Model Generation for Quantified Formulas: A Taint-Based Approach

Benjamin Farinier[1,2], Sébastien Bardin[1],
Richard Bonichon[1], and Marie-Laure Potet[2]

[1] CEA, LIST, Software Safety and Security Lab, Université Paris-Saclay, France
`firstname.lastname@cea.fr`
[2] Univ. Grenoble Alpes, Verimag, France
`firstname.lastname@univ-grenoble-alpes.fr`

Abstract. We focus in this paper on generating models of quantified first-order formulas over built-in theories, which is paramount in software verification and bug finding. While standard methods are either geared toward proving the absence of a solution or targeted to specific theories, we propose a generic and radically new approach based on a reduction to the quantifier-free case. Our technique thus allows to reuse all the efficient machinery developed for that context. Experiments show a substantial improvement over state-of-the-art methods.

1 Introduction

Context. Software verification methods have come to rely increasingly on reasoning over logical formulas modulo theory. In particular, the ability to generate models (i.e., find solutions) of a formula is of utmost importance, typically in the context of bug finding or intensive testing — symbolic execution [21] or bounded model checking [7]. Since *quantifier-free first-order formulas* on well-suited theories are sufficient to represent many reachability properties of interest, the Satisfiability Modulo Theory (SMT) [6,25] community has primarily dedicated itself to designing solvers able to efficiently handle such problems.

Yet, universal quantifiers are sometimes needed, typically when considering preconditions or code abstraction. Unfortunately, most theories handled by SMT-solvers are undecidable in the presence of universal quantifiers. There exist dedicated methods for a few decidable quantified theories, such as Presburger arithmetic [9] or the array property fragment [8], but there is no general and effective enough approach for the model generation problem over universally quantified formulas. Indeed, generic solutions for quantified formulas involving heuristic instantiation and refutation are best geared to proving the unsatisfiability of a formula (i.e., absence of solution) [13,20], while recent proposals such as local theory extensions [2], finite instantiation [31,32] or model-based instantiation [29,20] either are too narrow in scope, or handle quantifiers on free sorts only, or restrict themselves to finite models, or may get stuck in infinite refinement loops.

Goal and challenge. Our goal is to propose a generic and efficient approach to the model generation problem over arbitrary quantified formulas with support for theories commonly found in software verification. Due to the huge effort made by the community to produce state-of-the-art solvers for quantifier-free theories (QF-*solvers*), it is highly desirable for this solution to be compatible with current leading decision procedures, namely SMT approaches.

Proposal. Our approach turns a quantified formula into a quantifier-free formula with the guarantee that any model of the latter contains a model of the former. The benefits are threefold: the transformed formula is easier to solve, it can be sent to standard QF-solvers, and a model for the initial formula is deducible from a model of the transformed one. The idea is to ignore quantifiers but strengthen the quantifier-free part of the formula with an *independence condition* constraining models to be independent from the (initially) quantified variables.

Contributions. This paper makes the following contributions:

We propose a novel and generic framework for model generation of quantified formula (Sec. 5, Alg. 1) relying on the inference of *sufficient independence condition* (Sec. 4). We prove its *correctness* (Thm. 1, mechanized in Coq) and its *efficiency* under reasonable assumptions (Prop. 4 and 5). Especially our approach implies only a linear overhead in the formula size. We also briefly study its *completeness*, related to the notion of *weakest independence condition*.

We define a taint-based procedure for the inference of independence conditions (Sec. 5.2), composed of a theory-independent core (Alg. 2) together with theory-dependent refinements. We propose such refinements for a large class of operators (Sec. 6.2), encompassing notably arrays and bitvectors.

Finally, we present a concrete implementation of our method specialized on arrays and bitvectors (Sec. 7). Experiments on SMT-LIB benchmarks and software verification problems notably demonstrate that we are able not only to very effectively lift quantifier-free decision procedures to the quantified case, but also to supplement recent advances, such as finite or model-based quantifier instantiation [31,32,29,20]. Indeed, we concretely supply SMT solvers with the ability to efficiently address an extended set of software verification questions.

Discussions. Our approach supplements state-of-the-art model generation on quantified formulas by providing a more generic handling of satisfiable problems. We can deal with quantifiers on any sort and we are not restricted to finite models. Moreover, this is a lightweight preprocessing approach requiring a single call to the underlying quantifier-free solver. The method also extends to *partial* elimination of universal quantifiers, or reduction to *quantified-but-decidable* formulas (Sec. 5.4).

While techniques *a la* E-matching allow to lift quantifier-free solvers to the unsatisfiability checking of quantified formulas, this works provides a mechanism to lift them to the satisfiability checking and model generation of quantified formulas, yielding a more symmetric handling of quantified formulas in SMT. This new approach paves the way to future developments such as the definition of more precise inference mechanisms of independence conditions, the identification of interesting subclasses for which inferring weakest independence conditions is feasible, and the combination with other quantifier instantiation techniques.

2 Motivation

Let us take the code sample in Fig. 1 and suppose we want to reach function `analyze_me`. For this purpose, we need a model (a.k.a., solution) of the reachability condition $\phi \triangleq ax + b > 0$, where a, b and x are symbolic variables associated to the program variables `a`, `b` and `x`. However, while the values of `a` and `b` are user-controlled, the value of `x` is not. Therefore if we want to reach `analyze_me` in a reproducible manner, we actually need a model of $\phi_\forall \triangleq \forall x.ax + b > 0$, which *involves universal quantification.* While this specific formula is simple, model generation for quantified formulas is notoriously difficult: PSPACE-complete for booleans, undecidable for uninterpreted functions or arrays.

```
int main () {
  int a = input ();
  int b = input ();

  int x = rand ();

  if (a * x + b > 0) {
    analyze_me();
  }
  else {
    ...;
  }
}
```

Quantified reachability condition

(1) $\forall x.ax + b > 0$

Taint variable constraint

(2) $a^\bullet \wedge b^\bullet \wedge \neg\,(x^\bullet)$ $\quad$ ($a^\bullet, b^\bullet, x^\bullet$: fresh boolean)

Independence condition

(3) $((a^\bullet \wedge x^\bullet) \vee (a^\bullet \wedge a = 0) \vee (x^\bullet \wedge x = 0)) \wedge b^\bullet$

(4) $((\top \wedge \bot) \vee (\top \wedge a = 0) \vee (\bot \wedge x = 0)) \wedge \top$

(5) $a = 0$

Quantifier-free approximation of (1)

(6) $(ax + b > 0) \wedge (a = 0)$

Fig. 1: Motivating example

Reduction to the quantifier-free case through independence. We propose to ignore the universal quantification over x, but *restrict models to those which do not depend on x.* For example, model $\{\mathtt{a} = 1, \mathtt{x} = 1, \mathtt{b} = 0\}$ does depend on x, as taking $x = 0$ invalidates the formula, while model $\{\mathtt{a} = 0, \mathtt{x} = 1, \mathtt{b} = 1\}$ is *independent of x.* We call constraint $\psi \triangleq (a = 0)$ an *independence condition*: any interpretation of ϕ satisfying ψ will be independent of x, and therefore a model of $\phi \wedge \psi$ will give us a model of $\phi_\forall$.

