HAL
open science

Test Data Generation for Programs with Quantified First-Order Logic Specifications

Christoph D. Gladisch

To cite this version:

Christoph D. Gladisch. Test Data Generation for Programs with Quantified First-Order Logic Specifications. 22nd IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS), Nov 2010, Natal, Brazil. pp.158-173, 10.1007/978-3-642-16573-3_12 . hal-01055252

HAL Id: hal-01055252

https://inria.hal.science/hal-01055252v1

Submitted on 12 Aug 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Test Data Generation For Programs with Quantified First-order Logic Specifications

Christoph D. Gladisch*

University of Koblenz-Landau
Department of Computer Science
Germany

Abstract. We present a novel algorithm for test data generation that is based on techniques used in formal software verification. Prominent examples of such formal techniques are symbolic execution, theorem proving, satisfiability solving, and usage of specifications and program annotations such as loop invariants. These techniques are suitable for testing of small programs, such as, e.g., implementations of algorithms, that have to be tested extremely well.
In such scenarios test data is generated from test data constraints which are first-order logic formulas. These constraints are constructed from path conditions, specifications, and program annotation describing program paths that are hard to be tested randomly. A challenge is, however, to solve quantified formulas. The presented algorithm is capable of solving quantified formulas that state-of-the-art satisfiability modulo theory (SMT) solvers cannot solve. The algorithm is integrated in the formal verification and test generation tool KeY.

1 Introduction

Testing has been influenced in the last decade by formal methods. Prominent examples of such formal techniques are symbolic execution, theorem proving, satisfiability solving, and the usage of formal specifications and program annotations such as loop and class invariants. Formal testing techniques can achieve a high code coverage or they can generate a low number of tests that very likely exhibit software faults. Such techniques generate test data constraints which are first-order logic (FOL) formulas. These constraints are constructed from path conditions, specifications, and program annotation and describe program paths that are hard to be tested randomly.

Satisfiability modulo theory (SMT) solvers are state-of-the-art techniques for generating models of FOL formulas. A model is a FOL interpretation in which a formula evaluates to true. A major bottleneck is, however, the handling of quantifiers (see, e.g., [21, 22]). SMT solvers can often create models for quantified formulas if *one* theory is involved. Quantifiers and multiple theories, however, often lead to problems that are not in the decidable fragments of the solvers. In such cases an SMT solver cannot generate a model for the formula.

* gladisch@uni-koblenz.de

```
—— Java + JML ————————
public class C{
 private String[] s;
 /*@ invariant
      s.length>=10;*/
 ...
}
———————— Java + JML ——
```

$$\left(\begin{array}{l}\forall o : \texttt{C}.o \neq \texttt{null} \rightarrow \\ \quad \forall i : \texttt{int}.0 \leqslant i \leqslant \texttt{length}(\texttt{s}(o)) \rightarrow \\ \qquad\qquad \texttt{acc}_{[]}(\texttt{s}(o), i) \neq \texttt{null}\end{array}\right) \quad (1)$$

$$\forall o : \texttt{C}.o \neq \texttt{null} \rightarrow \texttt{length}(\texttt{s}(o)) \geqslant 10 \quad (2)$$

Fig. 1. (left) A field declaration and a class invariant; (right) Quantified formulas occurring in test data constraints generated by KeY

For example, Figure 1 shows a Java class with a field declaration and a JML [18] specification of a class invariant. From the field declaration and the class invariant the tool KeY[2, 16] generates the formulas (1) and (2), respectively. These formulas are part of test data constraints. In this approach Java-fields and arrays are modelled as uninterpreted functions in first-order logic, hence, FOL interpretations and program states are the same concept. Formula (1) follows JML's semantics and expresses that the elements of the array field `s` are not `null`. Formula (2) expresses the class invariant, that for all objects of class `C` the array `s` has 10 or more elements.

When generating a test for some method of class `C`, the test data constraints have to be satisfied by the test data. The problem is, however, that state-of-the-art SMT solvers, concretely we have tested Z3 [5], CVC3 [1], Yices [9], are not capable to solve (1) or (2). Although SMT solvers can solve quantified formulas in certain cases, (1) and (2) are not in the decidable logic fragment of the solvers. Note, that a different translation of the code in Figure 1 could create formulas that are solvable by an SMT solver, but the general problem of solving quantified formulas remains.

The contribution of this paper is not a technique to derive test cases or test data constraints. Those techniques are cited in Section 1.1. Instead, our contribution is the handling of quantified formulas for solving provided test data constraints. We propose a model generation algorithm that is not explicitly restricted to a specific class of formulas and which can therefore solve more general formulas than SMT solvers can solve.

The basic idea of our approach is to generate a partial FOL model in which a quantified formula that we want to eliminate evaluates to true. SMT solvers can then solve the remaining ground formulas. The representation of a model in our approach is a program. Our technique generates candidate programs and checks if a candidate program satisfies the test data constraint. For instance, in order to satisfy formula (2) we could generate the following Java statement

```
for (C o : Cs){o.s = new String [10] ;}        (3)
```

where `Cs` is a collection of objects of class `C`, and verify it against formula (2). A programming language such as Java is, however, not *directly* suitable for this

task because (a) function and predicate symbols are usually not part of such languages and (b) loop invariants would have to be generated for the verification proof. A language and a calculus that are suitable for our purpose exist, however, in the verification system KeY. The language consists of so-called *updates*.

Structure of the paper. In the following we describe the background of our work as well as related work. In Section 2 the underlying formalism of our approach is introduced. The main section is Section 3 where we describe our algorithm. In Section 4 we report on experiments with our approach and provide conclusions and further research plans in Section 5.

1.1 Background and Related Work

The work presented in the paper has been developed in the KeY project [16]. The KeY system [2, 16] is a verification and test generation tool for Java. The tool can automatically generate JUnit tests from proof structures [10]. The test data is generated from FOL constraints which combine execution path conditions with annotations such as method specification, class invariants, and loop invariants [13, 14]. Hence, the approach is a grey-box testing technique. So far we have used the theorem prover Simplify [7] to generate test data but the so-generated test data is not always guaranteed to satisfy the constraints as will be shown below. On the other hand, we found that more recent SMT solvers such as Z3 [5], CVC3 [1], Yices [9] are not capable to solve the constrains either, which was the motivation for this work.

