

Could we have chosen a better Loop Invariant or Method Contract?

Christoph Gladisch*

University of Koblenz-Landau
Department of Computer Science
Germany

Abstract. The method contract and loop invariant rules (*contract rules*) are an important software verification technique for handling method invocations and loops. However, if a verification condition resulting from using a contract rule turns out to be falsifiable, then the user does not know if she could have chosen a stronger contract to verify the program or if the program is not verifiable due to a software bug. We approach this problem and present a novel technique that unifies verification and software bug detection.

1 Introduction

Software verification is an iterative and time consuming task. A software correctness proof usually does not succeed on the first proof attempt because often a program has either a) an error, i.e. the program does not satisfy the specification (Figure 1.1), or b) the program is correct but inappropriate *auxiliary* formulas, i.e. loop invariants or method contracts, are used. The user then does not know if she should search for a bug in the program or search for a different loop invariant or method contract.

The technique described in this paper tries to show the correctness of a program and in case of verification failure it tries to show program incorrectness, i.e. it tries to exclude case (b) as the source for verification failure by showing case (a). Furthermore, we propose a unified method for verification and bug detection that is based on verification and on failed proof attempts. The idea is that even if a verification proof fails, it contains useful information that can help to find a program error, if one exists, and in this way the work for verification is reused for bug finding – subsuming testing. This is basically done by checking if an unproved verification condition has a counterexample, i.e., if it is falsifiable. The problem of existing techniques following this approach is that such counterexamples don't necessarily imply a program error but they only guide the user to find the problem.

The contribution of this paper is a method that checks if a falsifiable verification condition implies a program error. The bottlenecks that prevent from concluding the existence of program errors directly from a falsifiable verification

* gladisch@uni-koblenz.de

— JAVA + JML (1.1) —	— JAVA + JML (1.2) —
<pre> /*@ public normal_behavior requires x>=0; ensures \result * \result <= x && (\result+1)*(\result+1)>x; diverges true; @*/ public int sqrtA(int x){ int i=0; /*@ loop_invariant (i-1)*(i-1)<=x i==0; modifies i; @*/ while(i*i<=x){i++;} return i;} //bug </pre>	<pre> /*@ public normal_behavior requires x>=0; ensures \result * \result <= x && (\result+1)*(\result+1)>x; diverges true; @*/ public int sqrtB(int x){ int i=0; /*@ loop_invariant (i-1)*(i-1)<=x; modifies i; @*//weak invariant while(i*i<=x){i++;} return i-1; } </pre>
— JAVA + JML —	— JAVA + JML —

Fig. 1. Motivating examples

condition are the so called contract rules (loop invariant rule, method contract rule) of the verification calculus. These rules are important to reason about programs with loops and method invocation. However, in contrast to first-order logic rules and most other rules that transform the program to a first-order formula, the contract rules are *not always* falsifiability preserving. Falsifiability preservation is the property of the rules that enables us concluding that the verification condition at the beginning of the proof attempt is falsifiable. The core of our method is to check if contract rules that occur in a sequence of rule applications are falsifiability preserving for given contracts and in this way it enables us to conclude the existence of program error from falsifiable verification conditions.

For example, trying to verify the programs in listings (1.1) and (1.2) using the given loop invariants fails because the loop invariant rule, as will be shown in this paper, creates a falsifiable verification condition. The reason for the failure is however different in both cases. The method `sqrtA()` has a bug and cannot be verified with any loop invariant whereas method `sqrtB()` is correct but the loop invariant is inappropriate. The method described in this paper tries to show if a contract rule with a given loop invariant or method contract is falsifiability preserving. In listing (1.1) this is the case and indeed our approach detects that the method `sqrtA()` has a bug.

Whether a contract rule is falsifiability preserving or not depends on the *strength* of the auxiliary formulas, e.g. loop invariant, method contract which instantiate the loop invariant or method contract rule in the respective case. The stronger an auxiliary formula is, the more detailed information it contains about the loop or method invocation it describes. Consequently the described method is capable to detect bugs only if contracts are sufficiently strong. The method does not check if a stronger contract exists that would complete the proof or make a contract rule falsifiability preserving. In this paper we further assume the existence of a method that determines if a first-order logic formula is falsifiable. For this an SMT solver can be used or the user can simply decide

if a formula is falsifiable. The latter scenario could be considered in interactive proving, for example.

Plan of the Paper. Section 2 introduces the foundation of our approach which are a logic for reasoning about programs and a verification calculus. Section 3 describes our approach in detail. In Section 3.1 the falsifiability of contract rule premises is analysed. In Section 3.2 the approach is described in a general way. Sections 3.1 and 3.2 form the basis for our main contribution that is described in Section 3.3 where also our central theorem is proved. An example of using our approach is given in Section 4. Related work is described in Section 5 and finally Section 6 concludes our work.

2 Dynamic Logic and the Verification Calculus

2.1 Overview of JAVA CARD DL

The work presented here is based on a Dynamic Logic [15] for JAVA CARD (JAVA CARD DL [4]) but it can be adapted to similar logics like Hoare Logic and for different programming languages. JAVA CARD DL is the program logic of the KeY-System [6, 1], which is a combined interactive and automatic verification and test generation system with symbolic execution rules for a subset of JAVA.

Dynamic Logic is an extension of first-order logic where a formula φ can be *preended* by the modal operators $\langle \mathbf{p} \rangle$ and $[\mathbf{p}]$ for every program \mathbf{p} . The formula $[\mathbf{p}]\varphi$ means that if \mathbf{p} terminates, then φ holds in the state after the execution of \mathbf{p} . As we consider only sequential and deterministic JAVA programs the meaning of $\langle \mathbf{p} \rangle \varphi$ is that the program terminates and that $[\mathbf{p}]\varphi$ is true. Thus $[\mathbf{p}]\varphi \wedge \langle \mathbf{p} \rangle true$ is equivalent to $\langle \mathbf{p} \rangle \varphi$. In the following we denote the set of DL-formulas by Fml , and the set of first-order logic formulas by Fml_{FOL} . All variables in formulas are bound with quantifiers.