Inference of independence conditions through tainting. Fig. 1 details in its right part a way to infer such independence conditions. Given a quantified reachability condition (1), we first associate to every variable v a (boolean) *taint variable $v^\bullet$* indicating whether the solution may depend on v (value $\top$) or not (value $\bot$). Here, $x^\bullet$ is set to $\bot$, $a^\bullet$ and $b^\bullet$ are set to $\top$ (2). An independence condition (3) — a formula modulo theory — is then constructed using both initial and taint variables. We extend taint constraints to terms, $t^\bullet$ indicating here whether t may depend on x or not, and we require the top-level term (i.e., the formula) to be tainted to $\top$ (i.e., to be indep. from x). Condition (3) reads as follows: in order to enforce that $(ax + b > 0)^\bullet$ holds, we enforce that $(ax)^\bullet$ and $b^\bullet$ hold, and for $(ax)^\bullet$ we require that either $a^\bullet$ and $x^\bullet$ hold, or $a^\bullet$ holds and $a = 0$ (absorbing the value of x), or the symmetric case. We see that $\cdot^\bullet$ is defined recursively and combines a *systematic part* (if $t^\bullet$ holds then $f(t)^\bullet$ holds, for any f) with a *theory-dependent part* (here, based on ×). After simplifications (4), we obtain $a = 0$ as an independence condition (5) which is adjoined to the reachability condition freed of its universal quantification (6). A QF-solver provides a model of (6) (e.g., $\{\mathtt{a} = 0, \mathtt{b} = 1, \mathtt{x} = 5\}$), lifted into a model of (1) by discarding the valuation of x (e.g., $\{\mathtt{a} = 0, \mathtt{b} = 1\}$).

In this specific example the inferred independence condition (5) is the most generic one and (1) and (6) are equisatisfiable. Yet, in general it may be an under-approximation, constraining the variables more than needed and yielding a correct but incomplete decision method: a model of (6) can still be turned into a model of (1), but (6) might not have a model while (1) has.

3 Notations

We consider the framework of many-sorted first-order logic with equality, and we assume standard definitions of sorts, signatures and terms. Given a tuple of variables $\boldsymbol{x} \triangleq (x_1, \ldots, x_n)$ and a quantifier $\mathcal{Q}$ ($\forall$ or $\exists$), we shorten $\mathcal{Q}x_1 \ldots \mathcal{Q}x_n.\Phi$ as $\mathcal{Q}\boldsymbol{x}.\Phi$. A formula is in *prenex normal form* if it is written as $\mathcal{Q}_1\boldsymbol{x}_1 \ldots \mathcal{Q}_n\boldsymbol{x}_n.\Phi$ with Φ a quantifier-free formula. A formula is in *Skolem normal form* if it is in prenex normal form with only universal quantifiers. We write $\Phi(\boldsymbol{x})$ to denote that the free variables of Φ are in $\boldsymbol{x}$. Let $\boldsymbol{t} \triangleq (t_1, \ldots, t_n)$ be a term tuple, we write $\Phi(\boldsymbol{t})$ for the formula obtained from Φ by replacing each occurrence of x_i in Φ by t_i. An *interpretation* $\mathcal{I}$ associates a domain to each sort of a signature and a value to each symbol of a formula, and $[\![\Delta]\!]_{\mathcal{I}}$ denotes the evaluation of term Δ over $\mathcal{I}$. A *satisfiability relation* $\models$ between interpretations and formulas is defined inductively as usual. A *model* of Φ is an interpretation $\mathcal{I}$ satisfying $\mathcal{I} \models \Phi$. We sometimes refer to models as "solutions". Formula Ψ *entails* formula Φ, written $\Psi \models \Phi$, if every interpretation satisfying Ψ satisfies Φ as well. Two formulas are equivalent, denoted $\Psi \equiv \Phi$, if they have the same models. A *theory* $\mathcal{T} \triangleq (\Sigma, \boldsymbol{\mathcal{I}})$ restricts symbols in Σ to be interpreted in $\boldsymbol{\mathcal{I}}$. The quantifier-free fragment of $\mathcal{T}$ is denoted QF-$\mathcal{T}$.

Convention. Letters $a, b, c \ldots$ denote uninterpreted symbols and variables. Letters $x, y, z \ldots$ denote quantified variables. $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ denote sets of uninterpreted symbols. $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \ldots$ denote sets of quantified variables. Finally, $\mathtt{a}, \mathtt{b}, \mathtt{c} \ldots$ denote valuations of associated (sets of) symbols.

In the rest of this paper, we assume w.l.o.g. that all formulas are in Skolem normal form. Recall that any formula ϕ in classical logic can be normalized into a formula ψ in Skolem normal form such that any model of ϕ can be lifted into a model of ψ, and vice versa. This strong relation, much closer to formula equivalence than to formula equisatisfiability, ensures that our correctness and completeness results all along the paper hold for arbitrarily quantified formula.

Companion technical report. *Additional technical details (proofs, experiments, etc.) are available online at* `http://benjamin.farinier.org/cav2018/`.

4 Musing with independence

4.1 Independent interpretations, terms and formulas

A solution $(\mathtt{x}, \mathtt{a})$ of Φ does not depend on $\boldsymbol{x}$ if $\Phi(\boldsymbol{x}, \boldsymbol{a})$ is always true or always false, for all possible valuations of $\boldsymbol{x}$ as long as $\boldsymbol{a}$ is set to $\mathtt{a}$. More formally, we define the independence of an interpretation of Φ w.r.t. $\boldsymbol{x}$ as follows:

Definition 1 (Independent interpretation).

- *Let $\Phi(\boldsymbol{x}, \boldsymbol{a})$ a formula with free variables $\boldsymbol{x}$ and $\boldsymbol{a}$. Then an interpretation $\mathcal{I}$ of $\Phi(\boldsymbol{x}, \boldsymbol{a})$ is independent of $\boldsymbol{x}$ if for all interpretations $\mathcal{J}$ equal to $\mathcal{I}$ except on $\boldsymbol{x}$, $\mathcal{I} \models \Phi$ if and only if $\mathcal{J} \models \Phi$.*
- *Let $\Delta(\boldsymbol{x}, \boldsymbol{a})$ a term with free variables $\boldsymbol{x}$ and $\boldsymbol{a}$. Then an interpretation $\mathcal{I}$ of $\Delta(\boldsymbol{x}, \boldsymbol{a})$ is independent of $\boldsymbol{x}$ if for all interpretations $\mathcal{J}$ equal to $\mathcal{I}$ except on $\boldsymbol{x}$, $[\![\Delta(\boldsymbol{x}, \boldsymbol{a})]\!]_{\mathcal{I}} = [\![\Delta(\boldsymbol{x}, \boldsymbol{a})]\!]_{\mathcal{J}}$.*

Regarding formula $ax + b > 0$ from Fig. 1, $\{a = 0, b = 1, x = 1\}$ is independent of x while $\{a = 1, b = 0, x = 1\}$ is not. Considering term $(t\,[a \leftarrow b])\,[c]$, with t an array written at index a then read at index c, $\{a = 0, b = 42, c = 0, t = [\ldots]\}$ is independent of t (evaluates to 42) while $\{a = 0, b = 1, c = 2, t = [\ldots]\}$ is not (evaluates to $t\,[2]$). We now define independence for formulas and terms.

Definition 2 (Independent formula and term).

