

# Satisfiability Solving and Model Generation for Quantified First-order Logic Formulas

Christoph D.Gladisch\*

Karlsruhe Institute of Technology (KIT)  
Institute for Theoretical Informatics  
Germany

**Abstract.** The generation of models, i.e. interpretations, that satisfy first-order logic (FOL) formulas is an important problem in different application domains, such as, e.g., formal software verification, testing, and artificial intelligence. Satisfiability modulo theory (SMT) solvers are the state-of-the-art techniques for handling this problem. A major bottleneck is, however, the handling of quantified formulas.

Our contribution is a model generation technique for quantified formulas that is powered by a verification technique. The model generation technique can be used either stand-alone for model generation, or as a precomputation step for SMT solvers to eliminate quantifiers. Quantifier elimination in this sense is sound for showing satisfiability but not for refutational or validity proofs. A prototype of this technique is implemented.

## 1 Introduction

Showing the satisfiability of a first-order logic (FOL) formula means to show the existence of an interpretation in which the formula evaluates to true. This is an important and long studied problem in different application domains such as formal software verification, software testing, and artificial intelligence. In software verification and testing the models, i.e. interpretations, are used as counter examples to debug programs and specifications and to generate test data respectively.

Satisfiability modulo theory (SMT) solvers are the state-of-the-art techniques for showing satisfiability of FOL formulas and to generate models for FOL formulas. A major bottleneck is, however, the handling of quantifiers (see, e.g., [10, 23, 13, 24]). Quantifiers often lead to problems that are not in the decidable fragments of SMT 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.

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

---

\* gladisch@uni-koblenz.de

solvers can solve in cases where it terminates. As a motivating example, assume we want to show the satisfiability of the formula  $\phi_1$  which we define as

$$\forall x.(x \geq 0 \rightarrow prev(next(x)) = x) \tag{1}$$

where *prev* and *next* are uninterpreted function symbols. Some state-of-the-art SMT solvers —concretely we have tested Z3 [9], CVC3 [1], Yices [12]— are in contrast to the proposed technique not capable to solve this formula. The reason is that this formula is not in the decidable fragment of the solvers because it combines arithmetics, uninterpreted functions, and quantification.

The proposed technique is also capable of generating only 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 precomputation 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 handling of quantifiers in refutational and validity proofs powerful instantiation based techniques already exist.

While model generation is not a new idea, the novelty of our approach are (1) the choice of language to syntactically represent (partial) interpretations, (2) the technique for construction of models, and (3) the means to evaluate (quantified) formulas under these interpretations. Since satisfiability solving and model generation for ground formulas is already well studied, we concentrate on the handling of quantified formulas.

We experience that in software verification much time is spend with fixing and adjusting the programs, specifications, and annotations. For instance, Figure 2 shows an unprovable verification condition with subformulas similar to formula  $\phi_1$ . It is invaluable to detect if such verification conditions have counter examples. Once a program is correct and annotations are strong enough a verification tool can afterwards prove the correctness of the program usually automatically. The generation of counter examples is further important in counter example guided abstraction refinement (CEGAR) [7] and for checking the consistency, i.e. contradiction-freeness, of axiomatizations and of preconditions in specifications.

This paper is based on the technical report [17] where we propose our approach for the first time. In [18], which is a paper following this paper, we describe an algorithm that implements our approach. While in [18] we describe the application of the algorithm to test data generation and its evaluation, the contributions of this paper are the description of the theory of our approach and a soundness proof.

## 1.1 Background and Related Work

One has to distinguish between different quantifiers in different contexts, namely between those that can be skolemized and those that cannot be skolemized. For instance, in an attempt to show the validity of the formula  $\forall x.\varphi(x)$ , the variable  $x$  can be skolemized, i.e. replaced by a fresh constant, because all symbols of the

signature are implicitly universally quantified in this context. When showing the validity of  $\exists x.\varphi(x)$ , then skolemization is not possible. In contrast, when showing satisfiability, then skolemization is allowed for  $\exists x.\varphi(x)$  but not for  $\forall x.\varphi(x)$ . Thus, assuming the formulas being in prenex form, the tricky cases are the handling of (a) existential quantification when showing validity and (b) universal quantification when showing satisfiability. 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) 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 [23] 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 [21]) as shown in the following example.

