A Nelson-Oppen based Proof System using Theory Specific Proof Systems*

Frédéric Besson, Pierre-Emmanuel Cornilleau, David Pichardie
INRIA Rennes – Bretagne Atlantique, France

Abstract

SMT solvers are nowadays pervasive in verification tools. When the verification is about a critical system, the result of the SMT solver is also critical and cannot be trusted. The SMT-LIB 2.0 is a standard interface for SMT solvers but does not specify the output of the `get-proof` command. We present a proof system that is geared towards SMT solvers and follows their conceptually modular architecture. Our proof system makes a clear distinction between propositional and theory reasoning. Moreover, individual theories provide specific proof systems that are combined using the Nelson-Oppen proof scheme. We propose specific proof systems for linear real arithmetic (LRA) and uninterpreted functions (EUF) and discuss proof generation and proof checking. We have evaluated the cost of generating proofs in our proof system. Our experiments on benchmarks taken from the SMT-LIB library show that the simple mechanisms used in our approach suffice for a large majority of the selected benchmarks.

1 Introduction

Modern *Satisfiability Modulo Theory* (SMT) solvers (*e.g.,* CVC3 [2], VeriT [5], Yices [11] or Z3 [7]) are able to automatically discharge formula of industrial size combining various logic fragments such as linear (real or integer) arithmetic, the theory of uninterpreted function symbols or the theory of arrays. The SMT-LIB 2.0 format [1] is a standard interface for SMT solvers. It provides a unified syntax for SMT problems and a rich interface for interacting with SMT solvers. The command `check-sat` tests the satisfiability of the problem and is the minimal information that is expected from a SMT solver. More advanced features are unsat cores (`get-unsat-core`) or models (`get-model`).

In case the problem is unsat, the command `get-proof` outputs a *proof* of this fact. The answer to the `get-proof` command is unspecified and is therefore prover-specific. Actually, the SMT solvers CVC3, veriT and Z3 all use a different syntax and semantics for their proofs. Moreover, the granularity of the proofs greatly differ. This hinders proof exchanges and significantly complicates proof checking by third-party entities. Several works show that checking proofs generated by SMT provers in skeptical proof-assistants (see *e.g.,* [12, 13, 4]) requires substantial (retro-)engineering.

In this paper, we advocate for a very structured proof system that mimics the (conceptual) modular architecture of SMT solvers. We provide:

- A new methodology to obtain unsatisfiability proofs from an untrusted, non proof-producing, SMT solver. Our proof format is modular: it separates boolean reasoning from theory reasoning. Each multi-theory proof is itself decomposed (using the Nelson-Oppen proof scheme) into mono-theory proofs.

- A prototype prover that generate proofs. The prover only requires a SMT solver that extracts unsat cores and boolean models, as expected by the SMT-LIB 2 format. A SMT

*This work was funded by the ANR Decert projet

solver is used to obtain unsat multi-theory cores and any proof-generating multi-theory prover can be used to obtain certificates for theory specific lemmas.

For uninterpreted functions (EUF) and linear real arithmetic (LRA) we propose specific proof systems and discuss how to generate proofs using state-of-the-art decision procedures.

We have done preliminary experiments to assess the viability of our proof generation. Using SMT-LIB 2.0 scripts, we have implemented a lazy SMT loop [9]: a first SMT solver acting as a SAT solver; the second SMT solver acting as a Theory-reasoner. Such a set-up amounts to disabling many optimisations and forbidding, for instance, any global pre-processing or theory-propagation. Nonetheless, the results are rather encouraging as we are able to generate for most of the benchmarks a proof with an acceptable overhead.

The remainder of this paper is organised as follows. Section 2 covers the needed SMT solving background and describe a simple SMT proof search. Section 3 defines our proof systems and describe their interactions. Section 4 presents some experimental evaluation results. We discuss related work in Section 5 and conclude in Section 6 with a discussion on further work.

2 Background

In this section, we give an overview of some concepts useful to describe the interactions between the Boolean and the theory part of a SMT proof search.

2.1 Separating Boolean and Theory reasoning

We consider multi-theory unquantified first-order formulas, with terms belonging to combinations of theories. Such a formula will be called *T-formula*. The following formula is an example of T-formula combining uninterpreted functions and arithmetic:

$$f(f(x) - f(y)) \neq f(z) \ \wedge \ x \leq y \ \wedge \ ((y + z \leq x \wedge z \geq 0) \vee (y - z \leq x \wedge z < 0)) \qquad (1)$$

Boolean Abstraction. A simple approach to solve a T-formula is to consider its Boolean abstraction and search for propositional models, eliminating along the search any model leading to a contradiction at the theory level. To obtain the Boolean abstraction, the T-formula terms from the underlying theories are substituted for propositional variables. We will refer to the resulting propositional formula as the *propositional abstraction* of the initial T-formula. Each variable corresponds to a theory literal. For example, the propositional abstraction of the T-formula (1) is $A \wedge B \wedge ((C \wedge D) \vee (E \wedge \neg D))$, with the following *$T$-mapping*:

$$A \rightsquigarrow f(f(x) - f(y)) \neq f(z) \qquad B \rightsquigarrow x \leq y \qquad C \rightsquigarrow y + z \leq x$$
$$D \rightsquigarrow z \geq 0 \qquad E \rightsquigarrow y - z \leq x$$

If the abstracted formula does not have a model, *i.e.*, the propositional abstraction is unsatisfiable, then the T-formula is unsatisfiable at the Boolean level. But if the abstraction has a model, this model needs to be validated at the theory level. To do that, we transform this model in a conjunction of theory atoms, called *T-conjunction*, according to the T-mapping between propositional variables and corresponding atoms. Consider the following propositional model of the T-formula (1):

$$A \rightsquigarrow True \qquad B \rightsquigarrow True \qquad C \rightsquigarrow True \qquad D \rightsquigarrow True \qquad E \rightsquigarrow False \qquad (2)$$

The corresponding T-conjunction is

$$f(f(x) - f(y)) \neq f(z) \wedge x \leq y \wedge y + z \leq x \wedge z \geq 0 \wedge \neg(y - z \leq x) \tag{3}$$

This formula is unsatisfiable (see Section 2.2 for involved theory reasoning), hence model (2) leads to a contradiction at the theory level, and has to be removed from the search.