There exist several other tools that follow similar ideas as the KeY tool to generate test data constraints and have therefore similar requirements on test data generation. KUnit [6] is an extension of Bogor/Kiasan which combines symbolic execution, model checking, theorem proving, and constraint solving. Check'n'Crash generates JUnit tests for assertions that could not be proved using ESC/Java2 [4]. Java PathFinder is an explicit-state model checker and features the generation of test inputs [24]. Non-trivial FOL formulas may also occur in functional testing or model-based testing. A survey of search-based test data generation techniques is given in [19]. These techniques are powerful for traditional testing approaches but they do not handle test data generation for constraints with quantified FOL formulas.

The main contribution of our work is the handling of quantifiers. One has to distinguish between different quantifiers in different contexts, namely between those that can be skolemized and those that cannot be skolemized. The tricky cases are the handling of (a) existential quantification when showing validity and (b) universal quantification when generating models. In order to handle case (a) some instantiation(s) of the quantified formulas can be created *hoping* to complete the proof. Soundness is preserved by any instantiation. The situation in case (b) which occurs when generating test data is, however, worse when using instantiation-based methods, because these methods are sound only if a complete instantiation of the quantified formula is guaranteed.

A popular instantiation heuristic is E-matching [21] which was first used in the theorem prover Simplify. E-matching is, however, not complete in general.

In general a quantified formula $\forall x.\varphi(x)$ cannot be substituted by a satisfiability preserving conjunction $\varphi(t_0) \wedge \ldots \wedge \varphi(t_n)$ where $t_0 \ldots t_n$ are terms computed via E-matching. For this reason Simplify may produce unsound answers (see also [17]) as shown in the following example.

$$\forall h.\forall i.\forall v.select(store(h,i,v),i) = v \tag{4}$$

$$\forall h.\forall j.0 \leqslant select(h,j) \wedge select(h,j) \leqslant 2^{32} - 1 \tag{5}$$

Formula (4) is an axiom of the theory of arrays and (5) specifies that all array elements of all arrays have values between 0 and $2^{32} - 1$. The first axiom is used to specify heap memory in [20]. Formula (5) seems like a useful axiom to specify that all values in the heap memory have lower and upper bounds, as it is the case in computer systems. However, the conjunction (4) $\wedge$ (5) is inconsistent, i.e. it is false, which can be seen when considering the following instantiation $[h := store(h_0, k, 2^{32}), j := k]$, (see [20]). Simplify, however, produces a counter example for $\neg((4) \wedge (5))$, which means that it satisfies the *false* formula (4) $\wedge$ (5). E-matching may be used for sound satisfiability solving when a complete instantiation of quantifiers is ensured. For instance, completeness of instantiation via E-matching has been shown for the Bernays-Schönfinkel class in [11]. An important fragment of FOL for program specification which allows a complete instantiation is the Array Property Fragment [3].

Quantifier elimination techniques, in the *traditional* sense, replace quantified formulas by *equivalent* ground formulas, i.e. without quantifiers. Popular methods are, e.g., the Fourier-Motzkin quantifier elimination procedure for linear rational arithmetic and Cooper's quantifier elimination procedure for Presburger arithmetic (see, e.g., [12] for more examples). These techniques are, in contrast to the proposed technique, not capable of eliminating the quantifier in, e.g., (1) or (2). Since first-order logic is only semi-decidable, equivalence preserving quantifier elimination is possible only in special cases. The transformation of formulas by our technique is not equivalence preserving. The advantage of our technique is, however, that it is not restricted to a certain class of formulas.

Finally, Finite Model Finding methods such as [25] regard the finite domain version of the satisfiability problem in first-order logic. Our approach handles, however, also infinite domains.

2 KeY's Dynamic Logic with Updates

The KeY system is based on the logic Java Card DL, which is an instance of Dynamic Logic (DL) [15]. Dynamic Logic is an extension of first-order logic with modal operators. The ingredients of the KeY system that are needed in this paper are first-order logic (FOL) extended by the modal operators *updates* [23].

Notation. We use the following abbreviations for syntactic entities: $\boldsymbol{V}$ is the set of (logic) variables; $\boldsymbol{\Sigma^f}$ is the set of function symbols; $\boldsymbol{\Sigma^f_r} \subset \Sigma^f$ is the set of rigid function symbols, i.e. functions with a fixed interpretation such as, e.g., '0', 'succ', '+'; $\boldsymbol{\Sigma^f_{nr}} \subset \Sigma^f$ is the set of non-rigid function symbols, i.e.

uninterpreted functions; $\boldsymbol{\Sigma^p}$ is the set of predicate symbols; $\boldsymbol{\Sigma}$ is the signature consisting of $\Sigma^f \cup \Sigma^p$; $\boldsymbol{Trm_{FOL}}$ is the set of FOL terms; $\boldsymbol{Trm} \supset Trm_{FOL}$ is the set of DL terms; $\boldsymbol{Fml_{FOL}}$ is the set of FOL formulas; $\boldsymbol{Fml} \supset Fml_{FOL}$ is the set of DL formulas; $\boldsymbol{U}$ is the set of updates; $\doteq$ is the equality predicate; and $=$ is syntactic equivalence. The following abbreviations describe semantic sets: $\boldsymbol{\mathcal{D}}$ is the FOL domain or universe; $\boldsymbol{\mathcal{S}}$ is the set of states or equivalently the set of FOL interpretations. To describe semantic properties we use the following abbreviations: $\boldsymbol{val_s(t)} \in \mathcal{D}$ is the valuation of $t \in Trm$ and $\boldsymbol{val_s(u)} \in \mathcal{S}$ is the valuation of $u \in U$ in $s \in \mathcal{S}$; $\boldsymbol{s} \models \boldsymbol{\varphi}$ means that φ is true in state $s \in \mathcal{S}$; $\models \boldsymbol{\varphi}$ means that φ is valid, i.e. for all $s \in \mathcal{S}$, $s \models \varphi$; and $\equiv$ is semantic equivalence.

Updates capture the essence of programs, namely the state change computed by a program execution. States and FOL interpretations are the same concept. An update changes the interpretation of symbols Σ^f_{nr} (such as uninterpreted functions). Hence, updates represent partial states and can be used to represent (partial) models of formulas. The set Σ^f_r represents rigid functions whose interpretation is fixed and cannot be changed by an update.