An implication of the form $pre \rightarrow [\mathbf{p}]post \in Fml$ with $pre, post \in Fml_{FOL}$ corresponds to the Hoare triple $\{pre\}\mathbf{p}\{post\}$ in Hoare logic. If the precondition pre is true in the state before the execution of the program and the program terminates, then the postcondition $post$ holds after the execution of the program; if the precondition does not hold before the execution of the program, then no statement is made about the post-state. The implication $pre \rightarrow \langle \mathbf{p} \rangle post$ states additionally that \mathbf{p} terminates. Dynamic logic allows pre and $post$ to *contain* programs in contrast to Hoare logic: if $pre, post \in Fml$ then $pre \rightarrow [\mathbf{p}]post \in Fml$.

Program variables are modelled in JAVA CARD DL as *non-rigid* function symbols $f \in \Sigma_{nr} \subset \Sigma$ of the signature Σ . Different program states are therefore realized as different first-order interpretations of the non-rigid function symbols. For instance let $a, o, i, acc_{\square} \in \Sigma_{nr}$. In this case a program variable \mathbf{a} is represented by a logical non-rigid constant a , an expression like $\mathbf{o}.\mathbf{a}$, that accesses an object attribute, is modeled as the term $a(o)$, and in case of an array access $\mathbf{o}.\mathbf{a}[\mathbf{i}]$ the corresponding term is $acc_{\square}(a(o), i)$. We use constant domain semantics which means that in all states terms are evaluated to values of the same universe. Logical variables are always *rigid*, i.e., they have the same value in all program

states. We write $\models \varphi$ to denote that the formula φ is valid, i.e. it is true under all interpretations.

JAVA CARD DL extends classical Dynamic Logic [15] with updates [4]. An update represents a state change and has the form $\{l_1 := t_1 \parallel \dots \parallel l_m := t_m\}$. Each elementary update $l_i := t_i$, with $1 \leq i \leq m$, assigns the value of the term t_i (also called symbolic value) to the term l_i . The top-level function symbol of l_i must be non-rigid since the update changes its interpretation.

Updates allow an efficient way of handling the aliasing problem and making side-effects explicit. Consider the formula $i \doteq 0 \rightarrow \langle \mathbf{i}++; \mathbf{a}[\mathbf{i}]=\mathbf{i} \rangle acc_{\square}(a, i) \doteq 1$. Transforming the modal operator of this formula into an update yields the formula $i \doteq 0 \rightarrow \{i := i + 1 \parallel acc_{\square}(a, i + 1) := i + 1\} acc_{\square}(a, i + 1) \doteq 1$ that can be simplified using update application to $i \doteq 0 \rightarrow i + 1 \doteq 1$.

A (quantified) update [20] of the form $\{f(x_1, \dots, x_m) := g(x_1, \dots, x_m)\}$ assigns all values of $g \in \Sigma$ to $f \in \Sigma$ at the respective argument positions and is effectively a substitution $[g/f]$. In contrast to substitutions, updates allow referring to a pre- and a post-state. We abbreviate such an update with $\{f := g\}$. Furthermore the notation $\{A := B\}\varphi$ with $A, B \subset \Sigma$ is equivalent to the substitution $[b_1/a_1, \dots, b_n/a_n]\varphi$ where the function symbols $a_1, \dots, a_n \in A$ are replaced by the function symbols $b_1, \dots, b_n \in B$ respectively.

2.2 Modifier Sets and Anonymous Updates

A modifier set for a program is a set of function symbols that model program variables. The purpose of using a modifier set as part of a specification is to specify which program variables are modified by the program.

Definition 1. *The minimal modifier set of a program \mathbf{p} is denoted by $Mod(\mathbf{p}) \subset \Sigma$ and it consists exactly of those function symbols that can be modified by \mathbf{p} .*

A correct modifier set $M \subset \Sigma_{nr}$ contains at least the function symbols that are modifiable by \mathbf{p} , i.e. $M \supseteq Mod(\mathbf{p})$.

A modifier set $M \subset \Sigma_{nr}$ can be used to create an *anonymous update* of the form $\{M := M_{sk}\}$ (or shorter $\{M_{sk}\}$) which replaces each function symbol $f \in M$ by a fresh function symbol $f_{sk} \in M_{sk}$. Anonymous updates enable us to replace programs by abstractions. Consider the formula $a \doteq 7 \rightarrow \langle \mathbf{m}() \rangle (a \leq b \vee b \leq 7)$. If it is known that the method $\mathbf{m}()$ terminates and modifies only \mathbf{b} , then the modal operator can be replaced by the anonymous update $\{\{b_{sk}\}\}$ resulting in the formula $a \doteq 7 \rightarrow \{b := b_{sk}\}(a \leq b \vee b \leq 7)$. Update application then yields $a \doteq 7 \rightarrow a \leq b_{sk} \vee b_{sk} \leq 7$. Note that more general updates, than those used in the paper representing simple substitutions, allow also more powerful anonymous updates where the modifier set may contain terms. Then, assuming that the method $\mathbf{m2}()$ modifies only $\mathbf{q.x}$, i.e. $x(q)$, it is possible to prove, e.g., $a(p) \doteq 7 \wedge p \neq q \rightarrow \langle \mathbf{m2}() \rangle a(p) \doteq 7$.

2.3 Specifications

Definition 2. A specification is a triple $(pre, post, M)$ where $pre \in Fml_{FOL}$ is the precondition, $post \in Fml_{FOL}$ is the postcondition and $M \in \Sigma$ is a modifier set.

