leq x < \mathit{length}(a) \rangle, a[x] \rangle \}$ is not correct for the above program, since i is actually changed by the program.

4 Quantified Updates

The rules in calculi for deductive program verification (such as Hoare logic or Dynamic Logic) in a certain sense symbolically execute the program to be verified. And, usually, a state update, i.e., an assignment like $x := t$, is done by applying a substitution that replaces occurrences of x by t. This straightforward method works fine for simple programming languages but causes problems for more complex languages like JAVA CARD. In JAVA CARD (as in all other object-oriented programming languages) the same object may be referenced by several different reference variables (*aliasing*). We face the aliasing problem already for our simple while-language, because it contains arrays. An assignment $a[i] := 5$ changes the value of $a[j]$ if $i \doteq j$, i.e., $a[i]$ and $a[j]$ reference the same same array element. As a consequence, every array assignment causes a case distinction making verification infeasible. This is even more true for object-oriented languages where every assignment to an object attribute causes case distinctions. The solution to this problem proposed in [4] and implemented in the KeY System are so-called *updates*. The idea is to not immediately perform substitutions for assignments. Rather assignments are collected as state updates and not applied before the program has been completely symbolically executed. The advantage of this method is that assignments often cancel out previous ones rendering case distinctions for alias analysis unnecessary.

Definition 5 (Syntax of updates). *The set of Dynamic Logic formulas is extended as follows. For all non-rigid ground terms t, and all terms v, if ϕ is a formula, then $\{t := v\}\phi$ is a formula as well. The expressions $\{t := v\}$ are called* updates.

The formula $\{t := v\}\phi$ has the same semantics as $[t := v;]\phi$. Thus, one might ask why updates are introduced as a separate syntactic category instead of using assignments. Indeed, the goal of postponing the symbolic execution of state changes can be achieved without updates. However, there are some immediate extensions to updates that cannot be mimicked with assignments. E.g., one can introduce *quantified updates* that use a logical formula to describe the state change. This is a useful extension in the current context and is introduced below.

Anyway, it is important to note that updates are introduced for efficiency reasons but do not make the logic more expressive. A formula ϕ containing updates can always be transformed (in a uniform way) into an formula ϕ' without updates such that ϕ is valid iff ϕ' is valid. Therefore, the idea of modifier sets for loop bodies and the rule we introduce in the following section work just as well in calculi without updates.

The transformation for removing an update basically works by performing the symbolic execution that the state update represents (i.e., it does what updates try to avoid). It introduces new variables for preserving the old values of the changed variables (the value before the update is applied). However, due to aliasing the set of variables (or locations) that is affected by an update cannot be determined syntactically. Rather, all references (of compatible types) have to be checked for whether they point to the location that is updated or not.

Example 2. We consider the DL formula

$$(a[i] \doteq 0 \land a[j] \doteq 0) \rightarrow \{a[i] := a[i] + 1\}a[j] \doteq 0$$

which holds iff $i \not\doteq j$. The transformed formula without updates is

$$\begin{aligned}(a'[i] \doteq 0 \land a'[j] \doteq 0) \;\rightarrow\; (\; & a[i] \doteq a'[i] + 1 \land \\ & (i \not\doteq j \rightarrow a[j] \doteq a'[j]) \land \\ & (i \doteq j \rightarrow a[j] \doteq a'[i] + 1)\;) \rightarrow\; a[j] \doteq 0 \ .\end{aligned}$$

We now extend the idea of updates to *quantified updates*, a generalised form of updates proposed in [19] that allows to update arbitrary sets of locations described by a characteristic formula.

Definition 6 (Syntax of quantified updates). *The set of Dynamic Logic formulas is extended as follows. For all DL formulas χ, terms $f(t_1, \ldots, t_n)$ with a non-rigid function symbol f, and (arbitrary) terms v, if ϕ is a DL formula, then $\{\chi \ ?\ f(t_1, \ldots, t_n) := v\}\phi$ is a DL formula as well. The expressions $\{\chi \ ?\ f(t_1, \ldots, t_n) := v\}$ are called* quantified updates*.*

Example 3. The quantified update $\{0 \leq i < \mathit{length}(a) \ ?\ a[i] := 0\}\phi$ assigns 0 to all elements of the array a.