We eliminate model (2) from the propositional SAT search by adding to the propositional abstraction, as a new clause, called a *conflict clause*: the negation of the abstraction of the T-conjunction (3), *i.e.*, $A \wedge B \wedge C \wedge D \wedge \neg E \implies False$. We refer to the conjunction of the propositional abstraction and the discovered conflict clauses as the *propositional abstraction set*. At the beginning of the search, this set only contains the propositional abstraction of the T-formula. We can now continue the search by looking for another model of the propositional abstraction set, until either the set is unsatisfiable, or a model of the initial T-formula is found.

Shorter Conflict Clauses. Notice that in our example the atom $\neg(y-z \leq x)$ is not necessary to prove T-conjunction (3) unsatisfiable. The T-conjunction

$$f(f(x) - f(y)) \neq f(z) \wedge x \leq y \wedge y + z \leq x \wedge z \geq 0 \tag{4}$$

is already unsatisfiable, it is in fact an unsatisfiable core. The T-conjunction (3) being redundant it leads to a *weak* conflict clause that does not eliminate the following model:

$$A \rightsquigarrow True \qquad B \rightsquigarrow True \qquad C \rightsquigarrow True \qquad D \rightsquigarrow True \qquad E \rightsquigarrow True \tag{5}$$

By building the conflict clauses from unsatisfiability cores (unsat-cores) instead of whole T-conjunctions, we eliminate more models, and accelerate the search. If we use unsat-cores in our example, the conflict clause to add, in order to eliminate model (2), is $A \wedge B \wedge C \wedge D \implies False$, and it also eliminate model (5). The propositional abstraction set is then

$$A \wedge B \wedge ((C \wedge D) \vee (E \wedge \neg D))$$
$$A \wedge B \wedge C \wedge D \implies False$$

A model of this propositional formula is

$$A \rightsquigarrow True \qquad B \rightsquigarrow True \qquad C \rightsquigarrow True \qquad D \rightsquigarrow False \qquad E \rightsquigarrow True$$

and the corresponding T-conjunction is $f(f(x)-f(y)) \neq f(z) \wedge x \leq y \wedge y+z \leq x \wedge z < 0 \wedge y-z \leq x$. This is an unsatisfiable formula, and its unsat-core is

$$x \leq y \wedge z < 0 \wedge y - z \leq x \tag{6}$$

This unsat-core leads to the conflict clause $B \wedge \neg D \wedge E \implies False$. Once we have added this conflict clause to the propositional abstraction set, the set becomes unsatisfiable, and the model search ends.

Concluding the Search. Any model of the propositional abstraction set is a model of the propositional abstraction, because the conflict clauses we add to the set only eliminate models. Conversely, any model of the propositional abstraction which is not a model of the propositional abstraction set corresponds to an unsatisfiable T-conjunction. As a result, if the T-conjunction corresponding to a propositional model is satisfiable, we can obtain a model of the initial T-formula, *i.e.*, a proof of satisfiability. On the contrary, if all propositional models translate into unsatisfiable T-conjunctions, the initial T-formula is unsatisfiable. In such case, when the search ends the propositional abstraction set is an unsatisfiable propositional formula. It is composed of:

- the propositional abstraction; in our example $A \wedge B \wedge ((C \wedge D) \vee (E \wedge \neg D))$
- all the conflict clauses; in our example we found two of them:

$$A \wedge B \wedge C \wedge D \implies False$$
$$B \wedge \neg D \wedge E \implies False$$

Each conflict clause corresponds to an unsatisfiable T-conjunction. In our example, the two conflict clauses come from the T-conjunctions unsat-cores (4) and (6).

A conflict clause is the abstraction of a tautology, *i.e.*, the negation of an unsatisfiable T-conjunction. In fact, we could add to the propositional abstraction set the abstraction of any tautology, conflict clause or not, without endangering the soundness of our proof search. Adding more clauses to the propositional abstraction would eliminate more models from the search and accelerate the procedure. Conflict clauses can be more generally seen as abstraction of theory lemma, *i.e.*, valid formulas whose abstractions are necessary to prove the unsatisfiability of the T-formula. To optimise the search, other kinds of theory lemmas could be useful, and modern SMT solvers do use more theory reasoning than mere conflict clauses. Some SMT solvers check partial models incrementally against the theory in order to build similar subsets. In this example, it is useless to assign a boolean value to E to obtain a theory conflict. Second, the multi-theory solver may be able to discover propagation lemmas, *i.e* theory literals that are consequence of partial models. In a boolean form, such lemmas allow the SAT solver to perform efficient unit propagation and reduce its research tree.

2.2 Multi-Theory Conjunction Proofs

We now give an overview of the Nelson-Oppen equality exchange, used to prove unsatisfiability of T-conjunctions. We illustrate the proof search on the T-conjunction (4) from the previous example[1].

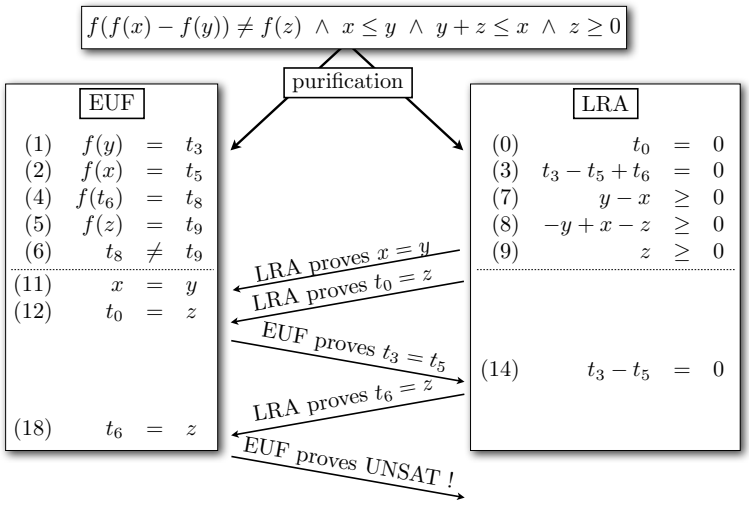


Figure 1: Example of Nelson-Oppen equality exchange

In this example, we combine the theories of Equality and Uninterpreted Function (EUF) and Linear Real Arithmetic (LRA). For EUF, a literal is an equality between multi-sorted ground