- *Let $\Phi(\boldsymbol{x}, \boldsymbol{a})$ a formula with free variables $\boldsymbol{x}$ and $\boldsymbol{a}$. Then $\Phi(\boldsymbol{x}, \boldsymbol{a})$ is independent of $\boldsymbol{x}$ if $\forall \boldsymbol{x}. \forall \boldsymbol{y}. (\Phi(\boldsymbol{x}, \boldsymbol{a}) \Leftrightarrow \Phi(\boldsymbol{y}, \boldsymbol{a}))$ is true for any value of $\boldsymbol{a}$.*
- *Let $\Delta(\boldsymbol{x}, \boldsymbol{a})$ a term with free variables $\boldsymbol{x}$ and $\boldsymbol{a}$. Then $\Delta(\boldsymbol{x}, \boldsymbol{a})$ is independent of $\boldsymbol{x}$ if $\forall \boldsymbol{x}. \forall \boldsymbol{y}. (\Delta(\boldsymbol{x}, \boldsymbol{a}) = \Delta(\boldsymbol{y}, \boldsymbol{a}))$ is true for any value of $\boldsymbol{a}$.*

Def. 2 of formula and term independence is far stronger than Def. 1 of interpretation independence. Indeed, it can easily be checked that if a formula Φ (resp. a term Δ) is independent of $\boldsymbol{x}$, then any interpretation of Φ (resp. Δ) is independent of $\boldsymbol{x}$. However, the converse is false as formula $ax + b > 0$ is not independent of x, but has an interpretation $\{a = 0, b = 1, x = 1\}$ which is.

4.2 Independence conditions

Since it is rarely the case that a formula (resp. term) is independent from a set of variables $\boldsymbol{x}$, we are interested in *Sufficient Independence Conditions*. These conditions are additional constraints that can be added to a formula (resp. term) in such a way that they make the formula (resp. term) independent of $\boldsymbol{x}$.

Definition 3 (Sufficient Independence Condition (SIC)).

- *A Sufficient Independence Condition for a formula $\Phi(\boldsymbol{x}, \boldsymbol{a})$ with regard to $\boldsymbol{x}$ is a formula $\Psi(\boldsymbol{a})$ such that $\Psi(\boldsymbol{a}) \models (\forall \boldsymbol{x}. \forall \boldsymbol{y}. \Phi(\boldsymbol{x}, \boldsymbol{a}) \Leftrightarrow \Phi(\boldsymbol{y}, \boldsymbol{a}))$.*
- *A Sufficient Independence Condition for a term $\Delta(\boldsymbol{x}, \boldsymbol{a})$ with regard to $\boldsymbol{x}$, is a formula $\Psi(\boldsymbol{a})$ such that $\Psi(\boldsymbol{a}) \models (\forall \boldsymbol{x}. \forall \boldsymbol{y}. \Delta(\boldsymbol{x}, \boldsymbol{a}) = \Delta(\boldsymbol{y}, \boldsymbol{a}))$.*

We denote by $\textsc{sic}_{\Phi,\boldsymbol{x}}$ (resp. $\textsc{sic}_{\Delta,\boldsymbol{x}}$) a Sufficient Independence Condition for a formula $\Phi(\boldsymbol{x},\boldsymbol{a})$ (resp. for a term $\Delta(\boldsymbol{x},\boldsymbol{a})$) with regard to $\boldsymbol{x}$. For example, $a = 0$ is a $\textsc{sic}_{\Phi,x}$ for formula $\Phi \triangleq ax + b > 0$, and $a = c$ is a $\textsc{sic}_{\Delta,t}$ for term $\Delta \triangleq (t\,[a \leftarrow b])\,[c]$. Note that $\perp$ is always a SIC, and that SIC are closed under $\wedge$ and $\vee$. Prop. 1 clarifies the interest of SIC for model generation.

Proposition 1 (Model generalization). *Let $\Phi(\boldsymbol{x},\boldsymbol{a})$ a formula and Ψ a $\textsc{sic}_{\Phi,\boldsymbol{x}}$. If there exists an interpretation $\{\mathtt{x},\mathtt{a}\}$ such that $\{\mathtt{x},\mathtt{a}\} \models \Psi(\boldsymbol{a}) \wedge \Phi(\boldsymbol{x},\boldsymbol{a})$, then $\{\mathtt{a}\} \models \forall\boldsymbol{x}.\Phi(\boldsymbol{x},\boldsymbol{a})$.*

Proof (sketch of). Appendix C.1 of the companion technical report.

For the sake of completeness, we introduce now the notion of *Weakest Independence Condition* for a formula $\Phi(\boldsymbol{x},\boldsymbol{a})$ with regard to $\boldsymbol{x}$ (resp. a term $\Delta(\boldsymbol{x},\boldsymbol{a})$). We will denote such conditions $\textsc{wic}_{\Phi,\boldsymbol{x}}$ (resp. $\textsc{wic}_{\Delta,\boldsymbol{x}}$).

Definition 4 (Weakest Independence Condition (WIC)).

- *A Weakest Independence Condition for a formula $\Phi(\boldsymbol{x},\boldsymbol{a})$ with regard to $\boldsymbol{x}$ is a $\textsc{sic}_{\Phi,\boldsymbol{x}}$ Π such that, for any other $\textsc{sic}_{\Phi,\boldsymbol{x}}$ Ψ, $\Psi \models \Pi$.*
- *A Weakest Independence Condition for a term $\Delta(\boldsymbol{x},\boldsymbol{a})$ with regard to $\boldsymbol{x}$ is a $\textsc{sic}_{\Delta,\boldsymbol{x}}$ Π such that, for any other $\textsc{sic}_{\Delta,\boldsymbol{x}}$ Ψ, $\Psi \models \Pi$.*

Note that $\Omega \triangleq \forall\boldsymbol{x}.\forall\boldsymbol{y}.\,(\Phi(\boldsymbol{x},\boldsymbol{a}) \Leftrightarrow \Phi(\boldsymbol{y},\boldsymbol{a}))$ is always a $\textsc{wic}_{\Phi,\boldsymbol{x}}$, and any formula Π is a $\textsc{wic}_{\Phi,\boldsymbol{x}}$ if and only if $\Pi \equiv \Omega$. Therefore all syntactically different WIC have the same semantics. As an example, both SIC $a = 0$ and $a = c$ presented earlier are WIC. Prop. 2 emphasizes the interest of WIC for model generation.

Proposition 2 (Model specialization). *Let $\Phi(\boldsymbol{x},\boldsymbol{a})$ a formula and $\Pi(\boldsymbol{a})$ a $\textsc{wic}_{\Phi,\boldsymbol{x}}$. If there exists an interpretation $\{\mathtt{a}\}$ such that $\{\mathtt{a}\} \models \forall\boldsymbol{x}.\Phi(\boldsymbol{x},\boldsymbol{a})$, then $\{\mathtt{x},\mathtt{a}\} \models \Pi(\boldsymbol{a}) \wedge \Phi(\boldsymbol{x},\boldsymbol{a})$ for any valuation $\mathtt{x}$ of $\boldsymbol{x}$.*

Proof (sketch of). Appendix C.2 of the companion technical report.

From now on, our goal is to infer from a formula $\forall\boldsymbol{x}.\Phi(\boldsymbol{x},\boldsymbol{a})$ a $\textsc{sic}_{\Phi,\boldsymbol{x}}$ $\Psi(\boldsymbol{a})$, find a model for $\Psi(\boldsymbol{a}) \wedge \Phi(\boldsymbol{x},\boldsymbol{a})$ and generalize it. This $\textsc{sic}_{\Phi,\boldsymbol{x}}$ should be as weak — in the sense “less coercive” — as possible, as otherwise $\perp$ could always be used, which would not be very interesting for our overall purpose.