For instance, the formula $(\{a := b\}a = c) \in Fml$, where $a \in \Sigma^f_{nr}$ and $b, c \in \Sigma^f$ consists of the (function) update $a := b$ and the *application* of the update modal operator $\{a := b\}$ on the formula $a = c$. The meaning of this *update application* is the same as that of the weakest precondition $wp(a := b, a = c)$, i.e. it represents all states such that after the assignment $a := b$ the formula $a = c$ is true which is equivalent to $b = c$.

Definition 1. *Syntax. The sets U, Trm and Fml are inductively defined as the smallest sets satisfying the following conditions. Let $x \in V$; $u, u_1, u_2 \in U$; $f \in \Sigma^f_{nr}$; $t, t_1, t_2 \in Trm$; $\varphi \in Fml$.*

- *Updates. The set U of updates consists of: function updates $(f(t_1, \ldots, t_n) := t)$, where $f(t_1, \ldots, t_n)$ is called the* location term *and t is the* value term; *parallel updates $(u_1 \,||\, u_2)$; conditional updates* (`if` φ; u)*; and quantified updates* (`for` x; φ; u).
- *Terms. The set of Dynamic Logic terms includes all FOL terms, i.e. $Trm \supset Trm_{FOL}$; and $\{u\}t \in Trm$ for all $u \in U$ and $t \in Trm$.*
- *Formulas. The set of Dynamic Logic formulas includes all FOL formulas, i.e. $Fml \supset Fml_{FOL}$; $\{u\}\varphi \in Fml$ for all $u \in U$, $\varphi \in Fml$; and sequents $\Gamma \Longrightarrow \Delta$, where $\Gamma \subset Fml$ is called antecedent and $\Delta \subset Fml$ is called succedent.*

A sequent $\Gamma \Longrightarrow \Delta$ is equivalent to the formula $(\gamma_1 \wedge \ldots \wedge \gamma_n) \rightarrow (\delta_1 \vee \ldots \vee \delta_m)$, where $\gamma_1, \ldots, \gamma_n \in \Gamma$ and $\delta_1, \ldots, \delta_m \in \Delta$. Sequents are normally, e.g. in [2] , not included in the set of formulas. However, in this work it is convenient to include them to the set of formulas as *syntactic sugar*.

Definition 2. *Semantics. We use the notation from Def. 1, further let $s, s' \in \mathcal{S}$; $v, v_1, v_2 \in \mathcal{D}$; $x, x_i, x_j \in V$; and $\varphi(x)$ and $u(x)$ denote a formula resp. an update with an occurrence of x.*

Terms and Formulas:

- $val_s(\{u\}t) = val_{s'}(t)$, *where* $s' = val_s(u)$

- $val_s(\{u\}\varphi) = val_{s'}(\varphi)$, *where* $s' = val_s(u)$

  Updates:

- $val_s(f(t_1, \ldots, t_n) := t) = s'$, *where* $s' = s$ *except the interpretation of* f *is changed such that* $val_{s'}(f(t_1, \ldots, t_n)) = val_s(t)$
- $val_s(u_1; u_2) = s'$, *there is* s'' *with* $s'' = val_s(u_1)$ *and* $s' = val_{s''}(u_2)$
- $val_s(u_1 \,||\, u_2) = s'$. *We define* s' *by the interpretation of terms* t.
  Let $v_0 = val_s(t)$, $v_1 = val_s(\{u_1\}t)$, *and* $v_2 = val_s(\{u_2\}t)$.

  $$\textit{If } v_0 \neq v_2 \textit{ then } val_{s'}(t) = v_2 \textit{ else } val_{s'}(t) = v_1.$$

- $val_s(\texttt{if}\ \ \varphi;\, u) = s'$, *if* $val_s(\varphi) = true$ *then* $s' = val_s(u)$, *otherwise* $s' = s$.
- *Intuitively, a quantified update* $(\texttt{for}\ x;\, \varphi(x);\, u(x))$ *is equivalent to the infinite composition of parallel updates (parallel updates are associative):*

  $$\ldots \,||\, (\texttt{if}\ \ \varphi(x_i);\, u(x_i)) \,||\, (\texttt{if}\ \ \varphi(x_j);\, u(x_j)) \,||\, \ldots$$

  satisfying some global order $\succ$ *such that* $\beta(x_i) \succ \beta(x_j)$, *where* $\beta : V \to \mathcal{D}$.

A complete and formal definition of quantified updates cannot be given in the scope of this paper; we refer the reader to [23, 2] for a complete definition of the language and the simplification calculus. In the following some examples are shown of how updates, terms, and formulas are evaluated in KeY respecting the given semantics in Def 2.

- $\{f(1) := a\}f(2) = f(1)$ simplifies to $f(2) = a$.
- $\{f(b) := a\}P(f(c))$ simplifies to $(b \doteq c \to P(a)) \wedge (\neg b \doteq c \to P(f(c)))$.
- $\{f(a) := a\}f(f(f(a)))$ simplifies to a.
- $\{u_1; f(t_1, \ldots, t_n) := t\}$ is equivalent to $\{u_1 \,||\, f(\{u\}t_1, \ldots, \{u\}t_n) := \{u\}t\}$.
- $\{f(1) := a \,||\, f(2) := b\}f(2) = f(1)$ simplifies to $b = a$.
- $\{f(1) := a \,||\, f(1) := b\}f(2) = f(1)$ simplifies to $f(2) = b$, i.e. the last update *wins* in case of a conflict.
- $\{\texttt{if}\ \ \varphi;\, f(b) := a\}P(f(c))$ simplifies to $\varphi \to \{f(b) := a\}P(f(c))$.
- $\{\texttt{for}\ x;\, 0 \leqslant x \wedge x \leqslant 1;\, f(x) := x\}$ is equivalent to $\{f(1) := 1 \,||\, f(0) := 0\}$.

3 Test Data Generation For Quantified Formulas

The basic idea of our approach is to generate a partial FOL model in which a quantified formula that we want to eliminate evaluates to true. A set of quantified formulas can be eliminated, i.e. evaluated to true, by successive extensions of the partial model. This approach can be continued also on ground formulas to generate complete models. While this basic idea is simple, the interesting questions are: how to represent the interpretations, how to generate (partial) models, and what calculus is suitable in order to evaluate formulas under those (partial) interpretations.

