Preprocessing Techniques for First-Order Clausification

Krystof Hoder
Computer Science Department
University of Manchester, UK
hoderk@cs.man.ac.uk

Zurab Khasidashvili
Intel Israel (74) Ltd.
Haifa 31015, Israel
zurabk@iil.intel.com

Konstantin Korovin, Andrei Voronkov
Computer Science Department
University of Manchester, UK
korovin@cs.man.ac.uk, andrei@voronkov.com

***Abstract*—It is well known that preprocessing is crucial for efficient reasoning on large industrial problems. Although preprocessing is well developed for propositional logic, it is much less investigated for first-order logic. In this paper we introduce several preprocessing techniques for simplifying first-order formulas aimed at improving clausification. These include definition inlining and merging, simplifications based on a new data structure, quantified AIG, and its combination with BDDs. We implemented our preprocessing methods and evaluated them over encodings of industrial hardware verification problems into the effectively propositional (EPR) fragment of first-order logic and over standard first-order (TPTP) and SMT (SMT-LIB) benchmarks. We also investigated preprocessing methods that help obtain EPR-resulting clausification in cases where standard clausification would lead outside the EPR fragment. We demonstrate that our methods enable one to considerably reduce the number of clauses obtained after clausification and by that help speedup first-order reasoning.**

I. Introduction

First-order logic solvers are increasingly used in industrial verification applications. These uses include model checking of large real-life hardware systems. It is well known that hardware designs have many redundancies from the logical point of view. Many powerful techniques have been developed for propositional logic problems to eliminate these redundancies. These techniques include use of efficient representations for propositional formulas, such as AIGs (And-Inverter Graphs) [17], simplification transformations for AIGs, such as BDD-sweeping, SAT-sweeping, AIG-rewriting [17], [16], [5], [7], and various pre- and in-processing techniques, e.g., [12], [10] which aim to simplify propositional problems for SAT and QBF solving. In this work, motivated by attempts to improve the performance and capacity of a model-checking algorithm we have recently developed [9], we seek to develop general simplification techniques for first-order logic problems.

First-order definitions are frequently used in many formalizations. For example, in hardware verification most generated formulas are definitions. An abundance of definitions can considerably slowdown the reasoning process. Many definitions in such problems are redundant, defining equivalent formulas, or can be eliminated without increasing the formula size. Moreover, direct clausal transformation of definitions can lead outside target fragments such as the effectively propositional (EPR) fragment (see definition in the next section). In this paper we introduce and discuss several methods for eliminating and simplifying definitions that also result in EPR-preserving clausification.

We further lift some of the propositional redundancy elimination techniques discussed above to first-order logic. In particular, we introduce quantified AIGs as an efficient data structure that enables sharing equivalent sub-formulas and facilitates implementation of simplification transformations for first-order logic formulas. On QAIGs, we implement BDD-sweeping, SAT-sweeping, and several rewriting transformations that help reduce the size of the problem after clausification and thus making the problem much simpler to solve.

Our improved clausification algorithm (which, as preprocessing steps, performs the above mentioned simplification transformations) is implemented in Vampire, a theorem prover for first-order logic [11]. We have evaluated the new clausification algorithm on three different benchmark sets: industrial hardware designs, quantified SMT problems and a TPTP problem set [22]. The experiments demonstrate the usefulness of our simplification transformations.

The paper is organized as follows. In Section II we recall basic definitions from first-order logic used throughout the paper. Sections III to XI are devoted to a range of simplification techniques for first-order logic formulas. Quantified AIGs are introduced and studied in Section XII. Experimental results are reported in Section XIII. Conclusions appear in Section XIV.

II. Preliminaries

We say that a formula φ is rectified if the following holds: (i) no variable occurs both free and bound in φ, and (ii) a variable can have at most one binding occurrence in φ. For simplicity of exposition, we assume that our formulas are rectified unless otherwise specified. In particular, this requirement is dropped in sections concerned with shared representation of formulas, such as AIGs and OBBDs, in order to increase sharing between subformulas.

We consider first-order formulas which are built from atoms using connectives $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ and quantifiers $\exists$ and $\forall$. We assume the standard semantics of first-order formulas. Polarity of a subformula occurrence at a position π will be denoted by $pol(\varphi, \pi) \in \{-1, 0, 1\}$, where 1 stands for positive, -1 for negative, and 0 for neutral polarity, which is defined inductively as follows. For any formula φ, $pol(\varphi, \epsilon) = 1$.

Consider $\varphi \mid_\pi = \psi$ and assume $pol(\varphi, \pi)$ is defined, then if ψ is of the form

- $Qx\ \psi_1$, where $Q \in \{\exists, \forall\}$ then $pol(\varphi, \pi.1) = pol(\varphi, \pi)$;
- $\psi_1 \star \psi_2$, where $\star \in \{\wedge, \vee\}$ then $pol(\varphi, \pi.1) = pol(\varphi, \pi.2) = pol(\varphi, \pi)$;
- $\psi_1 \rightarrow \psi_2$ then $pol(\varphi, \pi.1) = -pol(\varphi, \pi)$ and $pol(\varphi, \pi.2) = pol(\varphi, \pi)$;
- $\psi_1 \leftrightarrow \psi_2$ then $pol(\varphi, \pi.1) = pol(\varphi, \pi.2) = 0$.

Algorithms in this paper are parameterized by a Skolemization procedure $\mathcal{SK}$ and a clausification procedure $\mathcal{CL}$. In this paper we are not concerned how $\mathcal{SK}$ is realized, assuming only that $\mathcal{SK}$ transforms every first-order formula into an equi-satisfiable universal formula. We refer to [20], [2] for Skolemization and clausification techniques. As an example, we take an $\mathcal{SK}$ that applies miniscoping (moving all quantifiers inside the formula as far as possible) and eliminates existential quantifiers, as in inner Skolemization from left-to-right (resulting in flat Skolem terms). Similarly, we only require the clausification $\mathcal{CL}$ to transform universal formulas into equi-satisfiable sets of clauses.