[1]The formula is taken from [14].

terms and a formula is a conjunction of positive and negative literals. The axioms of this theory are reflexivity, symmetry and transitivity, and the congruence axiom $\forall a \forall b, a = b \Rightarrow f(a) = f(b)$ for functions. Such a theory is infinitely stable and decidable using an efficient extension of the union-find algorithm to compute congruence closures [10]. The only way for a set of literals to be unsatisfiable is to deduce from positive literals an equality trivially negated by one of the negative literals. For LRA, a literal is a linear constraint $c_0 + c_1 \cdot x_1 + \cdots + c_n \cdot x_n \bowtie 0$ where $(c_i)_{i=0..n} \in \mathbb{Q}$ is a sequence of rational coefficients, $(x_i)_{i=1..n}$ is a sequence of real unknowns and $\bowtie \in \{=, >, \geq\}$[2]. Here, a formula is a conjunction of positive literals. Such a theory is also infinitely stable and decidable using the Simplex procedure [10].

The Nelson-Oppen algorithm is a sound and complete decision procedure for combining infinitely stable theories with disjoint signatures. Figure 1 presents the deduction steps of this procedure on an example. We start from the formula at the top of Figure 1 and first apply a *purification* step that introduces sufficiently many intermediate variables to flatten each term and dispatch *pure* formulas to each theory. Then, each theory exchanges new equalities with the others, until a contradiction is found.

3 Proof Systems

In this section we discuss the proof system for multi-theory formulas. We begin with a general discussion on proof searches for whole formulas, then detail what is intended by Nelson-Oppen proofs. We follow with instances of uninterpreted functions (EUF) and linear real arithmetic (LRA) proofs.

3.1 Proof Scheme

Preprocessing. The first step of SMT solving is to handle Boolean abstraction and purification. Depending on the SAT proof system we use, we also need to put the propositional formulas in Conjunctive Normal Form (CNF). We can either give a proof for all these preprocessings, or make sure the checker will be able to find the normal forms itself, by using the same algorithms in the proof-producing prover and in the checker.

SMT Proofs. Once we are sure that the proof-producing prover and the proof checker agree on the preprocessing of the formula, the proof of unsatisfiability is composed of two parts:

- a proof of unsatisfiability of the propositional abstraction set, including all conflict clauses;
- the set of unsatisfiable T-conjunctions with their proofs.

With the theory proofs we can check the validity of the theory lemmas, and with the propositional proof we can check the unsatisfiability of the formula at the Boolean level.

Proof Generation. The proof generation would be facilitated if state-of-the-art SMT solvers would give direct access to the conflict clauses discovered during a search, or to any kind of theory reasoning for that matter. Still, we would have to link these discovered formulas to the initial problem, which would require to take into account any preprocessing done by the solver. Anyway, using the SMT-LIB 2.0 standard we can access models discovered by a SAT solver and unsatisfiability cores using a SMT solver. Then we can use off-the-shelf solvers to generate

[2]Following the Simplify [10] approach, disequality is managed on the EUF side.

proofs, if non optimal ones, and try to evaluate our scheme. See Section 4 for experimental results.

3.2 Propositional SAT Proof System

One part of a SMT proof is a proof of unsatisfiability of the propositional abstraction set. Unsatisfiability proofs of propositional formulas have already been discussed in the literature. Several proof systems [19] and checking procedures [22] exist. State-of-the-art solvers like zChaff [18] or PicoSAT [3] can output checkable proofs. Formats may vary and we will not go into details, but all proof systems are based on the resolution rule:

$$\frac{\neg x \vee C \qquad x \vee C'}{C \vee C'}$$

The variable x is called the resolution variable and C and C' are clauses. Using resolution chains, new clauses are deduced. Once the empty clause has been deduced, the initial set of clauses has been proved unsatisfiable; hence a proof is a list of resolution chains, and the checker uses them to produce new clauses until it reaches the empty clause. Using optimised algorithms, resolution proofs can be checked efficiently [20]. Other proof systems exist *e.g.,* Reverse Unit Propagation proofs [21], for checking propositional unsatisfiability.

3.3 Nelson-Oppen Proofs

The second part of a SMT proof is a set of T-conjunctions and their proofs of unsatisfiability. We have seen on an example in Section 2.2 how to solve such conjunctions and we will now introduce Nelson-Oppen based proofs using the same example.

Step 1 (LRA)	(11) $x = y$ because (7) gives $y - x \geq 0$ and (8) + (9) gives $x - y \geq 0$
Step 2 (EUF)	(14) $t_3 = t_5$ because of the following rewriting steps $t_3 \xrightarrow{\text{trans. with (1)}} f(y) \xrightarrow{\text{congr. with (11)}} f(x) \xrightarrow{\text{trans. with (2)}} t_5$
Step 3 (LRA)	(18) $t_6 = z$ because (3) + (7) + (8) − (14) gives $t_6 - z \geq 0$ and (7) + (8) + 2 · (9) + (14) − (3) gives $z - t_6 \geq 0$
Step 4 (EUF)	*False* by contradiction of (6) with the following rewriting steps $t_8 \xrightarrow{\text{trans. with (4)}} f(t_6) \xrightarrow{\text{congr. with (18)}} f(z) \xrightarrow{\text{trans. with (5)}} t_9$

Figure 2: Example of Nelson-Oppen proof

Proof Generation. Figure 2 presents the proofs we consider. The proof generation only has to consider useful exchanges, based on the whole history of exchanges. In this example, $t_0 = z$ is not required in the final proof. A LRA proof of $a = b$ is made of two Farkas proofs [17] of $b - a \geq 0$ and $a - b \geq 0$. Each inequality is obtained by a linear combination of hypotheses that preserves signs. A EUF proof of $a = b$ is made of a sequence of rewriting steps that allows to reach b from a. Each proof is expressed in a theory-specific proof format that is complete w.r.t. to the theory, *i.e.*, if a formula is unsatisfiable, there exists a proof of it.