$$\forall h.\forall i.\forall v.rd(wr(h, i, v), i) = v \tag{2}$$

$$\forall h.\forall j.0 \leq rd(h, j) \wedge rd(h, j) \leq 2^{32} - 1 \tag{3}$$

Formula (2) is an axiom of the theory of arrays and (3) 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 [22]. Formula (3) 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  $(2) \wedge (3)$  is inconsistent, i.e. it is false, which can be easily seen when considering the following instantiation  $[h := wr(h_0, k, 2^{32}), j := k]$ , (see [22]). Simplify, however, produces a counter example for  $\neg((2) \wedge (3))$ , which means that it satisfies the *false* formula  $(2) \wedge (3)$ . 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 [14]. An important fragment of FOL for program specification which allows a complete instantiation is the Array Property Fragment [6]. E-matching is used in state-of-the-art SMT solvers such as Z3 [9], CVC3 [1], Yices [12], and others (see [8]). Formula  $\phi_1$  which is solvable with our technique is, however, neither in the Bernays-Schönfinkel class nor in the Array Property Fragment.

Another set of approaches for finding instantiations of quantified formulas is based on free-variables (see e.g. [16]). These approaches focus, however, on validity or respectively unsatisfiability proofs and not on satisfiability solving. More precisely, they do not guarantee a complete instantiation of quantifiers in general case.

Satisfiability of a formula can be shown by weakening the formula with existential quantifiers and then showing its validity, instead of satisfiability. This idea is followed in [27] for proving the existence of a state that reveals a software bug. The approach uses free variables in order to compute instantiations of the existentially quantified variables.

Model generation theorem provers (MGTP) are similar to SMT solvers as their underlying technique is DPLL lifted to FOL. For instance, the theorem prover Darwin [2] is an instantiation-based prover which is sound and complete for the unsatisfiability of FOL, i.e. without theories. For the satisfiability part it decides Bernays-Schönfinkel formulas.

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., [15] for more examples). These techniques are, in contrast to the proposed technique, not capable of eliminating the quantifier in, e.g.,  $\phi_1$ . 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.

Quantified constraint satisfaction problem (QCSP) solvers 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.

Finite and infinite model building techniques are described in [25]. The authors distinguish between enumeration-based methods corresponding to the above mentioned instantiation techniques and deduction-based methods which are in the main focus of the book. Deductive methods produce syntactic representations of models in some logical language. Nitpick which is the counter example generator of Isabel/HOL uses first-order relational logic (FORL) [5]. FORL extends FOL with relational calculus operators and the transitive closure. The approach we propose is a deduction-based method which differs from existing approaches in the representation and generation of models.

**Structure of the paper.** In Section 2 the basic idea of our approach is explained. In Section 3 the underlying formalism of our approach is introduced. The main sections are Section 4 and 5 where the approach is described in more detail and where we identify the crucial problems that have to be solved. The solution to the problems described in Section 4 is given in form of a theorem and the soundness of the theorem is proved. In Section 6 we report on our preliminary experiments with our approach and provide conclusions and further research plans.

## 2 The Basic Idea of our Approach

The basic idea of our approach is to generate a partial FOL model, i.e. a partial interpretation, 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 weakest precondition computation in order to evaluate the quantified formulas to true. Weakest precondition is a well-known concept in formal software verification and symbolic execution based test generation. A weakest precondition  $wp(p, \varphi)$ , where  $p$  is a program and  $\varphi$  is a formula, expresses all states such that execution of  $p$  in any of these states results in states in which  $\varphi$  evaluates to true. Here, program states and FOL interpretations are understood as the same concept. Our approach is to generate for a given quantified formula  $\varphi$  a program  $p$  such that the final states of  $p$  satisfy  $\varphi$ . Thus a technique for program generation is one of our contributions.

For example, in order to solve  $\phi_1$ , we could generate the following program (assuming, e.g., JAVA-like syntax and semantics):

```
for(i=0;true;i++){ next[i]=new T(); next[i].prev=i; } (4)
```

and compute the weakest precondition of  $\phi_1$  with respect to this program, i.e.  $wp((4), \phi_1)$ . Using a verification calculus the weakest precondition of the quantified subformula can be evaluated to true. Thus, effectively the quantified formula is eliminated and a partial interpretation represented in form of a program is obtained.

A typical programming language such as JAVA is, however, not *directly* suitable for this task because function and predicate symbols are usually not representable in such languages. A verification calculus may also require extensions because loops are usually handled by the loop invariant rule and the loop invariant may introduce new quantified formulas.

