

Model Generation for Quantified Formulas with Application to Test Data Generation

Christoph D. Gladisch*

Karlsruhe Institute of Technology (KIT)
Faculty of Informatics
Germany

The date of receipt and acceptance will be inserted by the editor

Abstract We present a new model generation approach and technique for solving first-order logic (FOL) formulas with quantifiers in unbounded domains. Model generation is important, e.g., for test data generation based on test data constraints and for counterexample generation in formal verification. In such scenarios, quantified FOL formulas have to be solved stemming, e.g., from formal specifications.

Satisfiability modulo theories (SMT) solvers are considered as the state-of-the-art techniques for generating models of FOL formulas. Handling of quantified formulas in the combination of theories is, however, sometimes a problem. Our approach addresses this problem and can solve formulas that were not solvable before using SMT solvers.

We present the model generation algorithm and show how to convert a representation of a model into a test preamble for state initialization with test data. A prototype of this algorithm is implemented in the formal verification and test generation tool KeY.

1 Introduction

In the last decade, testing has been influenced 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. On the other hand, several formal verification tools have been extended to generate tests based on unproved verification conditions [10,6,38,16]. The resulting 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. The problem of solving these constraints is the problem of model generation.

A model in this context is an interpretation or assignment of values to function symbols (representing memory locations) which satisfies a given FOL formula. In software testing, models represent assignments of test data to memory locations. In verification, models are used as counterexamples of unproved verification conditions which are also used for deriving tests. Hence, in the following we refer to test data that satisfies a test data constraint or to counterexamples of verification conditions as models.

Satisfiability modulo theory (SMT) solvers are considered as the state-of-the-art techniques for generating models of FOL formulas. A bottleneck is, however, the handling of quantifiers (see, e.g., [9,33,17,34]). SMT solvers can often create models for large quantified formulas but they can fail on small and simple examples. Test data constraints and verification conditions usually include quantified formulas belonging to the combinations of multiple theories. Such formulas can lead to problems that are not in the decidable fragments of the solvers. In such cases an SMT solver returns the result *unknown*, which means that the solver cannot determine if the formula is satisfiable or not.

For example, Figure 1 shows a JAVA class with a field declaration and a JML [30] specification of a class invariant. From the field declaration and the class invariant the tool KeY [3,28] generates the formulas (1) and (2), respectively. These formulas are part of verification conditions and 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

* e-mail: gladisch@ira.uka.de

```

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

```

$$\left(\begin{array}{l} \forall o : C.o \neq \text{null} \rightarrow \\ \forall i : \text{int}.0 \leq i \leq \text{length}(s(o)) \rightarrow \\ \quad \text{acc}_{\square}(s(o), i) \neq \text{null} \end{array} \right) \quad (1)$$

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

Figure 1. (top) A field declaration and a class invariant; (bottom) Quantified formulas occurring in verification conditions and test data constraints generated by KeY

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 [8], CVC3 [1], Yices [13], 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, test data constraints, or verification conditions. Those techniques are cited in Section 2. Instead, our contribution is handling of quantified formulas for solving provided test data constraints. We propose a model generation technique that is not explicitly restricted to a specific class of formulas. Consequently, the technique is not a decision procedure, i.e., it may not terminate. However, it can solve more general formulas than SMT solvers can solve in cases where it terminates.

The proposed technique is also capable of generating partial interpretations that satisfy only the quantified formulas, and return a residue of ground formulas that is to be shown satisfiable. In this mode the technique acts as a pre-computation step for SMT solvers to eliminate quantifiers. Quantifier elimination in this sense is sound for showing satisfiability but not for refutational or validity proofs. However, for the handling of quantifiers in refutational and validity proofs powerful instantiation based techniques already exist, e.g. [33, 11, 18, 21, 2]. These can be combined with the proposed technique in order to create semi-decision procedures.

While model generation is not a new idea, the novelties of our approach are (1) the choice of language to represent

(partial) interpretations, (2) the technique for the construction of models, and (3) the means to evaluate (quantified) formulas under these interpretations. Since satisfiability solving and model generation for ground formulas is well studied [8, 1, 13], we concentrate only on the handling of quantified formulas.

This paper is based on the paper [25] and extends it with contributions from the paper [24]. The first paper describes the model generation algorithm, applies it to test data generation, and provides an evaluation and comparison of the algorithm to SMT solvers. The second paper presents the theory of the approach with a soundness proof. In this paper we additionally describe a technique for test preamble generation from a syntactic representation of a model.

Overview of the Paper. The following section describes the background and related work. Section 3 explains the basic idea of our approach and can be understood without special knowledge. Section 4 introduces the formalism that we use in the rest of the paper. Sections 5 to 8 alternate between theoretical description and implementation of our model generator. The model generator is divided into two parts. The main algorithm (Sec. 5 – 6) simplifies a formula into smaller pieces and assembles partial solutions into a solution for the original formula. Section 5 describes the rationale for this approach and provides a soundness proof for an abstract representation of the algorithm. The latter is described in Section 6. Sections 7 to 8 describe how the partial solutions needed by the main algorithm are obtained. Section 7 describes our heuristics and Section 8 describes the implementation. Section 9 describes the transformation of a syntactical representation of a model into a test preamble. The algorithm is evaluated in Section 10. We conclude in Section 11 with future research directions.

2 Background and Related Work

2.1 Verification-based Test Generation

The work presented in this paper has been developed in the KeY project [28]. The KeY system [3, 28] is a verification and test generation tool for JAVA. The tool can automatically generate JUnit tests from proof structures [16]. Figure 2 shows the components of the test generator. The test data is generated from FOL constraints which combine execution path conditions with annotations such as method specification, class invariants, and loop invariants [22, 23]. Hence, the approach is a gray-box testing technique where information about the program is contained in a proof tree. Test data constraints are extracted from the proof tree and a model generator is activated to solve them. A test-preamble generator then converts the model into a test preamble which initializes the program state accordingly. The

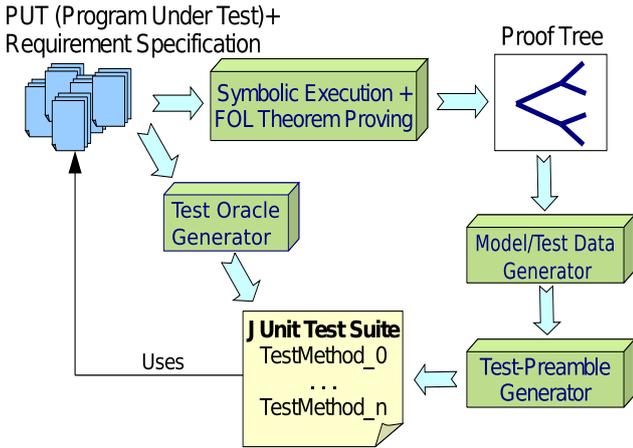


Figure 2. Verification-based test generation (VBT) in KeY

generated test oracle evaluates the postcondition that the program must ensure after running a test.

So far we have used the theorem prover Simplify [11] to generate test data but the so-generated test data is not always guaranteed to satisfy the constraints as will be explained below. Our approach was to instantiate quantified formulas and to solve the resulting ground formulas. This approach is, however, unsound and results in false positive tests¹ which mistakenly satisfy the precondition. On the other hand, we found that more recent SMT solvers such as Z3 [8], CVC3 [1], Yices [13] are not capable to solve the constraints either, which was the motivation for this work.

Several other tools exist that follow similar ideas to that of the KeY tool to generate test data constraints and have therefore similar requirements on test data generation. KUnit [10] 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 [6]. Java PathFinder is an explicit-state model checker and features the generation of test inputs [38]. 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 [31]. These techniques are powerful for traditional testing approaches but they do not handle test data generation for constraints with quantified FOL formulas.

2.2 Handling of Quantified Formulas

The main contribution of our work is 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 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 [33] which was first used in the theorem prover Simplify [11]. 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 $\varphi(t_0) \wedge \dots \wedge \varphi(t_n)$ where $t_0 \dots t_n$ are terms computed via E-matching. For this reason, Simplify may produce unsound answers (see also [29]) as shown in the following example.

$$\forall h.\forall i.\forall v.select(store(h,i,v),i) = v \quad (3)$$

$$\forall h.\forall j.0 \leq select(h,j) \wedge select(h,j) \leq 2^{32} - 1 \quad (4)$$

Formula (3) is an axiom of the theory of arrays and (4) 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 [32]. Formula (4) 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 (3) \wedge (4) is unsatisfiable, which can be easily seen when considering the following instantiation $[h := store(h_0, k, 2^{32}), j := k]$, (see [32]). Simplify, however, produces a counterexample for $\neg((3) \wedge (4))$, which means that it satisfies the *false* formula (3) \wedge (4). 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 [18].

An important fragment of FOL for program specification which allows a complete instantiation is the Array Property Fragment [5]. E-matching is used in state-of-the-art SMT solvers such as Z3 [8], CVC3 [1], Yices [14, 13]. Formulas (1) or (2) which is solvable with our technique is, however, neither in the Bernays-Schönfinkel class nor in the Array Property Fragment.

Saturation-based theorem provers such as SPASS [39] can decide the satisfiability of different fragments of first-order logic. The satisfiability of quantified first-order logic formulas with equality (without other theories) can be decided using superposition calculi [35]. Superposition is an instance of resolution with ordered rewriting rules for equational formulas. The ordering relation allows limiting the number of clauses that are created during resolution to a finite saturated set. If the empty clause, i.e. *false*, is not derived, then the set of clauses, and hence the input formula, is satisfiable.

Early SMT solvers were not capable of showing the satisfiability of quantified formulas. In order to overcome this limitation some SMT solvers, e.g. Z3, combine the ground procedures with a saturation engine [7].

¹ A positive test is regarded here as a test that detects an error.

Such combinations can generate in many cases models for large quantified formulas. However, as described in the introduction even these techniques sometimes cannot show the satisfiability of formulas that can be shown with our approach.

Satisfiability of a formula can also be shown by weakening the formula with existential quantifiers and then showing its validity, instead of satisfiability. This idea is followed in [37] for proving the existence of a state that reveals a software fault. For instance, let $\phi(t)$ be a formula with an occurrence of the term t , then the approach generates the formula $\exists x.\{t := x\}\phi(t)$. This transformation is repeated also for other terms of ϕ . The approach uses then free variables for computing instantiations of the existentially quantified variables.

Quantified constraint satisfaction problem (QCSP) solvers, e.g., [19], primarily regard the finite version of the satisfiability problem, whereas our approach handles infinite domains. Some of the work, e.g. [4], also considers continuous domains, however, these techniques do not handle uninterpreted function symbols other than constants. The finite domain version of the satisfiability problem in first-order logic is handled by finite model finding methods such as [40].

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., [20] 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 approach is, however, that it is not restricted to a certain class of formulas. Hence, our approach can solve formulas that other approaches cannot solve.

3 The Basic Idea of our Approach

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.

The approach that we suggest is to use programs to represent partial models and to use formal verification

in order to evaluate the quantified formulas to true. Our approach is to regard a given quantified formula φ as a postcondition and to generate for φ a program p such that the final states of p satisfy φ . Thus, one of our contributions is a program generation technique.

For instance, in order to satisfy formula (2) we could generate the following JAVA statement

```
for (C o : Cs){o.s = new String [10] ;} (5)
```

where Cs is an infinite collection of objects of class C , and verify it against formula (2). If the verification calculus is capable to prove that after the execution of (5) the formula (2) is satisfied, then the formula evaluates to true and consequently the quantified formula is eliminated. Furthermore, the program represents a partial model for the quantified formula. The formula (2) is a test data constraint and (5) is the test preamble that satisfies it. Hence, the satisfiability problem is replaced by a program generation and verification problem.