For EUF+LRA, unsatisfiability can always be proved without resorting to case-splits. EUF and LRA are said to be *convex* theories. In the general case of non-convex theories (such as linear integer arithmetic or theories of arrays), disjunctions of equalities may be generated and case splits are necessary.

The Nelson-Oppen Proof System. The proof system we propose for a combination of n theories T_1,..., T_n is given below.

$$\frac{\Gamma_i \vdash_{T_i} prf_i : (\Gamma'_i, eqs) \quad \bigwedge_{x_k = y_k \in eqs} (\Gamma_1[j \mapsto x_k = y_k], \ldots, \Gamma'_i, \ldots, \Gamma_n[j \mapsto x_k = y_k] \vdash_{NO} sons[k] : \mathit{False})}{\Gamma_1, \ldots, \Gamma_n \vdash_{NO} (prf_i, sons) : \mathit{False}}$$

In this judgement Γ_i represents an environment of pure literals of theory T_i. Each theory is equipped with is own deduction judgement $\Gamma_i \vdash_{T_i} prf_i : (\Gamma'_i, eqs)$ where Γ_i and Γ'_i are environments of theory T_i, prf_i is a proof specific to theory T_i and *eqs* is a list of equalities between variables. Such a judgement reads as follows: assuming that all the literals in Γ_i hold, we can prove that all the literals in Γ'_i hold and the disjunction equalities in *eqs* can be proved from Γ_i. The judgement $\Gamma_1, \ldots, \Gamma_n \vdash_{NO} (prf_i, sons) : \mathit{False}$ holds if given an environment $\Gamma_1, \ldots, \Gamma_n$ of the joint theory $T_1 + \ldots + T_n$, the proof $(prf_i, sons)$ allows to exhibit a contradiction, *i.e.*, *False*. Suppose that proof prf_i establishes a judgement of the form $\Gamma_i \vdash_{T_i} prf_i : (\Gamma'_i, eqs)$. If the list *eqs* is empty, we have a proof that Γ_i is contradictory and therefore the joint environment $\Gamma_1, \ldots, \Gamma_n$ is contradictory and the judgement holds. An important situation is when the list is always a singleton. This corresponds to the case of convex theories for which the Nelson-Oppen algorithm never perform case-splits. In the general case, we recursively exhibit a contradiction for each equality $(x_k = y_k)$ using the kth proof of *sons*, *i.e.*, $sons[k]$ for a joint environment $(\Gamma_1[j \mapsto x_k = y_k], \ldots, \Gamma'_i, \Gamma_n[j \mapsto x_k = y_k])$ enriched with the equality $(x_k = y_k)$. For completeness, the index j used to store the equality $(x_k = y_k)$ should be fresh. The judgement holds if all the branches of the case-split over the equalities in *eqs* reach a contradiction.

3.4 Proof Checking and Generation for EUF

In this section we introduce a proof system and checker for EUF and present an overview of the proof-producing procedure. We then propose an overview of an alternative EUF proof system. After preprocessing and purification, EUF formulas can be encoded with the following types:

```
type var = int
type term = Var of var | Apply of var * var list
type formula = Eq of term * term | Neq of term * term
```

The fact that terms are purified and flat is an invariant maintained by the proof-producing procedure.

Proof System. A proof is a list of commands executed in sequence. Each command operates on the state of the checker, which is a pair (Γ, eq). The assumption set Γ is a mapping from indices to assumptions, written $\Gamma(i) \mapsto a = b$, and eq is the *current equality*, *i.e.*, the last one we proved. Each command corresponds to an axiom or a combination of axioms of the EUF theory. The syntax of the commands is the following:

```
type command =
 | Refl of term  | Trans of index * bool
 | Congr  of index * position * bool | Push index
```

The semantics is given by rules of the form $(\Gamma, eq) \xrightarrow{\texttt{cmd}} (\Gamma', eq')$ where (Γ', eq') is the state obtained after executing the command `cmd` from the state (Γ, eq). The Boolean s in `Trans` and `Congr` commands make explicit symmetry: if $\Gamma(i) \mapsto t = t'$ then we have $\Gamma(i)^{true} \mapsto t' = t$ and $\Gamma(i)^{false} \mapsto t = t'$.

$$\frac{}{\Gamma, . = . \xrightarrow{\texttt{Refl}(y)} \Gamma, y = y} \quad \frac{\Gamma(i)^s \mapsto t = t'}{\Gamma, x = t \xrightarrow{\texttt{Trans}(i,s)} \Gamma, x = t'} \quad \frac{\Gamma' = \Gamma[i \mapsto x = t]}{\Gamma, x = t \xrightarrow{\texttt{Push(i)}} \Gamma', x = t}$$

$$\frac{\Gamma(i)^s \mapsto a_p = a'_p}{\Gamma, x = f(a_0..a_p..a_n) \xrightarrow{\texttt{Congr}(i,p,s)} \Gamma, x = f(a_0..a'_p..a_n)}$$

The command $\texttt{Refl}(y)$ corresponds to the reflexivity axiom and initialises the current equality with the tautology $y = y$, whatever the previous equality. Subsequent commands will then rewrite the right hand side of this equality. The command $\texttt{Trans}(i, s)$ updates the right hand side of the current equality. If we can prove that $x = t$ (current equality) and we know that $t = t'$ (equality indexed by i) then we can deduce $x = t'$. The command $\texttt{Congr}(i, p, s)$ rewrites a sub-term of the right hand side. In any given context if we can prove $x = f(y)$ (current equality) and we know that $y = z$ (equality indexed by i) then we can deduce $x = f(z)$ and make it the new current equality. The parameter p is used to determine where to rewrite. The command $\texttt{Push}(i)$ is used to update the assumption set Γ with the current equality $x = t$, creating a new context $\Gamma' = \Gamma[i \mapsto x = t]$ to be used to evaluate the next commands. It allows us some factorisation of sub-proofs and is mandatory to keep the terms flat.