A language and a calculus that are suitable for our purpose exist, however, in the verification system KeY. The language consists of so-called *updates*. In the following sections we introduce this language and describe our technique for construction of updates that evaluate quantified formulas to true while reducing the chance of introducing new quantified formulas.

### 3 KeY's Dynamic Logic with Updates

The KeY system [3, 20] is a verification and test generation system for a subset of JAVA. The system is based on the logic JAVA CARD DL, which is an instance of Dynamic Logic (DL) [19]. 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* [26].

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

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

Updates capture the essence of programs, namely the state change computed by a program execution. States and FOL interpretations are the same concept. An update is a modality which moves the interpretation to a new Kripke state and we say an update *changes* an interpretation. The states differ in the interpretation of symbols  $\Sigma_{nr}^f$  such as uninterpreted functions. Updates represent partial states and can be used to represent (partial) models of formulas. The set  $\Sigma_r^f$  represents rigid functions whose interpretation is fixed and cannot be changed by an update.

For instance, the formula  $(\{a := b\}a \doteq c) \in Fml$ , where  $a \in \Sigma_{nr}^f$  and  $b, c \in \Sigma^f$  consists of the (function) update  $a := b$  and the *application* of the update modal operator  $\{a := b\}$  on the formula  $a \doteq c$ . 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  $x \in V$ ;  $u, u_1, u_2 \in U$ ;  $f \in \Sigma_{nr}^f$ ;  $t, t_1, t_2 \in Trm$ ;  $\varphi \in Fml$ .

- *Updates.* The set  $U$  of updates consists of: neutral update  $\varepsilon$ ; 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  $(u_1 || u_2)$ ; conditional updates  $(\text{if } \varphi; u)$ ; and quantified updates  $(\text{for } x; \varphi; u)$ .
- *Terms.* The set of Dynamic Logic terms includes all FOL terms, i.e.  $Trm \supset Trm_{FOL}$ ; and  $\{u\}t \in Trm$  for all  $u \in U$  with no free variables and  $t \in Trm$ .
- *Formulas.* The set of Dynamic Logic formulas includes all FOL formulas, i.e.  $Fml \supset Fml_{FOL}$ ;  $\{u\}\varphi \in Fml$  for all  $u \in U$  with no free variables and  $\varphi \in Fml$ ; sequents  $\Gamma \Rightarrow \Delta \in Fml$ , where  $\Gamma, \Delta \subset Fml$ ; and all  $\varphi \in Fml$  are closed by quantifiers, i.e.  $\varphi$  has no free variables, if not stated otherwise.

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  $\gamma_1, \dots, \gamma_n \in \Gamma$  and  $\delta_1, \dots, \delta_m \in \Delta$  are closed formulas. Sequents are normally, e.g. in [3], not included in the set of formulas. However, in this work it is convenient to include them to the set of formulas as *syntactic sugar*.

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

*Terms and Formulas*

- $val_s(\{u\}t) \equiv val_{s'}(t)$ , where  $s' \equiv val_s(u)$
- $val_s(\{u\}\varphi) \equiv val_{s'}(\varphi)$ , where  $s' \equiv val_s(u)$

*Updates*

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

*If  $v_0 \not\equiv v_2$  then  $val_{s'}(t) \equiv v_2$  else  $val_{s'}(t) \equiv v_1$ .*

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

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

*satisfying a well-ordering  $\succ$  such that  $\beta(x_i) \succ \beta(x_j)$ , where  $\beta : V \rightarrow \mathcal{D}$ .*

A complete and formal definition of quantified updates cannot be given in the scope of this paper; we refer the reader to [26, 3] for a complete definition of the language and the simplification calculus. In the following some examples are shown of how updates, terms, and formulas are evaluated in KeY respecting the given semantics in Def 2.

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

## 4 Model Generation by Iterative Update Construction

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

Our main contribution is a technique for generating (partial) models for quantified formulas. As this work was developed in the context of KeY which is based on a sequent calculus, we consider the model generation problem of a quantified formula  $\forall x.\phi(x)$  in a sequent  $\varphi = (\Gamma, \forall x.\phi(x) \Rightarrow \Delta)$ . Such sequents occur frequently as open branches of failed proof attempts. The reason for proof failure is often unclear and it is desired to determine if  $\varphi$  has a counter example, i.e. if a model exists for  $\neg\varphi$ . The goal is therefore given by the following problem description.