A specification typically describes the behavior of a method but it can specify the behavior of any statement or sequence of statements. For instance a loop invariant $I \in Fml$ is the pre- and postcondition of a loop's body and the loop itself. A stronger postcondition of the loop is $I \wedge \neg lc$ where $lc \in Fml$ is the loop condition, i.e. the loop iterates while lc is true. The specification of a loop is therefore the triple $(I, I \wedge \neg lc, M)$.

2.4 The underlying verification technique

The underlying software verification technique of our approach is based on a sequent calculus. The calculus consists of sequent rules of the form

$$\frac{\Gamma_1 \Rightarrow \Delta_1 \quad \dots \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma_0 \Rightarrow \Delta_0}$$

where $\Gamma_i \Rightarrow \Delta_i$, with $i \in \{0, \dots, n\}$, are the sequents, or verification conditions and Γ_i and Δ_i are sets of formulas. A sequent has the form

$$\gamma_1, \dots, \gamma_k \Rightarrow \delta_1, \dots, \delta_l$$

with $\gamma_1, \dots, \gamma_k, \delta_1, \dots, \delta_l \in Fml$ and is equivalent to the implication

$$((\gamma_1) \wedge \dots \wedge (\gamma_k)) \rightarrow ((\delta_1) \vee \dots \vee (\delta_l))$$

When the context is clear, we use Δ_i and Γ_i to denote formulas: Γ_i denotes the conjunction of the formulas it includes and Δ_i denotes respectively a disjunction. Similarly we allow combining sequents with formulas with the obvious meaning.

The sequents $\Gamma_i \Rightarrow \Delta_i$, with $i \in \{1, \dots, n\}$, are the rule premises and the sequent $\Gamma_0 \Rightarrow \Delta_0$ is the rule conclusion. Soundness of the calculus ensures that if all rule premises are valid, then also the rule conclusion is valid.

In sequent calculus, proofs are constructed by applying sequent rules bottom-up, i.e., in order to prove $\Gamma_0 \Rightarrow \Delta_0$ the new proof obligations $\Gamma_1 \Rightarrow \Delta_1 \quad \dots \quad \Gamma_n \Rightarrow \Delta_n$ are created by rule application that have to be proved instead. In this way a tree structure is created, also called a proof tree, consisting of proof branches of the form S_0, \dots, S_n where S_0 is the root sequent of the tree and S_n is the leaf sequent of a branch. We use the notation S_i to refer to a node or the sequent contained in that node depending on the context.

In a verification proof the *root* of a proof tree has the form

$$\Gamma \Rightarrow \{U\} \langle p \rangle_{\pi} post, \Delta \tag{1}$$

where \mathbf{p} is the *target program*, the formula $\Gamma \wedge \neg \Delta$, with $\neg \Delta \equiv \bigwedge_{\delta \in \Delta} \neg \delta$, is the *current precondition*, and $\langle \rangle$ stands for $\langle \rangle$ or $\llbracket \rrbracket$ depending on whether total or

partial correctness is to be proved in the respective case. The subscript π denotes the *context* which is a stack of method invocations and JAVA blocks and is needed for the handling of jump statements like **return**, **throw**, **break**, and **continue**. In the KeY-system the context is embedded inside the modal operator. We omit writing the context whenever possible. The current precondition of the root consists of the precondition of the requirement specification and U is initially empty.

For example the root in a verification attempt of the method `sqrtA()` from Figure 1 with the given requirement specification is (let $(X)^2$ denote $X * X$):

$$x \geq 0 \Rightarrow \{ \} [\text{sqrtA } (x)] \overbrace{((\backslash result)^2 \leq x \wedge x < (\backslash result + 1)^2)}^E : S_0$$

Applying *symbolic execution* rules on the program part results in the sequents

$$x \geq 0 \Rightarrow [i=0;]_{\pi} [\text{while}(c)\{i++; \}]_{\pi} [\text{return } i]_{\pi} E : S_1$$

$$x \geq 0 \Rightarrow \{i := 0\} [\text{while}(c)\{i++; \}]_{\pi} [\text{return } i]_{\pi} E : S_2$$

where the context π consists of `sqrtA`. At this point consider using the loop invariant rule from Figure 2 (see also [7]) with the contract

$$\overbrace{((i-1)^2 \leq x \vee i \doteq 0)}^I, \overbrace{((i-1)^2 \leq x \vee i \doteq 0) \wedge i^2 > x, \{i\}}^{I \wedge \neg c, M}$$

Recall that the contract of a method is $(pre_m, post_m, M)$, the contract of a loop is $(I, I \wedge \neg c, M)$, and we assume that M is a correct modifier set in the respective case throughout the paper. The resulting rule premises are

$$\begin{aligned} \mathbf{1:} & \overbrace{x \geq 0}^{\Gamma} \Rightarrow \overbrace{\{i := 0\}}^U \overbrace{(i-1)^2 \leq x \vee i \doteq 0}^I & : PO1_{S_2} \\ \mathbf{2:} & x \geq 0 \Rightarrow \{i := 0\} \{i := i_{sk1}\} (I \wedge c \rightarrow [i++]_{\pi} I) & : PO2_{S_2} \\ \mathbf{3:} & x \geq 0 \Rightarrow \{i := 0\} \{i := i_{sk1}\} (I \wedge \neg c \rightarrow [\text{return } i]_{\pi} E) & : PO3_{S_2} = S_3 \end{aligned}$$

where the abbreviations $PO1_{S_2}$, $PO2_{S_2}$, and $PO3_{S_2}$ are defined as follows.