For the sake of simplicity, previous definitions omit to mention the theory to which the SIC belongs. If the theory $\mathcal{T}$ of the quantified formula is decidable we can always choose $\forall\boldsymbol{x}.\forall\boldsymbol{y}.\,(\Phi(\boldsymbol{x},\boldsymbol{a}) \Leftrightarrow \Phi(\boldsymbol{y},\boldsymbol{a}))$ as a SIC, but it is simpler to directly use a $\mathcal{T}$-solver. *The challenge is, for formulas in an undecidable theory $\mathcal{T}$, to find a non-trivial* SIC *in its quantifier-free fragment* QF-$\mathcal{T}$.

Under this constraint, we cannot expect a systematic construction of WIC, as it would allow to decide the satisfiability of any quantified theory with a decidable quantifier-free fragment. Yet informally, the closer a SIC is to be a WIC, the closer our approach is to completeness. Therefore this notion might be seen as a fair gauge of the quality of a SIC. *Having said that, we leave a deeper study on the inference of* WIC *as future work.*

5 Generic framework for SIC-based model generation

We describe now our overall approach. Alg. 1 presents our SIC-based generic framework for model generation (Sec. 5.1). Then, Alg. 2 proposes a taint-based approach for SIC inference (Sec. 5.2). Finally, we discuss complexity and efficiency issues (Sec. 5.3) and detail extensions (Sec. 5.4), such as partial elimination.

From now on, we do not distinguish anymore between terms and formulas, their treatment being symmetric, and we call targeted variables the variables we want to be independent of.

5.1 SIC-based model generation

Algorithm 1: SIC-based model generation for quantified formulas

```
Parameter: solveQF
    Input: $\Phi(v)$ a formula in QF-$\mathcal{T}$
    Output: SAT $(\mathtt{v})$ with $\mathtt{v} \models \Phi$, UNSAT or UNKNOWN
Parameter: inferSIC
    Input: $\Phi$ a formula in QF-$\mathcal{T}$, and $\boldsymbol{x}$ a set of targeted variables
    Output: $\Psi$ a SIC$_{\Phi,\boldsymbol{x}}$ in QF-$\mathcal{T}$
Function solveQ:
    Input: $\forall \boldsymbol{x}.\Phi(\boldsymbol{x}, \boldsymbol{a})$ a universally quantified formula over theory $\mathcal{T}$
    Output: SAT $(\mathtt{a})$ with $\mathtt{a} \models \forall \boldsymbol{x}.\Phi(\boldsymbol{x}, \boldsymbol{a})$, UNSAT or UNKNOWN
    Let $\Psi(\boldsymbol{a}) \triangleq \mathtt{inferSIC}(\Phi(\boldsymbol{x}, \boldsymbol{a}), \boldsymbol{x})$
    match solveQF $(\Phi(\boldsymbol{x}, \boldsymbol{a}) \wedge \Psi(\boldsymbol{a}))$
        with SAT $(\mathtt{x}, \mathtt{a})$ return SAT $(\mathtt{a})$
        with UNSAT
            if $\Psi$ is a WIC$_{\Phi,\boldsymbol{x}}$ then return UNSAT
            else return UNKNOWN
        with UNKNOWN return UNKNOWN
```

Our model generation technique is described in Alg. 1. Function `solveQ` takes as input a formula $\forall \boldsymbol{x}.\Phi(\boldsymbol{x}, \boldsymbol{a})$ over a theory $\mathcal{T}$. It first calculates a SIC$_{\Phi,\boldsymbol{x}}$ $\Psi(\boldsymbol{a})$ in QF-$\mathcal{T}$. Then it solves $\Phi(\boldsymbol{x}, \boldsymbol{a}) \wedge \Psi(\boldsymbol{a})$. Finally, depending on the result and whether $\Psi(\boldsymbol{a})$ is a WIC$_{\Phi,\boldsymbol{x}}$ or not, it answers SAT, UNSAT or UNKNOWN. `solveQ` is parametrized by two functions `solveQF` and `inferSIC`:

`solveQF` is a decision procedure (typically a SMT solver) for QF-$\mathcal{T}$. `solveQF` is said to be *correct* if each time it answers SAT (resp. UNSAT) the formula is satisfiable (resp. unsatisfiable); it is said to be *complete* if it always answers SAT or UNSAT, never UNKNOWN.

`inferSIC` takes as input a formula Φ in QF-$\mathcal{T}$ and a set of targeted variables $\boldsymbol{x}$, and produces a SIC$_{\Phi,\boldsymbol{x}}$ in QF-$\mathcal{T}$. It is said to be *correct* if it always returns a SIC, and *complete* if all the SIC it returns are WIC. A possible implementation of `inferSIC` is described in Alg. 2 (Sec. 5.2).

Function `solveQ` enjoys the two following properties, where correctness and completeness are defined as for `solveQF`.

Theorem 1 (Correctness and completeness).

- *If* `solveQF` *and* `inferSIC` *are correct, then* `solveQ` *is correct.*
- *If* `solveQF` *and* `inferSIC` *are complete, then* `solveQ` *is complete.*

Proof (sketch of). Follow directly from Prop. 1 and 2 (Sec. 4.2).

5.2 Taint-based SIC inference

```
Algorithm 2: Taint-based SIC inference
Parameter: theorySIC
    Input: $f$ a function symbol, its parameters $\phi_i$, $\boldsymbol{x}$ a set of targeted variables
           and $\psi_i$ their associated $\textsc{sic}_{\phi_i,\boldsymbol{x}}$
    Output: $\Psi$ a $\textsc{sic}_{f(\phi_i),\boldsymbol{x}}$
    Default: Return $\bot$
Function inferSIC($\Phi$,$\boldsymbol{x}$):
    Input: $\Phi$ a formula and $\boldsymbol{x}$ a set of targeted variables
    Output: $\Psi$ a $\textsc{sic}_{\Phi,\boldsymbol{x}}$
    either $\Phi$ is a constant  return $\top$
    either $\Phi$ is a variable $v$  return $v \notin \boldsymbol{x}$
    either  $\Phi$ is a function $f\,(\phi_1, . , \phi_n)$
        Let $\psi_i \triangleq$ inferSIC $(\phi_i, \boldsymbol{x})$ for all $i \in \{1, . , n\}$
        Let $\Psi \ \triangleq$ theorySIC $(f, (\phi_1,., \phi_n)\,, (\psi_1,., \psi_n)\,, \boldsymbol{x})$
        return $\Psi \vee \bigwedge_i \psi_i$
```

Alg. 2 presents a taint-based implementation of function `inferSIC`. It consists of a (syntactic) core calculus described here, refined by a (semantic) theory-dependent calculus `theorySIC` described in Sec. 6. From formula $\Phi\,(\boldsymbol{x}, \boldsymbol{a})$ and targeted variables $\boldsymbol{x}$, `inferSIC` is defined recursively as follow.