**Definition 3.** *Problem Description.* Given a sequent  $(\Gamma, \forall x.\phi(x) \Rightarrow \Delta)$  the goal is to generate an update  $u$  such that:

$$(\{u\}(\Gamma, \forall x.\phi(x) \Rightarrow \Delta)) \equiv (\{u\}(\Gamma, true \Rightarrow \Delta)) \quad (5)$$

If this problem is solved by a technique, then this technique can be applied iteratively to all quantified formulas occurring in  $\Gamma$  and  $\Delta$  resulting in a sequent  $\Gamma' \Rightarrow \Delta'$  that consists only of ground formulas. Note that non-skolemizable quantified formulas occurring in  $\Delta$  are those with existential quantifiers and they can be moved to  $\Gamma$  using the following equivalence:  $(\Gamma \Rightarrow \exists x.\phi(x), \Delta) \equiv (\Gamma, \forall x.\neg\phi(x) \Rightarrow \Delta)$ .

We have implemented different algorithms that follow this approach. Unfortunately, only in rare cases the problem formulated in Def. 3 was solved by early algorithms. Based on experiments with early algorithms we have identified two important problems that we state in form of the following informal proposition.

**Proposition 1.** *The following description follows the notation of Def. 3.*

- a) *In general cases of  $\forall x.\phi(x)$ , it is not feasible to construct an update  $u$  such that  $\models \{u\}\forall x.\phi(x)$ , without analysing the semantic properties of the matrix  $\phi(x)$ .*
- b) *The theorem prover defined in [3] is not sufficiently powerful to simplify  $(\Gamma, \{u\}\forall x.\phi(x) \Rightarrow \Delta)$  to  $(\Gamma, true \Rightarrow \Delta)$  if  $\models \{u\}\forall x.\phi(x)$  and  $u$  is a quantified update.*

Some possibilities to analyse the semantic properties of  $\phi(x)$  are to test instances of  $\phi(x)$  or to use free variables (see, e.g., [16]). We have experimented with the latter approach and could solve problem (a) in several cases but we describe a better approach in this paper. The reason for problem (b) is that in order to simplify the matrix  $\phi(x)$  the sequent calculus requires semantic information about  $\phi(x)$  to be available on the sequent level, i.e. in the formulas  $\Gamma \cup \Delta$ .

We have implemented an algorithm that solves both problems of Proposition 1. The algorithm is described and evaluated in [18] but without a soundness proof. In this section we provide a theorem that formalizes only the crucial problem simplification technique of the algorithm and prove it.

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  $notZero \in \Sigma^f$  with the axiom  $\neg(notZero \doteq 0)$ . With this axiom it is, e.g., possible to write an update  $a := b + notZero$ , with  $a, b \in Trm_{FOL}$ , expressing a general assignment to  $a$  with a value different from  $b$ . Each update  $u_i$  is therefore associated with an axiom  $\alpha_i$ .

**Definition 4.** *Given a sequent  $\varphi = (\Gamma, \forall x.\phi(x) \Rightarrow \Delta)$ , where  $\Gamma, \Delta \subset Fml$  and  $\phi(x)$  is an arbitrary formula with an occurrence of  $x \in V$ , i.e.  $\phi$  is not restricted to  $\phi \in \Sigma^p$ . Let  $u_0, \dots, u_m \in U$ ;  $\alpha_0, \dots, \alpha_m \in Fml$ , with  $m \in \mathbb{N}$ , be closed by quantifiers. The formulas  $\psi_m, \varphi'_m, \varphi_m \in Fml$ , are defined recursively as:*

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

Definition 4 describes an abstract search technique for a sequence of updates  $u_m; \dots; u_0$ ,  $m \in \mathbb{N}$ , for solving the problem of Def. 3. The updates  $u_m; \dots; u_0$  constitute the update  $u$  in Def. 3 and  $\varphi_0 \equiv \varphi$  is the original sequent that is to be shown falsifiable. In the following theorem we assume  $\gamma = \underline{\forall x.\phi(x)}$ .

**Theorem 1.** *Let  $\varphi = (\Gamma, \gamma \Rightarrow \Delta)$  and  $\psi_m, \varphi'_m, \varphi_m \in Fml$ , with  $m \in \mathbb{N}$ , be defined according to Def. 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  $s = val_{s_m}(u_m; \dots; u_1; \varepsilon)$  and  $s \models \neg\varphi$ .*

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 Def. 4, then from  $s_m$  a model for  $\neg\varphi$  can be derived by evaluation of the updates  $u_0, \dots, u_m$ . Hence,  $\neg\varphi_m$  can be used to show the satisfiability of  $\neg\varphi$ .