The rules below detail the transitive closure of the previous relation, explaining how to evaluate a list of commands prf.

$$\frac{}{\Gamma', eq' \xrightarrow{nil}_* \Gamma', eq'} \quad \frac{\Gamma, eq \xrightarrow{cmd} \Gamma', eq' \quad \Gamma', eq' \xrightarrow{prf}_* \Gamma'', eq''}{\Gamma, eq \xrightarrow{cmd::prf}_* \Gamma'', eq''}$$

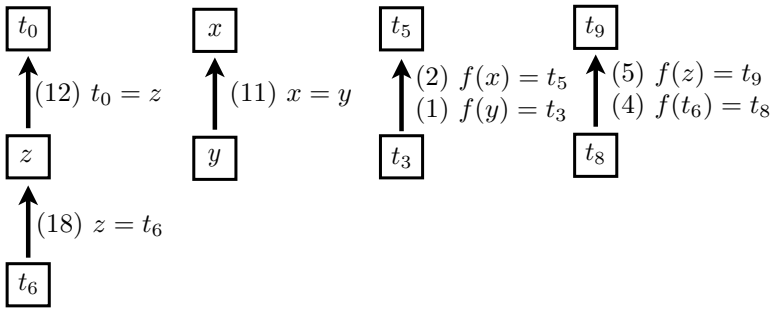
The relation $\Gamma \vdash_{EUF} prf_{EUF} : (\Gamma', eqs)$ implements the theory specific judgement seen in Section 3.3.

$$\frac{\Gamma, z = z \xrightarrow{prf}_* \Gamma', x = y}{\Gamma \vdash_{EUF} \texttt{EUF\_Eq}(prf) : (\Gamma', [x = y])} \quad \frac{\Gamma, z = z \xrightarrow{prf}_* \Gamma', x = y \quad \Gamma(i) \mapsto x \neq y}{\Gamma \vdash_{EUF} \texttt{EUF\_False}(i, prf) : (\Gamma', nil)}$$

Suppose that we obtain a state $(\Gamma, x = y)$ after processing a list prf of commands. The proof $\texttt{EUF\_False}(i, prf)$ deduces a contradiction if $\Gamma(i) \mapsto x \neq y$ and the proof $\texttt{EUF\_Eq}(prf)$ deduces the equality $x = y$.

Proof Generation. Proof generation follows closely [16] where the proof-producing prover maintains a *proof forest* that keeps track of the reasons why two nodes are merged. Besides the usual *merge* and *find* operations, the data structure has a new operator $\texttt{explain}(a, b, \texttt{forest})$ which outputs a *proof* that $a = b$ based on `forest`. In our case, proofs are lists of commands, while in the original approach they were unsatisfiable unordered sets of assumptions.

We show below the proof forest corresponding to the example of Section 2.2. Trees represent equivalence classes and each edges is labelled by assumptions. The prover updates the forest with each *merge*. Two distinct classes can be merged for two reasons: an equality between variables is added or two terms are equal by congruence.



Suppose for example that the problem contains (2) $f(x) = t_5$ and (1) $f(y) = t_3$ and we add the equality (11) $x = y$. First, we have to add an edge between x and y, labelled by the reason of this merge, *i.e.*, assumption (11). Then, we have to add an edge between t_3 and t_5, and label it with the two assumptions that triggered that merge by congruence, *i.e.*, (1) and (2).

To output a proof that two variables are equal, we travel the path between the two corresponding nodes, and each edge yields a list of commands. An edge labelled by an equality corresponds to a simple transitivity: $t_6 \xrightarrow{(18)} z$ yields

$$[\texttt{Trans}(18, true)]$$

An edge labelled by two equalities makes use of the congruence: $t_3 \xrightarrow{(1)(2)} t_5$ yields

$$[\texttt{Trans}(1, false); \texttt{Congr}(11, 1, true); \texttt{Trans}(2, true)]$$

If the equality that triggered the congruence was *discovered* by EUF and not an assumption, we have to explain it, and then update the environment accordingly, using the `Push` command. This could lead to factorisation issues. We can ensure that any intermediate result is checked only once during proof-producing, but this may not be enough. We may want to ensure that any *connection* between variables, reflected by an edge in the proof forest, is only checked once, but this is trickier.

Alternative EUF Checker. We now briefly expose a second EUF proof verifier, which aims at maximum factorisation of subproof. The proof forest maintained by our proof-producing prover is a compact array-based structure, on which it is very easy and efficient to check equalities of variables while sharing subproofs. Arrays may be a sensitive data structure depending on the proof verification context. In Coq for example, only functional style arrays are provided, and they may not behave like traditional arrays. But if our checker is able to efficiently manipulate arrays, the proof forest itself is a fine proof. To check an equality $a = b$, the checker only has to travel between the trees to ensure that the nodes corresponding to the variables a and b are in the same equivalence class, *i.e.*, have the same root. During this computation of the root of a node, any node on the path can store that information. Once the checker is aware of the root of a node, it doesn't have to compute it again, hence a high rate of subproof sharing if we can ensure that any edge in the forest is only crossed once. The forest being linear in the number of assumptions, we have achieved linear complexity in checking. The checker algorithm mimics the initial congruence closure algorithm, without any decision making or reordering of the forest. We take the forest for granted and fail as soon as it does not reflect any needed equality. In particular the choice of the roots is made by the prover, and the checker relies on it. With this simplification comes the reduction of algorithmic complexity.

A Nelson-Oppen compatible EUF checker needs to be incremental. We need to check equalities between variables, then to assert equalities discovered by other theories, and then to check more equalities. Fortunately, the proof forest obtained at the end of a Nelson-Oppen cycle reflects its history, *i.e.*, a path between two variables only uses equalities asserted or discovered earlier. We can compute the *temporary root* of a node, instead of its real root, by stopping as soon as an edge in the forest is not labelled by an available assumption. We can then check early equalities without breaking any temporal constraint, and *unroot* nodes as soon as a new assumption is available.