A suitable language for this purpose with a simplification calculus are updates. In order to generate test data that satisfies a test data constraint $\varphi \in Fml_{FOL}$, our approach is to generate an update u, such that $\{u\}\varphi$ evaluates to true. If such an update exists, then φ is satisfiable and the update represents a test preamble initializing a state in which φ evaluates to true (see Section 3.3).

Example 1. Referring to Figure 1, models for satisfying formulas (2) and (1) can be represented by the following update applications, respectively.

$$\{\texttt{for}\ o : \texttt{C};\ \neg o \doteq \texttt{null};\ \texttt{l}(\texttt{s}(o)) := 10\}(2) \tag{6}$$

$$\{\texttt{for}\ o : \texttt{C};\ \neg o \doteq \mathit{null};\ (\texttt{for}\ i : \texttt{int};\ 0 \leqslant i \leqslant \texttt{l}(\texttt{s}(o));\ \texttt{acc}_{[]}(\texttt{s}(o), i) := \mathit{obj}_C(i))\}(1)$$

where `l` is an abbreviations of `length`, $\texttt{acc}_{[]}(\texttt{s}(o), i)$ encodes the array access `o.s[i]`, and $\mathit{obj}_C : \texttt{int} \to \texttt{C}$ is an injective function from numbers to objects.

Our technique uses heuristics for construction of candidate updates u and then verifies $\{u\}\varphi$. The advantage of using updates is the availability of the powerful update simplification calculus to automatically verify $\{u\}\varphi$. In particular it is not required to generate loop invariants in order to handle quantified updates. In this way different heuristics can be used in a search procedure for u while guaranteeing correctness of the test preamble in the end. The technique requires a theorem prover for FOL and an implementation of updates.

Definition 3. ***Procedure Th****. The procedure* `Th` *represents a theorem prover.*

- *Given a formula $\vartheta \in \mathit{Fml}$ as input,* $\texttt{Th}(\vartheta)$ *returns a set $\Theta \subset \mathit{Fml}_{FOL}$.*
- *If $\Theta = \emptyset$, then $\models \vartheta$.*
- *Each $\vartheta' \in \Theta$ is either a literal or a quantified formula.*
- *The prover may use only local equivalence rules. Global rules ensure that satisfiability of $\neg\vartheta'$ for any $\vartheta' \in \Theta$ ensures satisfiability of $\neg\vartheta$. Local equivalence rules ensure additionally that $\models (\neg\vartheta') \to (\neg\vartheta)$.*

The KeY system uses a sequent calculus. Therefore, in the following sections we assume that the set Θ returned by `Th` consists of sequents (see Def. 1). In the following two algorithms are described. Algorithm 1 in Section 3.1 extracts information from (quantified) formulas for update construction and invokes a theorem prover to verify $\{u\}\varphi$. Algorithm 1 queries Algorithm 2 to construct candidate updates based on information obtained from Algorithm 1. Algorithm 2 is described in Section 3.2. In order to keep the pseudo-code small we use indeterministic choice points, marked by the keyword **choose**, and assume a backtracking control-flow wrt. to these choice points. In this way we also separate the basic algorithm from concrete search heuristics. If a choice at a choice point cannot be made, e.g. when trying to select an element of an empty set, then the algorithms backtracks or stops with the result "unknown" resp. "$\emptyset$".

3.1 The Model Search Algorithm

Assume we want to generate test data resp. a model satisfying the input formula ϕ_{in}. The Algorithm 1 reformulates this problem as counter example generation for φ' (Line 1). In Line 4 the algorithm attempts to show $\models \varphi'$ (Line 4). If φ' is valid, then $\Phi = \emptyset$ and the algorithm stops (Line 5) because φ' has no counter example and ϕ_{in} is unsatisfiable. The other case is that the proof attempt of φ' results in a set of open, i.e. unproved, proof obligations Φ (Line 5).

Algorithm 1 modelSearch(ϕ_{in})

1: $\varphi' := \neg\phi_{in}$
2: solution := $\bot$
3: **loop**
4: $\quad \Phi := \texttt{Th}(\varphi')$
5: $\quad$ **choose** $\varphi \in \Phi$
6: $\quad$ **if** φ is ground **then**
7: $\quad\quad$ **if** GROUNDPROC($\neg\varphi$) = ("sat", groundmodel) **then**
8: $\quad\quad\quad$ **return** ("sat", groundmodel, solution)
9: $\quad\quad$ **else**
10: $\quad\quad\quad$ **backtrack** or **return** ("unknown", $\bot$, $\bot$)
11: $\quad\quad$ **end if**
12: $\quad$ **end if**
13: $\quad$ normalize φ such that quantified formulas appear only in the antecedent of φ
14: $\quad$ **choose** a quantified formula $\forall x.\phi(x)$ in φ, i.e. let $\varphi = (\Gamma, \underline{\forall x.\phi(x)} \Longrightarrow \Delta)$
15: $\quad \varphi' := (\Gamma, \underline{true} \Longrightarrow \Delta)$
16: $\quad \psi := (\Gamma \Longrightarrow \underline{\forall x.\phi(x)}, \Delta)$
17: $\quad \Psi := \texttt{Th}(\psi)$
18: $\quad$ **while** $\Psi \neq \emptyset$ **do**
19: $\quad\quad$ **choose** $\psi' \in \Psi$
20: $\quad\quad \Upsilon :=$ formulaToUpdate(ψ')
21: $\quad\quad$ **choose** $(u, \alpha) \in \Upsilon$
22: $\quad\quad$ solution := append (u, α) to solution
23: $\quad\quad \varphi' := (\alpha \rightarrow \{u\}\varphi')$
24: $\quad\quad \psi := (\alpha \rightarrow \{u\}\psi)$
25: $\quad\quad \Psi := \texttt{Th}(\psi)$
26: $\quad$ **end while**
27: **end loop**

In this case it is unknown if a model of ϕ_{in} exists or not. The proof obligations Φ result from case distinctions in the proof structure created by Th and contain valuable information, because they describe situations in which φ' potentially has counter examples.