The EPR fragment, also called the Bernays-Schönfinkel-Ramsey fragment, consists of first-order formulas with no occurrences of function symbols other than constants, and which when written in prenex normal form have the quantifier prefix $\exists^*\forall^*$. Skolemization applied to EPR formulas can introduce only constant function symbols; this can be used to show decidability of the EPR fragment. Several important verification problems have been encoded into EPR [19], [13], [8], [9], [1], benefiting from the succinct representations possible in this fragment. The transformations considered in this paper can help to produce equi-satisfiable EPR formulas when the given formula is not necessarily EPR. Such transformations turned out to be crucial for the performance of first-order solvers on encodings of real-life hardware verification problems.

III. Definition simplifications

A (non-recursive) *predicate definition* $def(pol, p, \varphi)$ is a first-order formula of the form

$$def(pol, p, \varphi) \stackrel{\text{def}}{=} \begin{cases} \forall \bar{x}\ (p(\bar{x}) \leftrightarrow \varphi(\bar{x})), & \text{if } pol = 0, \\ \forall \bar{x}\ (\varphi(\bar{x}) \rightarrow p(\bar{x})), & \text{if } pol = 1, \\ \forall \bar{x}\ (p(\bar{x}) \rightarrow \varphi(\bar{x})), & \text{if } pol = -1, \end{cases} \quad (1)$$

where p is a predicate symbol, φ is a first-order formula with free variables $FV(\varphi) \subseteq \{\bar{x}\}$, $pol \in \{-1, 0, 1\}$ and p does not occur in φ. Let us note that $def(0, p, \varphi) \equiv (def(1, p, \varphi) \wedge def(-1, p, \varphi))$; we call $def(1, p, \varphi)$ *positive* and $def(-1, p, \varphi)$ *negative* subdefinition of $def(0, p, \varphi)$. The variable condition $FV(\varphi) \subseteq \{\bar{x}\}$ can be omitted without loss of generality but doing so would add a syntactic burden not essential to this exposition.

First we consider unused definition elimination, presented in Table I.

Theorem 1: UDE is a satisfiability preserving and terminating transformation.

$\varphi \wedge def(pol, p, \psi)$	$\Rightarrow$	φ, where p does not occur in φ.
$\varphi \wedge def(0, p, \psi)$	$\Rightarrow$	$\varphi \wedge def(pol, p, \psi)$, where $pol \neq 0$ and all occurrences of p in φ are of polarity $-pol$.

TABLE I
Unused Definition Elimination (UDE)

Proof: (Sketch) Termination is trivial since each application removes one (sub)definition. Let us show that UDE is satisfiability preserving. Consider the case $\varphi \wedge def(0, p, \psi) \Rightarrow \varphi \wedge def(-1, p, \psi)$, where all occurrences of p in φ are of polarity 1. The rest of the cases are similar. The only non-trivial direction is to show that if $\varphi \wedge def(-1, p, \psi)$ is satisfiable then $\varphi \wedge def(0, p, \psi)$ is also satisfiable. First note that if a predicate occurs only positively in a formula $\chi(\bar{x})$ then the formula is monotone wrt. this predicate in the following sense. Consider a first-order interpretation I such that $I \models \chi(\bar{a})$. Then $I' \models \chi(\bar{a})$ for any I' which is obtained from I by changing the interpretation of p such that $p^I \subseteq p^{I'}$. Now assume that $\varphi \wedge def(-1, p, \psi)$ is satisfiable in a model I and p occurs only positively in φ. Let I' be obtained from I by changing the interpretation of p such that $I' \models p(\bar{a})$ iff $I \models \psi(\bar{a})$. It is easy to check that $I' \models def(0, p, \psi)$ and $p^I \subseteq p^{I'}$ since p does not occur in ψ. Finally we have $I' \models \varphi$ since $p^I \subseteq p^{I'}$ and φ is monotone wrt. p. ∎

Example 1: Consider a definition

$$def(0, p, \psi) \stackrel{\text{def}}{=} \forall x\ (p(x) \leftrightarrow (\forall y\ (q(x, y) \leftrightarrow s(x, y)))). \quad (2)$$

Such definitions frequently occur in encodings of hardware verification into first-order logic where, e.g., $p(x)$ can represent equivalence between two bit-vectors $q(x, y)$ and $s(x, y)$ at time x. After Skolemization and clausification of $def(0, p, \psi)$ we obtain clauses outside of the EPR fragment due to non-constant Skolem functions, thanks to the negative occurrence of the $\forall$ quantifier in the positive subdefinition of $def(0, p, \psi)$. Now if all other occurrences of p in our formula are positive we can apply UDE and simplify our definition to

$$def(-1, p, \psi) \stackrel{\text{def}}{=} \forall x\ (p(x) \rightarrow (\forall y\ (q(x, y) \leftrightarrow s(x, y)))). \quad (3)$$

It is easy to see that after Skolemization of this simplified definition we obtain an EPR formula.

IV. Definition resolution

A *resolvent* of two definitions $def(1, p, \psi)$ and $def(-1, p, \psi')$ is a universal closure of the formula $\psi \rightarrow \psi'$, denoted as $def(1, p, \psi) \otimes def(-1, p, \psi')$. In Table II we define *definition resolution transformation* (DRT) which can be used to eliminate a definition of a predicate based on exhaustive application of resolution. DRT is similar to the variable elimination rule well studied in the propositional case (we refer to [12] for a comprehensive survey of propositional preprocessing techniques).

Theorem 2: DRT is a satisfiability preserving and terminating transformation.

$\varphi \wedge \bigwedge_{i:1\leq i\leq n} def(1,p,\psi_i) \bigwedge_{j:1\leq j\leq m} def(-1,p,\gamma_j) \Rightarrow$
$\varphi \wedge \bigwedge_{i,j:1\leq i\leq n;1\leq j\leq m} def(1,p,\psi_i) \otimes def(-1,p,\gamma_j)$,
where p does not occur in φ.