A typical programming language such as JAVA is, however, not *directly* suitable for this task. A syntactical problem is that function and predicate symbols are usually not part of such languages. Regarding automation the problem is that for the verification of programs with loops, loop invariants have to be generated. Finally, a semantic problem is that an infinite loop does not terminate so that a final state satisfying the target formula is never reached.²

We found that a language and a calculus that are suitable for our purpose are KeY’s updates and the update simplification calculus built into KeY. Quantified updates can represent models for many quantified formulas. This is because they can assign values to infinitely many locations at the same time rather than iteratively and therefore they terminate. However, (quantified) updates are less expressive than a Turing-complete while-language and therefore the updates simplification calculus of KeY can fully automatically reduce FOL formulas with updates to pure FOL formulas. Loop invariants do not have to be generated for the verification step. Another important advantage of updates is that updates are syntactically and semantically closer to first-order logic than general programming languages. The latter aspect simplifies the generation of updates from formulas.

4 KeY’s Dynamic Logic with Updates

The KeY system is a theorem prover and a verification tool for JAVA. It is based on the logic JAVA CARD DL, which is an instance of Dynamic Logic (DL) [27]. 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* [36].

² We address the usage of a Turing-complete language for model representation again at the end of Section 11.

*Notation.*³ We use the following abbreviations for syntactic entities: **VSym** is the set of (logic) variables; **FSym** is the set of function symbols; **FSym_r** \subset **FSym** is the set of rigid function symbols, i.e. functions with a fixed interpretation such as, e.g., '0', 'succ', '+'; **FSym_{nr}** \subset **FSym** is the set of non-rigid function symbols, i.e. uninterpreted functions; **PSym** is the set of predicate symbols; **Σ** is the signature consisting of **FSym** \cup **PSym**; **Trm_{FOL}** is the set of FOL terms; **Trm** \supset **Trm_{FOL}** is the set of DL terms; **Fml_{FOL}** is the set of FOL formulas; **Fml** \supset **Fml_{FOL}** is the set of DL formulas; **U** is the set of updates; \doteq is the equality predicate; and $=$ is syntactic equivalence. The following abbreviations describe semantic sets: **D** is the FOL domain or universe; **S** is the set of states or equivalently the set of FOL interpretations. To describe semantic properties we use the following abbreviations: $\mathit{val}_s(t) \in \mathcal{D}$ is the valuation of $t \in \mathit{Trm}$ and $\mathit{val}_s(u) \in \mathcal{S}$ is the valuation of $u \in U$ in $s \in \mathcal{S}$; $s \models \varphi$ means that φ is true in state $s \in \mathcal{S}$; $\models \varphi$ means that φ is valid, i.e. for all $s \in \mathcal{S}$, $s \models \varphi$; and \equiv is semantic equivalence.

4.1 Skolemizable and Non-Skolemizable Quantified Formulas in Sequents

KeY implements a sequent calculus. A sequent $\Gamma \Rightarrow \Delta$ is equivalent to the formula $(\gamma_1 \wedge \dots \wedge \gamma_n) \rightarrow (\delta_1 \vee \dots \vee \delta_m)$, where $\{\gamma_1, \dots, \gamma_n\} = \Gamma$ is called the antecedent and $\{\delta_1, \dots, \delta_m\} = \Delta$ is called the succedent.

In sequent calculus the goal is to prove the validity of a sequent, which is in contrast to a tableaux-based calculus where unsatisfiability is proven. This has implications on the semantics of quantified formulas in a sequent as we will explain in the following.

Formulas can be moved between the antecedent and succedent by negating them, i.e., $(\gamma \Rightarrow \delta) \equiv (\neg\delta \Rightarrow \neg\gamma)$. This explains the following lemma.

Lemma 1.

$$(\Gamma \Rightarrow \forall x.\phi, \Delta) \equiv (\Gamma, \exists x.\neg\phi \Rightarrow \Delta) \quad (6)$$

$$(\Gamma \Rightarrow \exists x.\phi, \Delta) \equiv (\Gamma, \forall x.\neg\phi \Rightarrow \Delta) \quad (7)$$

Skolemization of the quantified formulas in (6) is allowed because it is a global equivalence transformation as explained in the following. Let $S2$ be a sequent derived from a sequent $S1$ of the form (6) using Skolemization; for instance let $S1 = (\forall x.P(x) \Rightarrow \forall y.P(y))$ and $S2 = (\forall x.P(x) \Rightarrow P(sk))$. Global equivalence between the two sequents means that $\models S1$ iff $\models S2$. Skolemization of the formulas in (6) is allowed for removing the quantifiers because if $S2$ is valid, then $S1$ is valid; and if $S2$ has a counterexample (countermodel), then $S1$ has a counterexample as well.

³ Bold font is used to improve readability and is not used to give symbols different meanings.

The problematic quantified formulas are those in (7) because Skolemization of these quantified formulas significantly changes their semantics. For instance, the sequent $\forall x.P(x) \Rightarrow P(1)$ is valid but validity is not preserved when the quantified formula is Skolemized, i.e. $P(sk) \Rightarrow P(1)$ is not valid. We call such formulas *non-Skolemizable*. Non-Skolemizable formulas can be instantiated with arbitrary terms, e.g. $P(t), \forall x.P(x) \Rightarrow P(1)$. This is what the instantiations techniques described in Section 2 do for validity proofs but it does not allow removal of quantified formulas which is required for satisfiability proofs.

4.2 Updates

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 **FSym_{nr}** (such as uninterpreted functions). Hence, updates represent partial states and can be used to represent (partial) models of formulas. The set **FSym_r** represents rigid functions whose interpretation is fixed and cannot be changed by an update.

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

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

- Updates. The set U of updates consists of: function updates $(f(t_1, \dots, t_n) := t)$, where $f(t_1, \dots, t_n)$ is called the *location term* and t is the *value term*; parallel updates $(u_1 \parallel u_2)$; and quantified updates (**for** x ; φ ; u).
- Terms. The set of Dynamic Logic terms includes all FOL terms, i.e. $\mathit{Trm} \supset \mathit{Trm}_{FOL}$; $\{u\}t \in \mathit{Trm}$ for all $u \in U$ and $t \in \mathit{Trm}$; and **if** ϕ **then** t_1 **else** $t_2 \in \mathit{Trm}$ with $\phi \in \mathit{Fml}$ and $t_1, t_2 \in \mathit{Trm}$.
- Formulas. The set of Dynamic Logic formulas includes all FOL formulas, i.e. $\mathit{Fml} \supset \mathit{Fml}_{FOL}$; $\{u\}\varphi \in \mathit{Fml}$ for all $u \in U$, $\varphi \in \mathit{Fml}$; and sequents $\Gamma \Rightarrow \Delta$, where $\Gamma \subset \mathit{Fml}$ is called antecedent and $\Delta \subset \mathit{Fml}$ is called succedent.

Sequents are normally, e.g. in [3], not included in the set of formulas. However, in this work it is convenient to include them into the set of formulas as *syntactic sugar*.

For the definition of the semantics of updates we need the notion of semantic updates.

Definition 2. Let $f \in \text{FSym}_{nr}$ be of arity $n \geq 0$ and let $d_1, \dots, d_n, d \in \mathcal{D}$.

- A *semantic update* is a triple $(f, (d_1, \dots, d_n), d)$.
- A set CU of semantic updates is called *consistent* if for all $(f, (d_1, \dots, d_n), d), (f', (d'_1, \dots, d'_m), d') \in CU$, $d=d'$ if $f = f'$, $n = m$, and $d_i = d'_i (1 \leq i \leq n)$.
- CU denotes the set of consistent semantic updates.
- For any set $CU \in CU$ of consistent semantic updates, the *modification* $CU(s)$ is defined as the interpretation s' such that (note that states and interpretations are the same concept):

$$s'(f)(d_1, \dots, d_n) = \begin{cases} d, & \text{if } (f, (d_1, \dots, d_n), d) \in CU \\ s(f)(d_1, \dots, d_n), & \text{otherwise} \end{cases}$$

Definition 3. Semantics. We use the notation from the description in this section and from Definitions 1 and 2, further let $s, s' \in \mathcal{S}$; $v, v_1, v_2 \in \mathcal{D}$; $x, x_i, x_j \in V$; $\beta : V \rightarrow \mathcal{D}$ be a variable assignment; $val_{s,\beta} : Trm \rightarrow \mathcal{D}$; $val'_{s,\beta} : U \rightarrow \mathcal{S}$; and $val'_{s,\beta} : U \rightarrow CU$.

Terms and Formulas

- $val_{s,\beta}(\{u\}t) = val'_{s,\beta}(t)$, where $s' = val_{s,\beta}(u)$
- $val_{s,\beta}(\{u\}\varphi) = val'_{s',\beta}(\varphi)$, where $s' = val_{s,\beta}(u)$

Updates

- $val_{s,\beta}(u) = val'_{s,\beta}(u)(s)$ (modification, see Def. 2)
- $val'_{s,\beta}(f(t_1, \dots, t_n) := t) = \{(f, (d_1, \dots, d_n), d)\}$ with $d_i = val'_{s,\beta}(t_i)$ for $(1 \leq i \leq n)$ and $d = val'_{s,\beta}(t)$.
- $val'_{s,\beta}(u_1 ; u_2) = ((U_1 \cup U_2) \setminus C)$ where $U_1 = val'_{s,\beta}(u_1)$ and $U_2 = val'_{s',\beta}(u_2)$ with $s' = val_{s,\beta}(U_1)$. (The set C is defined at the bottom of this definition.)
- $val'_{s,\beta}(u_1 \parallel u_2) = ((U_1 \cup U_2) \setminus C)$ where $U_1 = val'_{s,\beta}(u_1)$ and $U_2 = val'_{s,\beta}(u_2)$
- $val'_{s,\beta}(\text{for } x; \phi; u) = \{(f, (d_1, \dots, d_n), d) \mid$
there is $a \in \mathcal{D}$ such that
 $((f, (d_1, \dots, d_n), d), a) \in dom$ and
 $b \not\leq a$ for all $((f, (d_1, \dots, d_n), d'), b) \in dom\}$

with $dom = \{(val'_{s,\beta}(u), a) \mid a \in \{d \in \mathcal{D} \mid s, \beta^d \models \phi\}\}$, and \leq is a well-ordering.

The set C is defined as

$$C = \{(f, (d_1, \dots, d_n), d) \mid (f, (d_1, \dots, d_n), d) \in U_1 \text{ and } (f, (d_1, \dots, d_n), d') \in U_2 \text{ for some } d' \neq d\}.$$

□

Remark. The variable assignment β is only needed for the definition of quantified updates and will be omitted in the following.

In order to help the reader understanding the semantics of updates we give the following examples.

- The sequential update $a := 2; f(3) := a$ evaluates to the set of semantic updates $\{(a, (), 2)\} \cup \{(f, (3), val_{\{(a,(),2)\}(s)}(a))\} \setminus \emptyset$ which can be simplified to $\{(a, (), 2), (f, (3), 2)\}$. Hence, the result of $val_s(\{a := 2; f(3) := a\}f(3))$ is 2.

- If two different values are assigned to the same location term, then the set C comes into play. The update $a := 2; a := 3$ evaluates to the set of semantic updates $\{(a, (), 2)\} \cup \{(a, (), 3)\} \setminus C$, where $C = \{(a, (), 2)\}$. Hence, the latter update *wins* and $val_s(\{a := 2; a := 3\}a)$ evaluates to 3.
- The subupdates of a parallel update are applied at the same time rather than sequentially. For instance, $(\{a := b \mid b := a\}b) \equiv a$ but $(\{a := b; b := a\}b) \equiv b$
- Intuitively, a quantified update (**for** $x; \varphi(x); u(x)$) is equivalent to the infinite composition of parallel updates (parallel updates are associative):

$$\dots \parallel (\text{if } \varphi(x_i); u(x_i)) \parallel (\text{if } \varphi(x_j); u(x_j)) \parallel \dots$$

satisfying a global order \leq on the domain \mathcal{D} such that $\beta(x_j) \leq \beta(x_i)$, where $(\text{if } \varphi_x; u)$ represents (informally) an update u with the condition φ .

