

Verification-based Test Case Generation for Full Feasible Branch Coverage

Christoph Gladisch
University of Koblenz-Landau
Department of Computer Science
56070 Koblenz, Germany, gladisch@uni-koblenz.de

Abstract

The goal of this work is to improve the testing of programs that contain loops and complex methods. We achieve this goal with verification-based testing, which is a technique that can generate test cases not only from source code but also from loop invariants and method specifications provided by the user. These test cases ensure the execution of interesting program paths that are likely to be missed by existing testing techniques that are based on symbolic program execution. These techniques would require an exhaustive inspection of all execution paths, which is hard to achieve in presence of complex methods and impossible if loops are involved. Verification-based testing takes a different approach.

1. Introduction

This paper extends the work presented in [9] and [4] and focuses on the semantic properties of loop invariants and method specifications, and their implications for test case generation.

The goal is to improve existing software testing techniques that use symbolic program execution for test case generation and constraint solving for computing concrete test data. The improvement is the generation of test cases for testing program branches that are likely to be missed by the existing testing techniques. These would require an exhaustive inspection of all execution paths which is infeasible in the presence of complex methods and impossible in the presence of loops because loops represent infinitely many paths. The presented approach solves this problem by using loop invariants and method specifications that must be provided with the source code under test by the user. These are usually written when applying formal software development methodologies (e.g. [6, 2, 5]).

An example that shows the advantages of the presented approach is given in listing 1 of Figure 1. In order to execute $A()$ the loop body has to be executed at least 10 times

and in order to execute $C()$ it has to be executed exactly 20 times. In similar programs these numbers could be much larger or be the result of complex expressions requiring an exhaustive inspection of all paths in order to find the case where the branch conditions are satisfied. The situation is similar in listing 2 where an exhaustive inspection of $D()$ may be required in order to find a path such that after the execution of $D()$ the branch condition $i \doteq 20$ holds. Since exhaustive symbolic execution is not possible existing testing techniques are likely to miss these cases because they have a bound on the amount of inspected execution paths. For loops and recursive methods the typical approach is to symbolically execute the first k loop iterations or recursion steps, called k -bounded unwinding, where k is a limiting constant.

— JAVA (1) —	— JAVA + JML (2) —
<pre>void foo(int n) { int i=0; while(i < n) { if(i==10) { A(); } B(); i++; } if(i==20) { C(); } }</pre>	<pre>/*@ requires i<n; @ assignable i; @ ensures i==n; */ void D(int n) { while(i<n) ... } void foo(int n) { D(n); if(i==20) {C();} }</pre>
————— JAVA —————	————— JAVA + JML —————

Figure 1. Motivating examples

In practice programs like in Figure 1 occur for instance if constants or constant-sized datastructures are used, or if datastructures of a certain size are treated in a special way for optimization purposes. Such programs occur rather seldom. However, the presented technique does not replace testing techniques that use k -bounded symbolic execution for test case generation but instead it closes the gap of situations which existing techniques cannot handle. The approach is implemented in the KeY-system [5] and has been successfully tested on small examples.

1.1. Verification-based Testing

The preliminary work of our approach is *verification-based testing* [9] and *white-box testing by combining deduction-based specification extraction and black-box testing* [4] which incorporate verification technology into test case generation. Both approaches derive test cases from a proof tree that is constructed from a verification proof obligation. The proof tree is constructed through the application of calculus rules for first-order logic, integer arithmetics, and symbolic execution. Due to case distinctions stemming from the symbolic execution rules the branches of the proof tree correspond to program paths that are regarded here as test cases. The two approaches differ in the test data generation phase. In [9] test data is generated by a custom-built constraint solver and in [4] test cases are encoded in a specification such that black-box testing with this specification results effectively in white-box testing.

The presented approach replaces the symbolic execution of a complex method or loop by the application of a method specification or loop invariant rule in the proof tree. The purpose of both approaches is the same, namely to compute a precondition for a given branch condition (the condition that has to be fulfilled later on for that branch to be taken): the *branch precondition*.

The challenge in test case generation for programs with loops and complex methods like in Figure 1 is to compute the precondition of the loop or the method for a given *branch condition* which occurs within the loop (e.g. $i==10$) or after the loop or method (e.g. $i==20$). The precondition is important because it describes test data for establishing a program state *before* the loop or method that guarantees the execution of a program branch or path. If the precondition is known and represented as a first-order logic formula, then the desired test data can be computed using a constraint solver.

For example the desired preconditions of the loop in listing 1 that are computed with our approach by using the loop invariant $i^{old} \leq i \wedge i \leq n$ are: $i \leq 10 \wedge 10 < n$ to execute $A()$ and $i \leq n \wedge 20 \doteq n$ to execute $C()$, where i^{old} refers to the value of i before the loop. The loop is replaced in listing 2 with the method $D()$ which has the specification: if $i < n$ is satisfied before the execution of $D()$, then $i \doteq n$ is satisfied after its execution. Instead of searching for a program path in $D()$ with a path-condition that ensures the execution of $C()$ we use the given specification of $D()$. Our approach yields in this case the desired constraint $i \leq n \wedge 20 \doteq n$. These preconditions are not generated by symbolic execution if we choose for instance the bound $k \doteq 3$.

Precondition computation based on specifications or loop invariants is applicable in situations where the inspection of all execution paths with symbolic execution is in-

feasible or impossible. Computing the precondition for a given branch condition based on a loop invariant resp. a method specification are however essentially the same problem. The loop invariant is a pre- and postcondition of the loop's body and of the loop itself. We will therefore refer to the loop invariant or method specification just by specification. The presented technique is similar to the well known verification technique called weakest precondition computation except that for the purpose of test data generation the constraint solver does not require weak but rather *strong* constraints that reduce the search space for test data.

Typical use-cases of our approach are those where specifications of methods and loops are provided. This is for instance the case in specification driven software development methodologies (e.g. B method [6], SOCOS [2]) or if software verification is applied, e.g. [5].

1.2. Plan of the Paper

In the next section we describe the logic and the calculus on which we build our approach and give some definitions. The approach can however easily be adapted to other formalisms. The main part is Section 3 where we introduce two formulae that are built from a specification and a branch condition: the *disjunctive branch precondition* (DBPC) and the *conjunctive branch precondition* (CBPC). The CBPC is the desired precondition (constraint) for test data generation using constraint solving as described in Section 1.1. The DBPC is needed for detecting infeasible execution paths and to show how the CBPC can be used depending on the properties of the involved specification. In Section 4 we give examples of how to generate the desired preconditions of methods and loops for a given branch condition by constructing the CBPC. Finally in Section 5 we describe how this contribution relates to existing work and we draw conclusions in Section 6.