This second proof system is checked using different data structures, namely arrays. Depending on the tools available, one could choose either a very efficient checker, or a checker that does not rely on arrays. The switch between checkers is easy as long as both implement the primitives needed by the Nelson-Oppen checker.

3.5 Proof Checking and Generation for LRA

In this section we introduce the proof system for LRA and describe a proof-producing procedure. Literals are of the form $e \bowtie 0$ with e a linear expression manipulated in (Horner) normal form and $\bowtie \in \{\geq, >, =\}$.

Proof System. For linear real arithmetic, Farkas' lemma provides a sound and complete notion of proof that a conjunction of linear constraints is unsatisfiable [17, Corollary 7.1e]. The following proof system allows to prove an inequality with a list of commands (a Farkas proof). Each command is a pair $\texttt{Mul}(c, i)$ with c a coefficient (in type $\texttt{Z}$) and i the index of an assumption in the current assumption set. Such a command is used below in a judgement $\Gamma \Vdash e \bowtie 0 \xrightarrow{\texttt{Mul}(c,i)} e' \bowtie' 0$ with $\bowtie$ and $\bowtie'$ in $\{\geq, >\}$. $\Gamma \cup \{e \bowtie 0\}$ is the current set of assumptions and $e' \bowtie' 0$ is the new inequality that is deduced.

$$\frac{c > 0 \quad \Gamma(i) \mapsto e' \geq 0}{\Gamma \Vdash e \bowtie 0 \xrightarrow{\texttt{Mul}(c,i)} (c[*]e'[+]e) \bowtie 0} \qquad \frac{\Gamma(i) \mapsto e' = 0}{\Gamma \Vdash e \bowtie 0 \xrightarrow{\texttt{Mul}(c,i)} (c[*]e'[+]e) \bowtie 0}$$

$$\frac{c > 0 \quad \Gamma(i) \mapsto e' > 0}{\Gamma \Vdash e \bowtie 0 \xrightarrow{\texttt{Mul}(c,i)} (c[*]e'[+]e) > 0}$$

The operators $[*], [+], [-]$ model the standard arithmetic operations but maintain the normalised form of the LRA expressions. The previous rules follow the standard sign rules in arithmetic: for example, if e' is non-negative we can add it c times to the right part of the inequality $e \bowtie 0$, assuming c is strictly positive.

Contrarily to the EUF checker of Section 3.4, the LRA checker does not change the assumption set Γ; this difference motivates the use of a different type of judgement. It is completely transparent to the Nelson-Oppen checker as long as the judgement $\Gamma_i \vdash_{T_i} prf_i : (\Gamma_i^{'}, eqs)$ is implemented.

The transitive closure of the previous relations allows to prove an inequality with a list of command. It is formalised with the following rules.

$$\frac{}{\Gamma \Vdash nil : 0 \geq 0} \qquad \frac{\Gamma \Vdash (c_1 :: \cdots :: c_{n-1}) : e \bowtie 0 \quad \Gamma \Vdash e \bowtie 0 \xrightarrow{c_n} e' \bowtie' 0}{\Gamma \Vdash (c_1 :: \cdots :: c_{n-1} :: c_n) : e' \bowtie' 0}$$

A LRA proof is then either a proof of $0 > 0$ given by a list of commands or a proof of $x = y$ given by two lists of commands (one for $x - y \geq 0$ and one other for $y - x \geq 0$).

```
type LRA_proof =
 | LRA_False of command list | LRA_Eq of command list * command list
```

$$\frac{\Gamma \vdash l : 0 > 0}{\Gamma \vdash_{LRA} (\texttt{LRA\_False}(l)) : (\Gamma, nil)} \qquad \frac{\Gamma \vdash l_1 : e \geq 0 \quad e = x[-]y \quad \Gamma \vdash l_2 : [-]e \geq 0}{\Gamma \vdash_{LRA} (\texttt{LRA\_Eq}(l_1, l_2)) : (\Gamma, [x = y])}$$

Proof Generation. In order to produce Farkas proofs efficiently, we can use the Simplex algorithm used in Simplify [10]. This variant of the standard linear programming algorithm does not require all the variables to be non-negative, and directly handles inequalities (strict or not) and equalities. Each time a contradiction is found, one line of the Simplex tableau gives us the expected Farkas coefficients. The algorithm is also able to discover new equalities between variables. In this case again, the two expected Farkas proofs are read from the current tableau, up to trivial manipulations.

4 Experiments

For our approach to be viable we first need to make sure that proof generation is feasible. For the moment, our goal is not to evaluate the proof verifier; hence, to get an idea of what we can expect at best we used a high-performance solver instead of a solver complying to the proof systems prensented in Sections 3.4 and 3.5. For this reason we were able to test proof generation for linear integer arithmetic, whose proof system is left as further work.

Prototype. The SMT-LIB 2.0 standard defines scripts to be run by solvers. First, one declares the logic used, the types of the terms, then `asserts` formulas and checks for satisfiability with a `check-sat` command. The standard also defines utility commands to obtain more than a verdict from the solver. A solver can implement a `get-model` command, which output a valuation of the variables validating a satisfiable formula, and a `get-unsat-core` command, which output an unsatisfiable subformula. Our scheme would benefit from a `get-conflict-clauses` command, to obtain the conflict clauses discovered during the search, but we can already use `get-model` and `get-unsat-core` to emulate the simple search described in Section 2.1, with a SAT solver to discover models of the propositional abstraction and a SMT solver to obtain the unsatisfiability cores of formulas corresponding to models. Once the conflict clauses have been discovered, we can build their proofs using the proof-producing prover of our choice.

We have implemented our proof scheme in OCaml, the OCaml programme being in charge of the abstraction and the communication with the SMT-LIB 2.0 compatible, off-the-shelf SAT and SMT solvers (we chose Z3 for both). We have isolated several parts of this lazy SMT loop and distinguished accordingly four times of importance:

- the time spent solving the propositional abstraction and the conflict clauses, and obtaining the propositional models;
- the time spent obtaining the unsatisfiability cores from the models;
- the time spent obtaining the propositional proof of unsatisfiability;
- the time spent obtaining the proofs of the conflict clauses.