For instance, let  $\varphi \equiv (\neg a = b)$ , then a suitable pair  $(u_0, \alpha_0)$  to construct  $\varphi_1$  is, e.g.  $(a := b, true)$ . In this case  $\varphi_1$  has the form  $\{a := b\}(true \rightarrow (\neg a = b))$  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  $s = val_{s_1}(a := b)$ . Note, that choosing an update is a heuristic, e.g. the pair  $(b := a, true)$  or the pair  $(a := 1 \parallel b := 1, true)$  are also suitable candidates.

An important property of the statement for the construction of an update search procedure is that soundness of the statement is preserved by any pair  $(u, \alpha)$ . For instance, consider the pair  $(a := 1 \parallel b := 2, true)$  or the pair  $(a := b, false)$ . In both cases  $\varphi_1$  evaluates to *true*. Hence, there is no  $s_1 \in \mathcal{S}$  such that  $s_1 \models \neg\varphi_1$  and therefore no implication is made regarding the satisfiability of  $\varphi$ .

Based on statement (i) an algorithm can be constructed for the generation of models for ground formulas. The challenge is to generate a model that satisfies a quantified formula that cannot be skolemized. If  $\psi_m$  is valid, then the model generation problem for  $\neg\varphi_m$  can be replaced by the model generation problem for  $\neg\varphi'_m$  because  $\varphi_m$  and  $\varphi'_m$  are equivalent. Considering Def. 4, the statement is interesting because in  $\varphi'_m$  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 the problem in Def. 3. Theorem 1 states that the problem  $\varphi_m \equiv \varphi'_m$  can be solved by a validity proof of  $\psi_m$ . This allows solving the problems described in Proposition 1 because the quantified formula in  $\psi_m$  occurs negated wrt.  $\varphi_m$  and can therefore

be skolemized — note that  $(\Gamma, \forall x.\phi(x) \Rightarrow \Delta) \equiv (\Gamma \Rightarrow \neg\forall x.\phi(x), \Delta)$ . When  $\psi_m$  is skolemized, then it is (a) easy to analyse the semantics of  $\phi(sk)$ , where  $sk \in \Sigma^f$  is the skolem function, 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.

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

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

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

In the proof of Theorem 1 we use the following lemma.

**Lemma 1.** *Weakening Update.* Let  $u \in U$  and  $\varphi \in Fml$ . If  $\models \varphi$ , then  $\models \{u\}\varphi$ .

*Proof of Lemma 1.* Since for any  $s \in \mathcal{S}$ , holds  $s \models \varphi$ , it is also the case for  $s' = val_s(u)$  that  $s' \models \varphi$  because  $s' \in \mathcal{S}$ . ■

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

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

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

can be shown by using propositional transformation rules. In the following we simplify  $\varphi'_0 \leftrightarrow \varphi_0$  and derive by equivalence transformations  $\psi_0$ .

$$\begin{aligned} & ((\Gamma \wedge true) \rightarrow \Delta) \leftrightarrow ((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta) \\ & (\Gamma \rightarrow \Delta) \leftrightarrow ((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta) \\ \\ & (\Gamma \rightarrow \Delta) \rightarrow ((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta) \quad ((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta) \rightarrow (\Gamma \rightarrow \Delta) \\ & ((\Gamma \rightarrow \Delta) \wedge \Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta \quad (((\Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta) \wedge \Gamma) \rightarrow \Delta \\ & (\Delta \wedge \Gamma \wedge \forall x.\phi(x)) \rightarrow \Delta \quad ((\forall x.\phi(x) \rightarrow \Delta) \wedge \Gamma) \rightarrow \Delta \\ & (\Delta \wedge \Gamma) \rightarrow \Delta \quad ((\forall x.\phi(x) \rightarrow \Delta) \wedge \Gamma) \rightarrow \Delta \\ & \Delta \rightarrow \Delta \quad ((\neg\forall x.\phi(x) \wedge \Gamma) \rightarrow \Delta) \wedge ((\Delta \wedge \Gamma) \rightarrow \Delta) \\ & true \quad (\neg\forall x.\phi(x) \wedge \Gamma) \rightarrow \Delta \\ & \quad \quad \quad \Gamma \rightarrow (\forall x.\phi(x) \vee \Delta) \end{aligned}$$