TABLE II
DEFINITION RESOLUTION TRANSFORMATION (DRT)

$\varphi[p(\bar{t})]_\pi \wedge def(pol,p,\psi) \Rightarrow \varphi[\psi\sigma]_\pi \wedge def(pol,p,\psi)$,
where $\bar{x}\sigma = \bar{t}$, and either
(i) $pol = 0$, or
(ii) $pol \neq 0$ and all occurrences of p in $\varphi[p(\bar{t})]$ are of polarity $-pol$.

TABLE III
DEFINITION INLINING TRANSFORMATION (DIT)

We can slightly generalise DRT to definitions of the form $def(0,p,\psi)$ by splitting such definitions into positive and negative subdefinitions, applying DRT to the new definitions, and removing tautologies of the form $\psi \vee \neg\psi$.

Let us note that although DRT transformation is terminating, it can quickly increase the size of the formula and therefore is usually applied only in specific cases.

V. DEFINITION INLINING

One way of eliminating a predicate definition is to exhaustively *inline* it as defined in Table III. For related discussion we refer to [20] and in the QBF setting to [10].

Theorem 3: DIT is a satisfiability preserving transformation. Moreover, any sequence of DIT applications wrt. a given predicate definition is terminating.

After an exhaustive application of DIT wrt. a predicate definition $def(pol,p,\psi)$ we can eliminate this definition altogether by applying UDE.

Let us note that DIT can quickly increase the size of the resulting formula. We define a special case where such an increase stays linear wrt. size of the formula, called *non-growing definition inlining*.

Definition 1: A predicate definition $def(pol,p,\psi)$ is *non-growing* wrt. a formula φ, if either (i) p occurs only once in φ, or (ii) ψ is an EPR literal. An application of DIT $\varphi[p(\bar{t})]_\pi \wedge def(pol,p,\psi) \Rightarrow \varphi[\psi\sigma]_\pi \wedge def(pol,p,\psi)$ is *non-growing* (NDIT) if $def(pol,p,\psi)$ is non-growing wrt. $\varphi[p(\bar{t})]_\pi$.

Theorem 4: NDIT increases the size of the formula linearly wrt. the number of transformation steps.

Let us note that non-growing inlining is not confluent in general.

VI. EPR RESTORING INLINING

In Section III we saw that UDE can help obtain EPR resulting clausification. It turns out that for many problems, in particular those coming from hardware verification, applying UDE is not sufficient for obtaining EPR resulting clausification. Let us show how DIT can be used to restore EPR resulting clausification.

Example 2: Consider a definition

$$def(0,p,\psi) = \forall x\ (p(x) \leftrightarrow \forall y\ q(x,y))$$

and a formula

$$\varphi \quad = \quad [p(a) \rightarrow (\forall z\ (q(z,c) \leftrightarrow q(d,z)))] \wedge [\forall u\ (p(u) \vee q(c,d))].$$

After Skolemization and clausification of $def(0,p,\psi)$, we obtain two clauses $p(x) \vee \neg q(x, sk(x))$ and $\neg p(x) \vee q(x,y)$, corresponding to Skolemization of $def(1,p,\psi)$ and $def(-1,p,\psi)$ respectively. Let us note that the first clause is non-EPR. Moreover, p occurs both positively and negatively in φ and therefore we cannot apply UDE as we did in Example 1.

Let us discuss how one can restore EPR using inlining.

If we inline all non-positive occurrences of p in φ according to $def(0,p,\psi)$, we obtain

$$\varphi' \quad = \quad [(\forall y\ q(a,y)) \rightarrow (\forall z\ (q(z,c) \leftrightarrow q(d,z)))] \wedge [\forall u\ (p(u) \vee q(c,d))].$$

Let us note that after inlining, variable x in the definition of p became instantiated by a constant a. Therefore, standard clausification of φ' will result in an EPR formula. Moreover, now all occurrences of p in φ' are positive, and therefore we can apply UDE to $\varphi' \wedge def(0,p,\psi)$, obtaining $\varphi' \wedge def(-1,p,\psi)$. Finally, standard clausification of $def(-1,p,\psi)$ is also in EPR. This example demonstrates how definition inlining in combination with unused definition elimination can be used to obtain an EPR resulting clausification.

Definition 2: A predicate definition $def(pol,p,\varphi)$ is *pre-EPR* if $\mathcal{SK}(def(pol,p,\varphi))$ is not EPR and φ is of the form $Q\bar{y}\psi(\bar{x},\bar{y})$, where $FV(\varphi) = \{\bar{x}\}$, $Q \in \{\exists,\forall\}$ and ψ is quantifier free.

Let us note that for a pre-EPR predicate definition $def(0,p,\varphi)$, either its positive subdefinition is EPR and its negative subdefinition pre-EPR or vice versa.

A substitution σ is constant-grounding for a set of variables V if σ maps all variables in V to constants.

Lemma 1: Let $def(pol,p,\varphi)$ be a pre-EPR predicate definition. Then $\mathcal{SK}(\varphi\sigma)$ is EPR for any substitution that is constant grounding for $FV(\varphi)$.

The EPR restoring inlining strategy (ERI) consists of exhaustive application of inlining to pre-EPR definitions until pre-EPR (sub)definitions can be eliminated by UDE.

VII. EPR RESULTING CLAUSIFICATION FOR NON-CYCLING DEFINITIONS

Consider a set of definitions

$$\mathcal{D} = \{def(pol_1,p_1,\psi_1),\ldots,def(pol_k,p_k,\psi_k)\}.$$

Define a binary dependency relation between symbols as follows: $(p_i,p_j) \in dep$ if and only if p_j occurs in ψ_i. $\mathcal{D}$ is called *non-cycling* if the transitive closure of dep is a strict ordering. $\mathcal{D}$ is called *non-branching* if each predicate has at most one definition in $\mathcal{D}$.