If Φ is a constant it returns $\top$ as constants are independent of any variable. If Φ is a variable v, it returns $\top$ if we may depend on v (i.e., $v \notin \boldsymbol{x}$), $\bot$ otherwise. If Φ is a function $f\,(\phi_1, . , \phi_n)$, it first recursively computes for every sub-term ϕ_i a $\textsc{sic}_{\phi_i,\boldsymbol{x}}$ ψ_i. Then these results are sent with Φ to `theorySIC` which computes a $\textsc{sic}_{\Phi,\boldsymbol{x}}$ Ψ. The procedure returns the disjunction between Ψ and the conjunction of the ψ_i's. Note that `theorySIC` default value $\bot$ is absorbed by the disjunction.

The intuition is that if the ϕ_i's are independent of $\boldsymbol{x}$, then $f\,(\phi_1, . , \phi_n)$ is. Therefore Alg. 2 is said to be *taint-based* as, when `theorySIC` is left to its default value, it acts as a form of taint tracking [15,27] inside the formula.

Proposition 3 (Correctness). *Given a formula $\Phi\,(\boldsymbol{x}, \boldsymbol{a})$ and assuming that* `theorySIC` *is correct, then* `inferSIC`*$(\Phi, \boldsymbol{x})$ indeed computes a $\textsc{sic}_{\Phi,\boldsymbol{x}}$.*

Proof (sketch of). This proof has been mechanized in Coq[3].

Note that on the other hand, completeness does not hold: in general `inferSIC` does not compute a WIC, cf. discussion in Sec. 5.4.

[3] `http://benjamin.farinier.org/cav2018/`

5.3 Complexity and efficiency

We now evaluate the overhead induced by Alg. 1 in terms of formula size and complexity of the resolution — the running time of Alg. 1 itself being expected to be negligible (preprocessing).

Definition 5. *The size of a term is inductively defined as* $\mathit{size}(x) \triangleq 1$ *for* x *a variable, and* $\mathit{size}(f\,(t_1,.\,,t_n)) \triangleq 1 + \Sigma_i\,\mathit{size}(t_i)$ *otherwise. We say that* `theorySIC` *is bounded in size if there exists* K *such that, for all terms* Δ, $\mathit{size}(\mathtt{theorySIC}(\Delta,\cdot)) \leq K$.

Proposition 4 (Size bound). *Let* N *be the maximal arity of symbols defined by theory* $\mathcal{T}$*. If* `theorySIC` *is bounded in size by* K*, then for all formula* Φ *in* $\mathcal{T}$, $\mathit{size}(\mathtt{inferSIC}(\Phi,\cdot)) \leq (K+N)\cdot \mathit{size}(\Phi)$.

Proposition 5 (Complexity bound). *Let us suppose* `theorySIC` *bounded in size, and let* Φ *be a formula belonging to a theory* $\mathcal{T}$ *with polynomial-time checkable solutions. If* Ψ *is a* SIC$_{\Phi,\cdot}$ *produced by* `inferSIC`*, then a solution for* $\Phi \wedge \Psi$ *is checkable in time polynomial in size of* Φ.

Proof (sketch of). Appendices C.3 and C.4 of the companion technical report.

These propositions demonstrate that, for formula landing in complex enough theories, our method lifts QF-solvers to the quantified case (in an approximated way) without any significant overhead, as long as `theorySIC` is bounded in size. This latter constraint can be achieved by systematically binding sub-terms to (constant-size) fresh names and having `theorySIC` manipulates these binders.

5.4 Discussions

Extension. Let us remark that our framework encompasses partial quantifier elimination as long as the remaining quantifiers are handled by `solveQF`. For example, we may want to remove quantifications over arrays but keep those on bitvectors. In this setting, `inferSIC` can also allow some level of quantification, providing that `solveQF` handles them.

About WIC. As already stated, `inferSIC` does not propagate WIC in general. For example, considering formulas $t_1 \triangleq (x < 0)$ and $t_2 \triangleq (x \geq 0)$, then WIC$_{t_1,x} = \bot$ and WIC$_{t_2,x} = \bot$. Hence `inferSIC` returns $\bot$ as SIC for $t_1 \vee t_2$, while actually WIC$_{t_1 \vee t_2,x} = \top$.

Nevertheless, we can already highlight a few cases where WIC can be computed. (1) `inferSIC` does propagate WIC on one-to-one uninterpreted functions. (2) If no variable of $\boldsymbol{x}$ appears in any sub-term of $f(t,t')$, then the associated WIC is $\top$. While a priori naive, this case becomes interesting when combined with simplifications (Sec. 7.1) that may eliminate $\boldsymbol{x}$. (3) If a sub-term falls in a sub-theory admitting quantifier elimination, then the associated WIC is computed by eliminating quantifiers in $(\forall.\boldsymbol{x}.\boldsymbol{y}.\Phi(\boldsymbol{x},\boldsymbol{a}) \Leftrightarrow \Phi(\boldsymbol{y},\boldsymbol{a}))$. (4) We may also think of dedicated patterns: regarding bitvectors, the WIC for $x \leq a \Rightarrow x \leq x+k$ is $a \leq \mathtt{Max} - k$. *Identifying under which condition* WIC *propagation holds is a strong direction for future work.*

6 Theory-dependent SIC refinements

We now present theory-dependent SIC refinements for theories relevant to program analysis: booleans, fixed-size bitvectors and arrays — recall that uninterpreted functions are already handled by Alg. 2. We then propose a generalization of these refinements together with a correctness proof for a larger class of operators.

6.1 Refinement on theories

We recall `theorySIC` takes four parameters: a function symbol f, its arguments $(t_1, ., t_n)$, their associated SIC $(t_1^\bullet, ., t_n^\bullet)$, and targeted variables $\boldsymbol{x}$. `theorySIC` pattern-matches the function symbol and returns the associated SIC according to rules in Fig. 2. If a function symbol is not supported, we return the default value $\bot$. Constants and variables are handled by `inferSIC`. For the sake of simplicity, rules in Fig. 2 are defined recursively, but can easily fit the interface required for `theorySIC` in Alg. 2 by turning recursive calls into parameters.

Booleans and ite. Rules for the boolean theory (Fig. 2a) handles $\Rightarrow$, $\wedge$, $\vee$ and ite (if-then-else). For binary operators, the SIC is the conjunction of the SIC associated to one of the two sub-terms and a constraint on this sub-term that forces the result of the operator to be constant — e.g., to be equal to $\bot$ (resp. $\top$) for the antecedent (resp. consequent) of an implication. These equality constraints are based on absorbing elements of operators.

Inference for the ite operator is more subtle. Intuitively, if its condition is independent of some $\boldsymbol{x}$, we use it to select the $\text{SIC}_{\boldsymbol{x}}$ of the sub-term that will be selected by the ite operator. If the condition is dependent of $\boldsymbol{x}$, then we cannot use it anymore to select a $\text{SIC}_{\boldsymbol{x}}$. In this case, we return the conjunction of the $\text{SIC}_{\boldsymbol{x}}$ of both sub-terms and the constraint that the two sub-terms are equal.

$$\begin{aligned}
(a \Rightarrow b)^\bullet &\triangleq (a^\bullet \wedge a = \bot) \vee (b^\bullet \wedge b = \top)\\
(a \wedge b)^\bullet &\triangleq (a^\bullet \wedge a = \bot) \vee (b^\bullet \wedge b = \bot)\\
(a \vee b)^\bullet &\triangleq (a^\bullet \wedge a = \top) \vee (b^\bullet \wedge b = \top)\\
(\text{ite}\, c\, a\, b)^\bullet &\triangleq (c^\bullet \wedge \text{ite}\, c\, a^\bullet\, b^\bullet) \vee (a^\bullet \wedge b^\bullet \wedge a = b)
\end{aligned}$$