Quantified updates—in contrast to “simple” updates (Def. 5)—may contain clashes. For example, the update $\{0 \leq i \leq 1 \ ?\ c := i\}$ tries to assign to the non-rigid constant c both the values 0 and 1. We define that, in case of a clash, an arbitrary (unknown) but fixed element is used. However, the updates we consider in this paper cannot contain clashes by construction. And without clashes, the semantics of the formula $\{\chi \ ?\ t := v\}\phi$ is the same as that of the transformed formula $(\forall_{Cl})((\chi \rightarrow \{t := v\}\phi) \land (\neg\chi \rightarrow \phi))$. Thus, as with simple updates, a formula containing quantified updates can always be transformed into an equivalent formula without them.

Definition 7 (Semantics of quantified updates). *Let s be a state, and let*

$$\mathcal{U} = \{\chi \ ?\ f(t_1, \ldots, t_n) := v\}$$

be a quantified update.

The state $\mathcal{U}(s)$ is defined as follows: $\mathcal{U}(s)$ coincides with s except for the interpretation of the function symbol f, which is defined by

$$\begin{aligned}V(o_1, \ldots, o_n) = \{\mathit{val}_{s,\beta}(v) \mid \ & \mathit{val}_{s,\beta}(\chi) = \mathit{tt} \text{ and } \mathit{val}_{s,\beta}(t_i) = o_i \ (1 \leq i \leq n), \\ & \text{where } \beta \text{ is a variable assignment}\}\end{aligned}$$

$$f^{\mathcal{U}(s)}(o_1, \ldots, o_n) = \begin{cases} w & \text{if } V(o_1, \ldots, o_n) = \{w\} \\ f^s(o_1, \ldots, o_n) & \text{if } V(o_1, \ldots, o_n) = \emptyset \\ w \in V(o_1, \ldots, o_n) \text{ arbitrarily} & \text{otherwise} \end{cases}$$

for all elements $o_1, \ldots, o_n$ of the universe.

The semantics of the application $\mathcal{U}\phi$ of a quantified update $\mathcal{U}$ to a formula ϕ is defined by

$$s \models \mathcal{U}\phi \qquad \text{iff} \qquad \mathcal{U}(s) \models \phi \ .$$

5 Invariant Rule Using Change Information

5.1 Motivation

Before we present our invariant rule that uses modifier sets and the change information they encode, we recall what the invariant rule in Dynamic Logic (with updates) looks like:

$$\frac{\mathit{\Gamma} \vdash \mathcal{U}\mathit{Inv},\ \mathit{\Delta} \qquad \mathit{Inv},\ \epsilon \vdash [\alpha]\mathit{Inv} \qquad \mathit{Inv},\ \neg\epsilon \vdash \phi}{\mathit{\Gamma} \vdash \mathcal{U}[\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\ \texttt{od}]\phi,\ \mathit{\Delta}} \tag{1}$$

Intuitively the above rule states that, if one can find an invariant *Inv* such that the three premisses hold, which state that (a) *Inv* holds in the beginning, (b) *Inv* is indeed an invariant, and (c) the conclusion ϕ follows from *Inv* and the negated loop condition ϵ, then ϕ holds after executing the loop (provided it terminates).

As a motivation for why using change information is useful, consider the following example program p defined as

$$q;\ i \texttt{ := } 0;\ \texttt{while}\ (i < \mathit{length}(a))\ \texttt{do}\ a[i] \texttt{ := } 0;\ i \texttt{ := } i+1;\ \texttt{od}\ ,$$

where q is a (sub-)program. In order to prove some post-condition ϕ under the pre-condition ψ for p we have to show the validity of the DL formula $\psi \rightarrow [p]\phi$. Using our DL sequent calculus, symbolic execution of q results in a sequence $\mathcal{U}$ of updates describing the program state after execution of q. Then, considering that the while loop simply assigns all the elements of array a the value 0, an obvious invariant for the loop might be $i \leq \mathit{length}(a) \wedge (\forall x)(0 \leq x < i \rightarrow a[x] \doteq 0)$. In fact, this is an invariant for the loop (i.e., it holds at the beginning of the loop and holds after each iteration of the loop body) but it is not strong enough to entail the post-condition ϕ in general, i.e. the third premiss of the loop rule does not hold. The reason is that the second and the third premiss of the invariant rule omit the formulas $\mathit{\Gamma}, \mathit{\Delta}$ and the sequence $\mathcal{U}$ of updates, i.e., all information about the state reached before running the while loop is lost though it may be unrelated to the array a (one can construct similar examples where the second premiss does not hold). The only way to keep this information—as long as no modifier sets are used—is to add it to the invariant which, as already explained in the introduction, has several disadvantages.