Theorem 5: Consider a formula φ that can be split into $\varphi_{epr} \wedge \mathcal{D}$, where (i) $\mathcal{SK}(\varphi_{epr})$ is an EPR formula, (ii) $\mathcal{D}$ is a set of non-cycling pre-EPR definitions, and (iii) all occurrences

of predicates in $\mathcal{SK}(\varphi_{epr})$ are ground-matching in $\mathcal{D}$. Then the EPR restoring inlining strategy is EPR resulting on φ.

In order to obtain an EPR resulting clausification, we need to resort to DIT, which generally does not satisfy our non-growing criterium. In the next sections we consider techniques that simplify definitions and formulas further and are helpful in restoring the non-growing condition in practical cases.

VIII. Argument Collapsing

Consider a first-order formula φ and assume that all occurrences of an m-ary predicate p have distinct constants $c_1, \ldots, c_n$ at the k-th argument (for some k). Moreover assume that

$$\varphi \models c_i \neq c_j \quad \text{for} \quad 1 \leq i < j \leq n. \tag{4}$$

Then we can introduce new $m-1$-ary predicates $p_1, \ldots, p_n$ and replace each occurrence of p where c_i occurs as the k-th argument with the corresponding $m-1$-ary predicate p_i.

This transformation can lead to some equivalences becoming predicate definitions, and therefore eligible for all predicate definition-related transformations. This frequently happens for example in hardware encodings, where some predicate arguments are bit-blasted. Although in general checking condition (4) is as difficult as checking the satisfiability of the formula, in many cases this condition is trivially satisfied. For example, if $c_1, \ldots, c_n$ represent bit-indexes, then when all bit-indexes are enforced to be different this condition is automatically satisfied as in the case of (selective) bit-blasting.

IX. Conditional rewriting

In previous sections of this paper we were addressing unconditional predicate definitions, which were formulas of the form $p(\bar{x}) \star \psi(\bar{x})$, where $\star \in \{\leftrightarrow, \rightarrow, \leftarrow\}$. Some definitions, however, may hold only under certain assumptions; we would call these conditional, and they appear as formulas $\varphi(\bar{x}) \rightarrow (p(\bar{x}) \star \psi(\bar{x}))$. It is not sound to inline such definitions in the entire problem; however, if we have a formula which is also conditioned by φ, or more generally by some φ' such that $\varphi' \rightarrow \varphi$, we can safely perform the inlining there.

Moreover, this observation does not hold only for predicate definitions, but also for equalities. If we have $\varphi(\bar{x}) \rightarrow s = t$, we can use $s = t$ for rewriting terms in formulas conditioned by φ.

As an example, consider a typical formula which occurs in hardware encodings: $next(x, y) \rightarrow (p(x, y) \leftrightarrow \psi(x))$, which informally states that p holds at the consecutive states provided that ψ holds in the current state. Now we can inline this conditional definition of p in other formulas which are also conditioned by the next state predicate. For example $next(x, y) \rightarrow (p(x, y) \vee q(y)))$ can be rewritten by conditional inlining to $next(x, y) \rightarrow (\psi(x) \vee q(y)))$.

X. Definition merging

In our experience many problems from hardware formalisations contain predicates which are implicitly equivalent. Such predicates can be merged, considerably speeding up reasoning. More generally, we will consider implied non-growing predicate definitions NDI, shown in Table IV. Let us note that after application of NDI one can exhaustively apply DIT and UDE, eliminating the defined predicate from the problem. NDI covers the special case of implicitly equivalent predicates, since the equivalence of two predicates p and q can be represented as a non-growing definition $def(0, p, q(\bar{x}))$. In the following we consider the case of non-growing definitions of the form $def(pol, p, \psi)$, where ψ is an EPR literal.

$\varphi \Rightarrow \varphi \wedge def(pol, p, \psi)$, where (i) $\varphi \models def(pol, p, \psi)$, (ii) $def(pol, p, \psi)$ is non-growing and (iii) definition inlining is applicable to $def(pol, p, \psi)$ and φ

TABLE IV
Non-growing Definition Introduction (NDI)

In general, checking condition (i) of the applicability of NDI is undecidable, and therefore we need to resort to some heuristics. First we consider syntactic heuristics. Syntactic heuristics will be parameterized by a normalising function. A *normalising function* is a mapping of formulas into equivalent formulas. For example, a function that transforms formulas into negation normal form is a normalising function. There are many other useful normalising functions, e.g., removing double negations or eliminating some connectives. Let us fix a normalising function Δ. Then, two definitions $def(0, p_1, \psi_1)$ and $def(0, p_2, \psi_2)$ are *normalising equivalent* wrt. Δ if $\Delta(\psi_1)$ and $\Delta(\psi_2)$ are syntactically the same formulas.

Let us introduce *syntactic definition merging* as follows. Let $\varphi = \chi \wedge def(0, p_1, \psi_1) \wedge def(0, p_2, \psi_2)$, where ψ_1 and ψ_2 are normalising equivalent. Wlog assume $arity(p_1) \geq arity(p_2)$. Then $\varphi \Rightarrow \varphi \wedge def(0, p_1, p_2(\bar{x}))$ using NDI and we can eliminate p_1 from φ using DIT and UDE.

We implemented the following normalising function Δ_{syn} which (i) transforms formulas into negation normal form, (ii) flattens conjunctions and disjunctions, and (iii) bottom-up renames bound variables, applies sharing of subformulas, and orders conjuncts/disjuncts in disjunctions/conjunctions according to the ordering induced by sharing.

XI. SAT sweeping

In this section we discuss discovery of predicate definitions using propositional reasoning. The problem consists of two tasks. The first task is to convert the first-order problem into a propositional problem so that equivalences found between propositional variables will correspond to equivalences between first-order formulas. The second task is to efficiently find equivalences between variables in a given propositional problem.

Definition 3: Let τ be an injective map of first-order atoms to propositional variables. We extend τ so that it maps unquantified first-order formulas to propositional formulas in a straightforward way (e.g., $\tau(\varphi \wedge \rho) \mapsto \tau(\varphi) \wedge \tau(\rho)$). We further extend τ to universally quantified formulas in prenex form by dropping quantifiers.