Definition 3. *Definition of a contract rule premise occurrence (PO):*

- $PO1_S$, $PO2_S$, and $PO3_S$ denote respectively the 1st, 2nd, and 3rd premise of the contract rule application at the sequent S in a proof tree.
- $PO1_S$ is the union $PO1_S \cup PO2_S$

The sequents $PO1_{S_2}$ and $PO2_{S_2}$ are proved by further rule applications. Symbolic execution on S_3 is continued in Section 4.

Note that in case of recursion, application of the method contract rule does not terminate unless it is combined with induction.

Method Contract Rule	Loop Invariant Rule
1: $\Gamma \Rightarrow \{U\}pre_m, \Delta$	1: $\Gamma \Rightarrow \{U\}I, \Delta$
2: $\Gamma \Rightarrow \{U\}(pre_m \rightarrow \langle m() \rangle post_m), \Delta$	2: $\Gamma \Rightarrow \{U\}\{M\}(I \wedge c \rightarrow [b]I), \Delta$
3: $\Gamma \Rightarrow \{U\}\{M\}(post_m \rightarrow post), \Delta$	3: $\Gamma \Rightarrow \{U\}\{M\}((I \wedge \neg c) \rightarrow post), \Delta$
$\Gamma \Rightarrow \{U\}\langle m() \rangle post, \Delta$	$\Gamma \Rightarrow \{U\}[while(c)\{b;\}]post, \Delta$

Fig. 2. Contract Rules

3 Unified Verification and Bug Detection

3.1 Falsifiability of Contract Rule Premises

The method contract and loop invariant rules (*contract rules*) (see Figure 2) are a software verification technique for generating verification conditions, i.e. proof obligations, for programs with method calls and loops. However, if a verification condition resulting from using a contract rule turns out to be falsifiable, then one cannot always conclude if the target program has an error or not. This section explains how to interpret falsifiable proof branches, i.e. rule premises, that these rules generate upon rule application. Both rules are considered in parallel.

Branch 1 This branch ensures that the precondition of the contract is satisfied in the prestate of the method or loop. Falsifiability of this branch can be interpreted in two ways. If the verification approach requires the precondition to be satisfied, then falsifiability of this branch implies a program bug in the calling code. Otherwise, the contract is too weak to prove anything and the user has the option to strengthen the contract by weakening the formula pre_m resp. I , or to strengthen $\Gamma \wedge \neg \Delta$ by strengthening contracts occurring earlier in the proof tree.

Branch 2 This branch ensures that the contract of the method or loop is correct. Falsifiability of this branch can be interpreted in two ways depending on the verification approach. If the contract is regarded as an auxiliary and not as a requirement specification, then falsifiability of this branch means that the contract is wrong and the user has to choose a correct contract. If the contract is regarded as a requirement specification, then falsifiability of this branch implies an error in the target program and the user has to fix the target program.

Branch 3 In this branch the postcondition of the contract plays a surrogate for the description of the state transition by the target program. If this branch is falsifiable, then either the target program is not correct or the contract is just too weak to complete the proof.

Conclusion. Whether the falsifiability of branch 1 or 2 is caused by an error in the target program or in the contract depends on the role of the contract in the particular verification approach and is clear in the respective case. Falsifiability of branch 3 in contrast has in both cases an ambiguous meaning. The contract,

either auxiliary or required, may be too weak to complete the proof or the program has an error. Our main contribution, described in the next section, is to help to distinguish between these cases by checking if the program has an error if the sequent in third branch is falsifiable.

3.2 Falsifiability Preservation Checking

The approach described in this paper is to try to show the correctness of a program and in case of verification failure to show program incorrectness or to guide the user (see Algorithm 1). The main *contribution* of this paper is however a technique to find program errors based on information contained in proof trees of failed verification attempts. If a verification condition or sequent, is falsifiable, i.e. it has a counterexample, then we check a falsification preservation property of a branch (Def. 4), such that if the property is valid, then it is sound to conclude that the program has an error. The user then knows that she could *not* have chosen a better loop invariant or method contract to succeed the verification attempt and instead she should fix the bug in the program or its specification.

Definition 4. *Definition of falsifiability preservation conditions (FP):*

- The falsifiability preservation condition FP_P of a rule premise P is the formula $\neg P \rightarrow \neg C$, where C is the rule conclusion. A rule premise is falsifiability preserving if FP_P is valid.
- A rule R is falsifiability preserving iff for all premises P of R , $\models FP_P$.
- The falsifiability preservation condition of a sequence of sequents S_i, \dots, S_j , in a branch S_0, \dots, S_n , with $0 \leq i < j \leq n$, is the formula $\neg S_j \rightarrow \neg S_i$ denoted by $FP_{S_i}^{S_j}$. FP_S^S is trivially true.
- A proof branch S_0, \dots, S_n is falsifiability preserving iff $FP_{S_0}^{S_n}$, or short FP^{S_n}

In case a verification attempt has failed Algorithm 1 iterates via the outer loop over the open proof tree branches and analyses them in the inner loop. The inner loop (Lines 8-24) operates only on falsifiable branches and tries to prove falsifiability preservation from a leaf to the root. If premise 1 or 2 of a contract rule application is falsifiable (Lines 12-15), then the user is guided by the corresponding description in Section 3.1. The main part is the analysis of premise 3 of a contract rule application in Line 17. Checking falsifiability preservation of this premise can be done, e.g., using the formula $FP_{S_i}^{S_n}$ (see Def. 4). If the check is successful and S_n is a falsifiable sequent of a proof branch S_0, \dots, S_n , then it is easy to see that also S_0 is falsifiable. Consequently, if S_0 has the form (1) then the target program has a bug. Otherwise if checking FP^{S_n} fails, then the algorithm may return further information for the user to proceed.

3.3 Special Falsifiability Preservation Checking

This section describes our main contribution. We check with a special falsifiability preservation condition whether contracts that are used in contract rule

Algorithm 1 tryToVerifyOrToFindABug(root)

Require: All rules but the contract rules are known to be falsifiability preserving.