Hence, the evaluation of the quantified update

$$\text{for } x; 0 \leq x \wedge x \leq 1; f(x) := x$$

yields the semantic update $\{(f, (0), 0), (f, (1), 1)\}$ and **for** $x; 0 \leq x \wedge x \leq 1; a := x$ yields the semantic update $\{(a, (), 0)\}$.

Table 1 shows update simplification rules which are on the one hand a subset and on the other hand instances of more general – and thus more complicated – rules presented in [36,3]. The simplified calculus in Table 1 is, e.g., not powerful enough to simplify non-Skolemizable quantified formulas with updates. To further simplify the calculus we have omitted rules for parallel updates. Parallel updates are not required for completeness but rather to increase performance significantly.

In the following, some examples are shown of how updates, terms, and formulas are simplified in KeY.

- The term $\{f(a) := a\}f(f(f(a)))$ simplifies to a using the rules R6 and R5.
- The sequent $\Rightarrow \{f(b) := a\}P(f(c))$ simplifies to the two sequents

$$b \doteq c \Rightarrow P(a) \text{ and } \neg b \doteq c \Rightarrow P(f(c))$$

by using the rules R4, R6, R8, and R5.

- Consider the sequent:

$$\Rightarrow \{\text{for } x; 0 \leq x; f(x) := x\} \forall z. (0 \leq z \rightarrow f(z) \doteq z)$$

Skolemization of the quantified formula yields:

$$\Rightarrow \{\text{for } x; 0 \leq x; f(x) := x\} (0 \leq sk \rightarrow f(sk) \doteq sk)$$

Rules R2, R4, and R5 yield:

$$\Rightarrow (0 \leq sk \rightarrow \underbrace{\{\text{for } x; 0 \leq x; f(x) := x\}}_u f(sk)) \doteq sk$$

Using rule R7 with the substitution $[x \setminus sk]$ the term $\{u\}f(sk)$ simplifies to

$$\text{if } sk \doteq sk \wedge 0 \leq sk \text{ then } sk \text{ else } f(\{u\}sk).$$

R1:	$\{u_2\}\{u_1\}\chi \rightsquigarrow \{u_2; u_1\}\chi$	where χ is a term or formula.
R2:	$\{u\}(\phi \circ \psi) \rightsquigarrow (\{u\}\phi) \circ (\{u\}\psi)$	where $\phi, \psi \in Fml$ and “ \circ ” is a logic operator. (Similarly for “ \neg ”)
R3:	$\{u\}(p \circ q) \rightsquigarrow (\{u\}p) \circ (\{u\}q)$	where $p, q \in PSym$.
R4:	$\{u\}p(t_1, \dots, t_n) \rightsquigarrow p(\{u\}t_1, \dots, \{u\}t_n)$	where $p \in PSym$.
R5:	$\underbrace{\{f(t_1, \dots, t_n) := t\}}_U g(b_1, \dots, b_m) \rightsquigarrow g(Ub_1, \dots, Ub_m)$	if $f \neq g$ or $n \neq m$. (Similarly for quantified updates)
R6:	$\underbrace{\{f(t_1, \dots, t_n) := t\}}_U f(b_1, \dots, b_n) \rightsquigarrow \text{if } t_1 \doteq Ub_1 \wedge \dots \wedge t_n \doteq Ub_n \text{ then } t \text{ else } f(Ub_1, \dots, Ub_n)$	
R7:	$\underbrace{\{\text{for } x; \phi; f(t_1, \dots, x, \dots, t_n) := t\}}_U f(b_1, \dots, b, \dots, b_n) \rightsquigarrow \text{if } (t_1 \doteq Ub_1 \wedge \dots \wedge t_n \doteq Ub_n \wedge \phi)[x \setminus b] \text{ then } t[x \setminus b] \text{ else } f(Ub_1, \dots, Ub, \dots, Ub_n)$	
R8:		
	$\frac{\Gamma, U_2\phi \Rightarrow U_2\psi[t_1], \Delta \quad \Gamma, U_2\neg\phi \Rightarrow U_2\psi[t_2], \Delta}{\Gamma \Rightarrow U_2\psi[\text{if } \phi \text{ then } t_1 \text{ else } t_2], \Delta} \quad \frac{\Gamma, U_2\phi, U_2\psi[t_1] \Rightarrow \Delta \quad \Gamma, U_2\neg\phi, U_2\psi[t_2] \Rightarrow \Delta}{\Gamma, U_2\psi[\text{if } \phi \text{ then } t_1 \text{ else } t_2] \Rightarrow \Delta} \quad (*)$	
R9:		
	$\frac{\Gamma, U_2((\phi \wedge t \doteq U_1b)[x \setminus sk]) \Rightarrow U_2\psi[s[x \setminus sk]], \Delta \quad \Gamma, U_2(\forall x.(\phi \rightarrow t \neq U_1b)) \Rightarrow U_2(\psi[f(U_1b)]), \Delta}{\Gamma \Rightarrow U_2\psi[\underbrace{\{\text{for } x; \phi; f(t) := s\}}_{U_1} f(b)], \Delta} \quad (*)$	

(*) $\psi \in Fml_{FOL}$ is a ground formula, $\psi[X]$ represents an occurrence of X in ψ , the term t is injective with respect to x , and the update U_2 may be an arbitrary update.

Table 1. A subset of KeY’s update simplification rules

Rules R5 and R8 yield the two sequents

$$\begin{aligned} sk \doteq sk \wedge 0 \leq sk \Rightarrow 0 \leq sk \rightarrow sk \doteq sk \\ \neg(sk \doteq sk \wedge 0 \leq sk) \Rightarrow 0 \leq sk \rightarrow f(sk) \doteq sk \end{aligned}$$

which can be both simplified to *true*.

5 Model Generation by Iterative Update Construction

5.1 The Goal and the Challenges

In order to show the satisfiability of a formula ϕ_{in} , our approach is to generate an update u , such that $\models \{u\}\phi_{in}$. If such an update exists, then ϕ_{in} is satisfiable and the update represents a set of models of ϕ_{in} .

The main contribution of this paper is a technique for generating (partial) models for quantified formulas. As this work was developed in the context of KeY, we regard the model generation problem of a quantified formula $\forall x.\phi(x)$ in a sequent $\varphi = (\Gamma, \forall x.\phi(x) \Rightarrow \Delta)$. The formula $\phi(x) \in Fml_{FOL}$ denotes a formula with an occurrence of the variable $x \in VSym$ and $\Gamma, \Delta \subset Fml_{FOL}$. Such sequents occur frequently as open branches of failed proof attempts. The reason for proof failure is often unclear and it is desirable to determine whether φ has a counterexample and to generate a test whose test data satisfies the test data constraint $\neg\varphi$. The goal is therefore given by the following problem description.

Problem 1. Given a sequent $(\Gamma, \forall x.\phi(x) \Rightarrow \Delta)$ the goal is to generate an update u such that:

$$(\{u\}(\Gamma, \forall x.\phi(x) \Rightarrow \Delta)) \equiv (\Gamma', \text{true} \Rightarrow \Delta') \quad (8)$$

where Γ' and Δ' are obtained by applying $\{u\}$ on the formulas of the sets Γ and Δ , respectively.

If this problem is solved by a technique for given formulas, then this technique can be applied iteratively to all quantified formulas occurring in Γ and Δ resulting in a sequent $\Gamma'' \Rightarrow \Delta''$ that consists only of ground formulas. Note that this problem is undecidable in general because otherwise the satisfiability problem of first-order logic formulas would be decidable. A technique for solving this problem can also be used to build models for ground formulas but we concentrate mainly on the harder problem – the removal of quantified formulas from a sequent. Note that non-Skolemizable quantified formulas occurring in the succedent Δ are those with existential quantifiers and they can be *moved* to the antecedent Γ using Lemma 1.

We have implemented different algorithms that follow this approach. Unfortunately, only in rare cases the Problem 1 was solved by early algorithms we have developed. Based on experiments with these algorithms we have identified two practical problems which are the rationale for the design of our technique. The problems are stated in form of the following informal proposition.

Proposition 1. *a) In general cases of $\forall x.\phi(x)$, it is necessary to somehow analyze the semantic properties about the matrix $\phi(x)$ and to construct the update u*

based on this information in order to satisfy $\models \{u\}\forall x.\phi(x)$.

- b) The KeY theorem prover is often not sufficiently powerful to automatically simplify $(\Gamma', \{u\}\forall x.\phi(x) \Rightarrow \Delta')$ to $(\Gamma', \text{true} \Rightarrow \Delta')$ if $\models \{u\}\forall x.\phi(x)$ and u is a quantified update.

Using arbitrary updates u in order to satisfy the goal $\{u\}\forall x.\phi(x) \equiv \text{true}$ is too inefficient for automation. Hence, the problem (a) of Proposition 1 requires some systematic method that analyzes the semantic properties of the matrix $\phi(x)$. If $\phi(x)$ has a non-trivial syntactic or semantic structure, then a method is required that breaks this structure into smaller pieces. For instance, in the formula $\forall x.(f(x) > x \wedge g(x) < f(x))$ the function g depends on the function f , and the function f depends on the variable x . Such information has to be extracted from $\phi(x)$ and then be used for the construction of u .

Some possibilities to analyze the semantic properties of $\phi(x)$ are to create instances of $\phi(x)$ or to use free variables (see, e.g., [21]). We have experimented with the latter approach and could solve problem (a) in some cases.

Problem (b) of Proposition 1 states that KeY's simplification calculus often cannot simplify the left-hand side to the right-hand side of Equation (8). The reason is that the rules R8 and R9 of Table 1 are applicable only on ground formulas. As explained in Section 4.1 a universally quantified formula in the antecedent can be instantiated but it cannot be removed using Skolemization. In contrast, an existentially quantified formula in the antecedent or a universally quantified formula in the succedent can be Skolemized so that the rules R8 and R9 are applicable.

In some cases quantified updates may introduce quantified formulas in Γ' and Δ' of Equation (8). The quantified formula is introduced by the right branch of rule R9 shown in Table 1. In the left branch of the rule the case is considered where the quantified update is effective on a term, e.g. $\{\text{for } x; x \geq 0; f(x) := t_x\}f(0)$. The right branch considers the case where no value for x can be found that allows the application of the update on a function, e.g. $\{\text{for } x; x > 1; f(x) := t_x\}f(0)$. At least one of these branches always closes, i.e., the sequent is reduced to true. If the branch with the quantified formula remains open our approach is to backtrack and to try a different update. If rule R7 of Table 1 is applicable, then it should be used instead of rule R9 because it can not introduce a quantified formula.

5.2 The Solution

We have implemented an algorithm that solves both problems of Proposition 1 and is capable to solve Problem 1 in many cases. The algorithm is described in Section 6. In this section we provide a theorem that formalizes the crucial problem simplification technique of the

algorithm. The simplification technique is the core of the algorithm and we therefore prove the soundness of this simplification.

The idea of the algorithm is to generate an update u , such that $\{u\}\forall x.\phi(x)$ evaluates to true, and in this way to eliminate the quantified formula. The weakest condition under which $\forall x.\phi(x)$ evaluates to true in φ can be expressed as

$$\underbrace{(\Gamma, \forall x.\phi(x) \Rightarrow \Delta)}_{\varphi} \leftrightarrow \underbrace{(\Gamma, \text{true} \Rightarrow \Delta)}_{\varphi'} \quad (9)$$

which simplifies by equivalence transformations to

$$\underbrace{\Gamma \Rightarrow \forall x.\phi(x), \Delta}_{\psi} \quad (10)$$

Any model of (10) is also a model of (9). This means that in states which satisfy (10) the quantified formula $\forall x.\phi(x)$ can be replaced by true. This corresponds to the statement (i) for $m = 0$ of Theorem 1 (see below). The algorithm then iteratively extends φ, φ' , and ψ with updates u_1, \dots, u_m until (10) and thus (9) are satisfied. This iterative extension results in the formulas φ_m, φ'_m , and ψ_m for $m \geq 0$ in Definition 4.