Theorem 6: If $\tau(\varphi) \vdash \tau(\rho)$, then it holds that $\varphi \vdash \rho$.

The above theorem is a different formulation of a result given in [15] on under-approximating first-order reasoning using a SAT solver. To apply it to our case, we Skolemize and clausify the problem by using the default $\mathcal{SK}$ and $\mathcal{CL}$ procedures, and use the τ map to translate it to a first-order problem. Any equivalences between propositional variables implied by this problem can then be lifted back to first-order.

Now, given a satisfiable propositional formula φ and a set of interesting propositional variables V, our goal is to discover (some of) the equivalences implied by φ between literals of variables V. For convenience of notation, we consider the propositional constant $\top$ to be one of the interesting variables, which extends the approach also to discovery of true literals. We base our discovery of the propositional equivalences on the idea of simultaneous implicative SAT solving presented in [14], which allows discovery of implied implications (and equivalences) in one call to the SAT solver.

One problem to address is that the clausification process can extend the signature of the formula by introducing new symbols, for example Skolem constants. We use a naive way of dealing with this issue — when an equivalence contains a symbol that is not in the original signature, we discard the equivalence. There may be more advanced ways of eliminating these symbols; however, in our practical applications the presence of introduced symbols did not become a significant problem.

The above approach can be further extended to find equivalences between general sub-formulas, not only between atoms. To this end, we may do an additional transformation on the problem, before it is Skolemized and clausified. First we convert the formula to a QAIG graph (described in Section XII) and then use the Tseitin transformation on the graph, introducing a new name predicate for each node. When we later discover equivalences involving the introduced name predicates, we translate them back into the signature of the input formula by unfolding the introduced names.

XII. QAIG

Following the And-Inverter Graph (AIG) representation [17] of propositional problems widely used in propositional decision procedures, we introduce its counter-part data structure QAIG (Quantified And-Inverter Graphs). It is based on the AIG structure but contains an additional kind of node to represent quantifiers.

The set of QAIG terms on a set of atoms A can be defined as the smallest set of terms Q such that:

$$\begin{array}{l}\top \in Q\\ \forall a \in A : atom(a) \in Q\\ \forall q \in Q : neg(q) \in Q\\ \forall q_1, q_2 \in Q : conj(q_1, q_2) \in Q\\ \forall q \in Q, x \in \mathrm{free}(q) : quant(x, q) \in Q\end{array}$$

where $\mathrm{free}(q)$ is the set of free variables in the QAIG q. Below we will use q to denote QAIG nodes.

The QAIG data structure is a canonical in-memory representation of the QAIG terms. Canonicity of the structure means that if two terms are syntactically equal, they are represented by the same memory object. On top of this, we also normalize the order of the arguments in the $conj$ term and eagerly perform the local AIG simplifications proposed in [7].

Lemma 2: An arbitrary first-order formula can be converted to QAIG structure in a single linear-time traversal, assuming a constant-time access to a hash table.

Proof: The conversion is performed by bottom-up application of following transformation rules:

$$\begin{array}{l}\mathrm{ftq}(a) \Rightarrow atom(a) \text{ for atoms } a \in A\\ \mathrm{ftq}(\phi \wedge \psi) \Rightarrow conj(\mathrm{ftq}(\phi), \mathrm{ftq}(\psi))\\ \mathrm{ftq}(\phi \vee \psi) \Rightarrow neg(conj(neg(\mathrm{ftq}(\phi)), neg(\mathrm{ftq}(\psi)))\\ \mathrm{ftq}(\exists x : \phi) \Rightarrow neg(quant(x, neg(\mathrm{ftq}(\phi))))\\ \mathrm{ftq}(\phi \leftrightarrow \psi) \Rightarrow conj(neg(conj(neg(\mathrm{ftq}(\phi)), \mathrm{ftq}(\psi)),\\ \qquad\qquad neg(conj(\mathrm{ftq}(\phi), neg(\mathrm{ftq}(\psi))))\\ \ldots\end{array}$$

Rules for other logical connectives can be written analogously. Each of the rules transforms a formula into QAIG in constant time, assuming that its subformulas are already transformed and that construction of an QAIG term having its arguments is a constant time operation. ■

One thing to note is that in the rule for equivalence we see two occurrences of the $\mathrm{ftq}(\phi)$ (as well as of $\mathrm{ftq}(\psi)$) on the right-hand side. If we were using a flat representation to keep the QAIG terms, applying the rewriting rule would double the size of the term. However, as we use a canonical representation, we are interested in the number of distinct QAIG terms. This number grows only by a constant amount, so the size of the canonical QAIG structure will remain at most linear with the size of the formula.

A. QAIG Inlining

An important goal with the QAIG structure was to obtain a good infrastructure for performing definition inlining without exponential growth in the size of the problem. It can be used for implementing the inlining rules (N)DIT and ERI. QAIGs also provide better sharing of subformulas and definition merging.

At a high level, the QAIG inlining algorithm can be described as follows:

1) Collect the set of candidate rewrite rules $atom(a) \Rightarrow q$: This is done by a scan through the problem, looking for formulas in the shape of $\forall \bar{x}(a(\bar{x}) \leftrightarrow \phi(\bar{x}))$. Here $a(\bar{x})$ denotes an arbitrary non-equality atom with variables $\bar{x}$, and $\phi(\bar{x})$ stands for a formula with free variables being a subset of $\bar{x}$. In order to enable more definitions eligible for inlining, we also apply argument collapsing whenever possible, see Section VIII.
2) Instantiate candidate rules:
Whenever there is a rule in the form $atom(a(\bar{x})) \Rightarrow q(\bar{x})$ and there is an atom $a(\bar{t})$ where $\bar{t}$ is different from $\bar{x}$, we add an instance of the rule $atom(\bar{t}) \Rightarrow q(\bar{t})$ as another candidate rule. Let us note that such instantiated rules are used only for inlining; we do not add them to the resulting QAIG since they are subsumed by the original rules.