2. Dynamic Logic and the Verification Calculus

2.1. Overview of JAVA CARD DL and KeY's Sequent Calculus

The work presented here is based on a Dynamic Logic [10] for JAVA CARD (JAVA CARD DL [3]) 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 [5, 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 $\langle p \rangle$ and $[p]$ for every program p . We regard only deterministic programs here. The formula $[p]\varphi$ means that if p ter-

minates, then φ holds in the state after the execution of p . Using the termini in [8], $[p]\varphi$ is semantically equivalent to the weakest precondition $wlp(p, \varphi)$. The formula $\langle p \rangle \varphi$ states additionally that the program terminates. Thus $[p]\varphi \wedge \langle p \rangle true$ is equivalent to $\langle p \rangle \varphi$ which is again equivalent to the weakest precondition $wp(p, \varphi)$. In the following we denote the set of programs by π , the set of DL-formulae by Fml , and the set of first-order logic formulae by Fml_{FOL} . All variables in formulae are bound with quantifiers.

An implication of the form $pre \rightarrow [p]post \in Fml$ with $pre, post \in Fml_{FOL}$ corresponds to the Hoare triple $\{pre\}p\{post\}$ 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. The implication $pre \rightarrow \langle p \rangle post$ states additionally that p terminates. Dynamic logic allows pre and $post$ to *contain* programs in contrast to Hoare logic: if $pre, post \in Fml$ then $pre \rightarrow [p]post \in Fml$.

Program variables are modeled 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 $a, o, i, acc[] \in \Sigma_{nr}$. In this case a program variable a is represented by a logical non-rigid constant a , an expression like $o.a$, that accesses an object attribute, is modeled as the term $a(o)$, and in case of an array access $o.a[i]$ the corresponding term is $acc[](a(o), i)$. We use constant domain semantics which means that in all states terms are evaluated to values of the same universe. In contrast to interpretations a variable assignment β cannot be modified by a program so that logical variables are always *rigid*, i.e., they have the same value in all program states.

To express that a formula φ is true in a state $s \in S$ (S is the set of all states) under a variables assignment β we write $s, \beta \models \varphi$. Furthermore $s \models \varphi$ means that for all variable assignments β : $s, \beta \models \varphi$; and $\models \varphi$ means that φ is valid, i.e., for all $s \in S$ and all variable assignments β the statement $s, \beta \models \varphi$ holds. Non-standard, but important in this paper, is the case where $s_{\Sigma \setminus SK}$ is only a partial interpretation $s_{\Sigma \setminus SK} \in S_{\Sigma \setminus SK}$, where SK is intuitively a set of skolem functions or new temporary program variables. A partial interpretation $s_{\Sigma \setminus SK}$ gives a meaning to symbols from the subset $(\Sigma \setminus SK) \subset \Sigma$ of the signature Σ . In this case each partial interpretation $s_{SK} \in S_{SK}$ of the unspecified symbols $SK \subset \Sigma$ is combined with the *given* partial interpretation $s_{\Sigma \setminus SK} \in S_{\Sigma \setminus SK}$ resulting in a total interpretation $(s_{SK} \cup s_{\Sigma \setminus SK}) \in (S_{SK} \cup_{\times} S_{\Sigma \setminus SK}) = S_{\Sigma}$. Thus $s_{\Sigma \setminus SK}, \beta \models \varphi$ means that for all $s_{SK} \in S_{SK}$ the statement $(s_{SK} \cup s_{\Sigma \setminus SK}), \beta \models \varphi$ holds.

In this paper we understand *test data* as a partial state, i.e. a mapping from program variables Σ_{nr} to values.

Definition 1 Let $\Phi = \phi_1, \dots, \phi_n$ be a set of formulae and let Φ_{\wedge} be the conjunction $\phi_1 \wedge \dots \wedge \phi_n$. Let γ be a formula, S the set of all states (interpretations), and B the set of all variables assignments, then

– γ is a local consequence of Φ , written as $\Phi \models_l \gamma$, iff

for all $\beta \in B$: for all $s \in S$:
 $s, \beta \models \Phi_{\wedge}$ implies $s, \beta \models \gamma$

– Let $SK \subset \Sigma$ and let $S_{\Sigma \setminus SK}$ be the set of all partial states that are defined only for the symbols $\Sigma \setminus SK$. γ is a semi-local consequence of Φ , or alternatively, γ is a local consequence of Φ modulo the interpretation of $SK \subset \Sigma$, written as $\Phi \models_{SK} \gamma$, iff

for all $\beta \in B$: for all $s \in S_{\Sigma \setminus SK}$:
 $s, \beta \models \Phi_{\wedge}$ implies $s, \beta \models \gamma$

(note that $s, \beta \models \Phi_{\wedge}$ and $s, \beta \models \gamma$ quantify locally over the interpretations of the symbols SK).

For example $a \doteq sk \rightarrow sk \doteq b \models_{\{sk\}} a \doteq b$ but $a \doteq sk \rightarrow sk \doteq b \not\models_l a \doteq b$. A state $s \in S$ where local consequence fails in the latter case is, e.g., $s = \{a \mapsto 0, sk \mapsto 1, b \mapsto 1\}$.

For the sake of a shorter notation we use the sequent calculus notation. A formula of the form

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

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

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

A sequent rule is of the form

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

and it is locally correct iff $\Gamma_1 \Rightarrow \Delta_1 \quad \dots \quad \Gamma_n \Rightarrow \Delta_n \models_l \Gamma_0 \Rightarrow \Delta_0$ and locally correct modulo $SK \subset \Sigma$ (or semi-locally correct) iff $\Gamma_1 \Rightarrow \Delta_1 \quad \dots \quad \Gamma_n \Rightarrow \Delta_n \models_{SK} \Gamma_0 \Rightarrow \Delta_0$. 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 generated that have to be proved instead.

2.2. State Updates

In order to write that $f \in \Sigma$ is replaced by $g \in \Sigma$ in $\varphi \in Fml$ we could use the substitution $[g/f]\varphi$. This is a rather technical notion of our intention to express that

φ is evaluated in the state after assigning $f(x_1, \dots, x_n)$ to $g(x_1, \dots, x_n)$ for all argument values. A more intuitive notation allowing us to refer to a pre-state and a post-state of an assignment are (state) updates [3, 12] which are an extension of classical Dynamic Logic [10].