In Line 5 the algorithm selects a formula $\varphi \in \Phi$. The goal is to create a counter example for φ, i.e. satisfy $\neg\varphi$, in order to satisfy ϕ_{in}. Ground formulas should be preferred at this choice point because they can be efficiently checked by a ground procedure such as an SMT solver. Otherwise, we assume φ is not ground. After normalization at Line 13 the antecedent of φ contains at least one universally quantified formula and all formulas of the succedent are ground. This normalization can be easily achieved by the equivalence $(\Gamma \Longrightarrow \exists x.\phi(x), \Delta) \equiv (\Gamma, \forall x \neg\phi(x) \Longrightarrow \Delta)$. A counter example for φ must satisfy the formulas in the antecedent, i.e. Γ and $\forall x.\phi(x)$. The algorithm selects a quantified formula $\forall x.\phi(x)$ from the antecedent of φ (Line 14) for which a model is generated in the following.

The core idea of this algorithm is to create an update u, such that $\{u\}\forall x.\phi(x)$ evaluates to true, and in this way to eliminate the quantified formula. The weak-

est condition under which $\forall x.\phi(x)$ evaluates to true in φ can be expressed as

$$\underbrace{(\Gamma, \forall x.\phi(x) \Longrightarrow \Delta)}_{\varphi} \leftrightarrow \underbrace{(\Gamma, true \Longrightarrow \Delta)}_{\varphi'} \tag{7}$$

which simplifies by equivalence transformations to

$$\underbrace{\Gamma \Longrightarrow \forall x.\phi(x), \Delta}_{\psi} \tag{8}$$

Any model of (8) is also a model of (7), which means that in such states $\forall x.\phi(x)$ evaluates to true. Hence, in Line 15 the formula φ' is constructed where the quantified formula is replaced by true. Substituting φ by φ' in subsequent computation is sound only if (7) or equivalently (8) is valid. Therefore formula (8) is assigned to ψ in Line 16 and is checked by `Th` in Line 17. If ψ can be proved, then $\Psi = \emptyset$ and the algorithm continues in Line 4 where φ' (now without the quantified formula) is used instead of φ. Otherwise, if the proof of (8) of does not close (Line 17), then the result is a set of proof obligations Ψ.

The formulas Ψ (Line 19) describe potential states in which $\forall x.\phi(x)$ does not evaluate to true. The goal is therefore to construct an update u (Line 20) such that for each formula $\psi' \in \Psi$, $\models \{u\}\psi'$. If this is the case, then also $\models \{u\}\psi$ which allows us to eliminate the quantified formula by the equivalence (7). Instead of satisfying this condition in one step, our heuristic is rather to extend u iteratively. In each iteration of the inner loop one formula $\psi' \in \Psi$ is selected in Line 19 and Ψ is updated in Line 25 until Ψ *eventually* decreases to $\emptyset$.

Remark. For the construction of the updates it is sometimes necessary to introduce and axiomatize fresh function symbols. For instance, it may be required to introduce a fresh function $notZero \in \Sigma^f$ with the axiom $\neg(notZero \doteq 0)$. An update is therefore associated with an axiom α (see Line 21).

The goal of the inner loop is to generate an update u and a formula α (Line 20) and to check if

$$\alpha \rightarrow \{u\}\psi \tag{9}$$

evaluates to true. Formula 9 (see Line 24) is a weakening of (8). The procedure `formulaToUpdate` which is described in Section 3.2 generates candidate pairs (u, a) that are likely to satisfy (9).[1] In (8), respectively (9), the quantified formula occurs negated wrt. (7). An important consequence of this negation is that in Lines 17 and 25 the theorem prover can skolemize the quantified formula (8) resulting in

$$\Gamma \Longrightarrow \phi(sk), \Delta \tag{10}$$

where $sk \in \Sigma_r^f$ is a fresh symbol. In this way the formula $\phi(sk)$ can be simplified by the calculus and information contained in the structure of $\phi(sk)$ is extracted

[1] Note that since the procedure `formulaToUpdate` uses only one formula $\psi' \in \Psi$ to construct the pair (u, α), formula 9 may not evaluate true. In this case the inner loop continues iteration and unsolved formulas $\psi' \in \Psi$ reapear in the next iteration to be solved.

to the sequent level, i.e. the boolean structure of $\phi(sk)$ is flattened (see Def. 3). This information occurs in the formulas $\psi' \in \Psi$ (Line 19). The task of generating a pair (u, α) from ψ' for satisfying (9) by the procedure `formulaToUpdate` is considerably simpler than generation of the pair from the whole unsimplified quantified formula $\forall x.\phi(x)$.

Example 2. Let $\varphi = (A, \forall x.\phi(x) \Longrightarrow B)$, where $A, B \in Fml_{FOL}$ and $\phi(x) = (f(x) > x \wedge g(x) < f(x))$. Generating a model for φ in one step is complicated because the quantified formula cannot be skolemized. In contrast, in $\psi = (A \Longrightarrow \forall x.\phi(x), B)$ the quantified formula is negated (because $(F \Longrightarrow) \equiv (\Longrightarrow \neg F)$) and can therefore be skolemized. $\mathtt{Th}(\psi)$ yields $\Psi = \{(A \Longrightarrow f(sk) > sk, B), (A \Longrightarrow g(sk) < f(sk), B)\}$. The structure of each $\psi' \in \Psi$ is simpler than of ψ. The procedure `formulaToUpdate` can then generate, e.g., the following updates with axioms to satisfy the formulas in Ψ respectively:

$$\{((\mathtt{for}\ x;\ true;\ f(x) := x - 1), true), ((\mathtt{for}\ x;\ true;\ g(x) := f(x) + 1), true)\}.$$

Checking formula (9) as described above is imporant because it is equivalent to

$$(\alpha \rightarrow \{u\}\varphi) \leftrightarrow (\alpha \rightarrow \{u\}\varphi') \tag{11}$$

which in turn is a weakening of (7). Accordingly, in Line 23 the formula φ' is updated. If (9) is valid, which is checked in Line 25, then the inner loop terminates and the outer loop continues execution. Hence, the original counter example generation problem for φ is replaced by the counter example generation problem for $\alpha \rightarrow \{u\}\varphi'$ where the quantified formula is eliminated, i.e. replaced by true. This is sound because if (9) is valid, then (11) is valid and therefore a counter example for $\alpha \rightarrow \{u\}\varphi'$ is a counter example for $\alpha \rightarrow \{u\}\varphi$. The latter implies that φ has a counter example as well which is formalized by the following proposition.