(a) Booleans and ite

$$\begin{aligned}
(a_n \wedge b_n)^\bullet &\triangleq (a_n^\bullet \wedge a_n = 0_n) \vee (b_n^\bullet \wedge b_n = 0_n)\\
(a_n \vee b_n)^\bullet &\triangleq (a_n^\bullet \wedge a_n = 1_n) \vee (b_n^\bullet \wedge b_n = 1_n)\\
(a_n \times b_n)^\bullet &\triangleq (a_n^\bullet \wedge a_n = 0_n) \vee (b_n^\bullet \wedge b_n = 0_n)\\
(a_n \ll b_n)^\bullet &\triangleq (b_n^\bullet \wedge b_n \geq n)
\end{aligned}$$

(b) Fixed-size bitvectors

$$\begin{aligned}
(\text{select }(\text{store}\, a\, i\, e)\ j)^\bullet &\triangleq (\text{ite }(i = j)\ e\ (\text{select}\, a\, j))^\bullet\\
&\triangleq ((i = j)^\bullet \wedge (\text{ite }(i = j)\ e^\bullet\ (\text{select}\, a\, j)^\bullet)) \vee (e^\bullet \wedge (\text{select}\, a\, j)^\bullet \wedge (e = \text{select}\, a\, j))\\
&\triangleq (i^\bullet \wedge j^\bullet \wedge (\text{ite }(i = j)\ e^\bullet\ (\text{select}\, a\, j)^\bullet)) \vee (e^\bullet \wedge (\text{select}\, a\, j)^\bullet \wedge (e = \text{select}\, a\, j))
\end{aligned}$$

(c) Arrays

Fig. 2: Examples of refinements for `theorySIC`

Bitvectors and arrays. Rules for bitvectors (Fig. 2b) follow similar ideas, with constant $\top$ (resp. $\bot$) substituted by 1_n (resp. 0_n), the bitvector of size n full of ones (resp. zeros). Rules for arrays (Fig. 2c) are derived from the theory axioms. The definition is recursive: rules need be applied until reaching either a store at the position where the select occurs, or the initial array variable.

As a rule of thumb, good SIC can be derived from function axioms in the form of rewriting rules, as done for arrays. Similar constructions can be obtained for example for stacks or queues.

6.2 $\mathcal{R}$-absorbing functions

We propose a generalization of the previous theory-dependent SIC refinements to a larger class of functions, and prove its correctness.

Intuitively, if a function has an absorbing element, constraining one of its operands to be equal to this element will ensure that the result of the function is independent of the other operands. However, it is not enough when a relation between some elements is needed, such as with $(t[a \leftarrow b])\,[c]$ where constraint $a = c$ ensures the independence with regards to t. We thus generalize the notion of absorption to $\mathcal{R}$-absorption, where $\mathcal{R}$ is a relation between function arguments.

Definition 6. *Let $f : \tau_1 \times \cdots \times \tau_n \to \tau$ a function. f is $\mathcal{R}$-absorbing if there exists $\mathcal{I}_\mathcal{R} \subset \{1, \cdots, n\}$ and $\mathcal{R}$ a relation between $\alpha_i : \tau_i$, $i \in \mathcal{I}_\mathcal{R}$ such that, for all $b \triangleq (b_1, \ldots, b_n)$ and $c \triangleq (c_1, \ldots, c_n) \in \tau_1 \times \cdots \times \tau_n$, if $\mathcal{R}(b|_{\mathcal{I}_\mathcal{R}})$ and $b|_{\mathcal{I}_\mathcal{R}} = c|_{\mathcal{I}_\mathcal{R}}$ where $\cdot|_{\mathcal{I}_\mathcal{R}}$ is the projection on $\mathcal{I}_\mathcal{R}$, then $f(b) = f(c)$.*

$\mathcal{I}_\mathcal{R}$ is called the support of the relation of absorption $\mathcal{R}$.

For example, $(a, b) \mapsto a \vee b$ has two pairs $\langle \mathcal{R}, \mathcal{I}_\mathcal{R} \rangle$ coinciding with the usual notion of absorption, $\langle a{=}\top,\ \{1_a\}\rangle$ and $\langle b{=}\top,\ \{2_b\}\rangle$. Function $(x, y, z) \mapsto xy + z$ has among others the pair $\langle x{=}0,\ \{1_x, 3_z\}\rangle$, while $(a, b, c, t) \mapsto (t[a \leftarrow b])\,[c]$ has the pair $\langle a{=}c,\ \{1_a, 3_c\}\rangle$. We can now state the following proposition:

Proposition 6. *Let $f\,(t_1, \ldots, t_n)$ be a $\mathcal{R}$-absorbing function of support $\mathcal{I}_\mathcal{R}$, and let $t_i^\bullet$ be a $\text{SIC}_{t_i,\boldsymbol{x}}$ for some $\boldsymbol{x}$. Then $\mathcal{R}\,(t_{i\in\mathcal{I}_\mathcal{R}}) \bigwedge_{i\in\mathcal{I}_\mathcal{R}} t_i^\bullet$ is a $\text{SIC}_{f,\boldsymbol{x}}$.*

Proof (sketch of). Appendix C.5 of the companion technical report.

Previous examples (Sec. 6.1) can be recast in term of $\mathcal{R}$-absorbing function, proving their correctness (cf. companion technical report). Note that regarding our end-goal, we should accept only $\mathcal{R}$-absorbing functions in QF-$\mathcal{T}$.

7 Experimental evaluation

This section describes the implementation of our method (Sec. 7.1) for bitvectors and arrays (ABV), together with experimental evaluation (Sec. 7.2).

7.1 Implementation

Our prototype TFML (*Taint engine for ForMuLa*)[4] comprises 7 klocs of OCaml. Given an input formula in the SMT-LIB format [5] (ABV theory), TFML performs several normalizations before adding taint information following Alg. 1. The process ends with simplifications as taint usually introduces many constant values, and a new SMT-LIB formula is output.

Sharing with let-binding. This stage is crucial as it allows to avoid term duplication in `theorySIC` (Alg. 2, Sec. 5.3, and Prop. 4). We introduce new names for relevant sub-terms in order to easily share them.

Simplifications. We perform constant propagation and rewriting (standard rules, e.g. $x - x \mapsto 0$ or $x \times 1 \mapsto x$) on both initial and transformed formulas – equality is soundly approximated by syntactic equality.

Shadow arrays. We encode taint constraints over arrays through *shadow arrays*. For each array declared in the formula, we declare a (taint) shadow array. The default value for all cells of the shadow array is the taint of the original array, and for each value stored (resp. read) in the original array, we store (resp. read) the taint of the value in the shadow array. As logical arrays are infinite, we cannot constrain all the values contained in the initial shadow array. Instead, we rely on a common trick in array theory: we constrain only cells corresponding to a relevant read index in the formula.

Iterative skolemization. While we have supposed along the paper to work on skolemized formulas, we have to be more careful in practice. Indeed, skolemization introduce dependencies between a skolemized variable and all its preceding universally quantified variables, blurring our analysis and likely resulting in considering the whole formula as dependent. Instead, we follow an iterative process: 1. Skolemize the first block of existentially quantified variables; 2. Compute the independence condition for any targeted variable in the first block of universal quantifiers and remove these quantifiers; 3. Repeat. This results in full Skolemization together with the construction of an independence condition, while avoiding many unnecessary dependencies.