- 1: Create proof tree PT for $root$
- 2: **if** all branches in PT are closed **then**
- 3: **return** “verified”
- 4: **end if**
- 5: **for all** $B \in$ open branches of PT **do**
- 6: **if** B is falsifiable /*determined by user, SMT solver, or other method*/ **then**
- 7: let $(S_0, \dots, S_n) = B$
- 8: **for** $i = n - 1$ to 0 **do**
- 9: **if** S_i is the root **then**
- 10: **return** “program has a bug on trace ”+B
- 11: **else if** a contract rule was applied at S_i **then**
- 12: **if** S_{i+1} is $PO1_{S_i}$ /*i.e., 1st premiss of contract rule app*/ **then**
- 13: $msg = msg \cup$ “ S_{i+1} is falsifiable” + Description Branch 1 (Sec 3.1)
- 14: **else if** S_{i+1} is $PO2_{S_i}$ /*i.e., 2nd premiss of contract rule app*/ **then**
- 15: $msg = msg \cup$ “ S_{i+1} is falsifiable” + Description Branch 2 (Sec 3.1)
- 16: **else if** S_{i+1} is $PO3_{S_i}$ /*i.e., 3rd premiss of contract rule app*/ **then**
- 17: $fp_i^n =$ check falsifiability preservation from S_n to S_i according to Definition 4 or Definition 5.
- 18: **if** fp_i^n is not true **then**
- 19: $msg = msg \cup$ “contract at node (i) is too weak to show program correctness or incorrectness”
- 20: $i=0$ //terminate the inner loop
- 21: **end if**
- 22: **end if**
- 23: **end if**
- 24: **end for**
- 25: **end if**
- 26: **end for**
- 27: **return** msg

applications on a given branch are strong enough to reveal a software bug. A *conventional* proof of the formula FP^{S_n} , i.e. $\neg S_n \rightarrow \neg S_0$, that ensures falsification preservation of branch S_n , would require a transformation of the program in S_0 into a first-order logic formula requiring, e.g., symbolic execution. Instead, for a unified verification and bug detection approach we regard the branch S_0, \dots, S_n that is created by the verification attempt as a *test run* with symbolic values and we reuse information contained in this branch to prove its falsifiability preservation. This is achieved by replacing in Line 17 of Algorithm 1 the formula $FP_{S_i}^{S_n}$ with the more sophisticated formula $SFP_S^{S_n}$ that is defined next.

Definition 5. Let $(S_0, \dots, S_n) = B$ be a branch and S_i with $0 \leq i < n$ be a sequent that has either the form (case 1):

$$\Gamma_i \Rightarrow \{U\}\langle \mathbf{p} \rangle \varphi, \Delta_i$$

or the form (case 2):

$$\Gamma_i \Rightarrow \{U\}[p]\varphi, \Delta_i$$

and on S_i a contract rule is applied with the contract (pre, post, M). Let $M \supseteq \text{mod}(p)$. S_{i+1} is the 3rd branch of the contract rule, i.e. $S_{i+1} = PO3_{S_i}$.

The special falsifiability preservation condition $SFP_{S_i}^{S_n}$ is the conjunction of

$$((\{M^1 := M^2\}S_n) \wedge \{U\}\{M^2\}post) \rightarrow S_n \quad (2)$$

$$(\neg S_n \wedge \Gamma_i \wedge \neg \Delta_i) \rightarrow \{U\}\Psi \quad (3)$$

$$\neg S_n \rightarrow (\Gamma_i \wedge \neg \Delta_i) \quad (4)$$

where $\Psi = \text{true}$ in case 1 and $\Psi = \langle p \rangle \text{true}$ in case 2. The third formula is optional if it is known to be valid for all instances of S_n, Γ_i and Δ_i that may occur in a proof tree (as it is the case for here regarded calculus).

Theorem 1. Assuming that R is the contract rule applied at S_i with $0 \leq i < n$; the 1st and 2nd premises of R are proved; and $FP_{S_{i+1}}^{S_n}$ holds, then

$$\models SFP_{S_i}^{S_n} \text{ implies } \models FP_{S_i}^{S_n}$$

Theorem 1 is the key for using the formula SFP to prove the falsifiability preservation of a branch that contains occurrences of the third premiss of contract rules. The implication of the theorem is that the formula $SFP_{S_i}^{S_n}$, or even just Formula 2 as described below, can replace the formula $FP_{S_i}^{S_n}$, i.e. $\neg S_n \rightarrow \neg S_0$, in Line 17 of Algorithm 1. In contrast to $FP_{S_i}^{S_n}$, the formula $SFP_{S_i}^{S_n}$ contains no program parts, or it contains only program parts that have not yet been symbolically executed on the branch with the leaf S_n . This is achieved because $SFP_{S_i}^{S_n}$ has no occurrence of φ that may contain the rest of the program following p . Since $SFP_{S_i}^{S_n}$ is mainly built from formulas in S_n , all the symbolic execution that took place up to S_n (by a verification attempt) is *reused*. These properties of $SFP_{S_i}^{S_n}$ support the idea of unified verification and bug detection.

The main subformula of $SFP_{S_i}^{S_n}$ is (2). This formula extends the leaf S_n of branch B with the conjunction $((\{M^1 := M^2\}S_n) \wedge \{U\}\{M^2\}post)$ that is mainly built from an updated version of S_n and the postcondition from the contract rule applied at S_i . Formula (3) ensures that in case of a partial correctness verification attempt non-terminating programs are recognized as correct programs. In practice, however, even if a partial correctness proof is created non-terminating programs are regarded as incorrect. Formula (3) is therefore optional. By ignoring the formula non-terminating programs in partial correctness proofs are regarded as incorrect which is not sound but usually a welcome behavior. Formula (4) is optional as well because, e.g., in the KeY calculus with the contract rules in Figure (2) the Formula (4) is always valid. The formula is however given here to increase the generality of the approach.