Updates are assignments between terms (not between JAVA expressions) and are therefore free of side-effects allowing an efficient way of handling aliasing. An update has the form $\{t_1 := t_2\}$ and means that in the post-state of the update the non-rigid term t_1 has the value of the term t_2 which is evaluated in the pre-state. For instance in order to prove $i \doteq 0 \rightarrow \langle i++ \rangle i \doteq 1$ the diamond operator $\langle i++ \rangle$ is replaced with an update (because $i++$ has a side-effect) yielding $i \doteq 0 \rightarrow \{i := i + 1\} i \doteq 1$. Update application finally gives $i \doteq 0 \rightarrow i + 1 \doteq 1 \in Fml_{FOL}$. Updates allow also an extension of the presented approach for describing state-dependent sets of memory locations in a more structured way, than it is possible with first-order formulae.

In the scope of this paper we allow updates of the form $\{f := g\}\varphi$, with $f \in \Sigma_{nr}$ and $g \in \Sigma$, that assign the values of g to f for all argument values. Technically this is equivalent to the substitution $[g/f]\varphi$. The notation $\{A := B\}\varphi$ with $A, B \subseteq \Sigma$ is equivalent to the update application $\{a_1 := b_1, \dots, a_n := b_n\}\varphi$ which replaces in φ the function symbols $a_1, \dots, a_n \in A$ by the function symbols $b_1, \dots, b_n \in B$ respectively.

2.3. 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 2 The minimal modifier set of a program \mathfrak{p} is denoted by $Mod(\mathfrak{p}) \subseteq \Sigma_{nr}$ and it consists exactly of those function symbols that can be modified by \mathfrak{p} . A correct modifier set $M \subseteq \Sigma_{nr}$ contains at least the function symbols that are modifiable by \mathfrak{p} , i.e. $M \supseteq Mod(\mathfrak{p})$.

A modifier set $M \subseteq \Sigma_{nr}$ can be used to create an *anonymous update* of the form $\{M := M_{sk}\}$ which replaces each function symbol $f \in M$ by a fresh function symbol $f_{sk} \in M_{sk}$. Anonymous updates enable us to transform a postcondition into a precondition preserving only the common information of the pre-state and the post-state.

For instance the formula $\langle \mathfrak{p} \rangle (o.a \doteq c \wedge o.a \doteq d)$ specifies a pre-state such that after the execution of \mathfrak{p} the postcondition $o.a \doteq c \wedge o.a \doteq d$ holds. Let $\{a\} = Mod(\mathfrak{p})$, then as an approximation we can use the anonymous update $\{a := a_{sk}\}$ to replace the modal operator $\langle \mathfrak{p} \rangle$ resulting in the formula $\{a := a_{sk}\}(o.a \doteq c \wedge o.a \doteq d)$. This can be reduced to $o.a_{sk} \doteq c \wedge o.a_{sk} \doteq d$. Thus we have gained the additional information about the pre-state that $c \doteq d$.

2.4. Specifications

Definition 3 A specification is a triple $(pre, post, M)$ where $pre \in Fml_{FOL}$ is the precondition, $post \in Fml_{FOL}$ is the postcondition and $M \subseteq \Sigma_{nr}$ 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 $(I, I \wedge \neg lc, M)$ and a specification of a loop body before loop termination is the triple $(I \wedge lc, I \wedge lc, M)$.

In the next section we describe how to compute preconditions from a given specification and a branch condition in the program. The preconditions have different semantic properties depending on the semantic properties of the involved specifications that are described next.

Definition 4 A specification $(pre, post, M)$ is satisfied by a program \mathfrak{p} iff $M \supseteq Mod(\mathfrak{p})$ and

$$pre \rightarrow \langle \mathfrak{p} \rangle post$$

is valid. This is equivalent to the statement that the program \mathfrak{p} is (totally) correct wrt. the specification.

An important verification technique is to use a specification in a proof instead of a program which is allowed if the specification is satisfied by the specification.

In the following sections we will abbreviate the conjunction

$$\begin{aligned} \forall x_1 \dots \forall x_{m_1}. f_1(x_1, \dots, x_{m_1}) &\doteq f_1^{sk}(x_1, \dots, x_{m_1}) \\ &\vdots \\ \wedge \forall x_1 \dots \forall x_{m_n}. f_n(x_1, \dots, x_{m_n}) &\doteq f_n^{sk}(x_1, \dots, x_{m_n}) \end{aligned}$$

where $\{f_1, \dots, f_n\} = M \subseteq \Sigma_{nr}$ and $\{f_1^{sk}, \dots, f_n^{sk}\} = M_{sk} \subseteq \Sigma$ are disjoint, with the notation

$$M \doteq_{\forall} M_{sk}$$

The definition below defines the *strength* property of a specification that is important when we construct a precondition based on the specification.

Definition 5 Let \mathfrak{p} be a program and $\sigma = (pre, post, M)$ a specification with $M \supseteq (Mod(\mathfrak{p}) \cap \Sigma(pre, post))$, where $\Sigma(pre, post)$ denotes the set of symbols that are in pre or in post. We say that

– σ is strong wrt. \mathfrak{p} iff the following formula is valid

$$(pre \wedge \{M := M_{sk}\}post) \rightarrow \langle \mathfrak{p} \rangle M \doteq_{\forall} M_{sk} \quad (1)$$

- σ is strong wrt. p in state s iff $s \models (1)$.

For example the specification $(y \doteq y', y \doteq y' + 1, \{y\})$ is strong for the program $y=y+1$ but the specification $(y \doteq y', y' > 0 \rightarrow y \doteq y' + 1, \{y\})$ is weak, e.g., it is not strong in the state $s = \{y \mapsto 0, y' \mapsto 0, y_{sk} \mapsto 7\}$, where y_{sk} is the new symbol introduced by the anonymous update $\{M := M_{sk}\}$. If a specification is strong and $pre = true$, then it is the strongest specification.

3. Branch Preconditions

The goal of the presented approach is to improve existing software testing techniques which use symbolic program execution and constraint solving. As described in the first section the purpose of the symbolic execution is to compute preconditions for a given branch condition from which test data can be computed by using constraint solving.

Our approach is to replace the symbolic execution of a complex method or loop by the computation of a precondition based on a *specification* and a *branch condition* (see Section 1.1). We define two kinds of such preconditions: the *disjunctive branch precondition* (DBPC) and the *conjunctive branch precondition* (CBPC); and we describe their relation to the weakest precondition.

Depending on the properties of the involved specification the DBPC is suitable to detect infeasible paths or unsatisfiable test data constraints and the CBPC is suitable for test data generation. In Section 3.3 we define a special CBPC for branch conditions that occur within loops (e.g. Listing 1) and in Section 3.4 we describe the relation between the DBPC and the CBPC.

3.1. Disjunctive Branch Precondition

The disjunctive branch precondition (DBPC) is a formula that is suitable for detecting infeasible paths or unsatisfiable test data constraints. It fulfills this purpose however only if it is constructed from a satisfied specification giving rise to the full disjunctive branch precondition (F-DBPC) as explained below. In order to save space we define the DBPC and F-DBPC in one definition.