For the construction of the updates it is sometimes necessary to introduce and axiomatize fresh function symbols. For instance, it may be desired to introduce a fresh function $\text{notZero} \in \text{FSym}_{nr}$ with the axiom $\neg(\text{notZero} \doteq 0)$. With this axiom it is, e.g., possible to write an update $a := b + \text{notZero}$, with $a, b \in \text{Trm}_{\text{FOL}}$, expressing a general assignment to a with a value different from b . Each update u_i is therefore associated with an axiom α_i . Note that several axioms can be combined to one axiom by using a conjunction.

Definition 4. Given a sequent $\varphi = (\Gamma, \forall x.\phi(x) \Rightarrow \Delta)$, where $\Gamma, \Delta \subset \text{Fml}_{\text{FOL}}$ and $\phi(x) \in \text{Fml}_{\text{FOL}}$ is a formula with an occurrence of $x \in \text{VSym}$. Let $m \in \mathbb{N}$, $u_0, \dots, u_m \in U$; and $\alpha_0, \dots, \alpha_m \in \text{Fml}_{\text{FOL}}$. The formulas $\psi_m, \varphi'_m, \varphi_m \in \text{Fml}$, for $m \in \mathbb{N}$, are defined recursively as:

- $\varphi_0 = (\Gamma, \forall x.\phi(x) \Rightarrow \Delta)$
- $\varphi'_0 = (\Gamma, \text{true} \Rightarrow \Delta)$
- $\psi_0 = (\Gamma \Rightarrow \forall x.\phi(x), \Delta)$
- $\varphi_{m+1} = \alpha_m \rightarrow \overline{\{u_m\}\varphi_m}$
- $\varphi'_{m+1} = \alpha_m \rightarrow \overline{\{u_m\}\varphi'_m}$
- $\psi_{m+1} = \alpha_m \rightarrow \overline{\{u_m\}\psi_m}$

Definition 4 can be seen as an abstract search technique where the sequence of updates $u_m; \dots; u_0, m \in \mathbb{N}$, has to be found for solving the Problem 1. The updates $u_m; \dots; u_0$ constitute the update u in Problem 1 and $\varphi_0 \equiv \varphi$ is the original sequent that is to be shown falsifiable. In the following theorem we assume $\gamma = \overline{\forall x.\phi(x)}$.

Theorem 1. Let \mathcal{S} be the set of models. Let $\varphi = (\Gamma, \gamma \Rightarrow \Delta)$ and $\psi_m, \varphi'_m, \varphi_m \in \text{Fml}$ be defined according to Definition 4, then

- i.* $\models \psi_m \leftrightarrow (\varphi'_m \leftrightarrow \varphi_m)$
ii. If there is $s_m \in \mathcal{S}$ such that $s_m \models \neg\varphi_m$, then there exists $s \in \mathcal{S}$ with $s = \text{val}_{s_m}(u_m; \dots; u_1)(s_m)$ and $s \models \neg\varphi$.

The theorem is proven in Section 5.3.

The theorem describes under what condition a sequence (not sequent) of update and axiom pairs $(u_0, \alpha_0), \dots, (u_m, \alpha_m)$ evaluates a quantified formula to true; and the theorem describes how this sequence represents a partial model.

Formula $\neg\varphi$ is the formula for which a model shall be generated. Statement (ii) of Theorem 1 states that if there is a model $s_m \in \mathcal{S}$ for a formula $\neg\varphi_m$, according to Definition 4, then from s_m a model for $\neg\varphi$ can be derived by evaluation of the updates u_0, \dots, u_m . Hence, the satisfiability of $\neg\varphi_m$ can be used for showing the satisfiability of $\neg\varphi$.

For instance, let $\varphi \equiv (-a = b)$, then a suitable pair (u_0, α_0) to construct φ_1 is, e.g. $(a := b, \text{true})$. In this case φ_1 has the form $\text{true} \rightarrow \{a := b\}(-a = b)$ which can be simplified to false. Hence, any state $s_1 \in \mathcal{S}$ satisfies $s_1 \models \neg\varphi_1$ which implies that $\neg\varphi$ is satisfiable and a model $s \in \mathcal{S}$ for $\neg\varphi$ is $s = \text{val}_{s_1}(a := b)(s_1)$. Note, that choosing an update is a problem for which no general uniform solution exists, e.g., the pair $(b := a, \text{true})$ or the pair $(a := 1 \parallel b := 1, \text{false})$ are also suitable candidates. We provide heuristics for finding such updates in Section 7.

An important property of the statement (ii) for the construction of an update search procedure is that soundness of the statement is preserved by any pair (u, α) . I.e., in principle random updates could be *tried-out* by a search procedure. For instance, consider the pair $(a := 1 \parallel b := 2, \text{true})$ or the pair $(a := b, \text{false})$ where neither of them represents a model of φ , with $\varphi \equiv (-a = b)$. In both cases φ_1 evaluates to true. Hence, there is no $s_1 \in \mathcal{S}$ such that $s_1 \models \neg\varphi_1$ and therefore statement (ii) makes no implication regarding the satisfiability of $\neg\varphi$.

Based on statement (i) an algorithm can be constructed for the generation of models for ground formulas. The challenge is, however, to generate a model that satisfies a quantified formula that cannot be Skolemized. If ψ_m is valid then the model generation problem for $\neg\varphi_m$ can be replaced by the model generation problem for $\neg\varphi'_m$ because φ_m and φ'_m are equivalent. Considering Definition 4, the statement (i) is interesting because in φ'_m the quantified formula is eliminated, i.e., it is replaced by true. Together with statement (ii), $\neg\varphi'_m$ can be used to generate a model for $\neg\varphi$.

The problem is to check if $\varphi'_m \equiv \varphi_m$, which is a generalization of Problem 1. Theorem 1 states that the problem $\varphi'_m \equiv \varphi_m$ can be solved by a validity proof of ψ_m . This allows solving the problems described in Proposition 1 because the quantified formula in ψ_m occurs negated with respect to φ_m and can therefore be Skolemized. For instance, let $\varphi_m = (\forall x.\phi(x) \Rightarrow)$, then $\psi_m = (\Rightarrow \forall x.\phi(x))$. In contrast to φ_m , the latter can be sim-

plified to $(\Rightarrow \phi(sk))$, where $sk \in \text{FSym}_r$ is the Skolem function. When ψ_m is Skolemized, then it is (a) easy to analyze the semantics of $\phi(sk)$ and (b) the propositional structure of $\phi(sk)$ can be *flattened* to the sequent level which is necessary to simplify quantified updates. In this way both problems described in Proposition 1 are solved. For instance, using classical sequent calculus rules it is not possible to simplify $(\{u\}\forall x.(A \rightarrow (A \vee B)) \Rightarrow)$ to $(\text{true} \Rightarrow)$ but $(\Rightarrow \{u\}\forall x.(A \rightarrow (A \vee B)))$ can be simplified to $(\Rightarrow \text{true})$, for any update u . More examples showing these advantages are provided in the following sections such as Example 4 on page 12.

The approach can be generalized for the generation of models for ground formulas by using the more general Definition 5 instead of Definition 4 in Theorem 1.

Definition 5. Given a sequent $\varphi = (\Gamma, \gamma \Rightarrow \Delta)$, where $\gamma \in \text{Fml}_{\text{FOL}}$ and $\Gamma, \Delta \subset \text{Fml}_{\text{FOL}}$. Let $m \in \mathbb{N}$, $u_0, \dots, u_m \in U$; and $\alpha_0, \dots, \alpha_m \in \text{Fml}_{\text{FOL}}$. The formulas $\psi_m, \varphi'_m, \varphi_m \in \text{Fml}$, for $m \in \mathbb{N}$, are defined recursively as:

- $\varphi_0 = (\Gamma, \gamma \Rightarrow \Delta)$
- $\varphi'_0 = (\Gamma, \text{true} \Rightarrow \Delta)$
- $\psi_0 = (\Gamma \Rightarrow \gamma, \Delta)$
- $\varphi_{m+1} = \alpha_m \rightarrow \{u_m\}\varphi_m$
- $\varphi'_{m+1} = \alpha_m \rightarrow \{u_m\}\varphi'_m$
- $\psi_{m+1} = \alpha_m \rightarrow \{u_m\}\psi_m$

5.3 Soundness Proof of Theorem 1

Lemma 2. Let $\varphi \in \text{Fml}$, if $\models \varphi$, then for any update $u \in U$, $\models \{u\}\varphi$ holds.

Proof of Lemma 2. Since for any $s \in \mathcal{S}$, $s \models \varphi$ holds, it is also the case for $s' = \text{val}_s(u)$ that $s' \models \varphi$ because $s' \in \mathcal{S}$. ■

Lemma 3. Let $A, B, C \in \text{Fml}$. The following formula is valid

$$(A \rightarrow (B \leftrightarrow C)) \leftrightarrow ((A \rightarrow B) \leftrightarrow (A \rightarrow C)) \quad (11)$$

Proof of Lemma 3. Assume $A \equiv \text{true}$, then Formula (11) simplifies to $(B \leftrightarrow C) \leftrightarrow ((B) \leftrightarrow (C))$ which is valid. Now assume that $A \equiv \text{false}$, then Formula (11) simplifies trivially to true. ■

Proof of Theorem 1 The proof of Theorem 1 is based on induction on m .

Induction Base ($m = 0$) (i) Validity of

$$\underbrace{(\Gamma \Rightarrow \forall x.\phi(x), \Delta)}_{\psi_0} \leftrightarrow \underbrace{(\Gamma, \text{true} \Rightarrow \Delta)}_{\varphi'_0} \leftrightarrow \underbrace{(\Gamma, \forall x.\phi(x) \Rightarrow \Delta)}_{\varphi_0} \quad (12)$$

can be shown by using propositional transformation rules. Using the abbreviations $A = \forall x.\phi(x)$ and $B = \neg\Gamma \vee \Delta$ we obtain

$$\underbrace{(B \vee A)}_{\psi_0} \leftrightarrow \underbrace{((\text{true} \rightarrow B) \leftrightarrow (A \rightarrow B))}_{\varphi'_0} \quad (13)$$

If $A \equiv \text{true}$ we obtain $B \leftrightarrow B$ which is true. Otherwise if $A \equiv \text{false}$ we obtain $B \leftrightarrow (B \leftrightarrow \text{true})$ which is also true. Thus, Formula (12) is valid.

(ii) Since $\varphi_0 = \varphi$ and $s = s_0$ statement (ii) is trivially true.

