Simplification of Boolean Affine Formulas

Paul Feautrier

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Simplification of Boolean Affine Formulas

Paul Feautrier

N° 7689 — version 1

initial version July 2011 — revised version Juillet 2011

Domaine 2

ISSN 0249-6399 ISRN INRIA/RR--7689--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Simplification of Boolean Affine Formulas

Paul Feautrier *

Domaine : Algorithmique, programmation, logiciels et architectures
Équipe-Projet COMPSYS

Rapport de recherche n° 7689 — version 1 — initial version July 2011 —
revised version Juillet 2011 — 16 pages

Abstract: Boolean Affine Formulas, in which affine inequalities are combined by boolean connectives, are ubiquitous in computer science : static analysis, code and hardware generation, symbolic model checking and many other techniques use them as a compact representation of large or infinite sets. Common algorithms tend to generate large and highly redundant formulas, hence the necessity of a simplifier for keeping the overall complexity under control. Simplification is a difficult problem, at least as hard as SMT solving, with a worst case complexity exponential in the number of affine inequalities. This paper proposes a new method, based on path cutting in Ordered Binary Decision Diagrams, which is able to take advantage of any regularity in the subject formula to speed up simplification. The method has been implemented and was tested on benchmarks from several application domains.

Key-words: Simplification, affine inequalities, boolean expressions, binary decision diagrams, non-convex polyhedra

* Compsys, INRIA, ENS-Lyon, LIP, UMR 5668 CNRS, UCB-Lyon

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Simplification des expressions booléennes affines

Résumé : Les expressions affines booléennes, qui combinent des inégalités affines à l'aide d'opérateurs booléens, apparaissent dans de nombreux domaines de l'informatique comme l'analyse statique, la génération de code, la synthèse matérielle et le *model checking* symbolique. Elles permettent de représenter de façon compacte des ensembles très grands et même infinis. Les algorithmes qui permettent de les manipuler ont tendence à construire des formules fortement redondantes et de taille croissante ; il est donc nécessaire d'utiliser un simplifieur si l'on veut en maintenir la complexité dans des limites raisonables. La simplification est un problème difficile, au moins aussi difficile que le test de satisfiabilité, et donc de complexité exponentielle dans le pire cas. Cet article propose une nouvelle méthode de simplification, basée sur l'analyse des chemins dans un diagramme de décision ordonné. Cette méthode est capable d'exploiter les régularités d'une formule pour en accélérer la simplification. L'algorithme a été implémenté en Java et testé sur une série d'exemples en provenance de divers domaines d'application.

Mots-clés : Simplification, inégalités affines, expressions booléennes, diagrammes de décision, polyèdres non convexes

Simplification of Boolean Affine Formulas

Paul Feautrier

July 19, 2011

Abstract Boolean Affine Formulas, in which affine inequalities are combined by boolean connectives, are ubiquitous in computer science : static analysis, code and hardware generation, symbolic model checking and many other techniques use them as a compact representation of large or infinite sets. Common algorithms tend to generate large and highly redundant formulas, hence the necessity of a simplifier for keeping the overall complexity under control. Simplification is a difficult problem, at least as hard as SMT solving, with a worst case complexity exponential in the number of affine inequalities. This paper proposes a new method, based on path cutting in Ordered Binary Decision Diagrams, which is able to take advantage of any regularity in the subject formula to speed up simplification. The method has been implemented and was tested on benchmarks from several application domains.

1 Introduction and Motivation

1.1 Boolean Affine Formulas and their Use

An affine atom is a strict $f(\vec{x}) > 0$ or non-strict inequality $f(\vec{x}) \geq 0$, where f is an affine form $f(\vec{x}) = c.\vec{x} + d$. The variables (or unknowns) $\vec{x}$ may take rational or integer values. Since the coefficients c and d must be represented in a computer memory, they are rational, and can be assumed, without loss of generality, to be integers. Boolean Affine Formulas are constructed from affine atoms by the usual logical connectives, $\wedge, \vee, \neg$ and maybe others. A boolean affine formula in n variables may be taken as the representation of a set of points in $\mathbb{Q}^n$ or $\mathbb{Z}^n$. This set is a polyhedron, which may be convex or not.