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

Induction Step ( $m \geq 0$ ). (i) Assuming  $\models \psi_m \leftrightarrow (\varphi'_m \leftrightarrow \varphi_m)$ , we want to show  $\models \psi_{m+1} \leftrightarrow (\varphi'_{m+1} \leftrightarrow \varphi_{m+1})$ . If  $\models \psi_m \leftrightarrow (\varphi'_m \leftrightarrow \varphi_m)$ , then

$$\models \alpha_m \rightarrow (\psi_m \leftrightarrow (\varphi'_m \leftrightarrow \varphi_m)) \tag{6}$$

for any  $\alpha_m \in Fml$ . We use the equivalence

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

to derive the following statement that is equivalent to (6)

$$\models ((\alpha_m \rightarrow \psi_m) \leftrightarrow ((\alpha_m \rightarrow \varphi'_m) \leftrightarrow (\alpha_m \rightarrow \varphi_m))) \quad (7)$$

Due to Lemma 1, (7) implies

$$\models \{u_m\}((\alpha_m \rightarrow \psi_m) \leftrightarrow ((\alpha_m \rightarrow \varphi'_m) \leftrightarrow (\alpha_m \rightarrow \varphi_m))) \quad (8)$$

that can be simplified by to (for all operators  $\circ$ :  $\{u\}(A \circ B) \equiv \{u\}A \circ \{u\}B$ )

$$\models (\{u_m\}(\alpha_m \rightarrow \psi_m) \leftrightarrow (\{u_m\}(\alpha_m \rightarrow \varphi'_m) \leftrightarrow \{u_m\}(\alpha_m \rightarrow \varphi_m))) \quad (9)$$

Statement 9 is equivalent to  $\models \psi_{m+1} \leftrightarrow (\varphi'_{m+1} \leftrightarrow \varphi_{m+1})$ .

(ii) 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  $\varphi_m^- \in Fml$  that can be recursively defined as

$$\varphi_0^- = \neg(\Gamma, true \Rightarrow \Delta) \quad \varphi_{m+1}^- = \{u_m\}(\alpha_m \wedge \varphi_m^-)$$

Hence,  $s_{m+1} \models \neg\varphi_{m+1}$  is equivalent to  $s_{m+1} \models \varphi_{m+1}^-$  which is equivalent to  $s_{m+1} \models \{u_m\}(\alpha_m \wedge \varphi_m^-)$ . There is  $s_m \in \mathcal{S}$  with  $s_m = val_{s_{m+1}}(u_m)$  such that  $s_m \models \alpha_m \wedge \varphi_m^-$  and therefore  $s_m \models \varphi_m^-$ . Since  $\varphi_m^-$  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  $s = val_{s_m}(u_m; \dots; u_1; \varepsilon)$  such that  $s \models \neg\varphi$ . Because of  $s_m = val_{s_{m+1}}(u_m)$ , we conclude that if  $s_{m+1} \models \neg\varphi_{m+1}$ , then there exists  $s \in \mathcal{S}$  with  $s = val_{s_{m+1}}(u_{m+1}; u_m; \dots; u_1; \varepsilon)$  such that  $s \models \neg\varphi$ . ■

## 5 Heuristics for Update Construction from Formulas

While Section 4 describes a general sound framework for model generation, in this section we shortly describe some heuristics that we have implemented to construct concrete updates. In particular we give an intuition of how quantified updates can be constructed in order to satisfy quantified formulas. Important to note is that soundness of Theorem 1 is preserved by *any* sequence of updates with axioms. Hence, unsoundness cannot be introduced by any of the heuristics.

**Definition 6.** *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 subformula in a sequent  $\varphi = (\Gamma, \gamma \Rightarrow \Delta)$ . Let  $\psi = (\Gamma \Rightarrow \gamma, \Delta)$  and  $\varphi' = (\Gamma \Rightarrow \Delta)$ .

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

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

The sequent  $\psi$  is equivalent to  $\psi_0$  and  $\varphi'$  is equivalent to  $\varphi'_0$ , according to Def. 5. In a model search algorithm each time a pair  $(u_m, \alpha_m)$  is constructed, new formulas  $\varphi'_{m+1}, \varphi'_{m+1}$ , and  $\psi_{m+1}$  are generated according to Def. 5. These formulas must be simplified to  $\varphi, \psi$  and,  $\varphi'$ , respectively, such that a new formula  $\gamma \in Fml_{FOL}$  can be selected for update construction according to Def. 6. In the following subsections, case distinctions are made on the structure of  $\gamma$ .