Definition 6 Let $\sigma = (pre, post, M)$ be a specification of a program $p \in \pi$ with $M \supseteq Mod(p)$. Let $M_{sk} \subset \Sigma$ be new symbols for the symbols in M (see Section 2.3), and $\varphi \in Fml$ a branch condition.

The full disjunctive branch precondition (F-DBPC) is the conjunction of:

- the condition that σ is satisfied by p (see Def. 4)
- the disjunctive branch precondition (DBPC) for φ :

$$pre \rightarrow \{M := M_{sk}\}(post \rightarrow \varphi)$$

For example for the specification $(x' \doteq x \wedge y > 0, x' < x, \{x\})$, the program $x=x+y$; and the branch condition $x \doteq y$ the DBPC is:

$$\begin{aligned} x' \doteq x \wedge y > 0 &\rightarrow \{x := x_{sk}\}(x' < x \rightarrow x \doteq y) \\ x' \doteq x \wedge y > 0 &\rightarrow (x' < x_{sk} \rightarrow x_{sk} \doteq y) \text{ (simpl.)} \end{aligned} \quad (2)$$

Lemma 1 Let $M_{sk} \subset \Sigma$ be new symbols for $M \supseteq Mod(p)$. The following rule is locally correct modulo M_{sk} .

$$\frac{pre \Rightarrow \langle p \rangle post \quad pre \Rightarrow \{M := M_{sk}\}(post \rightarrow \varphi)}{pre \Rightarrow \langle p \rangle \varphi} \quad (3)$$

Note that the two premisses of the rule constitute the F-DBPC. Given a correct specification and branch condition φ the rule says that the $DBPC_\varphi$ is a precondition of $pre \rightarrow \langle p \rangle \varphi$. For the purpose of test data generation using a constraint solver the DBPC is not suitable due to its *semi-local correctness* (see Section 2.1), i.e. not every model of this formula (solution of the constraint solver) ensures that after executing p the branch condition φ holds. For instance the model $s = \{x \mapsto 0, x' \mapsto 0, x_{sk} \mapsto 0, y \mapsto 2\}$ satisfies the DBPC (2) but not $\langle p \rangle \varphi$, i.e. $s \not\models \langle p \rangle \varphi$.

The DBPC is however useful for solving the problem

“no test data can be generated by using the contract of p such that after the execution of p , φ is satisfied”

for identifying infeasible execution paths before the attempt to generate test data. To do so let's assume we want to generate test data which represents a state s such that it satisfies the precondition, i.e. $s \models pre$, and $s \models \langle p \rangle \varphi$. We can determine whether $pre \wedge \langle p \rangle \varphi$ is unsatisfiable by proving $\neg pre \vee [p] \neg \varphi$. This means, if $(pre, post, M)$ is a specification satisfied by p and we prove the validity of $pre \rightarrow \{M := M_{sk}\}(post \rightarrow \neg \varphi)$, then we know, without the need to inspect all paths of p with symbolic execution, that there is no state that satisfies the precondition and $\langle p \rangle \varphi$.

Other approaches, e.g. [16, 15, 7], compute a weakest precondition to detect infeasible execution paths so the question is how the weakest precondition is related to the DBPC. Assuming that $(pre, post, M_{sk})$ is satisfied by p , then the weakest precondition $wp(p, \neg \varphi)$ (which is semantically equivalent to $\langle p \rangle \neg \varphi$) is the formula:

$$\begin{aligned} &\overbrace{\text{weakest precondition for } p \text{ and } \neg \varphi} \\ &\quad \overbrace{DBPC_{\neg \varphi}^p} \\ &pre \wedge \overbrace{(pre \rightarrow \{M := M_{sk}\}(post \rightarrow \neg \varphi))} \\ &pre \wedge \{M := M_{sk}\}(post \rightarrow \neg \varphi) \quad \text{(simpl.)} \end{aligned} \quad (4)$$

A proof of $\Gamma \rightarrow DBPC_{\neg \varphi}^p$ does not imply $\Gamma \rightarrow wp(p, \neg \varphi)$, i.e. that p is correct wrt. $(\Gamma, \neg \varphi, M_{sk})$, where Γ is some

precondition under which we exercise p . The case where $\Gamma \rightarrow \text{DBPC}_{\neg\varphi}^p$ is true but $\Gamma \rightarrow wp(p, \neg\varphi)$ is false occurs when Γ is satisfied and the precondition of p 's contract (pre) is false. In fact a proof of $\Gamma \rightarrow wp(p, \neg\varphi)$ would be unsound in this case because the contract does not specify the behavior for all possible inputs that are allowed by Γ . A proof of $\Gamma \rightarrow \text{DBPC}_{\neg\varphi}^p$ tells us however that we cannot generate test data that would satisfy φ after executing p by using the given contract. Of course there may exist states satisfying $\langle p \rangle \varphi$ but not pre . In this case, however, the specification is useless for our approach because we can make no statements about the postcondition.

In order to prove Lemma 1 we need the following rule.

Lemma 2 *The following rule is locally correct modulo the new symbols $M_{sk} \subset \Sigma$ for the symbols $M \supseteq \text{Mod}(p)$*

$$\frac{\text{pre} \Rightarrow \{M := M_{sk}\} \text{post}}{\text{pre} \Rightarrow \langle p \rangle \text{post}} \quad (5)$$

We omit a detailed proof of rule (5) but its *semi-local correctness* (see Section 2.1) is obvious: if for all possible assignments of values (see Section 2.3) to the modifiable program variables the postcondition is true, i.e. $\{M := M_{sk}\} \text{post}$ is valid, then for any program variable assignments in p the postcondition must be true, i.e. $\langle p \rangle \text{post}$.

Proof. We start the proof of Lemma 1 with the tautology

$$\langle p \rangle \text{true} \rightarrow \langle p \rangle \left(\overbrace{(\text{pre} \wedge \text{post} \wedge (\text{post} \rightarrow \varphi))}^{\text{true}} \rightarrow \varphi \right)$$

and obtain through equivalence transformations

$$(\text{pre} \rightarrow (\langle p \rangle \text{post} \wedge \langle p \rangle (\text{post} \rightarrow \varphi))) \rightarrow (\text{pre} \rightarrow \langle p \rangle \varphi)$$

This is again equivalent to the local correctness of the rule:

$$\frac{\text{pre} \Rightarrow (\langle p \rangle \text{post}) \wedge \langle p \rangle (\text{post} \rightarrow \varphi)}{\text{pre} \Rightarrow \langle p \rangle \varphi}$$

The open branches of the following proof tree are the ones of rule (3).

$$\frac{\text{pre} \Rightarrow \langle p \rangle \text{post} \quad \frac{\text{pre} \Rightarrow \{M := M_{sk}\} (\text{post} \rightarrow \varphi)}{\text{pre} \Rightarrow \langle p \rangle (\text{post} \rightarrow \varphi)} \quad (R5)}{\text{pre} \Rightarrow (\langle p \rangle \text{post}) \wedge \langle p \rangle (\text{post} \rightarrow \varphi)} \quad \frac{}{\text{pre} \Rightarrow \langle p \rangle \varphi}$$

■

3.2. Conjunctive Branch Precondition

The *conjunctive branch precondition* is the precondition of branch conditions in the program that we suggest for test data generation using a constraint solver (see Section 1.1). To save space we define it within the definition of the *full conjunctive branch precondition* which adds a constraint on the involved specification.