3) Remove cyclic dependencies:
   If we have a chain of candidate rules $atom(a_0) \Rightarrow q_0, \ldots, atom(a_n) \Rightarrow q_n$ such that $atom(a_n)$ occurs in q_0 and, for each $0 \leq i < n$, $atom(a_i)$ occurs in q_{i+1}, we remove one of the candidate rules to remove the cycle.
4) Apply rules to the QAIG representation of the problem:
   We exhaustively apply generated inlining rules to the QAIG. In this step we must be careful when rewriting using instantiated rules due to variable sharing. In particular, to improve sharing we do not assume that QAIGs are rectified and variable instantiation becomes a non-trivial problem which we consider in the next subsection.

The second step, which involves instantiation, is the only step where the size of the QAIG structure may grow[1] and is potentially the most time consuming. Instantiation is also specific to QAIGs, as the original AIG structure works with propositional atoms where instantiation does not make sense. In the next subsection we focus on the algorithm for QAIG instantiation and discuss some of its properties.

B. QAIG Instantiation

During the instantiation of candidate rules we need to apply a substitution for free variables in a QAIG formula. This cannot be done by a straightforward bottom-up traversal of the QAIG graph, as an atom a may appear at various positions of the QAIG, having different variables bound by its ancestor quantifier nodes. For example take a QAIG

$$conj(atom(p(x)), quant(x, atom(p(x))))$$

In the first occurrence of the atom $p(x)$ the variable is free; however, in the second it is bound by a quantifier. Applying a substitution $\{a/x\}$ will therefore result in

$$conj(atom(p(a)), quant(x, atom(p(x))))$$

We can express the instantiation as a set of rewrite rules parameterized by a substitution:

$$\begin{aligned}
&T_\sigma(atom(a)) = atom(a\sigma)\\
&T_\sigma(neg(a)) = neg(T_\sigma(a))\\
&T_\sigma(and(a,b)) = and(T_\sigma(a), T_\sigma(b))\\
&T_\sigma(quant(x,a)) = quant(x, T_{\sigma'}(a)),\\
&\qquad\qquad \text{where } \sigma' \text{ is } \sigma \text{ with } x \text{ unbound.}
\end{aligned}$$

Lemma 3: If we denote the size of the QAIG structure by n, the size of the QAIG term (which can be exponential with the size of the DAG data structure) by N and the number of variables in the substitution by m, the application of the instantiation rules can be implemented with complexity $O(\min(N, 2^m.n))$.

Proof: The bound $2^m.n$ follows from the fact that with the last rule we may generate at most 2^m possible substitutions, and then we may cache the pairs of a QAIG term and the substitutions applied to it. The bound N is valid because apart from the possible speed up by the earlier mentioned caching, we traverse the QAIG as a term of length N, rather than as a data structure of size n. ■

It can be noted that if in no QAIG subgraph would any variable occur as both free and bound, the instantiation could be performed in $O(n)$, as we would know in advance which variables would be instantiated and which would be quantified. Such a representation can be achieved by variable renaming; however, this would decrease the amount of sharing in the QAIG structure. For example, if we consider QAIG

$$conj(atom(p(x)), quant(x, atom(p(x))))$$

its size is 3: the $conj$ node, $quant$ node and the $atom(p(x))$ node which is referred to twice, once by the $conj$ node and once by $quant$. In order to ensure that no variable occurs both as bound and free, we would need to rename one of the occurrences, obtaining

$$conj(atom(p(x)), quant(y, atom(p(y))))$$

Now the size is 4, as we have two $atom$ nodes, $atom(p(x))$ and $atom(p(y))$.

C. QAIG BDD Sweeping

Following the idea of BDD sweeping for propositional problems [17], we attempt to simplify QAIGs using BDDs.

We perform the simplification from simpler AIGs to more complex. When we process an AIG node q, we first check whether it hasn't been simplified into q' by simplifications on its parent nodes. Then, if the number of distinct atoms in the AIG is lower than a given threshold (16 in our implementation), we convert it into a BDD and then back, obtaining q''. If the DAG size of q'' is smaller than the size of q' we use q'' as the simplified node, otherwise we use q'. We also keep a map where for each BDD we store the most compact QAIG representation of it we have encountered. If we encounter several QAIGs with the same BDD, we replace them in the end by the most compact one.

The conversion of QAIGs into BDDs uses a straightforward bottom-up algorithm atb:

$$\begin{aligned}
&\mathrm{atb}(atom(a)) = bdd_{var}(atom(a))\\
&\mathrm{atb}(quant(x,q)) = bdd_{var}(quant(x, \mathrm{atb}(q)))\\
&\mathrm{atb}(neg(q)) = bdd_{neg}(\mathrm{atb}(q))\\
&\mathrm{atb}(conj(q_1,q_2)) = bdd_{and}(\mathrm{atb}(q_1), \mathrm{atb}(q_2))
\end{aligned}$$

When converting from BDD to an QAIG, we first extract from the BDD all literals L such that the BDD formula φ can be written as $\varphi \leftrightarrow L \wedge \varphi[L := \top]$ or $\varphi \leftrightarrow L \rightarrow \varphi[L := \top]$. Then we continue with the conversion on the simplified formula $\varphi[L := \top]$. When there are no more possible extractions, we perform a naive conversion

$$\begin{aligned}
\mathrm{bta}(ite(x,t,e)) = and(&neg(and(atom(x), neg(\mathrm{bta}(t)))),\\
&neg(and(neg(atom(x)), neg(\mathrm{bta}(e)))))
\end{aligned}$$

D. AIG Definition Introduction

We traverse an AIG in a top-down manner, and for each node we remember how many times it would appear in a tree-like representation of the AIG (which can be exponentially large, compared to the DAG representation). If the counter

[1] In the rewriting step we still create new nodes, but for every added node there is a node that was rewritten and therefore removed.