The sum of these four times is the proof *generation time*. We estimated these times by relaunching Z3 on the scripts generated by our OCaml programme. We do not take into account

the running time of our OCaml programme, whose only part was to make the SAT solver and the SMT solver communicate.

We have launched this proof-producing prover on SMT-LIB benchmarks to measure the times described earlier and the number of conflict clauses we discovered for each benchmark. We compared these measures with the time of a direct run of Z3 on the same benchmark, referred to as *direct solve time*, to understand the overhead induced by our scheme. We also counted the number of atoms of each conflict clause to evaluate the stress put on the multi-theory conjunctions solver.

We call *overhead factor* the number obtained through the following division:

$$\frac{\text{generation time}}{\text{direct solve time}}$$

Results. We used 574 unsatisfiable unquantified formulas from the SMT-LIB benchmarks, combining uninterpreted functions and linear real arithmetic (QF_UFLRA) or linear integer arithmetic (QF_UFLIA). Eight of the benchmarks hit timeout at 1000 seconds. They belong to the same category (QF_UFLIA/wisas). The only 3 benchmarks with more than 2000 conflict clauses, and which took the longest time to prove, belong to that category too. We believe that theory-propagation is needed to solve them efficiently. As soon as theory-propagation can be encoded by conflict clauses it is expressible in our proof system but would require a tighter integration with a SMT solver.

To evaluate the overhead factor of our approach we sort the benchmarks by overhead factor and draw in the right-hand side graphic of Figure 3 a point by benchmarks, with on Y the overhead factor and on X the benchmark index in the list of benchmarks. On the left-hand side graphic we do the same with the proof generation time. For 2/3 of the benchmarks the overhead

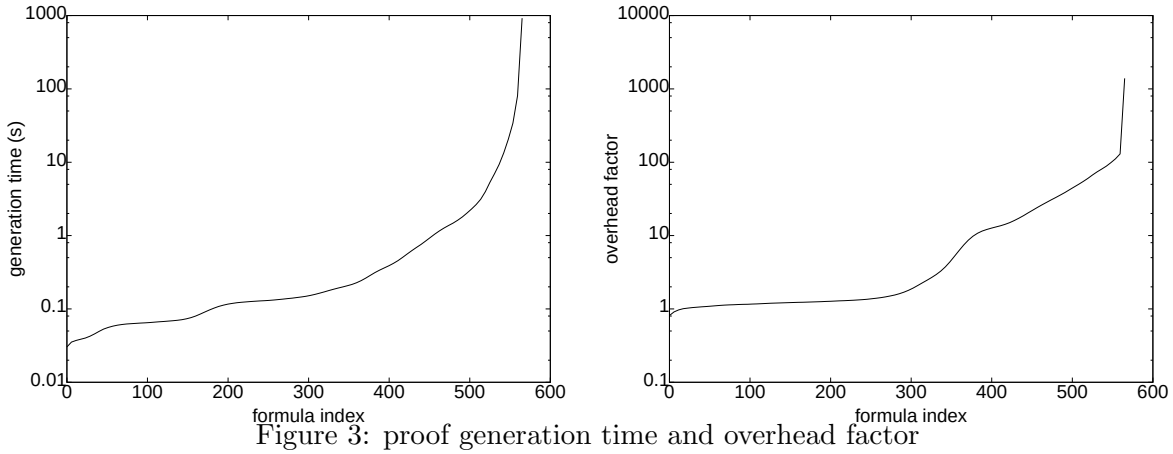


Figure 3: proof generation time and overhead factor

of the generation time w.r.t the solving time is less 10. For only 3%, the overhead climbs up to more than 100. For certain applications such as interactive theorem proving, wall clock is the critical factor not the overhead. If we only consider benchmarks that take more than a tenth of second to be solved, 4% have a overhead factor greater than 100. These cases represent 1.5% of the whole dataset.

Looking only at the generation time, 91% of the proofs are generated in less then 3 seconds, 96% in less then 30 seconds. Maybe surprisingly, for some benchmarks the generation time is inferior to the direct solve time, resulting in an overhead factor inferior to 1. These are

benchmarks solved by our prototype without any conflict clause, the abstraction being faster to solve and prove than the initial formula.

Overall, proof generation went quite well, considering how naive our prototype is. We can expect the overhead factor to vary less with each theory reasoning we take into account; but with only conflict clauses and no preprocessing, a lot of formulas can be certified in a reasonable amount of time.

For each benchmark, the number of conflict clauses vary between 0 (for 326 benchmarks of the QF_UFLRA category) and 29873 (only 3 benchmarks have more than 2000 clauses), the mean being 318.5 conflicts by benchmark and 86% of the benchmarks raising less than 100 conflicts. The mean size of the conflicts is 5.6 atoms by conjunction; therefore, we expect the proof generation of the conflict clause to amount for a little part of the whole generation time. In Figure 4 we consider the percentage of the generation time spent proving the conflict clauses. In 84% of the benchmarks the proof generation of the conflict clauses amounts for less than 10%

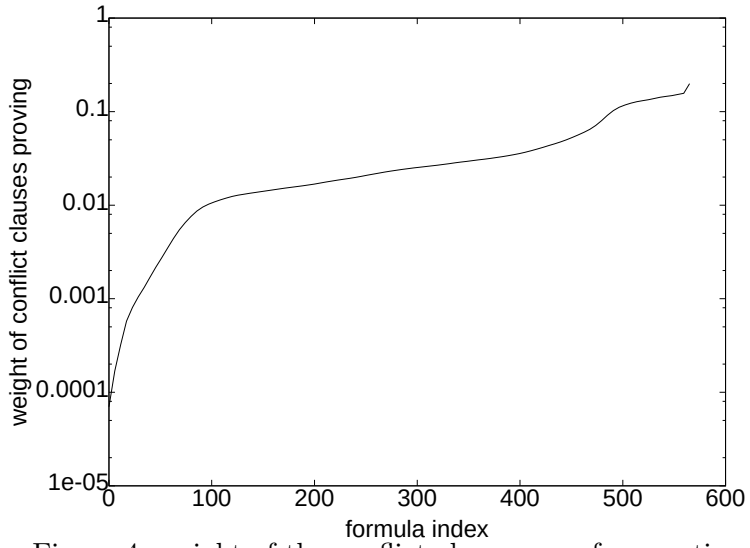


Figure 4: weight of the conflict clauses proof generation

of the generation time, and at most it amounts for less than 20% of the generation time. For this reason it seems that the proving multi-theory prover is not the bottleneck of our process, and we can focus on the quality of the proofs rather than the efficiency of the prover. Overall, once we have reduced a T-conjunction to its unsat-core, the remaining formula is very short and easy to prove.