Induction Step ($m \rightarrow m+1$) (i) Assuming $\models \psi_m \leftrightarrow (\varphi'_m \leftrightarrow \varphi_m)$, we want to show that $\models \psi_{m+1} \leftrightarrow (\varphi'_{m+1} \leftrightarrow \varphi_{m+1})$. If the statement

$$\models \psi_m \leftrightarrow (\varphi'_m \leftrightarrow \varphi_m) \quad (14)$$

holds, then according to Lemma 2 the following statement holds for any $u_m \in U$.

$$\models \{u_m\}(\psi_m \leftrightarrow (\varphi'_m \leftrightarrow \varphi_m)) \quad (15)$$

Using rule R2 of Table 1 we obtain

$$\models \{u_m\}\psi_m \leftrightarrow (\{u_m\}\varphi'_m \leftrightarrow \{u_m\}\varphi_m) \quad (16)$$

For any $\alpha \in Fml_{FOL}$, statement (16) implies

$$\models \alpha \rightarrow (\{u_m\}\psi_m \leftrightarrow (\{u_m\}\varphi'_m \leftrightarrow \{u_m\}\varphi_m)) \quad (17)$$

Using Lemma 3 the following equivalent statement is obtained.

$$\models \left(\begin{array}{c} (\alpha \rightarrow \{u_m\}\psi_m) \leftrightarrow \\ ((\alpha \rightarrow \{u_m\}\varphi'_m) \leftrightarrow (\alpha \rightarrow \{u_m\}\varphi_m)) \end{array} \right) \quad (18)$$

Statement 18 is equivalent to

$$\models \psi_{m+1} \leftrightarrow (\varphi'_{m+1} \leftrightarrow \varphi_{m+1}) .$$

(ii) Assuming that statement (ii) of the theorem holds for some $m \geq 0$ we show that it holds also for $m+1$. Assume there is $s_{m+1} \in \mathcal{S}$ such that $s_{m+1} \models \neg\varphi_{m+1}$. By propagating the negation of $\neg\varphi_{m+1}$ to the inside of the formula, loosely speaking, we obtain the equivalent formula $\hat{\varphi}_m \in Fml$ that can be recursively defined as

$$\hat{\varphi}_0 = \neg(\Gamma, \text{true} \Rightarrow \Delta) \quad \hat{\varphi}_{m+1} = (\alpha_m \wedge \{u_m\}\hat{\varphi}_m)$$

Hence, $s_{m+1} \models \neg\varphi_{m+1}$ is equivalent to $s_{m+1} \models \hat{\varphi}_{m+1}$ which is equivalent to $s_{m+1} \models (\alpha_m \wedge \{u_m\}\hat{\varphi}_m)$. The latter implies that $s_{m+1} \models (\alpha_m)$ and $s_{m+1} \models \{u_m\}\hat{\varphi}_m$. There is $s_m \in \mathcal{S}$ with $s_m = \text{val}_{s_{m+1}}(u_m)$ such that $s_m \models \hat{\varphi}_m$. Since $\hat{\varphi}_m$ is equivalent to $\neg\varphi_m$ we have $s_m \models \neg\varphi_m$.

According to the induction hypothesis there exists $s \in \mathcal{S}$ with $s = \text{val}_{s_m}(u_m; \dots; u_1)$ such that $s \models \neg\varphi$. Because of $s_m = \text{val}_{s_{m+1}}(u_m)$, we conclude that if $s_{m+1} \models \neg\varphi_{m+1}$, then there exists $s \in \mathcal{S}$ with $s = \text{val}_{s_{m+1}}(u_{m+1}; u_m; \dots; u_1)$ such that $s \models \neg\varphi$. ■

6 The Model Search Algorithm

Preliminaries. In the following two algorithms are described. Algorithm 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 8. 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” respectively “ \emptyset ”.

The technique requires a theorem prover for FOL and an implementation of updates.

Definition 6. Procedure Th. The procedure **Th** represents a theorem prover.

- Given a formula $\vartheta \in Fml$ as input, **Th**(ϑ) returns a set $\Theta \subset Fml_{FOL}$.
- If $\Theta = \emptyset$, then $\models \vartheta$.
- Each $\vartheta' \in \Theta$ is a sequent containing at most literals and quantified formulas.
- For all $\vartheta' \in \Theta$ must hold $\models (\neg\vartheta') \rightarrow (\neg\vartheta)$.

The set Θ represents open branches or open proof obligations. These are formulas that the theorem prover was not able to prove. The third statement requires that the propositional structure of formulas has been simplified which is usually one of the reasons why Θ is a set. The last condition requires that a counterexample of an open proof obligation is also a counterexample of ϑ .

Description of Algorithm 1. Assume we want to generate a model satisfying the input formula ϕ_{in} . The Algorithm 1 reformulates this problem as counterexample generation for $\neg\phi_{in}$ which is represented by φ' (Line 1). In Line 4 the algorithm attempts to show $\models \varphi'$. If φ' is valid, then $\Phi = \emptyset$ and the algorithm stops because φ' has no counterexample 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). 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 counterexamples.

Example 1. Let ϕ_{in} be defined as

$$\phi_{in} = f(1) \doteq a \wedge \forall x.(f(x) > x \wedge g(x) < f(x))$$

then $\varphi' = (f(1) \doteq a \wedge \forall x.(f(x) > x \wedge g(x) < f(x)) \Rightarrow)$. The theorem prover cannot simplify the sequent any further. In Line (4) we get:

$$\Phi = \{f(1) \doteq a \wedge \forall x.(f(x) > x \wedge g(x) < f(x)) \Rightarrow\}$$

Algorithm 1 modelSearch(ϕ_{in})

```

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

```

□

In Line 5 the algorithm selects a formula $\varphi \in \Phi$. The goal is to create a counterexample for φ , i.e. to satisfy $\neg\varphi$. The last condition of Definition 6 ensures that satisfaction of $\neg\varphi$ implies satisfaction of 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 which is the case in Example 1. After normalization at Line 13 the antecedent of φ contains at least one universally quantified formula. This normalization can be easily achieved using Lemma 1. A counterexample 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.

Example 2. Following Example 1, formula φ is defined as

$$\varphi = \underbrace{(f(1) \doteq a, \forall x. (f(x) > x \wedge g(x) < f(x)))}_{\Gamma} \Rightarrow \underbrace{(f(x) > x \wedge g(x) < f(x))}_{\phi(x)} \quad (19)$$

Generating a model for φ in one step is complicated because the quantified formula cannot be Skolemized (see Section 4.1).

□

Lines 14 to 26 implement the iterative construction of formulas according to Definition 4. In Line 15 the formula φ' is constructed where the quantified formula is replaced by true. Recall from Section 5.2 that

$$\underbrace{(\Gamma, \forall x.\phi(x) \Rightarrow \Delta)}_{\varphi} \leftrightarrow \underbrace{(\Gamma, \text{true} \Rightarrow \Delta)}_{\varphi'} \quad (9)$$

is equivalent to

$$\underbrace{\Gamma \Rightarrow \forall x.\phi(x), \Delta}_{\psi} \quad (10)$$

which is generalized in statement (i) of Theorem 1:

$$\models \psi_m \leftrightarrow (\varphi'_m \leftrightarrow \varphi_m)$$

Substituting φ by φ' in subsequent computation is sound only if (9) or equivalently (10) is valid. Therefore Formula (10) is assigned to ψ in Line 16 and is checked by Th in Line 17. If ψ can be proved, then the algorithm continues in Line 4 where φ' (now without the quantified formula) is used instead of φ . If the proof of (10) does not close (Line 17), then the result is a set of proof obligations Ψ .

In (10) the quantified formula occurs negated wrt. (9). As described in Section 5.2 an important consequence of this negation is that in Lines 17 and 25 the theorem prover can Skolemize the quantified formula (10) resulting in

$$\Gamma \Rightarrow \phi(sk), \Delta \quad (20)$$

where $sk \in \text{FSym}_r$ is a fresh symbol. In this way the formula $\phi(sk)$ can be simplified by the calculus.

Example 3. In our example ψ is defined as

$$\psi = (f(1) \doteq a \Rightarrow \forall x. (f(x) > x \wedge g(x) < f(x)))$$

The quantified formula is now negated wrt. φ in (19) (because $(F \Rightarrow) \equiv (\Rightarrow \neg F)$) and can therefore be Skolemized (see Section 4.1). $\text{Th}(\psi)$ yields

$$\Psi = \{(f(1) \doteq a \Rightarrow f(sk) > sk), \\ (f(1) \doteq a \Rightarrow g(sk) < f(sk))\}$$

Note that the structure of each $\psi' \in \Psi$ is simpler than the structure of 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 (9). 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 . Thus the algorithm

refines the class of models in each iteration. Note that Ψ may not decrease to \emptyset . In this case the algorithm does not terminate which means that it has no complexity bounds.

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

$$\alpha \rightarrow \{u\}\psi \quad (21)$$

evaluates to true. If we consider $\psi_m = \psi$, then Formula 21 (see Line 24) corresponds to ψ_{m+1} in Definition 4. The procedure `formulaToUpdate` which is described in Section 8 generates candidate pairs (u, a) that are likely to satisfy (21).⁴

Example 4. Using the formulas in the set Ψ from Example 3, the procedure `formulaToUpdate` can generate, e.g., the following updates with axioms to satisfy the formulas respectively:

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

□

Checking Formula (21) as described above is important because it is equivalent to

$$(\alpha \rightarrow \{u\}\varphi) \leftrightarrow (\alpha \rightarrow \{u\}\varphi') \quad (22)$$

which in turn is a weakening of (9). Accordingly, in Line 23 the formula φ' is updated. If (21) is valid, which is checked in Line 25, then the inner loop terminates and the outer loop continues execution. Hence, the original counterexample generation problem for φ is replaced by the counterexample generation problem for $\alpha \rightarrow \{u\}\varphi'$ where the quantified formula is eliminated, i.e. replaced by true. This is sound because if (21) is valid, then (22) is valid and therefore a counterexample for $\alpha \rightarrow \{u\}\varphi'$ is a counterexample for $\alpha \rightarrow \{u\}\varphi$. The latter implies that φ has a counterexample as well which is formalized by statement (ii) of Theorem 1.

Example 5. Continuing with Example 4, after two iterations of the inner loop the formula ψ is equivalent to:

$$(\mathbf{true} \rightarrow \{\mathbf{for} \ x; \mathbf{true}; g(x) := f(x) - 1\} \\ (\mathbf{true} \rightarrow \{\mathbf{for} \ x; \mathbf{true}; f(x) := x + 1\} \\ (f(1) \doteq a \implies \forall x. (f(x) > x \wedge g(x) < f(x))))))$$

This formula is valid and can be proved using the rules in Table 1 and first-order theorem proving rules with arithmetic. Hence, Ψ has decreased to \emptyset and the inner loop terminates. At this point formula φ' is defined as:

$$(\mathbf{true} \rightarrow \{\mathbf{for} \ x; \mathbf{true}; g(x) := f(x) - 1\} \\ (\mathbf{true} \rightarrow \{\mathbf{for} \ x; \mathbf{true}; f(x) := x + 1\}(f(1) \doteq a \implies)))$$

⁴ Note that since the procedure `formulaToUpdate` uses only one formula $\psi' \in \Psi$ to construct the pair (u, α) , Formula (21) may not evaluate to true. In this case the inner loop continues iteration and unsolved formulas $\psi' \in \Psi$ reappear in the next iteration to be solved.

and is simplified to $1 + 1 \doteq a$ in Line 4. Thus, the quantified formula is eliminated and it remains to show the satisfiability of $1 + 1 \doteq a$ using the ground procedure in Line 7. □

7 Heuristics for Update Construction from Formulas

While Section 5 describes a general sound framework for model generation, in this section we study how to generate a partial model for one selected formula γ in a sequent $\varphi = (\Gamma, \gamma \implies \Delta)$. In particular we give an intuition of how quantified updates can be constructed in order to satisfy quantified formulas. The described heuristics can then be used to iteratively extend the partial model for all formulas in the sequent. Important to note is that soundness of Theorem 1 is preserved by *any* sequence of update and axiom pairs. Hence, unsoundness cannot be introduced by any of the heuristics. However, in order to increase the probability of termination of Algorithm 1 we choose the most general update and axiom pairs that satisfy the following definition.

Definition 7. Update Construction. Let $\gamma \in Fml_{FOL}$ be the currently selected formula for which a partial model is to be created and which is a sub-formula in a sequent $\varphi = (\Gamma, \gamma \implies \Delta)$. Let $\psi = (\Gamma \implies \gamma, \Delta)$ and $\varphi' = (\Gamma \implies \Delta)$.

The goal of update construction from the formula γ is to create a pair (u, α) , with $u \in U$ and $\alpha \in Fml$, such that

- $\models \alpha \rightarrow \{u\}\psi$, and
- there is some $s \in \mathcal{S}$ with $s \models \neg(\alpha \rightarrow \{u\}\varphi')$

The first condition ensures that $\models (\alpha \rightarrow \{u\}\varphi) \leftrightarrow (\alpha \rightarrow \{u\}\varphi')$ which corresponds to statement (i) of Theorem 1. The second condition satisfies, in combination with the first condition, the assumption in the statement (ii) of the theorem. The second condition ensures that, e.g. the trivial pair $(\{\}, \mathbf{false})$ is not used to satisfy the first condition. In this case $\models \mathbf{false} \rightarrow \{\}\psi$ but there is no $s \in \mathcal{S}$ satisfying $s \models \neg(\mathbf{false} \rightarrow \{\}\varphi')$.