rule	full name
UDE	Unused Definition Elimination
DRT	Definition Resolution Transformation
(N)DIT	(Non-growing) Definition Inlining Transf.
ERI	EPR Restoring Inlining
NDI	Non-growing Definition Introduction
ED	Equivalence Discovery (or SAT sweeping)
AC	Argument Collapsing
ABS	AIG BDD Sweeping
ADI	AIG Definition Introduction
ACR	AIG Conditional Rewriting
VEP	Variable Equality Propagation

Fig. 1. Simplification transformations

of a node q reaches a certain threshold value (4 in our implementation), we introduce for it a definition $p(\bar{x})$ where $\bar{x}$ are all the free variables of the node q. We will use this definition in place of φ later when we convert the AIG representation back into the non-shared first-order formulas, which will be converted to $p(\bar{x}) \leftrightarrow \varphi_q(\bar{x})$, where $\varphi_q(\bar{x})$ is the formula corresponding to the node q.

E. QAIG Variable Equality Propagation

If a variable occurs in an equality, under some conditions we may propagate it into neighbouring subformulas. We perform this transformation on first-order formulas, but using the above AIG instantiation terminology it can be expressed as

$$and(x = s, b) \Rightarrow and(x = s, T_{x\to s}(b))$$
$$quant(x, neg(and(x = s, b))) \Rightarrow T_{x\to s}(b),$$

where x does not occur in s.

In the first case we cannot remove the equality $x = s$, as x may occur also elsewhere in the problem. However, in the second case (due to the quantifier) we know x does not appear elsewhere, so the equality can be removed. The second rule is also discussed in [23] in the context of simplifying quantified bit-vector formulas.

XIII. Experimental results

Figure 1 summarizes the main simplification transformations discussed in the paper. They are implemented in Vampire's clausifier. The implementation is flexible in that these options can be run in a different order, often repeatedly (or until fix-point) when useful.

We have evaluated the simplification techniques reported in the paper on three sets of benchmarks:

(A) EPR-based bounded model checking problems [9].
(B) The QA_UF problems from the SMT library [3].
(C) The FOF problems from the TPTP library [22].

In all the experiments, the time spent on pre-processing was negligible compared to the timeout used and is not reported.

A. Evaluation on EPR-based BMC problems

In [9] we studied an encoding of the BMC [4] problem into first-order logic. The BMC encoding there is called BMC1, as the transition relation is never enrolled explicitly (thus one deals with only one copy of the transition relation). In order to better explain the benchmark results below, let us briefly recall the encoding used for BMC1.

Design block	FOF size	Bound		CNF size		Time	
		Bln	Dft	Bln	Dft	Bln	Dft
BPB2	913	7	9	1977	2921	6994	8023
DCC1	1093	4	4	3209	1615	5999	8981
DCC2	431	7	10	861	370	6542	9465
DCI1	4678	0	1	15899	9852	149	3085
PMS1	574	5	7	1295	1016	8157	6771
ROB2	713	5	7	1717	1157	8239	6157
SCD1	736	8	9	1908	1328	7704	5366
SCD2	267	8	15	755	524	5691	6370
TOTAL	9404	44	62	27621	18783	49475	54218

TABLE V
BMC1 results on industrial benchmarks.

Let n be a non-negative integer. The *n-step unrolling of the transition system* is defined as follows. Take new constants $s_0, \ldots, s_n$ and a new binary predicate $next$. Denote by $In(\mathtt{S})$, $Fin(\mathtt{S})$, and $Trans(\mathtt{S}, \mathtt{S'})$ the initial and final state constraints, and the transition relation, respectively. The n-step unrolling of the transition system is defined as the set of formulas

$$In(s_0); Fin(s_n);$$
$$\forall \mathtt{S}, \mathtt{S'}(next(\mathtt{S}, \mathtt{S'}) \to Trans(\mathtt{S}, \mathtt{S'}));$$
$$next(s_0, s_1); next(s_1, s_2); \ldots next(s_{n-1}, s_n).$$

In BMC1, it is possible to solve the BMC problems incrementally per bound, and increasing the bound to $n + 1$ is expressed by adding an extra constant s_{n+1} and an axiom $next(s_n, s_{n+1})$. (There are a few more subtleties involved in unrolling, but they are irrelevant to the discussion here.)

Table V displays bounds reached by the iProver solver [15] on eight BMC1 benchmarks produced from actual Intel hardware designs. We also report the sizes of the original FOF problems and the sizes of the resulting CNFs, and the solver run-times. The timeout used was 1000 seconds. This data is given for the base-line (or Bln, for short) clausification algorithm of Vampire, and a reasonable default (or Dft for short) version to which we arrived as a result of experiments on BMC1 benchmarks. Unused definition elimination (UDE) is already part of the Vampire baseline clausifier. In the default version above, all the options listed in Figure 1 except for ACR are switched on. (Surprisingly to us, ACR didn't prove useful on BMC1, even if it helps simplifying within formulas ϕ in next-state axioms of the form $\forall \mathtt{S}, \mathtt{S'}(next(\mathtt{S}, \mathtt{S'}) \to \phi(\mathtt{S}, \mathtt{S'}))$.) As can be observed from the table, with the advanced clausification options the total CNF size was reduced from 27621 to 18783, and the number of solved bounds increased from 44 to 62, with only a slight increase in solving time. Note also that higher BMC bounds are much more difficult to solve than lower bounds. Thanks to DIT and ERI, all the resulting CNFs were EPR.