Boolean affine formulas are found in many fields of computer science, including static program analysis [5], symbolic model checking [15], dependence analysis and scheduling [10], code generation [2, 3] and hardware synthesis [1]. Usually, the authors take pains to restrict their formulas to conjunctions only, or, equivalently, to convex polyhedra, at the price of conservative approximations. Most analysis algorithms are iterative. As the iteration progress, formulas have a tendency to grow without limits, hence the necessity of systematic simplifications. Here, simplification is understood as the removal of redundant atoms, even if this entails an increase in the complexity of the boolean skeleton of the formula. This is especially important in the case of hardware synthesis : while the cost of a boolean connective is a few transistors, the evaluation of an affine atom needs adders, multipliers, and possibly several clock cycles.

Boolean affine simplification is a difficult problem. If the simplifier is complete, then an unsatisfiable formula must simplify to **false**, hence simplification

*Compsys, INRIA, ENS-Lyon, LIP, UMR 5668 CNRS, UCB-Lyon

is at least as difficult as satisfiability testing. Furthermore, a boolean formula, in which atoms are logical variables, can always be transformed into an equivalent boolean affine formula by replacing each logical variable p_i by the inequality $x_i \geq 0$. Hence, there is small hope of finding a simplification algorithm of less than exponential worst case complexity. One can either trade completeness for efficiency, or construct algorithms that take advantage of the peculiarities of the subject formula to reduce complexity. A similar phenomenon occurs when boolean formulas are represented by Reduced Ordered Binary Decision Diagrams [4]. ROBDDs are a canonical representation, and hence an unsatisfiable formula is represented by an ROBDD with only one false leaf. However, the ROBDD for an arbitrary boolean formula may have exponential size. ROBDD are interesting because most of the formulas one encounters in practice have small representations.

This paper is organized as follows : the next section describe notations and present some related works. A new approach is proposed in Sect. 2. The basic idea is that simplifications are associated to unfeasible paths in the associated OBDD. While still of exponential complexity in the worst case, this method can take advantage of structural properties of the subject formula to be much faster in practical situations. Sect. 3 reports on a preliminary implementation and benchmarks.

1.2 Notations

Logical formulas are written using the usual propositional connectives $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ in that order of precedence. Another useful connective is the ternary **if** x **then** y **else** z, written **ite**(x, y, z), which has the truth value of y if x is true, and that of z if x is false.

In the present work, one is given a fixed list of *atoms* $a_1, \ldots, a_n$, which are affine inequalities :

$$a(\vec{x}) = \sum_{i=1}^{m} c_i x_i + c_0 \succ 0. \tag{1}$$

The c_i can be taken as integers, and the comparator $\succ$ can be restricted, without loss of generality, to $>$ or $\geq$. The arithmetic variables $x_1, \ldots x_m$ may be integral or rational. Notice that the negation of an affine atom is an affine atom. Also, when the x_i are integral, each inequality can be reduced to lowest terms : if g is the gcd of the c_i, the above inequality is equivalent to :

$$\sum_i \frac{c_i}{g} x_i + \lfloor c_0/g \rfloor \succ 0. \tag{2}$$

1.3 Related Work

The simplification of pure Boolean formulas has been the subject of much research since the seminal work of W. V. O. Quine. For an extensive treatment and applications to logical synthesis, see de Micheli's textbook [7].

The problem of simplifying conjunctive boolean affine formulas is well known and has been much studied, especially in the context of Fourier-Motzkin elimination. A first approach is based on the fact that a_n is redundant in $a_1 \wedge \ldots \wedge a_n$ iff $a_1 \wedge \ldots \wedge a_{n-1} \wedge \neg a_n$ is unfeasible. In the second approach, one notices that the formula to be simplified defines a polyhedron, which can be represented without

redundancy as the convex hull of its vertices and rays. This representation can be computed by several algorithms, as for instance the Chernikova algorithm, and a second application of the same algorithm reconstructs a non redundant system of equivalent constraints. Which method is faster depends on the shape of the polyhedron.