Definition 7 *Let $\sigma = (\text{pre}, \text{post}, M)$ with $M \supseteq (\text{Mod}(p) \cap \Sigma(\text{pre}, \text{post}))$ be a specification for $p \in \pi$, where $\Sigma(\text{pre}, \text{post})$ denotes the symbols occurring in pre and post . The full conjunctive branch precondition (F-CBPC) for a formula φ is the conjunction of:*

- σ is strong for p (see Def. 5):
 $(\text{pre} \wedge \{M := M_{sk}\} \text{post}) \rightarrow \langle p \rangle M \doteq_{\forall} M_{sk}$
- the conjunctive branch precondition for φ (CBPC $_{\varphi}$):

$$\text{pre} \wedge \{M := M_{sk}\} (\text{post} \wedge \varphi)$$

Theorem 1 *Each state satisfying the F-CBPC for φ also satisfies $\langle p \rangle \varphi$.*

The CBPC $_{\varphi}$ is a precondition for $\langle p \rangle \varphi$ that is suitable for test data generation using constraint solvers: (1) if pre , post , and φ are first-order logic formulae, then CBPC $_{\varphi}$ can be trivially simplified to a first-order formula and (2) every model (test data) of the CBPC $_{\varphi}$ guarantees that after executing p the condition φ is satisfied if σ is strong in this state. For instance, for $\sigma = (x' \doteq x \wedge y > 0, x' < x, \{x\})$, the program $x = x + y$; and the branch condition $x \doteq y$ the CBPC is:

$$\begin{aligned} &x' \doteq x \wedge y > 0 \quad \wedge \quad \{x := x_{sk}\} (x' < x \wedge x \doteq y) \\ &x' \doteq x \wedge y > 0 \quad \wedge \quad (x' < x_{sk} \wedge x_{sk} \doteq y) \quad (\text{impl.})(6) \end{aligned}$$

A model of this condition is, e.g., the state $s = \{x \mapsto 0, x' \mapsto 0, x_{sk} \mapsto 1, y \mapsto 1\}$. It satisfies also the strength condition, i.e. $s \models (x' \doteq x \wedge y > 0 \wedge \{x := x_{sk}\} x' < x) \rightarrow \langle p \rangle x \doteq x_{sk}$, and therefore the formula $\langle p \rangle x \doteq y$ as well.

The CBPC is stronger than the DBPC and it is also stronger than the weakest precondition, which is our objective. Interesting is that in contrast to the F-DBPC, the F-CBPC *does not require* the involved specification to be satisfied by the program. Instead the specification must be strong. For example consider the specification $\sigma_2 = (x' \doteq x, x \doteq 2z \wedge x' \doteq z \wedge y \doteq z, M)$ with $M = \{x\}$. It is not satisfied by p but it is strong and therefore CBPC $_{x \doteq y}$ implies $\langle p \rangle x \doteq y$.

In [9] the problem of “fresh constants introduced in [anonymous] updates” was mentioned that replacing a program by the anonymous update $\{M := M_{sk}\}$, semantically seen, destroys information about the pre-state that is encoded in post and φ . The strength condition ensures, however, that just enough information is preserved to ensure the satisfaction of $\langle p \rangle \varphi$. In contrast to the F-DBPC the F-CBPC is more sensitive to the size of the modifier set. The specification σ is not strong if, e.g., $M = \{x, y\}$.

Instead of creating a model for the CBPC $_{\varphi}$ one could consider the generation of a counter example for the weakest precondition of the negated branch condition (see Formula 4). The latter approach is equivalent to generating a

model for:

$$\begin{aligned}
& \neg wp(\mathfrak{p}, \neg\varphi) \\
\Leftrightarrow & \neg(\text{pre} \wedge \{M := M_{sk}\}(\text{post} \rightarrow \neg\varphi)) \\
\Leftrightarrow & \text{pre} \rightarrow \{M := M_{sk}\}\neg(\text{post} \rightarrow \neg\varphi) \\
\Leftrightarrow & \text{pre} \rightarrow \{M := M_{sk}\}(\text{post} \wedge \varphi)
\end{aligned}$$

A model of $\neg wp(\mathfrak{p}, \neg\varphi)$ may falsify the precondition pre resulting in a test that can possibly, but not necessarily, satisfy φ after the execution of \mathfrak{p} . The $\text{CBPC}_{\varphi}^{\mathfrak{p}}$ is however stronger than $\neg wp(\mathfrak{p}, \neg\varphi)$ and prevents this case. This is our objective since we focus on test generation that is based on loop invariants and method specifications. Thus the $\text{CBPC}_{\varphi}^{\mathfrak{p}}$ is equivalent to $\text{pre} \wedge \neg wp(\mathfrak{p}, \neg\varphi)$.

Proof. We prove Theorem 1 by proving the validity of the implication

$$\left(\begin{aligned} & (\text{pre} \wedge \{M := M_{sk}\}\text{post}) \rightarrow \langle \mathfrak{p} \rangle M \dot{=}_{\forall} M_{sk} \\ & \text{pre} \wedge \{M := M_{sk}\}(\text{post} \wedge \varphi) \end{aligned} \right) \rightarrow \langle \mathfrak{p} \rangle \varphi$$

which can be simplified to (let $\varphi' = \{M := M_{sk}\}\varphi$):

$$\langle \mathfrak{p} \rangle M \dot{=}_{\forall} M_{sk} \wedge \varphi' \rightarrow \langle \mathfrak{p} \rangle \varphi$$

Since φ' is rigid for \mathfrak{p} we can use the equivalence $\varphi' \equiv \langle \mathfrak{p} \rangle \varphi'$ and obtain

$$\begin{aligned}
\langle \mathfrak{p} \rangle M \dot{=}_{\forall} M_{sk} \wedge \langle \mathfrak{p} \rangle \varphi' & \rightarrow \langle \mathfrak{p} \rangle \varphi \\
M \dot{=}_{\forall} M_{sk} \wedge \varphi' & \rightarrow \varphi
\end{aligned}$$

Applying $M \dot{=}_{\forall} M_{sk}$ on φ yields φ' resulting in $\varphi' \rightarrow \varphi$. ■

3.3. Form of the Strength Condition when dealing with Branches within a Loop

In the case of a branch condition that occurs *after* a loop or method the strength condition of the F-CBPC can be constructed from the specification of the loop or method as it is defined in Section 2.4. In order to apply Theorem 1 for the case of a branch condition that occurs *within* a loop we need to define what a strong specification of an arbitrarily long sequence of loop body concatenations is. A candidate for the pre- and postcondition of such a specification is a loop invariant. The following lemma is a version of Theorem 1 for a sequence of loop bodies, i.e. for loop iterations. We abbreviate programs of the form $\text{while}(\text{loop-cond})\{\text{loop-body}\}$, with $w(\text{lc})\{b\}$.