The sequent ψ is equivalent to ψ_0 and φ' is equivalent to φ'_0 , according to Definition 5. In the model search algorithm each time a pair (u_m, α_m) is constructed, new formulas φ'_{m+1} , φ'_{m+1} , and ψ_{m+1} are generated according to Definition 5. These formulas must be simplified by `Th` to φ , ψ and φ' , respectively, such that a new formula $\gamma \in Fml_{FOL}$ can be selected for update construction. We assume that φ is an open branch of `Th`, i.e. $\varphi \in \mathbf{Th}$, and it is therefore simplified according to Definition 6. Hence, γ is either a literal or a quantified formula. In the following subsections, case distinctions are made on the structure of γ .

7.1 Update Construction from Ground Formulas

In the following we ignore the context formulas Γ and Δ , i.e., we assume that $\Gamma = \Delta = \emptyset$. Since $\varphi' \equiv (\emptyset \Rightarrow \emptyset) \equiv \text{false}$ the second condition of Definition 7 is satisfied if $\alpha \neq \text{false}$.

Handling of Equalities Assume $t_1, t_2 \in \text{Term}_{\text{FOL}}$ are location terms (see Def. 1). If γ is of the form

$$t_1 = l \text{ or } l = t_1$$

where l is a literal, then the pair $(t_1 := l, \text{true})$ should be created because it satisfies the first condition of Definition 7 as $\models \text{true} \rightarrow \{t_1 := l\}(t_1 \doteq l \wedge l \doteq t_1)$. If γ is of the form

$$t_1 = t_2$$

then a choice has to be made between the pairs $(t_1 := t_2, \text{true})$ and $(t_2 := t_1, \text{true})$. In both cases the first condition of Definition 7 is satisfied as $\models \text{true} \rightarrow \{t_1 := t_2\}(t_1 \doteq t_2)$ and $\models \text{true} \rightarrow \{t_2 := t_1\}(t_1 \doteq t_2)$. The particular choice can have influence on the rest of the partial model construction. For instance, let $\psi = (F[b] \Rightarrow a \doteq b)$, where $F[b]$ is a formula with an occurrence of b . If we chose the update $a := b$, then in subsequent steps a model has to be created for the formula $F[b]$. Otherwise, if we chose the update $b := a$, then in subsequent steps a model has to be created for the formula $F[a]$.

Equality between terms can in some cases also be established, if the terms share the same top-level function symbol and have location terms as arguments. For instance, let $f(t_1), f(t_2) \in \text{Term}_{\text{FOL}}$ and $f \in \text{FSym}_{nr}$, then $\models \alpha \rightarrow \{u\}(f(t_1) = f(t_2))$ can be satisfied by the pair $(t_1 := t_2, \text{true})$ or by $(t_2 := t_1, \text{true})$.

Handling of Arithmetic Expressions Let $t_1, t_2 \in \text{Term}_{\text{FOL}}$ be arithmetic expressions composed of rigid and non-rigid function symbols. Several solutions exist to satisfy $\models \alpha \rightarrow \{u\}(t_1 \doteq t_2)$. Consider for instance the polynomial equation

$$2 * a + b * c = d - e$$

where $a, b, c, d, e \in \text{Term}_{\text{FOL}}$ are location terms. There are five most general updates evaluating this equation to true. These can be obtained by solving the polynomial equation for one of the location terms at a time. Our implementation enumerates those solutions during update search. An example for one of the solutions is

$$(a := (d - e - b * c) / 2, \text{true})$$

Note that a, b, c, d, e are not restricted to constants, i.e., terms consisting of a nullary function.

Handling of Inequalities Let $t_1, t_2 \in \text{Term}_{\text{FOL}}$ where t_1 is a location term. An inequation

$$\neg t_1 \doteq t_2$$

can be satisfied, e.g., by the pair $(t_1 := t_2 + 1, \text{true})$. A more general update is, however, $t_1 := t_2 + \text{notZero}$, where $\text{notZero} \in \text{FSym}_{nr}$ is a fresh-symbol representing a value different from 0. This is where the axiom part of a pair comes into play. A more general solution for the formula $t_1 \neq t_2$ is the pair

$$(t_1 := t_2 + \text{notZero}, \neg(\text{notZero} = 0))$$

This pair satisfies both conditions of Definition 7. We allow the constant notZero to be non-rigid, so that during model generation its value can be further concretized.

Inequations of the form

$$t_1 < t_2$$

can be handled by introducing a fresh symbol $gtZero \in \text{FSym}_{nr}$ with the axiom $gtZero > 0$.

7.2 Update Construction from Quantified Formulas

Our approach for creating models of quantified formulas is to generate quantified updates. For example, the quantified formula

$$\forall x. x > a \rightarrow f(x) = g(x) + x \quad (23)$$

is satisfiable in any state after execution of the quantified update

$$\text{for } x; x > a; f(x) := g(x) + x \quad (24)$$

i.e., $\models \{(24)\}(23)$. Notice the similar syntactical structure between (23) and (24). Another solution is

$$\text{for } x; x > a; g(x) := f(x) - x \quad (25)$$

for which $\models \{(25)\}(23)$ holds. It is easy to see that a translation can be generalized for other *simple* quantified formulas. Furthermore, the heuristics and case distinctions described in Section 7.1 can be reused to handle different arithmetic expressions and relations. For instance the formula

$$\forall x. f(x) \geq x \rightarrow (g(x) < f(x))$$

evaluates to true after execution of any of the following updates (with axioms)

$$\begin{aligned} & (\text{for } x; f(x) \geq x; g(x) := f(x) + gtZro, gtZro > 0) \\ & (\text{for } x; \neg(g(x) < f(x)); f(x) := x - gtZro, gtZro > 0) \end{aligned}$$

The update simplification calculus may in some cases introduce new quantified formulas. In such cases our approach has to be applied either recursively on the new quantified formulas or the heuristic has to choose different updates in a search procedure to prevent the introduction of new quantified formulas.

Algorithm 2 formulaToUpdate(ψ')

```

1: let  $sk$  = the Skolem function of  $\psi'$ 
2: let  $(\Gamma \Rightarrow \Delta) = \psi'$ 
3: let  $(\Gamma_{sk} \Rightarrow \Delta_{sk}) \subset (\Gamma \Rightarrow \Delta)$  (see description)
4: choose  $\vartheta_{sk} \in (\neg\Gamma_{sk} \cup \Delta_{sk})$  (elements in  $\Gamma$  are negated)
5: choose  $\vartheta'_{sk} \in \text{solve}(\vartheta_{sk})$ 
6: choose  $(u, \alpha) \in \text{concretize}(\vartheta'_{sk})$ 
7: choose  $(u', \alpha) \in$ 
    toQuanUpd( $sk, (u, \alpha), (\Gamma_{sk} \Rightarrow \Delta_{sk}), \vartheta_{sk}$ )
8: choose  $(u'', \alpha') \in \text{injectiveSubTerms}((u', \alpha))$ 
9: return  $(u'', \alpha)$ 

```

8 Update Generation for Satisfying Quantified Formulas

In this section we describe Algorithm 2 which is based on the ideas described in Section 7. The algorithm is used by Algorithm 1 to construct updates for satisfying quantified formulas. According to Section 6 this algorithm receives as input a sequent ψ' that is an open proof obligation of $\text{Th}(\psi)$. Algorithm 2 is queried for each open proof obligation and is expected to generate an update with an axiom (u, α) that is *likely* to satisfy the conditions in Definition 7.

As each pair (u, α) satisfies one of the open proof obligations $\psi' \in \text{Th}(\psi)$, a series of such pairs *eventually* satisfies Formula (21) (see page 12). Algorithm 2 returns a set of alternative solutions for each sequent ψ' . Recall 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 Definition 6 the sequent ψ' has been simplified such that all formulas on the sequent level are either quantified formulas or literals. We are interested in literals that were derived from $\forall x.\phi(x)$. Therefore our heuristic is to categorize as highly relevant for the construction of the update u those literals in ψ' that have an occurrence of the Skolem symbol sk , that was introduced in (20). Let ψ'_{sk} be defined as

$$\Gamma_{sk} \Rightarrow \Delta_{sk}$$

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 4, $(\Gamma_{sk} \Rightarrow \Delta_{sk})$ is either $(\Rightarrow f(sk) > sk)$ or $(\Rightarrow g(sk) < f(sk))$.

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 literal, whose interpretation we want to manipulate, as the *core literal* and it corresponds to the formula γ in Definition 7. Let ϑ_{sk} denote the chosen core literal (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).

Generation of Updates from Core Literals

Definition 8. The procedure **solve** is such that given a literal ϑ_{sk} , whose top-level predicate symbol is a relation $R \in \text{PSym}$, it constructs a set of literals Θ and for each literal $\vartheta'_{sk} \in \Theta$:

- $\vartheta'_{sk} = (\neg)R'(f(t_1, \dots, t_n), v)$, i.e. syntax
- $\vartheta'_{sk} \equiv \vartheta_{sk}$, i.e. semantics

where (\neg) is optional negation, $R' \in \text{PSym}_r$, $f \in \text{FSym}_{nr}$, $f \neq sk$, and $t_1, \dots, t_n, v \in \text{Trm}_{\text{FOL}}$.

The procedure **solve** creates normal forms from core atoms. For example, for the formula $\vartheta_{sk} = (f(sk) + 3 < g(sk) - sk)$, with “ $<$ ” $\in \text{PSym}$, $f, g \in \text{FSym}_{nr}$, the procedure **solve** may generate, e.g., the following set:

$$\{f(sk) < (g(sk) - sk - 3), g(sk) > (f(sk) + 3 + sk)\} \quad (26)$$

In the first core literal of this set $R' = “<”$ and in the second core literal $R' = “>”$. 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 literals may not be solvable by the procedure but the more results the procedure **solve** produces the better is the chance of generating a suitable update.

Definition 9. The procedure **concretize** is defined such that given a literal of the form $(\neg)R(f(t_1, \dots, t_n), v)$, with $R \in \text{PSym}_r$, $f \in \text{FSym}_{nr}$, and $t_1, \dots, t_n, v \in \text{Trm}_{\text{FOL}}$, it creates a set of pairs (u, α) , with $u \in U$, $\alpha \in \text{Fml}_{\text{FOL}}$, such that:

- $u = (f(t_1, \dots, t_n) := \text{value})$, where $\text{value} \in \text{Trm}_{\text{FOL}}$
- $\alpha \rightarrow \{u\}(\neg)R(f(t_1, \dots, t_n), v)$ 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 literal is of the form $t_1 \doteq t_2$, using infix notation, then the result of the procedure **concretize** is simply $((t_1 := t_2), \text{true})$. 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 literal $\vartheta'_{sk} = (f(sk) < (g(sk) - sk - 3))$ from the solution set of the previous example, the procedure **concretize** may produce, e.g., the following solution:

$$\underbrace{\{(f(sk) := (g(sk) - sk - 3) + sk_2), sk_2 < 0\}}_{u, \alpha} \quad (27)$$

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

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

Any term $(2 * c - sk - 3) + Z$, where Z is a negative integer literal, is also an admissible solution of the procedure. However, the introduction of the fresh constant sk_2 with the axiom $sk_2 < 0$ is a more general solution than using Z . If the top-level symbol of the core atom is “ \doteq ” and it occurs in the antecedent Γ , i.e., effectively it means “ \neq ”, then a fresh symbol $sk_3 \in \text{FSym}_r$ may be introduced with the axiom $\neg(sk_3 \doteq 0)$.