We regard the requirement that the 1st and 2nd branch of the respective contract rule application, i.e. $PO12_S$, must be closed, as a minor problem because

trying to close these branches is part of the verification process in Algorithm 1 so that no additional work is done for bug detection. Furthermore if the branches $PO12_S$ are not proved, then the algorithm provides guidelines for the user on how to proceed. The second requirement, i.e. $\models FP_{S_{i+1}}^{S_n}$, is ensured by an induction principle of the inner loop: as the loop iterates the program variable i is decreased and the validity of $FP_{S_{i+1}}^{S_n}, FP_{S_i}^{S_n}, \dots$ is ensured.

Proof Sketch of Theorem 1. In this proof the formula $SFP_{S_i}^{S_n}$ is constructed from a formula that is equivalent to $FP_{S_i}^{S_n}$ while making use of the assumptions in Theorem 1. The construction of $SFP_{S_i}^{S_n}$ ensures that $SFP_{S_i}^{S_n}$ implies $FP_{S_i}^{S_n}$.

By assumption of the theorem, $FP_{S_i}^{S_n}$ is valid and has the form

$$\neg S_n \rightarrow \neg \overbrace{(\Gamma_i \rightarrow ((\{U\}\{p\})\varphi) \vee \Delta_i)}^{S_i} \quad (5)$$

Equivalence transformations yield the conjunction of the two formulas

$$\neg S_n \rightarrow (\Gamma_i \wedge \neg \Delta_i) \quad (6)$$

$$(\neg S_n \wedge (\Gamma_i \wedge \neg \Delta_i)) \rightarrow \{U\}[\{p\}]\neg\varphi \quad (7)$$

where (6) is equal to (4). Formula (7) is equivalent to the conjunction of the following two formulas

$$(\neg S_n \wedge \Gamma_i \wedge \neg \Delta_i) \rightarrow \{U\}[p]\neg\varphi \quad (8)$$

$$(\neg S_n \wedge \Gamma_i \wedge \neg \Delta_i) \rightarrow \{U\}\Psi \quad (9)$$

where (9) is equal to (3) and $\Psi = \langle p \rangle true$ or $\Psi = true$ depending on the modal operator of the target program. Let $(pre, post, M)$ be the contract of R that was by assumption applied on S_i . The same contract rule with the same contract is now applied on (8). This yields three branches of which the first two are already proved. The third branch is

$$(\neg S_n \wedge \Gamma_i \wedge \neg \Delta_i) \rightarrow \{U\}\{M^2\}(post \rightarrow \neg\varphi) \quad (10)$$

and can be transformed via equivalence transformations to

$$\overbrace{((\Gamma_i \wedge \neg \Delta_i) \wedge \{U\}\{M^2\}(post \wedge \varphi))}^{\Phi} \rightarrow S_n \quad (11)$$

This formula is *implied by* the formula

$$\overbrace{(\{M^1 := M^2\}S_n \wedge \{U\}\{M^2\}post)}^{\Phi'} \rightarrow S_n \quad (12)$$

that is equal to (2). It remains to show that $\Phi \rightarrow \Phi'$. By the assumption $\models FP_{S_{i+1}}^{S_n}$ the following sequent is valid (we use sequents for a compact notation)

$$S_{i+1} \Rightarrow S_n \quad (13)$$

This sequent is extended to the valid sequent

$$(\{M^1 := M^2\}S_{i+1}) \wedge \phi \Longrightarrow (\{M^1 := M^2\}S_n) \wedge \phi \quad (14)$$

where $\phi = (\Gamma_i \wedge \neg \Delta_i) \wedge \{M^1 := M^2\}((\Gamma_i \wedge \neg \Delta_i) \rightarrow \{U\}\{M^1\}post)$. According to Figure 2, $\Gamma_i = \Gamma_{i+1}$ and $\Delta_i = \Delta_{i+1}$. The rule R ensures that the skolem functions in M^1 cannot occur neither in $\{U\}$ nor in $(\Gamma_{i+1} \wedge \neg \Delta_{i+1})$ and the same applies to the skolem functions in M^2 . A set of equivalence transformations of (14) yields

$$(\Gamma_i \wedge \neg \Delta_i) \wedge \{U\}\{M^2\}(post \wedge \varphi) \Longrightarrow \{M^1 := M^2\}S_n \wedge \{U\}\{M^2\}post$$

showing that $\Phi \rightarrow \Phi'$. A detailed proof exists in a longer version of this paper. ■

4 Example

The example illustrates the workflow of Algorithm 1 and in particular the construction of the formula $SFP_{S_i}^{S_n}$ from a falsifiable sequent S_n and a sequent S_i on which a contract rule was applied. From the user's

perspective the ordinary underlying verification process is extended with automatic or manual falsifiability checking of unproved proof obligations and automatic falsifiability preservation analysis of open proof branches. Thus the required amount of user interaction is relative to the required amount of user interaction of the underlying verification approach.