In the general case, one says that a_n can be eliminated from $f(a_1, \ldots, a_n)$ if there exists a formula g which does not depends on a_n such that

$$\forall \vec{x} : f(a_1, \ldots, a_n) \equiv g(a_1, \ldots, a_{n-1}).$$

Dillig et. al. [8] have proposed another definition of elimination : a_n can be eliminated from $f(a_1, \ldots, a_n)$ if one of the formulas $f(a_1, \ldots, a_{in-1}, \mathbf{true})$ or $f(a_1, \ldots, a_{n-1}, \mathbf{false})$ is equivalent to f. While this is clearly sound, it is not complete : the reader may care to check that the formula $\mathbf{ite}(x \geq 0, x \leq 1, x \geq -1)$ cannot be simplified in this way, i.e. that the atom $x \geq 0$ cannot be eliminated, but that this formula is equivalent to $-1 \leq x \leq 1$.

In fact, the two methods are equivalent for pure boolean atoms. If

$$f(a_1, \ldots, a_n) \equiv g(a_1, \ldots, a_{n-1}), \tag{3}$$

then

$$f(a_1, \ldots, a_n) \equiv g(a_1, \ldots, a_{n-1}) \equiv f(a_1, \ldots, \mathbf{true}) \equiv f(a_1, \ldots, \mathbf{false}).$$

This reasoning is sound, because in the boolean case, equation (3) is universally quantified over all boolean values for the atoms, and **true** is a legitimate value for a_n. In the affine case, the formula is quantified over the $\vec{x}$ variables, and (3) does not imply that $f(a_1, \ldots, \mathbf{true}) \equiv g(a_1, \ldots, a_{n-1})$ *for all values of* $\vec{x}$. Hence, the present method simplifies all formulas which are simplified by Dillig et. al., but the reverse is not true.

The work of Scholl et. al. [13] is closest to the present work : they use the same definition of simplification. They start by devising a formula which, if unsatisfiable, implies that one of the atoms can be eliminated. This formula is submitted to an SMT solver. The eventual proof of unsatisfiablility gives indications on how to build the simplified formula. The disadvantage of this method is that it necessitates a satisfiability check on a formula of almost double the size of the original formula. This drawback is also present in Dillig's method, but it can be mitigated by a detailed analysis of the subject formula.

2 Binary Decision Diagrams

It has been noticed [4] that the **if then else** ternary connective and the constants **true** and **false** are complete for Boolean algebra, since :

$$\begin{aligned} \neg\, x &= \mathbf{ite}(x, \mathbf{false}, \mathbf{true}), \\ x \wedge y &= \mathbf{ite}(x, y, \mathbf{false}), \\ x \vee y &= \mathbf{ite}(x, \mathbf{true}, y). \end{aligned}$$

A formula using only the **ite** connective can be seen as a tree whose interior nodes are decorated with atoms and whose leaves are either 0 or 1 (**false** or **true**, respectively). Such a *binary decision diagram* is ordered if, on any path from the root to a leaf, the atoms always occur in the same order. An ordered BDD is reduced if the following two rules have been applied as much as possible :

1. No node has isomorphic left and right subtrees,
2. Isomorphic subtrees are stored only once ; if this rule is applied, the diagram is no longer a tree but a DAG.

The important point is that reduced ordered BDDs (ROBDDs) are a normal form for boolean expressions : two boolean expressions which have the same truth table have isomorphic ROBDDs.

If the boolean values of all atoms are known, one can evaluate the BDD by a kind of pointer chasing algorithm : start from the root, and move to the left son if the current atom has the value **true**, or to the right son if **false**, until a leaf is reached.