### 5.1 Update Construction from Ground Formulas

*Handling of Equalities.* Assume  $t_1, t_2 \in Trm_{FOL}$  are location terms (see Def. 1). If  $\gamma$  is of the form  $t_1 = l$  or  $l = t_1$ , where  $l$  is a literal, then the pair  $(t_1 := l, true)$  should be created because  $\models \{t_1 := l\}(true \rightarrow (t_1 \doteq l \wedge l \doteq t_1))$ . If  $\gamma$  is of the form  $t_1 = t_2$ , a choice has to be made between the pairs  $(t_1 := t_2, true)$  and  $(t_2 := t_1, true)$ . 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  $f(t_1), f(t_2) \in Trm_{FOL}$  and  $f \in \Sigma^f$ , then  $\models \{u\}(\alpha \rightarrow f(t_1) = f(t_2))$  can be satisfied by the pair  $(t_1 := t_2, true)$  or by  $(t_2 := t_1, true)$ .

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

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

where  $a, b, c, d, e \in \Sigma_{nr}^f$  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  $((a := (d - e - b * c) / 2, true)$ .

*Handling of Inequalities.* Let  $t_1, t_2 \in Trm_{FOL}$  where  $t_1$  is a location term. An inequation  $t_1 \neq t_2$  can be satisfied, e.g., by the pair  $(t_1 := t_2 + 1, true)$ . A more general update is, however,  $t_1 := t_2 + notZero$ , where  $notZero \in \Sigma^f$  is a fresh-symbol representing a value different from 0. This is where the axiom part of a pair comes into play. A general solution for the formula  $t_1 \neq t_2$  is the pair  $(t_1 := t_2 + notZero, \neg(notZero = 0))$ . Inequations of the form  $t_1 < t_2$  can be handled by introducing a fresh symbol  $gtZero \in \Sigma_{nr}^f$  with the axiom  $gtZero > 0$ .

### 5.2 Update Construction from Quantified Formulas

Our approach to create models for quantified formulas is to generate quantified updates. For example, the quantified formula  $\phi_{10}$ :

$$\forall x. x > a \rightarrow f(x) = g(x) + x \tag{10}$$

is satisfiable in any state after execution of the quantified update  $u_{11}$ :

$$\mathbf{for} \ x; \ x > a; \ f(x) := g(x) + x \quad (11)$$

i.e.  $\models \{\phi_{10}\}u_{11}$ . Notice the similar syntactical structure between  $\phi_{10}$  and  $u_{11}$ . Another solution is  $u_{12}$ :

$$\mathbf{for} \ x; \ x > a; \ g(x) := f(x) - x \quad (12)$$

for which  $\models \{u_{12}\}\phi_{10}$  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 5.1 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)

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

The KeY tool implements a powerful update simplification calculus for quantified updates. The 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 use backtracking when new quantified formulas are introduced. A limitation is the handling of recursively defined functions. For instance, the following update application, as well as any other, does not eliminate the quantified subformula of the following formula:

$$\{\mathbf{for} \ x; \ x > 0; \ h(x) := h(x - 1) + 1\} \forall x. x > 0 \rightarrow h(x) = h(x - 1) + 1$$

Finally, the initial example of the paper, i.e. Formula  $\phi_1$ , can be solved by the following quantified update application which KeY simplifies to true.

$$\{(\mathbf{for} \ x_1; \ x_1 \geq 0; \ next(x_1) := x_1); (\mathbf{for} \ x_2; \ x_2 \geq 0; \ prev(next(x_2)) := x_2)\}\phi_1$$

## 6 Experiments, Conclusions, and Future Work

We have proposed a model generation approach for quantified first-order logic (FOL) formulas that is based on weakest-precondition computation. The language we propose for representing models is KeY's update language. 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.

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.

```

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

```

---

**Fig. 1.** 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  $x \leq i$

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

**Fig. 2.** Quantified formulas in a sequent resulting from a failed verification attempt of the code in Figure 1; 21 additional ground formulas are abbreviated by ‘...’

Based on Theorem 1 and Definitions 4 and 5 an algorithm for model generation can be derived. The technique can be used in two ways. On the one hand, it can be used as a precomputation step to SMT solvers by restricting the computation of the formulas  $\psi_m$ ,  $\varphi'_m$ , and  $\varphi_m$  to Def 4. In this case the technique eliminates quantified formulas and leaves a residue of ground formulas or alternative quantified formulas to be solved by a different method, e.g. an SMT solver. On the other hand, the technique can be used stand-alone for model generation by using the general Def. 5.