The invariant rule proposed in this paper allows to keep as much context information as possible without explicitly encoding the context in the invariant. This is achieved by only throwing away those parts of $\mathit{\Gamma}, \mathit{\Delta}$ and $\mathcal{U}$ (i.e., of the descriptions of the initial state) that may be changed by the loop. Anything that remains unchanged is kept and can be used to establish the invariant (second premiss) and the post-condition (third premiss).

Our new rule is still available if, for some reason, no modifier sets is known for the loop body. In that case, it assumes that the loop potentially changes everything, and it then coincides with the traditional invariant rule. However, programmers usually know what is changed by a piece of code and can (or even *should*) annotate the code with the appropriate information.

An important advantage of using modifier sets is that usually a loop only changes few locations and only these locations must be put in a modifier set. On the other hand, using the traditional rule, all locations that do *not* change and whose value is of importance have to be included in the invariant and, typically, the number of locations that are not changed by the loop is much bigger than the number of locations that are actually changed. Of course, in general not everything that remains unchanged is needed to establish the post-condition in the third premiss. But when applying the invariant rule it is often not obvious what information must be preserved, in particular if the loop is followed by a non-trivial program. That can lead to repeated failed attempts to find the right invariant that allows to complete the proof. Whereas, to figure out the locations that are possibly changed by the loop, it is usually enough to look at the small piece of code in the loop body.

5.2 The New Invariant Rule for Dynamic Logic

Let *Mod* be a modifier set that is correct for the loop body α. The basic idea of the new version of the loop rule we define in this section is that the context $\Gamma, \Delta, \mathcal{U}$ is *not* removed from the second and third premiss. Then, however, information on locations appearing in the context $\Gamma, \Delta, \mathcal{U}$ that are mentioned in *Mod* must not be used. It must be removed. To meet this requirement, we introduce so-called *anonymous updates* which assign an arbitrary unknown value (represented by a Skolem symbol) to the locations mentioned in the modifier set and, thus, since nothing is known about the new unknown values, destroy the information on these (and only these) locations.

Definition 8 (Anonymous Update). *Let*

$$\mathit{Mod}_p = \{\langle \chi_1, f_1(t_1, \ldots, t_{n_1})\rangle, \ldots, \langle \chi_m, f_m(t_1, \ldots, t_{n_m})\rangle\}$$

be a correct modifier set for a program p. For every f_i, let f_i^{sk} be a fresh rigid function symbol with the same arity as f_i. Then, the sequence $\mathcal{V} = \mathcal{V}_1 \cdots \mathcal{V}_m$ of quantified updates where

$$\mathcal{V}_i \;=\; \{\chi_i \;?\; f_i(t_i, \ldots, t_{n_i}) := f_i^{sk}(t_1, \ldots, t_{n_i})\}$$

is called an anonymous update *with respect to Mod_p. By abuse of terminology we call the new function symbols f_i^{sk} Skolem functions.*

Now, we can proceed to define the new invariant rule for while loops using change information:

$$\frac{\Gamma \vdash \mathcal{U}\mathit{Inv},\ \Delta \quad \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \wedge \epsilon) \vdash \mathcal{U}\mathcal{V}[\alpha]\mathit{Inv},\ \Delta \quad \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \wedge \neg\epsilon) \vdash \mathcal{U}\mathcal{V}\phi,\ \Delta}{\Gamma \vdash \mathcal{U}[\mathtt{while}\ \epsilon\ \mathtt{do}\ \alpha\ \mathtt{od}]\phi,\ \Delta} \tag{2}$$

where $\mathcal{V}$ is an anonymous update (Def. 8) w.r.t. the modifier set *Mod*, which is correct for the loop body α (Def. 4).