Proposition 1. *Let $u \in U$, $\alpha, \varphi' \in Fml$. If there is an $s \in \mathcal{S}$ such that $s \vDash \neg(\alpha \rightarrow \{u\}\varphi)$, then there is $s' \in \mathcal{S}$ such that $s' \vDash \neg\varphi$.*

Proof. Assume there is $s \in \mathcal{S}$ such that $s \vDash \neg(\alpha \rightarrow \{u\}\varphi)$, which implies that $s \vDash \alpha$ and $s \vDash \neg\{u\}\varphi$. The following is an equivalence in Dynamic Logic: $\neg\{u\}\phi \equiv \{u\}\neg\phi$ (see [2]). Using this equivalence we obtain the statement: there is $s \in \mathcal{S}$ such that $s \vDash \{u\}(\neg\varphi)$. According to Def. 2, there is $s' \in \mathcal{S}$ such that $s' = val_s(u)$, and $s' \vDash \neg\varphi$. ■

3.2 Update Generation For Satisfying Quantified Formulas

In this section we describe Algorithm 2 which is used by Algorithm 1 to construct updates for satisfying quantified formulas. According to Section 3.1 this algorithm receives as input a sequent ψ' that is an open proof obligation of $\mathtt{Th}\ (\psi)$. Algorithm 2 is queried for each open proof obligation and is expected to generate an update and axiom pair (u, α) that is *likely* to satisfy

$$\alpha \rightarrow \{u\}\psi'$$

Algorithm 2 formulaToUpdate(ψ')

1: let sk = the skolem function of ψ'
2: let $(\Gamma \Longrightarrow \Delta) = \psi'$
3: let $(\Gamma_{sk} \Longrightarrow \Delta_{sk}) \subset (\Gamma \Longrightarrow \Delta)$ (according to the description)
4: **choose** $\vartheta_{sk} \in (\neg\Gamma_{sk} \cup \Delta_{sk})$ (negation is applied to each element in Γ)
5: **choose** $\vartheta'_{sk} \in \texttt{solve}(\vartheta_{sk})$
6: **choose** $(u, \alpha) \in \texttt{concretize}(\vartheta'_{sk})$
7: **choose** $(u', \alpha) \in \texttt{toQuanUpd}(sk, (u, \alpha), (\Gamma_{sk} \Longrightarrow \Delta_{sk}), \vartheta_{sk})$
8: **return** (u', α)

Considering Example 2, if $\psi' = (A \Longrightarrow f(sk) > sk, B)$, then a suitable (u, α) pair is, e.g., $((\texttt{for}\ x;\ true;\ f(x) := x - 1), true)$.

As each pair (u, α) satisfies one of the open proof obligations $\psi' \in \texttt{Th}(\psi)$, a series of such pairs *eventually* satisfies formula (9). The algorithm returns a set of alternative solutions for each sequent ψ'. Important to note is that soundness of the approach is guaranteed by any pair (u, α) because the inner loop of Algorithm 1 does not terminate until a model is generated for ψ.

According to Def. 3 the sequent ψ' has been simplified such that all formulas on the sequent level are either quantified formulas or atoms. We are interested in atoms that were derived from $\forall x.\phi(x)$. Therefore our heuristic is to categorize those atoms in ψ' as highly relevant for the construction of the update u that have an occurrence of the skolem symbol sk, that was introduced in (10). Let ψ'_{sk} be defined as

$$\Gamma_{sk} \Longrightarrow \Delta_{sk}$$

such that it coincides with ψ', except that all quantified formulas and formulas that do not contain an occurrence of sk are removed in ψ'_{sk} (Line 3 of Algorithm 2). Hence, all formulas in Γ_{sk} and Δ_{sk} are ground formulas with an occurrence of sk. Following the Example 2, $(\Gamma_{sk} \Longrightarrow \Delta_{sk})$ is either $(\Longrightarrow f(sk) > sk)$ or $(\Longrightarrow g(sk) < f(sk))$.

The goal is to create an update u such that $\models (\{u\}\psi'_{sk})$. Note that $(\{u\}\psi'_{sk}) \rightarrow (\{u\}\psi')$. In order to evaluate $\{u\}\psi'_{sk}$ to true the update u must either evaluate an atom in Γ_{sk} to $false$, or an atom in Δ_{sk} to $true$. We refer to the chosen atom, whose interpretation we want to manipulate, as the *core atom*. Let ϑ_{sk} denote the chosen core atom (Line 4). The task is to construct a function update u such that $\{u\}\vartheta_{sk}$ evaluates $true$. This task is divided into two steps realized by the algorithms `solve` (Line 5) and `concretize` (Line 6).

Definition 4. ***Procedure Solve.*** *Given an atom ϑ_{sk}, whose top-level symbol is a relation $R \in \Sigma^p$, the procedure* `solve` *constructs a set of atoms such that for each atom $\vartheta'_{sk} \in \Theta$ holds*

- $\vartheta'_{sk} = R'(\underline{f(t_1, \ldots, t_n)}, v)$, *i.e. syntactic equivalence*
- $\vartheta'_{sk} \doteq \vartheta_{sk}$, *i.e. semantic equivalence*

where $R' \in \Sigma^p_r$, $f \in \Sigma^f_{nr}$, $f \neq sk$, and $t_1, \ldots, t_n, v \in Trm$.

The procedure `solve` creates normal forms for core atoms. For example, for the formula $\vartheta_{sk} = (f(sk) + 3 < g(sk) - sk)$, with "$<$"$\in \Sigma^p$, $f, g \in \Sigma^f_{nr}$, the procedure `solve` should generate, e.g., the following set:

$$\{\underline{f(sk)} < (g(sk) - sk - 3), \underline{g(sk)} > (f(sk) + 3 + sk)\} \tag{12}$$

Note that the procedure `solve` is part of our heuristic and the resulting set is not strictly defined, it may also be empty. Some core atoms may be not solvable by the procedure but the more results the procedure `solve` produces the better is the chance of generating a suitable update.