Lemma 3 *Let $I \in \text{Fml}$ be a loop invariant of the loop $w(\text{lc})\{b\}$. Let $M_{old}, M_{sk} \subset \Sigma$ be new symbols for $M \supseteq \text{Mod}(b)$, $\varphi \in \text{Fml}$ be a branch condition, and $i, N \in \mathbb{N}$ are new symbols that indicate loop iteration numbers. The program b' is an extension of the original loop body b with the loop iteration counter i : $\{b'\} = \{b; i++\}$*

In any state where the

- *strength condition:*

$$(M_{old} \dot{=}_{\forall} M \wedge I \wedge \text{lc} \wedge \{M := M_{sk}\}I \wedge \text{lc}) \rightarrow i \dot{=} 0 \rightarrow \exists N. \left(\begin{aligned} & (\langle w(\text{lc})\{b'\} \rangle i \geq N) \\ & (\langle w(i < N)\{b'\} \rangle M \dot{=}_{\forall} M_{sk}) \end{aligned} \right)$$

- *loop branch precondition (LBPC):*

$$M_{old} \dot{=}_{\forall} M \wedge I \wedge \text{lc} \wedge \{M := M_{sk}\}(I \wedge \text{lc} \wedge \varphi)$$

are both satisfied, also the following condition is satisfied:

$$i \dot{=} 0 \rightarrow \exists N. \left(\begin{aligned} & (\langle w(\text{lc})\{b'\} \rangle i \geq N) \\ & (\langle w(i < N)\{b'\} \rangle \varphi) \end{aligned} \right) \quad (7)$$

The Formula 7 means that the loop with the original loop condition lc iterates at least N times and after N iterations (specified by $i < N$) the branch condition φ is satisfied.

The strength condition means that if the pre-state $s_{pre} \in S$ and a state $s_{post} \in S$ both satisfy the invariant and the loop condition, i.e. $s_{pre} \models I \wedge \text{lc}$ and $s_{post} \models I \wedge \text{lc}$, then the loop must iterate at least N times (specified by $i \dot{=} 0 \rightarrow \langle w(\text{lc})\{b'\} \rangle i \geq N$) such that after iteration N the loop is in state s_{post} (specified by $\langle w(i < N)\{b'\} \rangle M \dot{=}_{\forall} M_{sk}$). The formula $M_{old} \dot{=}_{\forall} M$ stores the values of program variables $f_1, \dots, f_n \in M$ in the interpretation of the new function symbols $f_1^{old}, \dots, f_n^{old} \in M_{old}$ so that the invariant and the loop condition can refer to the values of program variables in state s_{pre} .

The loop branch precondition (LBPC) is an adaption of the CBPC to the here regarded case, i.e. where the branch to be tested occurs within a loop. If a state s_{pre} satisfies the strength condition and also the LBPC $M_{old} \dot{=}_{\forall} M \wedge I \wedge \text{lc} \wedge \{M\}(I \wedge \text{lc} \wedge \varphi)$, then the execution of the loop in this state results in at least N loop iterations such that during these iterations a state s_{post} is reached where φ is true. The proof of this lemma is similar to the proof of Theorem 1.

The strength condition is hard to be proved and generally cannot be proved automatically unless an existing invariant proof can be reused. The computation of the strength condition is however *not* part of the test generation procedure. The generation of test data is achieved by applying constraint solving to the CBPC or the LBPC only. The strength condition is part of the theoretical description of the approach.

In the following we refer to the CBPC and the LBPC with CBPC.

3.4. Using the CBPC if the Specification is Satisfied but Weak

According to Lemma 1 the DBPC requires the involved specification to be satisfied and allows then the detection of infeasible execution paths using theorem provers. According to Theorem 1 the CBPC requires the involved specification to be strong to provide a constraint for test data generation using a constraint solver that ensures the satisfaction of a desired branch condition.

We show that the CBPC can be used for test data generation even in use-cases (e.g. it may stem from a verification proof) where it is known that the specification constituting the CBPC is satisfied but not whether it is strong. Assume that the specification is satisfied but we do not know whether it is strong. Generating test data, i.e. a partial state that gives meaning to program variables, for which the CBPC is unsatisfiable is undesired. The reason is that if the specification is correct for this state, then it is guaranteed that either the branch condition is unsatisfied or the precondition is unsatisfied. In the latter case we can make no assumptions about the postcondition.

Theorem 2 *Let $\sigma = (pre, post, M)$ be a specification, $p \in \pi$, $M \supseteq Mod(p)$, $\varphi \in Fml$, and let $s \in S$ be a state.*

If σ is satisfied by p and $s \not\models CBPC_{\varphi}^{\sigma}$, then either $s \not\models pre$ or $s \not\models \langle p \rangle \varphi$

The theorem states that if the specification is satisfied but eventually weak, then the CBPC reduces the search space of the constraint solver appropriately. The CBPC does not guarantee that every model $s \in S$ of the CBPC ensures $s \models \langle p \rangle \varphi$ but if a model does *not* satisfy the CBPC, then either (1) the precondition of the contract is not satisfied, making the contract useless, or (2) the branch condition is not satisfied which is undesired as well.