Depending on the particular proof goal, the context encoded in $\Gamma, \Delta, \mathcal{U}$ may only be needed in either the second or the third premiss of the rule and not in both of them. In that case, the premiss where the context is not needed can be simplified and replaced by the corresponding premiss from the classical Rule (1). If both premisses are simplified, Rules (2) and (1) become identical.

Theorem 1 (Soundness). *Let Inv be an arbitrary formula and $\mathcal{V}$ an anonymous update w.r.t. a correct modifier set Mod_α for the loop body α.*

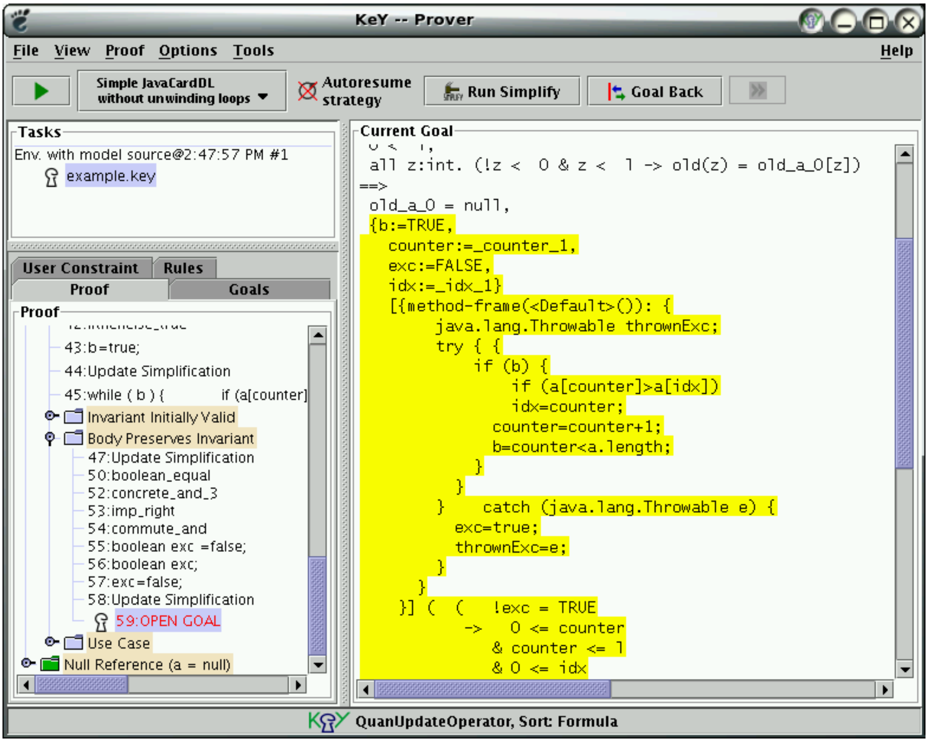
If all premisses of Rule (2) are valid in all states, then its conclusion is valid in all states.

Proof. See [5].

Even the main focus in this paper is on Dynamic Logic, the approach is not restricted to this particular logic. A version of the improved invariant rule for Hoare logic can be found in [5].

6 Implementation

We have implemented the invariant rule that uses change information in the KeY system for the programming language Java Card. Advanced features like



Env. with model source@2:47:57 PM #1
example.key

```
43:b=true;
44:Update Simplification
45:while ( b ) {            if (a[counter]
Invariant Initially Valid
Body Preserves Invariant
    47:Update Simplification
    50:boolean_equal
    52:concrete_and_3
    53:imp_right
    54:commute_and
    55:boolean exc =false;
    56:boolean exc;
    57:exc=false;
    58:Update Simplification
    59:OPEN GOAL
Use Case
Null Reference (a = null)
```

```
 all z:int. (!z <  0 & z <  1 -> old(z) = old_a_0[z])
==>
 old_a_0 = null,
 {b:=TRUE,
   counter:=_counter_1,
   exc:=FALSE,
   idx:=_idx_1}
   [{method-frame(<Default>()): {
        java.lang.Throwable thrownExc;
        try { {
            if (b) {
                if (a[counter]>a[idx])
                idx=counter;
              counter=counter+1;
              b=counter<a.length;
            }
          }
        }    catch (java.lang.Throwable e) {
          exc=true;
          thrownExc=e;
        }
      }
    }] (  (   !exc = TRUE
           ->   0 <= counter
              & counter <= 1
              & 0 <= idx
```