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

Update Generalization by Conversion to Quantified Updates

Definition 10. The procedure `toQuanUpd` is defined such that given a tuple (u, α) , with $u \in U$ and $\alpha \in \text{Fml}_{\text{FOL}}$, a sequent $\Gamma_{sk} \Rightarrow \Delta_{sk}$, a core literal ϑ'_{sk} , and a Skolem function sk , it creates the pair (u', α) where $u' \in U$ has the form (let $z \in \text{VSym}$)

$$\text{for } z; \text{guard}; u[sk \setminus z]$$

where $\text{guard} = \neg((\Gamma_{sk} \setminus \{\vartheta_{sk}\}) \Rightarrow (\Delta_{sk} \setminus \{\vartheta_{sk}\}))[sk \setminus z]$.

The substitution $[sk \setminus z]$ 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 (10) vs. (20) on page 8). The guard $\neg((\Gamma_{sk} \setminus \{\psi_{sk}\}) \Rightarrow (\Delta_{sk} \setminus \{\psi_{sk}\}))$ restricts the quantification domain of the update to the relevant cases.

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' :

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

Let $f(sk) + 3 < g(sk) - sk$ be the core literal ϑ_{sk} chosen in Line 4, then according to the previous examples procedures `solve` and `concretize` produce the intermediate result (27) that serves as input to the procedure `toQuanUpd`. We obtain the guard

$$\neg(\{sk > 4\} \setminus \{\vartheta_{sk}\}) \Rightarrow (\{\psi_{sk}\} \setminus \{\psi_{sk}\})[sk \setminus z]$$

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

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

Updating of Sub-Terms If we create an update of the form

$$\text{for } x; \text{guard}; f(g(x)) := h(x) \quad (28)$$

that is supposed to evaluate, e.g., $\forall x.f(g(x)) = h(x)$ to true, then usually the intention is that f is assigned different values at different argument positions. This does

not happen, however, if the function g is not injective. For instance if $\forall x.g(x) = 0$, then the update (28) just assigns a value to $f(0)$. Our heuristic is to create another update that is applied before (28) and that ensures injectivity of the argument of the updated function. In this example the argument is g and the updated function is f .

Definition 11. The procedure `injectiveSubTerms` denotes a procedure such that given an update

$$u = \text{for } x; \text{guard}; f(\tau_x) := t_x$$

where $\text{guard} \in \text{Fml}_{\text{FOL}}$, $f \in \text{FSym}_{nr}$, and $\tau_x, t_x \in \text{Trm}_{\text{FOL}}$ are terms with an occurrence of $x \in \text{FSym}$, it creates a set of tuples such that every tuple (u'', α) , with $\alpha \in \text{Fml}_{\text{FOL}}$ and $u'' \in U$:

$$\models \alpha \rightarrow \{u''\}\{u\}(\forall x.(\text{guard} \rightarrow f(\tau_x) \doteq t_x))$$

For example, a possible solution of `injectiveArgs` for (28) is

$$u'' = (\text{for } z; \text{guard}[x \setminus z]; g(z) := z, \text{true})$$

A more general solution is to introduce a fresh function symbol $f_{sk} \in \text{FSym}_{nr}$, axiomatize it with $\forall x.\forall y.x \neq y \rightarrow f_{sk}(x) \neq f_{sk}(y)$, and assign it to g , i.e. $\dots g(z) := f_{sk}(z)$. The axiom introduces a new quantified formula for which another update has to be created in order to evaluate this formula to true. In each step a more specific partial model has to be created until eventually no axioms with quantifiers are introduced. Hence, this technique has to be applied recursively on the arguments of locations.

9 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 $(u_m, a_m), \dots, (u_0, a_0)$. By the construction of the algorithm, the axioms are already satisfied by M . The conversion of M into a test preamble is rather simple 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. 3). Parallel updates were not created by

our algorithm, but they are created by the full update simplification calculus of KeY. They can be converted into sequential assignments with some additional temporary variable. Quantified updates are converted into loops. 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.

In the following we define and explain the transformation function τ which transforms an update to another update or to statements of a sequential imperative programming language. We do not define transformation function strictly mathematically. This is because not all updates can be transformed into equivalent programs and there are also different possibilities how to realize the transformation.

Function and Sequential Updates Function updates and sequential updates can be transformed trivially to a sequence of assignments. However, not all updates have to be transformed to the program. If it is known that a function symbol does not represent an actual memory location of the program the update can be ignored. Hence, the transformation function τ for function updates, sequential updates, and conditions updates is defined as follows.

Definition 12. The transformation function $\tau : U \rightarrow (U \cup \text{Programs})$, where *Programs* denotes the set of sequences of JAVA statements, has the following properties.

Let $(f(t_1, \dots, t_n) := t), u_1, u_2 \in U$ be function updates.

- The transformation $\tau(f(t_1, \dots, t_n) := t)$ is defined as:
 - an empty statement, if $f(t_1, \dots, t_n)$ does not represent any of the following: a program variable, an object field, the array access operator, or a temporary symbol introduced by τ as described in this section;
 - otherwise, it is defined as the assignment

$$f(t_1, \dots, t_n) := t$$

- The transformation $\tau(u_1 ; u_2)$ yields the sequential program $\tau(u_1) ; \tau(u_2)$.

Parallel Updates The transformation of parallel updates without quantified updates has been described in [15]. The approach is to transform a parallel update into a sequential update before it is transformed into a program. The approach requires that no sequential update occurs as a sub-update of the parallel update which is the normal form of updates in KeY. When evaluating parallel updates the arguments of both, the location terms and

the value terms of all function updates that occur below the parallel update, are evaluated in the pre-state of the update (see Def. 3).

In order to transform a parallel update into a sequential update, the approach is to store the values of the location term arguments and the value terms in temporary function symbols.

Definition 13. Let $u \in U$ be given as:

$$u = (f_1(\bar{t}_1) := s_1 \parallel \dots \parallel f_n(\bar{t}_n) := s_n)$$

The transformation $\tau(u)$ is defined as

$$\tau(u_{\text{prefix}} ; u')$$

where $u_{\text{prefix}} \in U$ is defined as the sequential update

$$(\bar{t}'_1 := \bar{t}_1 ; s'_1 := s_1) ; \dots ; (\bar{t}'_n := \bar{t}_n ; s'_n := s_n)$$

where $\bar{t}'_1, \dots, \bar{t}'_n, s'_1, \dots, s'_n \in \text{FSym}_{nr}$ are function symbols representing temporary program variables and $u' \in U$ is defined as the sequential update

$$f_1(\bar{t}'_1) := s'_1 ; \dots ; f_n(\bar{t}'_n) := s'_n$$

□

In the definition the update u_{prefix} stores the values of the terms \bar{t}_i and s_i with $1 \leq i \leq n$ in the pre-state of the update u . The notation \bar{t} represents a vector or list of terms. The values are stored in the constants \bar{t}'_i and s'_i which are used in the update u' . The update u' is the sequential version of u which ensures that the arguments of the location term and the value term are not influenced by the sequence. This is because \bar{t}'_i and s'_i cannot be modified by u' .

Quantified Updates A quantified update of the form $(\text{for } x; \text{true}; u)$ can be *informally* seen as an infinite composition of parallel updates of the form $u_{+\infty} \parallel \dots \parallel u_{-\infty}$ where each sub-update has its own variable assignment for x . A quantified update of the form $(\text{for } x; \phi; u)$ allows a sub-update to be effective only if the guard ϕ is true for the particular assignment of x , e.g., the quantified update $\text{for } x; 0 \leq x \leq 2; f(x) := x$ is equivalent to the update $f(2) := 2 \parallel f(1) := 1 \parallel f(0) := 0$. Hence, it seems reasonable to transform a quantified update into a loop statement which iterates over the values of x , and to use an if-statement to check whether ϕ is satisfied before executing the sub-update. Since loops are executed sequentially a conversion from the parallel composition to a sequential composition of updates is necessary. This conversion can be achieved with the approach followed in Definition 13. Based on this intuition we extend the transformation function τ for quantified updates as follows.

Definition 14. Let $u \in U$ be given as:

$$u = (\text{for } x; \phi; f(\bar{t}) := s)$$

The transformation $\tau(u)$ is defined as

$$\tau(\text{QUpToLoop}_{\preceq}(u_{\text{prefix}}); \text{QUpToLoop}_{\preceq}(u'))$$

where $u_{\text{prefix}} \in U$ is defined as

$$\text{for } x; \phi; (\bar{t}' := \bar{t}; s' := s)$$

$u' \in U$ is defined as

$$\text{for } x; \phi; f(\bar{t}') := s'$$

and $\text{QUpToLoop}_{\preceq}$ represents a program in form of Algorithm 3. The sets dom_x^ϕ contain all values of the variables x for which ϕ is satisfiable.

Algorithm 3 $\text{QUpToLoop}_{\preceq}(\mu)$

Require: $\mu \in U$ has the form $(\text{for } x; \phi; u)$.

```

1: for all  $x \in \text{dom}_x^\phi$  respecting the order relation  $\preceq$  do
2:   if  $\phi$  then
3:      $\tau(u)$ 
4:   end if
5: end for

```

□

In Definition 14 a general transformation is given which converts a quantified update with a function sub-update to a schema for a sequential program. Similarly as in Definition 13, the update u_{prefix} evaluates the terms \bar{t} and s in the pre-state of the update u . The values are stored in the constants \bar{t}' and s' allowing to break the parallelism of u . The update u' is the sequential version of u .

Algorithm 3 is a program schema. An implementation of a loop which iterates over all elements in dom_x^ϕ is, however, not practical or not possible in some cases. The problem of the transformation is the domain of the quantification.

One problem is that if the domain is infinite, Definition 14 may yield an infinite loop or a loop that iterates, e.g., over all integer values of the programming language. A practical transformation can be achieved if the formula ϕ restricts the domain of quantification to a finite set. This is, for instance, the case if ϕ specifies upper and lower bounds in case of quantification over the integer domain.

Another problem is handling of quantification over the domain of objects in the programming language. The problem is that it is not possible to determine and access all objects, e.g., of the Java Virtual Machine. In cases where a sufficient reduction of a quantification domain is not possible, KeY approximates the semantics of updates by performing instantiations of the quantified updates with primitive values or object references that occur in the test data constraint. Recall from Section 2 that using quantifier instantiation in first place would not solve our satisfiability problem as this is only an approximation of the more general first-order logic formulas (see Section 2).

10 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 but it can be fully automated. The implementation proposes a finite set of candidate updates to be selected by the user. 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. In more general test generation contexts a full automation can be achieved by automatic selection of the candidate updates from the finite set of choices and applying backtracking if formulas tend to become bigger rather than smaller.

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. For example, in Figure 3 the method `link` initializes the array `prev` such that for all indices `i` of the array `next` holds `prev[next[i]]==i`, i.e., the method creates a doubly linked list based on arrays. A verification attempt of the method with KeY fails with an open proof branch. Figure 4 shows the quantified formulas occurring in the verification condition of the proof branch. The goal was then to check if the verification condition has a counterexample and to generate test data in order to test if the program exhibits a fault. Although the verification condition is falsifiable, SMT solvers cannot generate counterexamples/models for the sequent because it contains quantified formulas with uninterpreted and arithmetic function symbols. Using our model generator we were able to generate a model which is shown in Figure 5. This example contained the biggest number of quantifiers and the duration for finding the right selection of updates was approximately 30 minutes. We believe that an automatic search and a more efficient implementation can find the same result in less than a minute. Repeating this user-interaction takes about five minutes.

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 [12]. The banking software contains JML specifications with quantified formulas. When applying KeY's test generation techniques [16,22,23], the quantified formulas are encountered as test data constraints. The algorithm ensures that test data is generated only for those constraints that are satisfiable, i.e. it ensures soundness and prevents false positive tests. Our goal was to test for how many of these formulas our algorithm can generate models. Table 2 shows the results.