Proof. Let $s_{\Sigma \setminus M_{sk}} \in S_{\Sigma \setminus M_{sk}}$ be a partial state where CBPC is unsatisfiable. This means that for each partial state $s_{M_{sk}} \in S_{M_{sk}}$ the CBPC is not satisfied in the total state $s = (s_{\Sigma \setminus M_{sk}} \cup s_{M_{sk}}) \in S$ (see Section 2.1):

$$\begin{aligned} s_{\Sigma \setminus M_{sk}} &\not\models pre \wedge \{M := M_{sk}\}(post \wedge \varphi) \\ s_{\Sigma \setminus M_{sk}} &\models \neg pre \vee \{M := M_{sk}\}\neg(post \wedge \varphi) \\ s_{\Sigma \setminus M_{sk}} &\models pre \rightarrow \{M := M_{sk}\}(post \rightarrow \neg \varphi) \end{aligned}$$

Since $pre \rightarrow \{M := M_{sk}\}post \rightarrow \neg \varphi$ is the DBPC $_{\neg \varphi}$ and since we assume that the specification is satisfied Lemma 1 implies $s_{\Sigma \setminus M_{sk}} \models \neg pre \vee \langle p \rangle \neg \varphi$. Note that $\langle p \rangle \neg \varphi$ implies $\neg \langle p \rangle \varphi$ (see Section 2.1). ■

4. Applications of the CBPC

4.1. Branch Precondition of Loops and Methods

In this section we show how to compute a precondition for a branch condition that follows a method call or loop by constructing the CBPC. Given is a pre- and postcondition $pre_{pq}, post_{pq} \in Fml_{FOL}$ for a sequence of statements $p; q;$, and a specification of p : $\sigma = (pre_p, post_p, M)$. Testing the program $p; q;$ wrt. $(pre_{pq}, post_{pq}, \Sigma_{nr})$ is equivalent to testing the DL-formula: $pre_{pq} \rightarrow \langle p; q; \rangle post_{pq}$.

The goal is to test the formula such that the test cases ensure the execution of each branch in q . Symbolic execution of q yields branches that using Dynamic Logic can be represented in form of formulae as follows

$$\begin{aligned} pre_{pq} &\rightarrow \langle p \rangle (\varphi_1 \rightarrow \langle q1 \rangle post_{pq}) \\ &\vdots \\ pre_{pq} &\rightarrow \langle p \rangle (\varphi_n \rightarrow \langle qN \rangle post_{pq}) \end{aligned}$$

where $\varphi_1, \dots, \varphi_n \in Fml$ are the *branch conditions* for executing the sub-programs $q1, \dots, qN \in \pi$ in q . Using the specification σ and a branch condition φ_i with $i \in 1 \dots n$ the CBPC $_{\varphi_i}$ can be constructed according to Def. 7. If σ is strong, then using Theorem 1 the precondition of a branch $pre_{pq} \rightarrow \langle p \rangle (\varphi_i \rightarrow \langle q1 \rangle post_{pq})$ is obviously the CBPC $_{\varphi_i}$:

$$pre_{pq} \wedge pre_p \wedge \{M := M_{sk}\}(post_p \wedge \varphi_i)$$

because it ensures in states where it is true that also $\langle p \rangle \varphi_i$ is true.

As an example we construct the branch precondition that ensures the execution of $C()$ in listing 2 of Figure 1. The initial formula to test is:

$$\underbrace{true}_{pre_{pq}} \rightarrow \underbrace{\langle i=0; D(); \rangle}_p \underbrace{\langle \text{if } (i==20) \{C(); \} \rangle}_q \underbrace{true}_{post_{pq}}$$

We make the trivial choices for pre_{pq} and $post_{pq}$ in order to simplify this example. Symbolic execution of the *if*-statement yields two branches according to the case distinction of the *if*-statement.

$$\underbrace{\langle i=0; D(); \rangle}_p \underbrace{(i \doteq 20)}_{\varphi_1} \rightarrow \underbrace{\langle C(); \rangle}_{q1} \underbrace{true}_{post_{pq}}$$

and

$$\underbrace{\langle i=0; D(); \rangle}_p \underbrace{(\neg(i \doteq 20))}_{\varphi_2} \rightarrow \underbrace{\langle \rangle}_{q2} \underbrace{true}_{post_{pq}}$$

The interesting branch is where $i \doteq 20$ is true after the execution of $D()$. In order to compute the precondition we use the specification of $D()$ consisting of the pre- and postconditions: $i < n$ and $i \doteq n$. Note that the postcondition of $D()$ and the branch condition $i \doteq 20$ are unrelated formulae, i.e. neither $i \doteq n$ implies $i \doteq 20$ nor does $i \doteq 20$ imply $i \doteq n$. The CBPC is

$$i < n \wedge \{i := i_{sk}\}(i \doteq n \wedge i \doteq 20)$$

Applying the update results in the formula $i < n \wedge i_{sk} \doteq n \wedge i_{sk} \doteq 20$ which in fact implies the desired branch precondition $i < 20 \wedge n \doteq 20$. This example is almost identical for listing 1 using a loop invariant.

4.2. Loop Branch Precondition

The loop branch precondition (LBPC) is a precondition of a loop that ensures that *during* the execution of the loop a certain branch is taken, e.g. to execute $A()$ in listing 1 of Figure 1. In the following example we don't prove the strength condition but make use of Theorem 2. The theorem tells us that by using a satisfied specification in the construction of the LBPC (or CBPC) the search space of the constraint solver is reduced in an appropriate way to increase the likelihood for satisfying the branch condition φ .

JAVA (3)

```

1 MyHashMap incHM(IDObj[] a) {
2   int i = 0; MyHashMap map = new MyHashMap();
3   while(i < a.length) {
4     if(map.count > (map.size*3)/4) {
5       tmp = new MyHashMap(map.size*2);
6       tmp.copyFrom(map); map = tmp;
7     }
8     map.put(a[i].id, a[i]); i++;
9   }
10 }
```

JAVA

We assume the following behavior of the program in listing 3. The constructor $\text{MyHashMap}()$ creates a hash map object with the initial capacity $\text{map.size} \doteq 8$. The field map.count tracks the current number of elements in the map. If during loop iteration the branch condition $\text{map.count} > (\text{map.size} * 3) / 4$ becomes true, then a new hash map of size $\text{map.size} * 2$ is created. The statement $\text{tmp.copyFrom}(\text{map})$; copies all elements from the old hash map to the new hash map so that $\text{tmp.count} \doteq \text{map.count}$. The method $\text{put}(x, y)$ stores y in the hash map under key x and overwrites any previously stored entry under the same key.