The approach was developed in the context of a formal software verification and test generation project. Verification attempts often fail, i.e., they are interrupted by a timeout. For instance, Figure 1 shows a JAVA method with a JML specification. A verification attempt of the method results in a set of open proof obligations. One of them is shown in Figure 2 that we abbreviate as  $\varphi$ . For a verification engineer it is important to know if the open proof obligation has a counter example or not. State-of-the-art approaches use SMT solvers to try answering such questions. These are, however, not powerful enough to solve for-

```

...
{for x : MyCls; (next(x) ≐ null ∧ ¬a(x) ≐ null ∧ ...); created(x) := false}
{for x : MyCls; (a(x) ≐ 0 ∧ ¬x ≐ null); created(x) := false}
{for x : int; (b ≥ 1 + x ∧ x ≤ -1); acc[] (next(self)) := -1 + c2}
{for x : int; x ≤ -1; i := acc[] (next(self)) - c0 * -1 + c1}
{for x : int; (x ≥ 0 ∧ x ≥ 1 + i0); acc[] (next(self)) := length(prev(self)) + c0}
{for x : int; (acc[] (next(self), x) = x ∧ x ≤ i0 ∧ ...); get0(prev(self), x) := x}

```

**Fig. 3.** A subset of generated updates satisfying the quantified formulas in Figure 2

mulas such as  $\varphi$ . Our experiments show that our method can generate counterexamples for formulas such as  $\varphi$  that SMT solvers cannot solve [18]. For instance, Figure 3 shows a part of an iterative update application that describes a model for  $\neg\varphi$  and was generated by an implementation of our approach.

What formulas can be solved by our general approach depends on the chosen language for model representation, the theorem prover in use, and the heuristics for model construction. Quantified formulas are suitable to represent models for certain kinds of quantified formulas. They are, however, not sufficient to represent models of inductively defined functions. This problem can be probably solved by an extension of updates which is future work.

## References

1. C. Barrett and C. Tinelli. CVC3. In *CAV*, 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, *CP*, volume 1894 of *LNCS*, pages 67–82. Springer, 2000.
5. J. C. Blanchette. Relational analysis of (co)inductive predicates, (co)algebraic datatypes, and (co)recursive functions. In *TAP*, volume 6143 of *LNCS*, pages 117–134. Springer, 2010.
6. A. R. Bradley, Z. Manna, and H. B. Sipma. What’s decidable about arrays? In *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.
7. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.
8. L. M. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
9. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
10. D. Déharbe and S. Ranise. Satisfiability solving for software verification. *STTT*, 11(3):255–260, 2009.
11. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical report, J. ACM, 2003.

12. B. Dutertre and L. de Moura. The YICES SMT solver. Technical report, Computer Science Laboratory, SRI International, 2006.
13. 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.
14. Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In *CAV*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.
15. S. Ghilardi. Quantifier elimination and provers integration. *Electr. Notes Theor. Comput. Sci.*, 86(1), 2003.
16. M. Giese. Incremental closure of free variable tableaux. In *IJCAR*, volume 2083 of *LNCS*, pages 545–560. Springer, 2001.
17. C. Gladisch. Satisfiability solving and model generation for quantified first-order logic formulas. Karlsruhe Reports in Informatics ISSN: 2190-4782, Fakultät für Informatik Institut für Theoretische Informatik (ITI), 2010.
18. C. Gladisch. Test data generation for programs with quantified first-order logic specifications. In *ICTSS*, volume 6435 of *LNCS*. Springer, 2010. To appear.
19. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, London, England, 2000.
20. KeY project homepage. At <http://www.key-project.org/>.
21. 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.
22. M. Moskal. *Satisfiability Modulo Software*. PhD thesis, University of Wrocław, 2009.
23. M. Moskal, J. Lopuszanski, and J. R. Kiniry. E-matching for fun and profit. *Electr. Notes Theor. Comput. Sci.*, 198(2):19–35, 2008.
24. R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Challenges in satisfiability modulo theories. In *RTA*, number 4533 in *LNCS*, pages 2–18. Springer, 2007.
25. C. Ricardo, L. Alexander, and P. Nicolas. *Automated Model Building*, volume 31 of *Applied Logic Series*. Springer, 2004.
26. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *LPAR*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.
27. P. Rümmer and M. A. Shah. Proving programs incorrect using a sequent calculus for java dynamic logic. In *TAP*, volume 4454 of *LNCS*, pages 41–60. Springer, 2007.