Definition 5. ***Procedure Concretize.*** *Let $R \in \Sigma^p_r$, $f \in \Sigma^f_{nr}$, and $t_1, .., t_n, v \in Trm$. Given an atom of the form $R(f(t_1, \ldots, t_n), v)$ the procedure* `concretize` *creates a set of pairs (u, α), with $u \in U$, $\alpha \in Fml$, such that:*

- $u = (f(t_1, \ldots, t_n) := value)$, *where* $value \in Trm$
- $\alpha \rightarrow \{u\}R(f(t_1, \ldots, t_n), v)$ *evaluates to true*

The procedure `concretize` creates for a given normalized core atom ϑ'_{sk} an update u that evaluates $\{u\}\vartheta'_{sk}$ to *true*. E.g., if the normalized core atom is of the form $t_1 = t_2$, using infix notation, then the result of the procedure `concretize` is simply $((t_1 := t_2), true)$. In some cases it is desired to introduce fresh symbols and to axiomatize them for the construction of the term *value*. Such axiomatizations are collected in the formula α.

For example, using the normalized core atom $\vartheta'_{sk} = (f(sk) < (g(sk) - sk - 3))$ from the solution set of the previous example, the procudure `concretize` may produce, e.g., the following solution:

$$\{(\underbrace{f(sk) := (g(sk) - sk - 3) + sk_2)}_{u}, \underbrace{sk_2 < 0}_{\alpha}\} \tag{13}$$

where $sk_2 \in \Sigma^f_r$ is a fresh constant. Using this solution, we can evaluate $\alpha \rightarrow \{u\}\vartheta'_{sk}$ as follows

$$\begin{aligned}
&sk_2 < 0 \rightarrow \{f(sk) := (g(sk) - sk - 3) + sk_2\}(f(sk) + 3 < g(sk) - sk)\\
&sk_2 < 0 \rightarrow (g(sk) - sk - 3) + sk_2 + 3 < g(sk) - sk\\
&sk_2 < 0 \rightarrow sk_2 < 0
\end{aligned}$$

The next step uses the result computed by the procedures `solve` and `concretize` in order to create a quantified update (Line 7).

Definition 6. ***Procedure toQuanUpd.*** *Given a tuple (u, α), with $u \in U$ and $\alpha \in Fml$, a sequent $\Gamma_{sk} \Longrightarrow \Delta_{sk}$, a core atom ϑ'_{sk}, and a (skolem) function sk. The procedure creates the pair (u', α) where $u' \in U$ has the form (let $z \in V$)*

$$\texttt{for}\ z;\ \neg((\Gamma_{sk} \backslash \{\vartheta_{sk}\}) \Longrightarrow (\Delta_{sk} \backslash \{\vartheta_{sk}\}))[z \backslash sk];\ u[z \backslash sk]$$

The substitution $[x \backslash sk]$ deskolemizes all formulas and terms in order to quantify functions and predicates at those argument positions as they were quantified in the original quantified formula (see (8) vs. (10)). The guard $\neg((\Gamma_{sk} \backslash \{\psi_{sk}\}) \Longrightarrow (\Delta_{sk} \backslash \{\psi_{sk}\}))$ restricts the application of the update in order create small models as explained in the following.

For example, assume we want to construct an update that evaluates the formula $\forall x.(x > 4 \rightarrow (f(x) + 3 < g(x) - x))$ to true. Algorithm 1 invokes Algorithm 2 with the following sequent ψ':

$$sk > 4 \Longrightarrow f(sk) + 3 < g(sk) - sk$$

Let $f(sk)+3 < g(sk)-sk$ be the core atom ϑ_{sk} chosen in Line 4, then according to the previous examples procedures `solve` and `concretize` produce the intermediate result (13) that serves as input to the procedure `toQuanUpd`. We obtain the guard

$$\neg((\{sk > 4\} \backslash \{\vartheta_{sk}\}) \Longrightarrow (\{\psi_{sk}\} \backslash \{\psi_{sk}\}))[z \backslash sk]$$

which simplifies to $\neg(z > 4 \Longrightarrow)$ and then to $z > 4$. The final result of the procedure `toQuanUpd` for this example is the (u, α)-pair:

$$((\texttt{for}\ z;\ z > 4;\ f(z) := (g(z) - z - 3) + sk_2), sk_2 < 0)$$

3.3 From Updates to a Test Preamble

A test preamble is part of a test harness and its goal is to initialize the program under test with a desired program state. Here we assume that the test preamble can directly write values to all relevant memory locations. For this purpose, e.g., the KeY tool uses Java's reflection API.

Assume the input formula for Algorithm 1 is a test data constraint ϕ_{in}. If the algorithm terminates with the answer "sat", then it also provides a ground model M, i.e. assignment of values to non-quantified terms, and a sequence S of update and axiom pairs $(u_m, a_m), \ldots, (u_0, a_0)$. By the construction of the algorithm the axioms are already satisfied by M. The conversion of M into a test preamble is a well-known technique and is not discussed here.

The choice of using updates to represent models of quantified formulas is also very convenient for test preamble generation. The reason is that updates can be viewed as a small imperative programming language with some special constructs. An algorithm that converts updates to a test preamble simply has to follow the semantics of updates (Def. 2). Conversion of function updates to assignments and conditional updates to `if`-statements is trivial. Parallel updates were not created by our algorithm, they were used only to define the semantics of quantified updates. Quantified updates are converted into loops, e.g. (6) is converted to (3). If a quantified update quantifies over integers, then the integer bounds have to be determined. If the update quantifies over objects, then our solution is iterate over all objects that were created by the preamble. This solution is, however, only an approximation as it does not initialize objects that are created later on during the execution of the program under test.