The upper sub-table of Table 2 shows numbers of quantified formulas that stem from class invariants of the respective classes and the lower sub-table shows numbers of quantified formulas that stem from method precondi-

— JAVA + JML —

```

1  /*@ public normal_behavior
2  @ requires
3  @   next!=null && prev!=null && next!=prev
4  @   && (\forall int k; true ;
5  @     0<=next[k] && next[k] < prev.length)
6  @   && (\forall int l; 0<=l && l<next.length;
7  @     next[l]==1);
8  @ ensures (\forall int j;
9  @   0<=j && j<next.length; prev[next[j]]==j);
10 @ modifies prev[*]; @*/
11 public void link(){
12 /*@ loop_invariant
13 (\forall int x;0<=x && x <= i; prev[next[x]]==x)
14   && (0<=i && i<next.length) ;
15   modifies prev[*],i; @*/
16 for(int i=0;i<next.length;i++){prev[next[i]]=i;}
17 }

```

— JAVA + JML —

Figure 3. An example of a JAVA method (of class MyCls) with a JML specification that is not verifiable because the underlined formula should be $x < i$ instead of $x \leq i$

$$\begin{aligned}
& \forall x : \text{int}. (x \leq -1 \vee \\
& x \geq 1 + i_0 \vee \text{get}_0(\text{prev}(\text{self}), \text{acc}_{\square}(\text{next}(\text{self}), x) \doteq x), \\
& \forall x : \text{MyCls}. (\text{prevAtPre}(x) \doteq \text{prev}(x)), \\
& \forall x : \text{MyCls}. (x \doteq \text{null} \vee \neg \text{created}(x) \vee \neg \text{a}(x) \doteq \text{null}), \\
& \forall x : \text{MyCls}. (x \doteq \text{null} \vee \neg \text{created}(x) \vee \neg \text{next}(x) \doteq \text{null}), \\
& \forall x : \text{MyCls}. (x \doteq \text{null} \vee \neg \text{created}(x) \vee \neg \text{prev}(x) \doteq \text{null}), \\
& \forall x : \text{int}. \text{acc}_{\square}(\text{next}(\text{self}), x) \geq 0), \\
& \forall x : \text{int}. \text{acc}_{\square}(\text{next}(\text{self}), x) \leq -1 + \text{length}(\text{prev}(\text{self})), \\
& \forall x : \text{int}. (l \leq -1 \vee \\
& l \geq \text{length}(\text{next}(\text{self})) \vee \text{acc}_{\square}(\text{next}(\text{self}), x) \doteq x), \\
& \dots \Rightarrow \dots
\end{aligned}$$

Figure 4. Quantified formulas in a sequent resulting from a failed verification attempt of the code in Figure 3 that have to be satisfied by test data; 21 additional ground formulas are abbreviated by ‘...’

```

...
{for x : MyCls; (next(x) \doteq null \wedge \neg a(x) \doteq null \wedge ...);
  created(x) := false}
{for x : MyCls; (a(x) \doteq 0 \wedge \neg x \doteq null);
  created(x) := false}
{for x : int; (b \ge 1 + x \wedge x \le -1);
  acc_{\square}(next(self)) := -1 + c_2}
{for x : int; x \le -1; i := acc_{\square}(next(self)) - c_0 * -1 + c_1}
{for x : int; (x \ge 0 \wedge x \ge 1 + i_0);
  acc_{\square}(next(self)) := length(prev(self)) + c_0}
{for x : int; (acc_{\square}(next(self), x) = x \wedge x \le i_0 \wedge ...);
  get_0(prev(self), x) := x}

```

Figure 5. A subset of generated updates representing a model for the quantified formulas in Figure 4

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

Table 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

tions 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 the SMT solvers could not 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. Annotation of the quantified formulas with patterns does not help because patterns are used for quantifier instantiation which is not an alternative technique. The user-interaction time was between 10 minutes and a few seconds, e.g. 5 clicks. Automatic updates selection and backtracking would increase the speed significantly. The implementation is not yet applicable on examples from SMT-LIB mainly because it lacks heuristics for handling other theories than arithmetic with uninterpreted functions.

11 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 formu-

las. 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 algorithm can be used as a pre-computation 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. With our approach we were able to achieve sound fault-detection and test generation in the context of formal verification with KeY [26]. In particular, the model generator ensures that unsound tests which do not satisfy preconditions are not generated.

We have identified problems (Proposition 1) that occur when the approach is implemented according to the *basic* description. Theorem 1 provides a solution to these problems. The theorem allows us to reformulate the basic model generation approach for quantified formulas into a semantically equivalent approach without the problems described in Proposition 1. Models generated by the algorithm are represented as updates. Quantified updates are suitable for representing models of quantified formulas. Our algorithm systematically analyzes quantified formulas and uses heuristics to generate candidate updates. The advantage of using updates is the possibility to express models for quantified formulas via quantified updates, and the availability of a powerful calculus for simplifying formulas with updates to FOL formulas. In particular, no loop invariants have to be generated in order to simplify quantified updates. Furthermore, updates are a convenient model representation language for test generation, because they have program semantics and can be converted into a test preamble.

Future work goes into two directions: automation and generalization. The prototype can be automated by performing random selections at the choice points of the model generator. To improve efficiency, heuristics have to be investigated. A heuristic that we have manually applied was to backtrack when formulas start getting bigger rather than smaller. A way to increase automation is also specialization of the heuristics for a particular tool. Problematic formulas for SMT solver are often specific for a particular tool or problem. If a sequence of updates for such formulas is found, this solution can be simply reused, e.g., when a test suite is generated and the model generator is queried multiple times. Another important direction is to investigate the combination of our approach with SMT solvers. As mentioned previously, our model generator can be used as a precomputation step for SMT solvers. The idea is to use our model generator for handling of formulas that are problematic for a given SMT solver and to let the solver handle the remaining formulas.

So far we have not regarded model generation for recursively defined functions. Quantified updates can in

some cases represent models for recursive function. To generate such updates automatically, closed forms of recursively defined functions have to be found. These could be provided in form of heuristics as was done in Section 7. Another research direction is to investigate the usage of a Turing-complete language for model representation. As mentioned in Section 3 an iterative loop does not terminate when constructing an infinite model. Hence, we cannot prove that the program reaches a state that satisfies the model. However, we conjecture that the following sentence is correct: If there is a program p that builds a model for an arbitrarily large but finite quantification domain \mathcal{D}_i for $\forall x \in \mathcal{D}_i. \varphi(x)$, and by increasing the domain (\mathcal{D}_{i+1}) the program yields an extended model for $\forall x \in \mathcal{D}_{i+1}. \varphi(x)$, then $\forall x. \varphi(x)$ is satisfiable in the infinite domain. This statement would allow using a Turing-complete language for model representation.

References

1. C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings, Computer Aided Verification, 19th International Conference, CAV 2007*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.
2. P. Baumgartner, A. Fuchs, and C. Tinelli. Implementing the model evolution calculus. *International Journal on Artificial Intelligence Tools*, 15(1):21–52, 2006.
3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.
4. F. Benhamou and F. Goualard. Universally quantified interval constraints. In R. Dechter, editor, *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore*, volume 1894 of *LNCS*, pages 67–82. Springer, 2000.
5. A. R. Bradley, Z. Manna, and H. B. Sipma. What’s decidable about arrays? In E. A. Emerson and K. S. Namjoshi, editors, *Proceedings, Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA*, volume 3855, pages 427–442. Springer, 2006.
6. C. Csallner and Y. Smaragdakis. Check ‘n’ Crash: combining static checking and testing. In *ICSE*, pages 422–431. ACM, 2005.
7. L. M. de Moura and N. Bjørner. Engineering DPLL(T) + saturation. In *IJCAR*, volume 5195 of *LNCS*, pages 475–490. Springer, 2008.
8. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
9. D. Déharbe and S. Ranise. Satisfiability solving for software verification. *STTT*, 11(3):255–260, 2009.
10. 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.
11. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
 12. L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. Case study in jml-based software validation. In *ASE*, pages 294–297. IEEE CS, 2004.
 13. 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 December 2010.
 14. B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In T. Ball and R. B. Jones, editors, *Proceedings, Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.
 15. C. Engel. Verification based test case generation. Master's thesis, University of Karlsruhe, Institut für Theoretische Informatik, 2006.
 16. C. Engel, C. Gladisch, V. Klebanov, and P. Rümmer. Integrating verification and testing of object-oriented software. In B. Beckert and R. Hähnle, editors, *Proceedings, Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy*, volume 4966 of *LNCS*, pages 182–191. Springer, 2008.
 17. Y. Ge, C. W. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *Ann. Math. Artif. Intell.*, 55(1-2):101–122, 2009.
 18. Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In A. Bouajjani and O. Maler, editors, *Proceedings, Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.
 19. I. P. Gent, P. Nightingale, and K. Stergiou. QCSP-Solve: A solver for quantified constraint satisfaction problems. In L. P. Kaelbling and A. Saffiotti, editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK (IJCAI 2005)*, pages 138–143. Professional Book Center, 2005.
 20. S. Ghilardi. Quantifier elimination and provers integration. *Electr. Notes Theor. Comput. Sci.*, 86(1):22–34, 2003.
 21. M. Giese. Incremental closure of free variable tableaux. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings, Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy*, volume 2083 of *LNCS*, pages 545–560. Springer, 2001.
 22. C. Gladisch. Verification-based test case generation for full feasible branch coverage. In A. Cerone and S. Gruner, editors, *Proceedings, Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa*, pages 159–168. IEEE Computer Society, 2008.
 23. C. Gladisch. Could we have chosen a better loop invariant or method contract? In C. Dubois, editor, *Proceedings, Tests and Proofs, Third International Conference, TAP 2009, Zurich, Switzerland*, volume 5668 of *LNCS*, pages 74–89. Springer, 2009.
 24. C. Gladisch. Satisfiability solving and model generation for quantified first-order logic formulas. In B. Beckert and C. Marché, editors, *Conf. Post. Proc., Formal Verification of Object-Oriented Software International Conference, FoVeOOS 2010, Paris, France*, volume 6528 of *LNCS*. Springer, 2010.
 25. C. Gladisch. Test data generation for programs with quantified first-order logic specifications. In A. Petrenko, A. da Silva Simão, and J. C. Maldonado, editors, *Proceedings, Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil*, volume 6435 of *LNCS*, pages 158–173. Springer, 2010.
 26. C. Gladisch. *Verification-based Software-fault Detection*. PhD thesis, Karlsruhe Institute of Technology (KIT), 2011.
 27. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, London, England, 2000.
 28. KeY project homepage. <http://www.key-project.org/>.
 29. 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.
 30. G. Leavens and Y. Cheon. Design by contract with JML, 2006. <http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf>. Visited December 2010.
 31. P. McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.
 32. M. Moskal. *Satisfiability Modulo Software*. PhD thesis, University of Wrocław, 2009.
 33. M. Moskal, J. Lopuszanski, and J. R. Kiniry. E-matching for fun and profit. *Electr. Notes Theor. Comput. Sci.*, 198(2):19–35, 2008.
 34. R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Challenges in satisfiability modulo theories. In F. Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France*, volume 4533 of *LNCS*, pages 2–18. Springer, 2007.
 35. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.
 36. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *LPAR*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.
 37. P. Rümmer and M. A. Shah. Proving programs incorrect using a sequent calculus for Java dynamic logic. In Y. Gurevich and B. Meyer, editors, *Proceedings, Tests and Proofs, First International Conference, TAP 2007*, volume 4454 of *LNCS*, pages 41–60. Springer, 2007.
 38. W. Visser, C. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, pages 97–107. ACM, 2004.
 39. C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischniewski. Spass version 3.5. In *CADE*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.
 40. J. Zhang and H. Zhang. Extending finite model searching with congruence closure computation. In B. Buchberger and J. A. Campbell, editors, *Proceedings, Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004, Linz, Austria*, volume 3249 of *LNCS*, pages 94–102. Springer, 2004.