The goal is to compute the CBPC for the specification of the loop and branch condition $\text{map.size} * 3 / 4 < \text{map.count}$, which we abbreviate with φ . The specification of loop body chains before loop termination is $\sigma = \{I \wedge lc, I \wedge lc, M\}$ (see Section 2.4), where $M = \{\text{map}, \text{size}, \text{count}, i\}$, the loop condition lc is $i < a.length$, and I consists of the invariants

$$\underbrace{\text{map}^{old}.size^{old} \leq \text{map.size}}_{inv_1} \wedge \underbrace{\text{map.count} \leq N}_{inv_2} \wedge \underbrace{\text{map.count} \leq i}_{inv_3}$$

For the precondition $pre_{incHM} \in Fml_{FOL}$ of $incHM()$ we assume that $a \neq null$ and that $N \in \mathbb{N}$ is a lower bound on the number of distinct elements in the array a formalized

by

$$a \neq null \wedge \forall x. \left(0 \leq x < N \leq a.length \rightarrow \exists y. 0 \leq y < a.length \wedge a[y].id = x \right)$$

For this setting the resulting CBPC is:

$$\begin{aligned} & \text{map}^{old} \doteq \text{map} \wedge \text{size}^{old} \doteq \text{size} \\ & \wedge \text{count}^{old} \doteq \text{count} \wedge i^{old} \doteq i \wedge I \wedge lc \\ & \wedge \{M := M_{sk}\}(I \wedge lc \wedge \varphi) \end{aligned} \quad (8)$$

where $\{M := M_{sk}\}$ abbreviates the corresponding updates

$$\{\{\text{map}, \text{size}, \text{count}, i\} := \{\text{map}_{sk}, \text{size}_{sk}, \text{count}_{sk}, i_{sk}\}\}$$

Applying the update on $(I \wedge lc \wedge \varphi)$ yields the conjunction

$$\begin{aligned} & \underbrace{\text{map}^{old}.size^{old} \leq \text{map}_{sk}.size_{sk}}_{inv'_1} \wedge \underbrace{i_{sk} < a.length}_{lc'} \\ & \wedge \underbrace{\text{map}_{sk}.count_{sk} \leq N}_{inv'_2} \wedge \underbrace{\text{map}_{sk}.count_{sk} \leq i_{sk}}_{inv'_3} \\ & \wedge \underbrace{(\text{map}_{sk}.size_{sk} * 3) / 4 < \text{map}_{sk}.count_{sk}}_{\varphi'} \end{aligned}$$

that can be simplified as follows. From inv'_1 , φ' , and inv'_2 follows $(\text{map}^{old}.size^{old} * 3) / 4 < N$. From phi' , inv'_3 , and lc' follows $(\text{map}^{old}.size^{old} * 3) / 4 + 1 < a.length$. Using the equations in (8) we can replace $\text{map}^{old}.size^{old}$ with map.size deriving the important constraint from the CBPC:

$$(\text{map.size} * 3) / 4 < N \wedge (\text{map.size} * 3) / 4 + 1 < a.length \quad (9)$$

In order to transfer the CBPC into the pre-state of $incHM()$ the effect of line 2 has to be taken into account. Symbolic execution of line 2 results in a sequence of updates. One of them is $\{\text{map.size} := 8\}$ according to our description of the listing. Applying the update on (9) and respecting pre_{incHM} results in the desired precondition of $incHM()$:

$$pre_{incHM} \wedge (8 * 3) / 4 < N \wedge (8 * 3) / 4 + 1 < a.length$$

This precondition of $incHM()$ guarantees that the *then*-branch of the *if*-statement is entered.

5. Related Work

This work is an extension of [9] and [4] —both developed within the KeY-project [5]. In [9] verification-based testing is introduced as a method for deriving test cases from verification proofs. In [4] we present a white-box testing approach which combines verification-based specification inference and black-box testing allowing to combine different coverage criteria. Both approaches consider the derivation

of test cases based on loop invariants by example but not in the depth as it is done in this work.

Other approaches using symbolic execution to derive test cases are, e.g. [16, 15, 7]. The tools Symstra [16] and Unit Meister [15] use a bound on the number of analyzed loop iterations and [7] uses a bound on the size of the analyzed data structures. The latter approach yields an implicit bound on set of symbolically executed paths.

How to compute a precondition for a given condition after the execution of a program has been a research topic especially in the context of weakest precondition calculi for the purpose of software verification [8, 14, 11]. The *disjunctive branch precondition* (DBPC) is weaker than the weakest preconditions and the *conjunctive branch precondition* CBPC is stronger than the negated weakest precondition of the negated branch condition.

The generation of test cases based on specifications has been suggested, e.g., in [2] and [13]. In [2] a software development methodology is described whose main artifacts are invariants. In [13] the generated test cases are based on *effect predicates* derived from executable specifications. *Effect predicates* are related to strong specifications in this paper. We use however first-order logic based specifications and executable JAVA source code.

6. Conclusion

We have shown how test cases can be generated for testing program branches that occur within loops and after the execution of loops or complex methods. The challenge is to generate a precondition for a branch condition of a program branch. This may be infeasible or impossible with precondition computation based on symbolic execution. Our approach is to compute the *conjunctive branch precondition* (CBPC), which is a precondition generated from a branch condition and a user-provided specification. The CBPC is stronger than the *disjunctive branch precondition* (DBPC) that we use for detecting infeasible execution paths.

Our contributions are the described approach, the analysis of the two kinds of preconditions, i.e. CBPC and DBPC, their relation to the weakest precondition, and the required properties of the involved specifications. We proved the correctness of a theorem for test data generation (Theorem 1), a rule for detecting infeasible execution paths (Rule 3), and another theorem showing the relation between the CBPC and DBPC (Theorem 2). The CBPC can improve test case generation even if the F-CBPC, which additionally requires the involved specification to be strong, is not satisfied. A suitable use-case for our approach is, e.g., the generation of test cases from verification proofs because the CBPC occurs, except for some details, as a sub-formula in proof branches. The approach is implemented in the KeY-system and has been successfully tested on small examples.

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] R.-J. Back, J. Eriksson, and M. Myreen. Testing and verifying invariant based programs in the SOCOS environment. In Y. Gurevich and B. Meyer, editors, *Proceedings, Tests and Proofs, Zürich, Switzerland*, LNCS. Springer, 2007.
- [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 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.
- [5] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.
- [6] D. Cansell and D. Méry. Foundations of the B method. *Computers and Artificial Intelligence*, 22(3), 2003.
- [7] 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.
- [8] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.
- [9] 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.
- [10] 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.
- [11] B. Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. *J. Log. Algebr. Program.*, 58(1-2):61–88, 2004.
- [12] P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *LPAR*, pages 422–436, 2006.
- [13] M. Satpathy, M. Butler, M. Leuschel, and S. Ramesh. Automatic testing from formal specifications. In Y. Gurevich and B. Meyer, editors, *Proceedings, Tests and Proofs, Zürich, Switzerland*, LNCS. Springer, 2007.
- [14] K.-D. Schewe and B. Thalheim. A generalization of djikstra’s calculus to typed program specifications. In *FCT*, pages 463–474, 1999.
- [15] N. Tillmann and W. Schulte. Parameterized unit tests with Unit Meister. In *ESEC/SIGSOFT FSE*, pages 241–244, 2005.
- [16] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings, Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Edinburgh, UK*, LNCS 3440, pages 365–381. Springer, 2005.