4.1 Verification Attempt

The example from Section 2.4 where Algorithm 1 is applied on the method `sqrtA()` (Fig. 1) is continued with symbolic execution of sequent S_3 (or $PO\beta_{S_2}$).

$$\begin{aligned} \overbrace{x \geq 0}^{\Gamma_2 = \Gamma_3} &\Longrightarrow \overbrace{\{i := 0\}}^U \overbrace{\{i := i_{sk^1}\}}^{M^1} (I \wedge \neg c \rightarrow \overbrace{[\text{return } i]_\pi E}^\varphi) & : S_3 \\ x \geq 0 &\Longrightarrow \{i := i_{sk^1}\} (I \wedge \neg c \rightarrow [\text{return } i]_\pi E) & : S_4 \\ x \geq 0 &\Longrightarrow \{i := i_{sk^1}\} (I \wedge \neg c) \rightarrow \{i := i_{sk^1}\} [\text{return } i]_\pi E & : S_5 \\ x \geq 0, \{i := i_{sk^1}\} (I \wedge \neg c) &\Longrightarrow \{i := i_{sk^1}\} [\text{return } i]_\pi E & : S_6 \end{aligned}$$

Rule applications on $\{i := i_{sk^1}\}[\text{return } i]_\pi E$ yield

$$x \geq 0, \{i := i_{sk^1}\} (I \wedge \neg c) \Longrightarrow \overbrace{(i_{sk^1})^2 \leq x \wedge (i_{sk^1} + 1)^2 > x}^{\Delta_6} \quad : S_6$$

Update simplification on $\{i := i_{sk^1}\} (I \wedge \neg c)$ yields (with $\Delta_7 = \Delta_6$)

$$x \geq 0, ((i_{sk^1} - 1)^2 \leq x \vee i_{sk^1} \doteq 0) \wedge (i_{sk^1})^2 > x \Longrightarrow \Delta_7 \quad : S_7$$

Running this example with the KeY-system yields one open branch that extends the sequence (S_0, \dots, S_7) to $(S_0, \dots, S_7, \dots, S_n)$ with $n > 7$. The sequent S_n in the KeY-system is shown (copied directly from the prover with adapted notation):

$$\begin{aligned}
& x \geq 1 + i_{sk^1} * (-2), \\
& (i_{sk^1})^2 \geq 0, \\
& i_{sk^1} \geq 1, \\
& (i_{sk^1})^2 \geq 1 + x, \\
& (i_{sk^1})^2 \leq i_{sk^1} * 2 + x - 1, \\
& x \geq 0 \\
& \Rightarrow
\end{aligned}$$

Note that the succedent of S_n is empty. The verification attempt has failed at this point and the user does not know the reason for the failure. The description of the example continues in the following section.

4.2 Checking Falsifiability Preservation

The sequent S_n cannot be proved by the system. The next step is to decide if S_n is actually falsifiable which can be done by the user, an SMT solver or, e.g., by using the built-in satisfiability solver for test generation of KeY.

KeY finds a counterexample for the sequent S_n which implies that the sequent is falsifiable. Instead of creating a JUnit test that has to create the correct initial state and then execute the program that has already been symbolically executed, our approach is to check falsifiability preservation of the branch S_n . In contrast to traditional testing our method does not require a concrete counterexample for S_n .

Algorithm 1 proceeds in Line 5 and selects the branch (S_0, \dots, S_n) . As the inner loop of the algorithm iterates it decreases the program variable i from n until it reaches the value 2 because at S_2 the first occurrence of a contract rule application is found. In Line 16 the if-condition is satisfied. In Line 17 the falsifiability preservation is checked by proving of $SFP_{S_2}^{S_n}$. According to Definition 5, $SFP_{S_2}^{S_n}$ is the conjunction of the formulas (2), (3), and (4). Note that also the side condition that $PO1_{S_2}$ and $PO2_{S_2}$ are proved must be checked. This was however done as part of the verification attempt (see Section 2.4).

The Formula (4) is always valid when using the KeY-calculus (see Section 3.3). In our concrete case, e.g., Δ_2 denotes an empty disjunction that is equivalent to false and Γ_2 is $x \geq 0$. The formula $\neg S_n \rightarrow (\Gamma_2 \wedge \neg \Delta_2)$ simplifies to $\Gamma_n \rightarrow x \geq 0$ which is valid because $(x \geq 0) \in \Gamma_n$.

The role of Formula (3) is to prevent the provability of $SFP_{S_2}^{S_n}$ in case the target program does not terminate because non-termination implies program correctness when using the modal operator \Box , (case 2 in the definition). In practice the reason for using the modal operator \Box is, however, to simplify a verification attempt. Even when using the modal operator \Box the user may implicitly regard non-terminating programs as incorrect. For terminating programs this formula causes a computationally expensive proof obligation. For these reasons formula (3) can be ignored in practice.

The important formula to be proved is (2) that has the form

$$\overbrace{\{i_{sk^1} := i_{sk^2}\}}^{M^1 := M^2} S_n \wedge \overbrace{\{i := 0\}}^U \overbrace{\{i := i_{sk^2}\}}^{M^2} \overbrace{((i-1)^2 \leq x \vee i \doteq 0) \wedge i^2 > x}^{I \wedge \neg c} \rightarrow S_n$$

in our example. The sequent S_n has the form $\Gamma_n \Longrightarrow$, which is equivalent to $\neg\Gamma_n$. The formula can therefore be rewritten by equivalence transformations as

$$\begin{aligned} & (\{i_{sk^1} := i_{sk^2}\} \neg\Gamma_n \wedge \{i := 0\} \{i := i_{sk^2}\} ((i-1)^2 \leq x \vee i \doteq 0) \wedge i^2 > x) \rightarrow \neg\Gamma_n \\ & (\Gamma_n \wedge ((i_{sk^2} - 1)^2 \leq x \vee i_{sk^2} \doteq 0) \wedge (i_{sk^2})^2 > x) \rightarrow \Gamma'_n \end{aligned} \quad (15)$$

where Γ'_n is obtained by applying the update $\{i_{sk^1} := i_{sk^2}\}$ on Γ_n . $SFP_{S_2}^{S_n}$ simplifies therefore to a first-order logic formula that is built based on the leaf node S_n and the post condition of the contract. KeY proves formula 15 fully automatically in 187 prove steps.