QuanUpdateOperator, Sort: Formula

Fig. 1. KeY prover window with the example from Sect. 7 after applying the invariant rule

abrupt termination, exceptions, side-effects of expressions, break- and continue-statements of a real object-oriented language like Java Card make the implemented rule more involved than the one presented above. For example, in case of side-effects the invariant rule cannot be applied directly. Beforehand, the following rule has to be applied that performs a program transformation and ensures that the loop condition does not have side-effects

$$\frac{\Gamma \vdash \mathcal{U}[\mathsf{boolean}\ b\ =\ \mathit{expr};\ \mathsf{while}\ (b)\ \{\alpha'; b = \mathit{expr};\}]\phi,\ \Delta}{\Gamma \vdash \mathcal{U}[\mathsf{while}\ (\mathit{expr})\ \{\alpha\}]\phi,\ \Delta}$$

where b is a new Boolean variable and α' is the result of inserting the statement $b = \mathit{expr}$; in front of every continue-statement in the loop body α.

Fig. 1 shows the KeY prover window with the example from Sect. 7. The lower left pane displays the proof tree with three open branches corresponding to the three premisses of the invariant rule. For better user interaction, the goals are labelled with “Invariant Initially Valid”, “Body Preserves Invariant”, and “Use Case”. The right pane shows the sequent that is currently under consideration. Rules can be applied automatically by pressing the button in the upper left corner or interactively using the mouse: pointing at a certain term or formula highlights the respective item and pressing the left mouse button offers (only) those rules that are applicable at this position.

7 Extended Example

The example in this section is based on the calculus and the loop rule implementation in the KeY tool, i.e., the target programming language is Java (more precisely Java Card but the difference does not matter here), and the specification language is UML/OCL [17,16] or—as in the example—JML [15].

The JML specification of the Java method swapMax (see Fig. 2) states that, if the pre-condition (*requires* clause, lines 1–2) consisting of

a. *a* is not *null* and
b. the length of *a* is greater than zero

holds in the beginning, then after the execution of swapMax the following post-condition (*ensures* clause, lines 3–7) holds:

a. there exists an index such that the elements of *a* at position index and zero are swapped,
b. the element at position zero is greater than or equal to the elements at all other positions, and
c. all elements at positions different from zero and the index remain unchanged.

In other words, the post-condition says that the method swaps the greatest element and the element at position zero and all other elements remain unchanged. In JML post-conditions, one can use *old(expr)* to refer to the value of *expr* at the beginning of the method.

```
    /*@ requires
 2    @ a!=null && a.length > 0;
      @ ensures
 4    @ (\ exists int idx;  0 <= idx && idx<\old(a).length;
      @ a[idx]==\old(a)[0] && a[0]==\old(a)[idx] &&
 6    @ (\ forall int i; 0 <= i && i<\old(a).length;
      @ a[0] >= a[i] && (i!=0 && i!=idx ==> a[i]==\old(a)[i])));
 8    @*/
    void swapMax(int[] a) {
10    int counter = 0, int index = 0;
      /*@ loop_invariant
12      @ 0<=counter && counter<=a.length &&
        @ 0<=index && index<a.length &&
14      @ (\ forall int x; x>=0 && x<counter; a[index]>=a[x]);
        @ assignable index, counter;
16      @*/
      while (counter<a.length) {
18      if (a[counter] > a[index])
          index = counter;
20      counter = counter+1;
      }
22    int tmp = a[index];
      a[index] = a[0];
24    a[0] = tmp;
    }
```

Fig. 2. JML specification and JAVA implementation of method swapMax

The body of swapMax is divided into two parts. In the first part (lines 17–21), we iterate through the elements of array *a* and store the index of the greatest element in variable *index*. In the second part (lines 22–24), the elements at position *index* and zero are swapped.