7.2 Evaluation

Objective. We experimentally evaluate the following research questions: *RQ1* How does our approach perform with regard to state-of-the-art approaches for model generation of quantified formulas? *RQ2* How effective is it at lifting quantifier-free solvers into (SAT-only) quantified solvers? *RQ3* How efficient is it in terms of preprocessing time and formula size overhead? We evaluate our method on a set of formulas combining arrays and bitvectors (paramount in software verification), against state-of-the-art solvers for these theories.

Protocol. The experimental setup below runs on an Intel(R) Xeon(R) E5-2660 v3 @ 2.60GHz, 4GB RAM per process, and a TIMEOUT of 1000s per formula.

[4] `http://benjamin.farinier.org/cav2018/`

Table 1: Answers and resolution time (in seconds, include TIMEOUT)

			Boolector•	CVC4	CVC4•	CVC4$_E$	CVC4$_E$•	Z3	Z3•	Z3$_E$	Z3$_E$•
SMT-LIB	#	SAT	**399**	84	242	84	242	261	366	87	366
		UNSAT	N/A	0	N/A	0	N/A	165	N/A	0	N/A
		UNKNOWN	870	1185	1027	1185	1027	843	903	1182	903
	total time		**349**	165	194 667	165	196 934	270 150	36 480	192	41 935
Binsec	#	SAT	**1042**	951	954	951	954	953	**1042**	953	**1042**
		UNSAT	N/A	62	N/A	62	N/A	319	N/A	62	N/A
		UNKNOWN	379	408	467	408	467	149	379	406	379
	total time		**1152**	64 761	76 811	64 772	77 009	30 235	11 415	135	11 604

solver•: solver enhanced with our method   Z3$_E$, CVC4$_E$: essentially E-matching

Metrics For *RQ1* we compare the number of SAT and UNKNOWN answers between solvers supporting quantification, with and without our approach. For *RQ2*, we compare the number of SAT and UNKNOWN answers between quantifier-free solvers enhanced by our approach and solvers supporting quantification. For *RQ3*, we measure preprocessing time and formulas size overhead.

Benchmarks We consider two sets of ABV formulas. First, a set of 1421 formulas from (a modified version of) the symbolic execution tool Binsec [12] representing quantified reachability queries (cf. Sec. 2) over Binsec benchmark programs (security challenges, e.g. `crackme` or vulnerability finding). The initial (array) memory is quantified so that models depend only on user input. Second, a set of 1269 ABV formulas generated from formulas of the QF-ABV category of SMT-LIB [5] – sub-categories `brummayerbiere`, `dwp formulas` and `klee selected`. The generation process consists in universally quantifying some of the initial array variables, mimicking quantified reachability problems.

Competitors For *RQ1*, we compete against the two state-of-the-art SMT solvers for quantified formulas CVC4 [4] (finite model instantiation [31]) and Z3 [14] (model-based instantiation [20]). We also consider degraded versions CVC4$_E$ and Z3$_E$ that roughly represent standard E-matching [16]. For *RQ2* we use Boolector [10], one of the very best QF-ABV solvers.

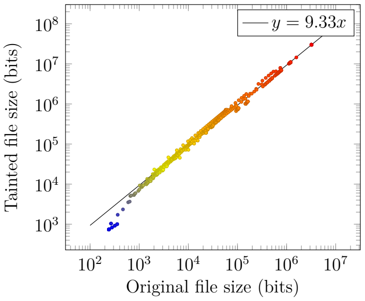
Table 2: Complementarity of our approach with existing solvers (SAT instances)

		CVC4•	Z3•	Boolector•
SMT-LIB	CVC4	-10 +168 [252]		-10 +325 [409]
	Z3		-119 +224 [485]	-86 +224 [485]
Binsec	CVC4	-25 +28 [979]		-25 +116 [1067]
	Z3		-25 +114 [1067]	-25 +114 [1067]

Results. Tables 1 and 2 and Fig. 3 sum up our experimental results, which have all been cross-checked for consistency. Table 1 reports the number of successes (SAT or UNSAT) and failures (UNKNOWN), plus total solving times. The • sign indicates formulas preprocessed with our approach. In that case it is impossible to correctly answer UNSAT (no WIC checking), the UNSAT line is thus N/A. Since Boolector does not support quantified ABV formulas, we only give results with our approach enabled. Table 1 reads as follow: of the 1269 SMT-LIB formulas, standalone Z3 solves 426 formulas (261 SAT, 165 UNSAT), and 366 (all SAT) if preprocessed. Interestingly, our approach always improves the underlying solver in terms of solved (SAT) instances, either in a significant way (SMT-LIB) or in a modest way (BINSEC). Yet, recall that in a software verification setting every win matters (possibly new bug found or new assertion proved). For Z3●, it also strongly reduces computation time. Last but not least, Boolector● (a pure QF-solver) turns out to have the best performance on SAT-instances, beating state-of-the-art approaches both in terms of solved instances and computation time.

Table 2 substantiates the complementarity of the different methods, and reads as follow: for SMT-LIB, Boolector● solves 224 (SAT) formulas missed by Z3, while Z3 solves 86 (SAT) formulas missed by Boolector●, and 485 (SAT) formulas are solved by either one of them.

Fig. 3 shows formula size averaging a 9-fold increase (min 3, max 12): yet they are easier to solve because they are more constrained. Regarding performance and overhead of the tainting process, *taint time is almost always less than 1s* in our experiments (not shown here), 4min for worst case, clearly dominated by resolution time. The worst case is due to a pass of linearithmic complexity which can be optimized to be logarithmic.



Maximal size ratio	12.48
Minimal size ratio	2.81
Average size ratio	8.73
Standard deviation	0.78

Fig. 3: Overhead in formula size

Pearls. We show hereafter two particular applications of our method. Table 3 reports results of another symbolic execution experiment, on the `grub` example. On this example, Boolector● completely outperforms existing approaches. As a second application, while the main drawback of our method is that it precludes proving UNSAT, this is easily mitigated by complementing the approach with another one geared (or able) to proving UNSAT, yielding efficient solvers for quantified formulas, as shown in Table 4.

Table 3: GRUB example

		Boolector●	Z3
#	SAT	**540**	1
	UNSAT	N/A	42
	UNKNOWN	355	852
total time		**16 732**	159 765

Conclusion. Experiments demonstrate the relevance of our taint-based technique for model generation. (*RQ1*) Results in Table 1 shows that our approach greatly facilitates the resolution process. *On these examples, our method performs better than state-of-the-art solvers but also strongly complements them (Table 2).* (*RQ2*) Moreover, Table 1 demonstrates that our technique is highly effective at lifting quantifier-free solvers to quantified formulas, in both number of SAT answers and computation time. *Indeed, once lifted, Boolector performs better (for* SAT*-only) than Z3 or CVC4 with full quantifier support.* Finally (*RQ3*) our tainting method itself is very efficient both in time and space, making it perfect either for a preprocessing step or for a deeper integration into a solver. In our current prototype implementation, we consider the cost to be low. *The companion technical report contains a few additional experiments on bitvectors and integer arithmetic, including the example from Fig. 1.*

Table 4: Best approaches

		former	new	
		Z3	B•	B• ▷ Z3
SMT-LIB	SAT	261	399	485
	UNSAT	165	N/A	165
	UNKNOWN	843	870	619
	time	270 150	350	94 610
BINSEC	SAT	953	1042	1067
	UNSAT	319	N/A	319
	UNKNOWN	149	379	35
	time	64 761	1 152	1 169

8 Related work

Traditional approaches to solving quantified formulas essentially involve either generic methods geared to proving unsatisfiability and validity [16], or complete but dedicated approaches for particular theories [8,36]. Besides, some recent methods [22,20,31] aim to be correct and complete for larger classes of theories.