B. Evaluation on SMT benchmarks

We used our clausification algorithms and then passed the clauses in a TPTP format to the Z3 [18] solver which was run

		Bln+ACR	Dft	Dft+ACR	Dft+ACR(ERI)
$\geq 2x$	faster	100	10	74	122
$> 1x$	faster	4890	1527	4847	4941
$\geq 2x$	slower	36	33	36	36

TABLE VI
PERFORMANCE RESULTS FOR QA_UF SMT PROBLEMS.

with a timeout of 30 seconds. Out of 93 problems that timed out with either the baseline or the advanced clausification algorithms, 3 problems were uniquely solved after baseline clausification, and 12 problems could only be solved using the advanced clausification options. Since these represent only a small fraction of the entire problem set, in Table VI we report runtime results, where ACR refers to full conditional rewriting and ACR(ERI) to conditional rewriting restricted to the EPR-restoring strategy. We can conclude that these preprocessing techniques can considerably speed up SMT solvers on a number of problems. We can also note that ACR is very useful both with the baseline and advanced clausification options.

C. Evaluation on TPTP benchmarks

We also evaluated the clausification configurations described in Table VI on all 14540 FOF problems of the TPTP library. We collectively refer to these configurations as advanced clausification configurations. We ran both Vampire and iProver solvers (using Vampire's clausifier) with 30 and 60 seconds timeouts, respectively. A decrease in the number of clauses after applying advanced clausification occurred in 2922 problems. All together, with the advanced clausification configurations Vampire solved 276 problems that it cannot solve (with the same strategies) with the baseline clausification, while it solved 76 problems with baseline clausification that cannot be solved with advanced clausification. In total, Vampire solved 11906 problems with advanced clausification configurations while with baseline clausification it solved 11706. Similarly, iProver solved 482 problems only when it used the advanced clausification configurations, and cannot solve 83 problems that it can solve with baseline clausification. In total, iProver solved 7178 problems with baseline clausification and 7577 problems with advanced clausification configurations. We note that 15 (resp. 7) problems uniquely solved with the advanced clausification configurations by Vampire (resp. iProver) have the rating 1 in TPTP 5.3.0; they cannot be solved within a 300 second timeout by any of the solvers that participated in CASC theorem proving competition in 2011.

XIV. CONCLUSIONS

Preprocessing is crucial when dealing with large industrial problems. In this paper we presented a number of preprocessing techniques for simplification of first-order formulas. One of the main goals was to investigate methods for simplifying first-order formulas so that Skolemization and clausification would result in clause sets that are simpler for first-order reasoners. We have investigated methods for definition simplification, EPR-preserving clausification based on definition inlining, discovery and merging of first-order definitions. We also introduced new data structures: quantified AIG, called QAIGs, and a combination of QAIGs and BDDs. We implemented all our techniques in Vampire[2]. Vampire can also be used as an intermediate preprocessing/clausification step for other solvers, in the same way as we used it with iProver and Z3.

We evaluated our techniques over a broad range of benchmarks, including industrial hardware verification benchmarks coming from real-life designs at Intel and largest problem collections for first-order logic (TPTP) and SMT (SMT-LIB). The results are very encouraging, showing that many problems can be solved only with the help of our preprocessing techniques.

There are many directions for future work. Let us only mention that we are planning to develop inprocessing techniques for FOL solvers, that is, we want to combine simplification and reasoning steps more tightly.

REFERENCES

[1] Alberti F., Armando A., Ranise S. ASASP: Automated Symbolic Analysis of Security Policies, CADE 2011, LNCS 6803, 26-33, Springer.
[2] Baaz M. Egly U., Leitsch A. Normal Form Transformations, in [21], pages 273-333.
[3] Barrett C., Stump A., Tinelli C., The SMT-LIB Standard: Version 2.0
[4] Biere A., Cimatti A., Clarke E., Zhu Y. Symbolic model checking without BDDs, TACAS 1999.
[5] Bjesse P., Boralv A. DAG-aware circuit compression for formal verification, ICCAD 2004.
[6] Brand D. Verification of large synthesized designs, ICCAD 1993.
[7] Brummayer R., Biere A. Local two-level And-Inverter Graph minimization without blowup, MEMICS 2006.
[8] Emmer M., Khasidashvili Z., Korovin K., Voronkov A. Encoding Industrial Hardware Verification Problems into Effectively Propositional Logic FMCAD'10, 2010.
[9] Emmer M., Khasidashvili Z., Korovin K., Sticksel C., Voronkov A. EPR-Based Bounded Model Checking at Word Level, IJCAR 2012.
[10] Giunchiglia E., Marin P., Narizzano M. sQueezeBF: An Effective Preprocessor for QBFs Based on Equivalence Reasoning, SAT 2010.
[11] Hoder K., Kovács L., Voronkov A. Invariant Generation in Vampire, TACAS 2011.
[12] Järvisalo M., Heule M.,3, and Biere A. Inprocessing Rules, IJCAR 2012
[13] Khasidashvili Z., Kinanah M., Voronkov A. Verifying Equivalence of Memories Using a First Order Logic Theorem Prover FMCAD'09, 2009.
[14] Khasidashvili Z., Nadel A. Implicative Simultaneous Satisfiability and Applications, HVC 2011.
[15] Korovin K. iProver–an instantiation-based theorem prover for first-order logic (system description), IJCAR, 2008.
[16] Kuehlmann, A. Dynamic Transition Relation Simplification for Bounded Property Checking, ICCAD 2004.
[17] Kuehlmann A., F. Krohm. Equivalence checking using cuts and heaps, DAC 1997.
[18] de Moura L., Bjorner N.: Z3: An Efficient SMT Solver. TACAS 2008: 337-340
[19] Navarro-Perez, J.A., Voronkov A. Encodings of Bounded LTL Model Checking in Effectively Propositional Logic, CADE 2007, 346-361, LNCS 4603, Springer.
[20] Nonnengart A., Weidenbach C. Computing Small Clause Normal Forms, in [21], pages 335-367.
[21] Robinson J. A., Voronkov A. Handbook of Automated Reasoning, Elsevier and MIT Press, 2001.
[22] Sutcliffe G. The 5th IJCAR automated theorem proving system competition @CASC-J5, AI Communications, Volume 24(1), pp. 75-89, 2011.
[23] Wintersteiger C.M., Hamadi Y., de Moura L.M. Efficiently solving quantified bit-vector formulas, FMCAD 2010.

[2]publicly available at http://www.vprover.org/