Using JML, it is possible to annotate loops with loop invariants. The invariant in our example states that

a. *counter* and *index* stay in the correct range (lines 12–13), and
b. the element at position *index* is greater than or equal to all elements at positions zero to $counter - 1$ (line 14).

The only locations that are modified in the loop body are *index* and *counter*. To make this information explicit we use the *assignable* clause of JML (line 15).[1]

The KeY tool is able to use the invariant given as annotation in the code when applying an invariant rule. Our example can be proved almost fully automatically using the above invariant. The only user interaction required is the

[1] Following the JML standard [15] assignable clauses, which are the JML-equivalent of modifier sets, are restricted to methods. Recent discussions on the JML mailing list suggest that the assignable clause will also be applicable to loops in the future.

simple instantiation of the existential quantifier in the post-condition with the term *index* at the end of the proof.

Using the traditional invariant rule, the above invariant is not strong enough. Fig. 3 shows the additional conjuncts that have to be added to the invariant in order to prove the post-condition using the classical loop rule.

```
  /*@ (\forall int x; x>=0 && x<counter; a[x]==\old(a)[x]) &&
2   @ a.length==\old(a.length) && a.length>0 && a==\old(a) && a!=null
    @*/
```

Fig. 3. Additional conjuncts for the invariant preserving the context information

Line 1 expresses that the elements in array *a* are the same before and after execution of the loop body. Line 2 states that the length of the array does not change and is greater than zero and that the array reference *a* is an invariant of the loop and is different from *null*.

As one can see, the invariant for the traditional rule is more complicated and has to contain information not directly related to the while loop (there is an indirect relationship, however, since the additional conjuncts express what the loop does *not* do).

8 Conclusion

We have extended the idea of modifier sets from to method specification to loops, and have defined a DL loop invariant rule that makes use of such change information. Our new definition of *quantified* modifier sets overcomes the restrictions from [6], where modifier sets could only describe location sets of fixed length. The new loop rule has been implemented in the KeY System and in experiments has proved to be a great improvement over rules not using change information.

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. *Software and System Modeling*, 4:32–54, 2005.
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. de Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer, 2000.
3. K. R. Apt. Ten years of Hoare logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 1981.
4. 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.

5. B. Beckert, S. Schlager, and P. H. Schmitt. An Improved Rule for While Loops in Deductive Program Verification. Technical Report in Computing Science 2005-26, Fakultät für Informatik, Universität Karlsruhe, Germany, September 2005. Available at `http://i12www.ira.uka.de/~schlager/publications/TRInvRule.ps.gz`.
6. B. Beckert and P. H. Schmitt. Program verification using change information. In *Proceedings, Software Engineering and Formal Methods (SEFM), Brisbane, Australia*, pages 91–99. IEEE Press, 2003.
7. A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, 1995.
8. 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.
9. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.
10. M. D. Ernst. *Dynamically Discovering Likely Program Invariants.* PhD thesis, University of Washington, Seattle, August 2000.
11. D. Harel. Dynamic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*. Reidel, 1984.
12. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* The MIT Press, 2000.
13. D. Kozen and J. Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 14, pages 89–133. Elsevier, 1990.
14. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer Academic Publisher, 1999.
15. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06z, Iowa State University, Department of Computer Science, December 2004.
16. Object Modeling Group. *UML 2.0 OCL Specification*, October 2003.
17. Object Modeling Group. *UML 2.0 Superstructure Specification*, October 2004.
18. J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 23–32, Newport Beach, CA, USA, November 2–4, 2004.
19. P. Rümmer. A Language for Sequential, Parallel and Quantified Updates of First-order Structures, 2005. Forthcoming.
20. R. Sasse. Proof obligations for correctness of modifies clauses. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, 2004. Available at `http://i12www.ira.uka.de/~key/doc/2004/sasse2004.pdf`.
21. N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Proceedings, Logic for Programming, Artificial Intelligence, and Reasoning(LPAR)*, LNAI 3452, pages 398–414. Springer, 2004.
22. F. Spoto and E. Poll. Static analysis for JML's assignable clauses. In *Proceedings, Foundations of Object-Oriented Languages (FOOL10)*, 2003.