Classes with invariants	T	A	B
`Account`	4	4	4
`AccountMan_src`	5	5	2
`Currency_src`	2	2	2
`SavingRule`	4	2	2
`SpendingRule`	4	2	2
`Transfers_src`	3	3	2
Total	22	18	14

Methods with Specifications	T	A	B
`AccountMan_src::IsValid()`	6	5	2
`AccountMan_src::B0delete()`	6	5	2
`AccountMan_src::isValidBank()`	5	4	2
`AccountMan_src::isValAcc()`	5	5	2
`AccountMan_src::getRef()`	5	3	2
Total	27	22	10

Fig. 2. Evaluation of the model generation algorithm applied to a banking software with JML specifications; T: total number of quantified formulas in one conjunction that occurred as test data constraints; A, B: maximum number of quantified formulas solved in a conjunction; A: our model generation algorithm; B: SMT solvers

4 Evaluation

We have implemented our algorithm, i.e. the combination of Algorithms 1 and 2, as well as a converter from updates to a test preamble, in an experimental version of the KeY tool. The technique is currently realized as an interactive model generator. The implementation proposes candidate updates to be selected by the user. The reason for this is two-fold. On the one hand, the interaction with the algorithm enables us to study heuristics for the model generation as well as to identify and understand limitations of the algorithm. On the other hand, it is the paradigm of the KeY tool to combine automation and interaction. In more general test generation contexts a full automation of the model generator is of course expected, and possible.

In order to test the algorithm we have used several examples from different sources. In the beginning, we have used hand-crafted formulas in order to test and develop the algorithm. A crucial improvement was achieved by the generation of formula (8) that allows skolemization of the quantified formulas as described in Section 3.1. Earlier approaches to generate models without formula (8) were not successful.

To test our algorithm on more realistic tests, we used a small banking software that was the subject in a case study on JML-based software validation [8]. The banking software contains JML specifications with quantified formulas. When applying KeY's test generation techniques [10, 13, 14], the quantified formulas are encountered as test data constraints. Our goal was to test for how many of these formulas our algorithm can generate models. Figure 2 shows the results.

The left table of Figure 2 shows numbers of quantified formulas that stem from class invariants of the respective classes and the right table shows numbers of quantified formulas that stem from method preconditions and loop invariants. Note that additional quantified formulas are generated by the KeY tool as shown in the motivating example in Figure 1. The column T shows the total number of quantified formulas that occurred in test data constraints in a *conjunction*, i.e. a complete model must satisfy the whole conjunction. Columns A and B

show the maximum number of quantified formulas that we found solvable in one conjunction which required us to test different combinations of the quantified formulas. Column A shows the results of our algorithm and column B shows respectively the best result achieved by any of the SMT solvers Z3, CVC3, and Yices. The evaluation shows that our algorithm can solve quantified formulas that state-of-the-art SMT solvers cannot solve. Furthermore, our algorithm was able to generate models for almost all of the quantified formulas when it was applied to the quantified formulas in separation, i.e. not in a conjunction. This simplification did not make an improvement on the SMT side.

5 Conclusions and Future Work

In formal test generation techniques quantified formulas occur in test data constraints. We have therefore developed a model generation algorithm for quantified formulas. The algorithm is not guaranteed to find a solution but on the other hand it is not restricted to a particular class of formulas. In this way, the algorithm can solve formulas that are otherwise not solvable by satisfiability modulo theories (SMT) solvers alone which is confirmed by our experiments. The algorithms can be used as a precomputation step for SMT solvers. In this case the algorithm generates only a partial model that satisfies only the quantified formulas and returns a residue of ground formulas to be solved.

Models generated by the algorithm are represented as updates. Quantified updates are suitable to represent models of quantified formulas. Our algorithm systematically analyses quantified formulas and uses heuristics to generate candidate updates. The KeY system implements a powerful update simplification calculus that allows us to check if a quantified formulas evaluates to true in a model represented by an update. Furthermore, updates are a convenient model representation language for test generation, because they have program semantics and can be converted into a test preamble.

The current implementation of the algorithm queries the user to select candidate updates from a list. We are developing heuristics in order to automate the search. The kind of formulas that is solvable by our general approach depends on the model representation language. Quantified updates are, e.g., not expressive enough to represent models of recursive functions. Our future research plans are therefore to extend the expressiveness of updates and to extend the simplification calculus.

References

1. C. Barrett and C. Tinelli. CVC3. In *CAV*, pages 298–302, 2007.
2. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.
3. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.
4. C. Csallner and Y. Smaragdakis. Check 'n' Crash: combining static checking and testing. In *ICSE*, pages 422–431. ACM, 2005.

5. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
6. X. Deng, Robby, and J. Hatcliff. Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.
7. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
8. L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. Case study in JML-based software validation. In *Proceedings, Automated Software Engineering*, pages 294–297. IEEE Computer Society, 2004.
9. B. Dutertre and L. de Moura. The YICES SMT solver. Technical report, Computer Science Laboratory, SRI International, 2006. `http://yices.csl.sri.com/tool-paper.pdf` Visited July 2010.
10. C. Engel, C. Gladisch, V. Klebanov, and P. Rümmer. Integrating verification and testing of object-oriented software. In *TAP 2008*, volume 4966 of *LNCS*, pages 182–191. Springer, 2008.
11. Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *CAV*, pages 306–320, 2009.
12. S. Ghilardi. Quantifier elimination and provers integration. *Electr. Notes Theor. Comput. Sci.*, 86(1), 2003.
13. C. Gladisch. Verification-based test case generation for full feasible branch coverage. In *SEFM*, pages 159–168, 2008.
14. C. Gladisch. Could we have chosen a better loop invariant or method contract? In *TAP 2009*, volume 5668 of *LNCS*, pages 74–89. Springer, 2009.
15. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, London, England, 2000.
16. KeY project homepage. `http://www.key-project.org/`.
17. J. R. Kiniry, A. E. Morkan, and B. Denby. Soundness and completeness warnings in ESC/Java2. In *Proc. Fifth Int. Workshop Specification and Verification of Component-Based Systems*, pages pp. 19–24, 2006.
18. G. Leavens and Y. Cheon. Design by contract with JML, 2006. `http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf`. Visited May 2010.
19. P. McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.
20. M. Moskal. *Satisfiability Modulo Software*. PhD thesis, University of Wrocław, 2009.
21. M. Moskal, J. Lopuszanski, and J. R. Kiniry. E-matching for fun and profit. *Electr. Notes Theor. Comput. Sci.*, 198(2):19–35, 2008.
22. R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Challenges in satisfiability modulo theories. In *RTA*, pages 2–18, 2007.
23. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *LPAR*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.
24. W. Visser, C. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, pages 97–107. ACM, 2004.
25. J. Zhang and H. Zhang. Extending finite model searching with congruence closure computation. In *AISC*, volume 3249 of *LNCS*, pages 94–102. Springer, 2004.