5 Related Work

For his Proof Carrying Code framework, Necula has pioneered the area of proof-generating decision procedures [14]. In his Touchstone theorem prover [15], Necula needed to derive complete proof terms in a unified language. In our approach, each decision procedure comes with its own proof language thus allowing to choose the level of details to be put in the proofs. Several authors have examined EUF proofs [8, 16]. They extend a pre-existing decision procedure with proof-producing mechanisms without degrading its complexity and achieving a certain level of irredundancy. However, their notion of proof is reduced to unsatisfiable cores of literals rather than proof trees. Our proof generation builds on such works to produce detailed explanations.

Like several modern SMT solvers (CVC3, VeriT), the solver Z3 has its own proof language [6]. It contains a lot of rules reflecting its internal reasoning with different levels of precision, some rules detailing each computation step, some others accounting for complex reasoning with no further details. Our approach advocates a strict discipline in the way the proof is conducted but simplifies its proof-checking. Moreover, we believe that SMT solvers could generate proofs in our proof system without too much hassle when certain optimisations are disabled.

Previous work has been devoted to reconstruct SMT solvers proofs in proof assistants. McLaughlin et al. [13] have combined CVC Lite and HOL light for quantifier-free first-order logic with equality, arrays and linear real arithmetic. Ge and Barrett have continued that work with CVC3 and have extended it to quantified formulas and linear integer arithmetic. This approach highlighted the difficulty for proof reconstruction to compare to straightforward implementation of decision procedures in HOL. Independently Fontaine et al. [12] have combined haRVey with Isabelle/HOL for quantifier free first-order formulas with equality and uninterpreted functions. Their scheme includes Isabelle solving of EUF sub-proof with hints provided by haRVey. Our EUF proof system is more detailed and does not require any decision on the checker side. Böhme and Weber [4] have built a proof reconstruction of Z3 proof in the theorem provers Isabelle/HOL and HOL4. Their implementation is particularly efficient but their fine profiling shows that a lot of time is spend re-proving sub-goals for which the Z3 proof does not give sufficient details.

6 Conclusion and Perspectives

We have presented a proof system for multi-theory unquantified first-order formulas that relies on theory-specific proofs. We have developed uninterpreted functions and linear real arithmetic checkers, and combined them using a Nelson-Oppen checker. The proof format of any theory can be changed as long as a checker is provided, with no modification of the combination scheme. We have examined feasibility of proof generation based on state-of-the-art SMT solvers, and implemented simple proof-producing provers to test proof generation for our EUF and LRA proof systems and combinations of them. Our prover use an extended Union-Find algorithm [16] for EUF and a Simplex algorithm [10] for LRA. The checkers for EUF, LRA and the generic Nelson-Oppen combination have been developed and proved in Coq to provide a new reflexive decision procedure.

As further work we intend to instantiate further the framework and examine checkers and proof systems for non-convex theories such as the theory of linear integer arithmetic and the theory of arrays. The Nelson-Oppen verifier is generic enough to handle such theories but we still need to design specialised checkers and examine proof generation. The experiments have shown that handling conflict clauses is not always enough to solve formulas in a reasonable time with a reasonable amount of resources, and we need to explore other kinds of theory reasoning to shorten the proof search. Closer interaction with SMT solvers and access to theory propagation decisions would be very beneficial for our proofs because theory propagation can readily be encoded in our proof system.

References

[1] C. Barret, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0, 2010.

[2] C. Barrett and C. Tinelli. CVC3. In *Proc. of CAV 2007*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[3] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4(2-4):75–97, 2008.

[4] S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In *Proc. of ITP 2010*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.

[5] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In *Proc. of CADE 2009*, LNCS. Springer, 2009.

[6] L. M. de Moura and N. Bjørner. Proofs and Refutations, and Z3. In *Proc. of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants*, volume 418. CEUR-WS.org, 2008.

[7] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[8] L. M. de Moura, H. Rueß, and N. Shankar. Justifying equality. *ENTCS*, 125(3):69–85, 2005.

[9] L. M. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proc. of CADE'02*, volume 2392 of *LNCS*, pages 438–455. Springer, 2002.

[10] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[11] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, 2006.

[12] P. Fontaine, J-Y. Marion, S. Merz, L. P. Nieto, and A. F. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In *Proc. of TACAS 2006*, volume 3920 of *LNCS*, pages 167–181. Springer, 2006.

[13] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. *ENTCS*, 144(2):43–51, 2006.

[14] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.

[15] G. C. Necula and P. Lee. Proof generation in the Touchstone theorem prover. In *Proc. of CADE 2000*, volume 1831 of *LNCS*, pages 25–44. Springer, 2000.

[16] R. Nieuwenhuis and A. Oliveras. Proof-producing congruence closure. In *Proc. of RTA 2005*, volume 3467 of *LNCS*, pages 453–468. Springer, 2005.

[17] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1998.

[18] Princeton University. http://www.princeton.edu/ chaff/zchaff.html.

[19] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *Elec. Proc. of ISAIM 2008*.

[20] Allen Van Gelder. Verifying propositional unsatisfiability: Pitfalls to avoid. In *Proc of SAT'07*, Lisboa, Portugal, 2007.

[21] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *Proc of ISAIM'08*, Fort Lauderdale, 2008. `http://isaim2008.unl.edu/index.php?page=proceedings`.

[22] Lintao Zhang. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proc. of DATE 2003*, pages 10880–10885, 2003.