The algorithm continues with two more loop iterations until $i = 0$ and exits at line 10 with the return message “program has a bug on trace (S_0, \dots, S_n) ”. By looking at the used symbolic execution rules the trace can be read as the program execution trace that leads to the bug. The user now knows that choosing a different contract would not have lead to a successful verification attempt because the program or the requirement specification has to be fixed.

5 Related Work

This work is an extension of the works [11, 5, 13, 10, 21] that were developed within the KeY-project [6]. In [11] verification-based testing is introduced as a method for deriving test cases from verification proofs. In [5] we present a white-box testing approach which combines verification-based specification inference and black-box testing enabling us to combine different coverage criteria. Both approaches consider the derivation of test cases based on loop invariants.

More directly this work is an extension of [13] that describes a test case generation technique for full feasible (program) branch coverage. In [13] branch coverage is ensured if contracts satisfy a *strength condition*. In this work we have extended this idea resulting in the *special falsifiability preservation condition* (*SFP*). Checking falsifiability preservation for contract rules is also possible with the *strength condition* of a contract. However, the *SFP* can be viewed as a *customized* strength condition for a contract that is valid in far more cases than the more general strength condition given in [13]. Furthermore, in contrast to the strength condition defined [13], *SFP reuses* symbolic execution that has already been performed. The latter property is the reason why we argue that our approach unifies verification and bug detection.

While our approach starts with a verification attempt, the approach in [21] tries to show program incorrectness by starting at the root of the proof tree with a formula that express program incorrectness. Thus in contrast to our approach the approach in [21] can only show program incorrectness.

Another related work that unifies verification and bug finding very closely is Synergy [14] that is an extensions of the Lee-Yannakakis algorithm [17] and is an improvement to SLAM [2] and BLAST [16]. While these approaches are based on abstraction and refinement, our approach is optimized for underlying verification techniques that are based on symbolic execution or weakest precondition computation. Furthermore, the approaches [14, 16] and, e.g., [18] are more

concerned with the automatic generation of annotations while in our work theorem proving and the challenges with user-provided loop invariants and method contracts are in focus. The latter applies also to [8] where in contrast to our work explicit program execution is used and also other reasons for proof failure than program error are considered. The main concern in [8] is however finding the right program input to detect a bug while in our approach we reason about the existence of such an input.

Approaches that start with a verification attempt and in case of failure generate counterexamples for the unproved verification conditions are e.g. Spec#[3], VCC [22], Caduceus [12], Krakatoa [19], Bogor/Kiasan[9]. These approaches have the problem that a counterexample for a verification condition has an ambiguous meaning, i.e. the used contracts can be too weak or the target program has an error. Our contribution in contrast deals with this problem and therefore it extends the existing approaches.

6 Summary and Conclusion

The presented work extends existing approaches that try to verify a program and in case of verification failure generate counterexamples for verification conditions. In contrast to existing approach our approach allows us to conclude the existence of a program bug from falsifiable verifications even if contract rules were used during the verification attempt. Furthermore, checking the existence of a program bug after the verification attempt does not require explicit program testing, symbolic execution, or weakest precondition computation. Instead we *reuse* information obtained from the verification attempt to reason about the existence of a program bug. In this way our technique unifies verification and bug detection. We have successfully tested several small example programs and specifications using KeY with some manual interaction. An extended evaluation of this approach is planned as future work.

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. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, LNCS 2057, pages 103–122, 2001.
3. M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. R. M. Leino, W. Schulte, and H. Venter. The SPEC# programming system: Challenges and directions. In *VSTTE*, pages 144–152, 2005.
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 and C. Gladisch. White-box Testing by Combining Deduction-based Specification Extraction and Black-box Testing. In Y. Gurevich and B. Meyer, editors, *Proceedings, Tests and Proofs, Zürich, Switzerland*, LNCS. Springer, 2007.
6. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.
7. B. Beckert, S. Schlager, and P. H. Schmitt. An improved rule for while loops in deductive program verification. In K.-K. Lau, editor, *Proceedings, Seventh International Conference on Formal Engineering Methods (ICFEM), Manchester, UK*, LNCS. Springer, 2005.
8. K. Claessen and H. Svensson. Finding counter examples in induction proofs. In *TAP*, pages 48–65, 2008.
9. X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In *ASE*, pages 157–166, 2006.
10. C. Engel, C. Gladisch, V. Klebanov, and P. Rümmer. Integrating verification and testing of object-oriented software. In *TAP*, pages 182–191, 2008.
11. C. Engel and R. Hähnle. Generating Unit Tests from Formal Proofs. In Y. Gurevich and B. Meyer, editors, *Proceedings, Tests and Proofs, Zürich, Switzerland*, LNCS. Springer, 2007.
12. J.-C. Filliâtre and C. Marché. Multi-prover Verification of C Programs. In *ICFEM*, pages 15–29, 2004.
13. C. Gladisch. Verification-based testing for full feasible branch coverage. In *Proc. 6th IEEE Int. Conf. Software Engineering and Formal Methods (SEFM'08)*. IEEE Computer Society Press.
14. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127, 2006.
15. D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, chapter 10, pages 497–604. Reidel, Dordrecht, 1984.
16. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
17. D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *STOC*, pages 264–274, 1992.
18. K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *APLAS*, pages 119–134, 2005.
19. C. Marché, C. Paulin, and X. Urbain. The Krakatoa tool for JML/Java program certification. Website at <http://krakatoa.lri.fr>, 2003.
20. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *LPAR*, pages 422–436, 2006.
21. P. Rümmer and M. A. Shah. Proving programs incorrect using a sequent calculus for java dynamic logic. In *TAP*, pages 41–60, 2007.
22. W. Schulte, X. Songtao, J. Smans, and F. Piessens. A glimpse of a verifying C compiler. In *C/C++ Verification Workshop*, 2007.