Generic method for unsatisfiability. Broadly speaking, these methods iteratively instantiate axioms until a contradiction is found. They are generic w.r.t. the underlying theory and allow to reuse standard theory solvers, but termination is not guaranteed. Also, they are more suited to prove unsatisfiability than to find models. In this family, E-matching [16,13] shows reasonable cost when combined with conflict-based instantiation [30] or semantic triggers [17,18]. In pure first-order logic (without theories), quantifiers are mainly handled through resolution and superposition [1,26] as done in Vampire [33,24] and E [34].

Complete methods for specific theories. Much work has been done on designing complete decision procedures for quantified theories of interest, notably array properties [8], quantified theory of bitvectors [36,23], Presburger arithmetic or Real Linear Arithmetic [9,19]. Yet, they usually come at a high cost.

Generic methods for model generation. Some recent works detail attempts at more general approaches to model generation.

Local theory extensions [22,2] provide means to extend some decidable theories with free symbols and quantifications, retaining decidability. The approach identifies specific forms of formulas and quantifications (bounded), such that these theory extensions can be solved using finite instantiation of quantifiers together with a decision procedure for the original theory. The main drawback is that the formula size can increase a lot.

Model-based quantifier instantiation is an active area of research notably developed in Z3 and CVC4. The basic line is to consider the partial model under construction in order to find the right quantifier instantiations, typically in a try-and-refine manner. Depending on the variants, these methods favors either satisfiability or unsatisfiability. They build on the underlying quantifier-free solver and can be mixed with E-matching techniques, yet each refinement yields a solver call and the refinement process may not terminate. Ge and de Moura [20] study decidable fragments of first-order logic modulo theories for which model-based quantifier instantiation yields soundness and refutational completeness. Reynolds *et al.* [30], Barbosa [3] and Preiner *et al.* [28] use models to guide the instantiation process towards instances refuting the current model. *Finite model quantifier instantiation* [31,32] reduces the search to finite models, and is indeed geared toward model generation rather than unsatisfiability. Similar techniques have been used in program synthesis [29].

We drop support for the unsatisfiable case but get more flexibility: we deal with quantifiers on any sort, the approach terminates and is lightweight, in the sense that it requires a single call to the underlying quantifier-free solver.

Other. Our method can be seen as taking inspiration from program taint analysis [15,27] developed for checking the non-interference [35] of public and secrete input in security-sensitive programs. As far as the analogy goes, our approach should not be seen as checking non-interference, but rather as inferring preconditions of non-interference. Moreover, our formula-tainting technique is closer to dynamic program-tainting than to static program-tainting, in the sense that precise dependency conditions are statically inserted at preprocess-time, then precisely explored at solving-time.

Finally, Darvas *et al.* [11] presents a bottom-up formula strengthening method. Their goal differ from ours, as they are interested in formula well-definedness (rather than independence) and validity (rather than model generation).

9 Conclusion

This paper addresses the problem of generating models of quantified first-order formulas over built-in theories. We propose a correct and generic approach based on a reduction to the quantifier-free case through the inference of independence conditions. The technique is applicable to any theory with a decidable quantifier-free case and allows to reuse all the work done on quantifier-free solvers. The method significantly enhances the performances of state-of-the-art SMT solvers for the quantified case, and supplements the latest advances in the field.

Future developments aim to tackle the definition of more precise inference mechanisms of independence conditions, the identification of interesting subclasses for which inferring weakest independence conditions is feasible, and the combination with other quantifier instantiation techniques.

References

1. L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *J. Log. Comput.*, 4(3):217–247, 1994.
2. K. Bansal, A. Reynolds, T. King, C. W. Barrett, and T. Wies. Deciding Local Theory Extensions via E-matching. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 87–105, 2015.
3. H. Barbosa. Efficient Instantiation Techniques in SMT (work in progress). In *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning co-located with International Joint Conference on Automated Reasoning (IJCAR 2016), Coimbra, Portugal, July 2nd, 2016.*, pages 1–10, 2016.
4. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, 2011.
5. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.
6. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 825–885. 2009.
7. A. Biere. Bounded Model Checking. In *Handbook of Satisfiability*, pages 457–481. 2009.
8. A. R. Bradley, Z. Manna, and H. B. Sipma. What's Decidable About Arrays? In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, pages 427–442, 2006.
9. A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. Beyond Quantifier-Free Interpolation in Extensions of Presburger Arithmetic. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 88–102, 2011.
10. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 174–177, 2009.
11. Á. Darvas, F. Mehta, and A. Rudich. Efficient Well-Definedness Checking. In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, pages 100–115, 2008.
12. R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 653–656, 2016.
13. L. M. de Moura and N. Bjørner. Efficient E-Matching for SMT Solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, pages 183–198, 2007.
14. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and*

Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, pages 337–340, 2008.

15. D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20(7):504–513, 1977.
16. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
17. C. Dross, S. Conchon, J. Kanig, and A. Paskevich. Reasoning with Triggers. In *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, pages 22–31, 2012.
18. C. Dross, S. Conchon, J. Kanig, and A. Paskevich. Adding Decision Procedures to SMT Solvers Using Axioms with Triggers. *J. Autom. Reasoning*, 56(4):387–457, 2016.
19. A. Farzan and Z. Kincaid. Linear Arithmetic Satisfiability via Strategy Improvement. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 735–743, 2016.
20. Y. Ge and L. M. de Moura. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 306–320, 2009.
21. P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *ACM Queue*, 10(1):20, 2012.
22. C. Ihlemann, S. Jacobs, and V. Sofronie-Stokkermans. On Local Reasoning in Verification. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 265–281, 2008.
23. M. Jonás and J. Strejcek. Solving Quantified Bit-Vector Formulas Using Binary Decision Diagrams. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 267–283, 2016.
24. L. Kovács and A. Voronkov. First-Order Theorem Proving and Vampire. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 1–35, 2013.
25. D. Kroening and O. Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.
26. R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 371–443. 2001.
27. P. Ørbæk. Can you Trust your Data? In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, pages 575–589, 1995.
28. M. Preiner, A. Niemetz, and A. Biere. Counterexample-Guided Model Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 264–280, 2017.
29. A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 198–216, 2015.

30. A. Reynolds, C. Tinelli, and L. M. de Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 195–202, 2014.
31. A. Reynolds, C. Tinelli, A. Goel, and S. Krstic. Finite Model Finding in SMT. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 640–655, 2013.
32. A. Reynolds, C. Tinelli, A. Goel, S. Krstic, M. Deters, and C. Barrett. Quantifier Instantiation Techniques for Finite Model Finding in SMT. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pages 377–391, 2013.
33. A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2-3):91–110, 2002.
34. S. Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2-3):111–126, 2002.
35. G. Smith. Principles of Secure Information Flow Analysis. In *Malware Detection*, pages 291–307. 2007.
36. C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura. Efficiently solving quantified bit-vector formulas. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 239–246, 2010.