Let $\mathrm{tt}(u)$ (resp. $\mathrm{ff}(u)$) be the left son (resp. right son) of node u, and let $a(u)$ be the atom which decorates u. By convention, the left node is associated to the truth of $a(u)$, and the right node is associated to its falsehood. One can extract a logical formula from a BDD, simply by converting each node to an **ite** operation :

$$\begin{aligned}\mathrm{extract}(u) &= \mathbf{ite}(a(u), \mathrm{extract}(\mathrm{tt}(u)), \mathrm{extract}(\mathrm{ff}(u))),\\ \mathrm{extract}(0) &= \mathbf{false},\\ \mathrm{extract}(1) &= \mathbf{true}.\end{aligned}$$

However, one can do slightly better by applying pattern matching rules, like :

$$\begin{aligned}\mathbf{ite}(x, y, 0) &= x \wedge y,\\ \mathbf{ite}(x, y, 1) &= x \Rightarrow y,\\ \mathbf{ite}(x, 1, y) &= x \vee y,\\ \mathbf{ite}(x, y, y) &= y.\end{aligned} \tag{4}$$

The last rule may seem to be redundant if the first reduction rule has been applied consistently. However, this may not be the case after simplification.

A *path* in a BDD is simply a word on the alphabet $\{0, 1\}$. To each path w one may associate a node $n(w)$ in the BDD : the root of the BDD is associated to the empty path ϵ, $n(w.1) = \mathrm{tt}(n(w))$ and $n(w.0) = \mathrm{ff}(n(w))$. Then , to each path, one associates a clause ψ_w by the following recursive definition :

$$\begin{aligned}\psi_\epsilon &= \mathbf{true}\\ \psi_{w.1} &= \psi_w \wedge a(n(w))\\ \psi_{w.0} &= \psi_w \wedge \neg a(n(w)).\end{aligned}$$

A path w is unfeasible if the associated clause ψ_w is unsatisfiable.

In this work, for reasons that will be clear later (see Theorem 1), reduction uses only the first rule and the diagram stays a tree. One can see that such an Ordered BDD is still a normal form for pure boolean formulas, the effect of rule 2 being only to reduce the memory footprint of the diagrams.

2.1 Simplification of OBDDs

For pure boolean formulas, the construction of an OBDD is by itself a simplification algorithm : if f and g are equivalent, and if g does not depends on

(a) : No simplification (b) : Simplification

FIGURE 1 – Order and Simplification

atom a_n, then a_n does not occur in the ROBDD for g, and therefore does not occur in the ROBDD for f since they are isomorphic. For boolean-affine formulas, further simplification can be achieved when the atoms are not independent, i.e. when some conjunction of atoms or their negation is unsatisfiable. In the context of BDDs, such clauses are associated to unfeasible paths. If a path is unfeasible, it will never be traversed when evaluating the BDD, and hence can be cut.

However, the existence of an unfeasible path in a BDD depends on the atom order. Consider for instance the toy formula $x \geq 1 \wedge x \geq 2$. Fig 1(a) represents the corresponding diagram for the order $x \geq 1$ before $x \geq 2$. The reader may care to check that this diagram has no unfeasible path. If the order is reversed (see Fig. 1(b)), the path 1.0 can be cut and the atom $x \geq 1$ can be eliminated.

This should not be taken as meaning that all $n!$ orders are to be tested :

Theorem 1 *Assume that atom a_n is redundant. If in the subject OBDD a_n occurs in the last position (i.e. just above the leaves), then a_n can be eliminated by path cutting.*

> **Proof** If the atom $a_n(x_1, \ldots, x_m)$ can be eliminated from a formula $f(a_1, \ldots, a_n)$, it means that there exists another formula $g(a_1, \ldots, a_{n-1})$ such that :
>
> $$f(a_1, \ldots, a_n) \equiv g(a_1, \ldots, a_{n-1}). \tag{5}$$
>
> Remember that rule 2 above has not be used, and hence that the BDD for f is a tree. Consider an occurrence of a_n. There exists a unique path w from the root to this occurence, which is associated to the clause ψ_w. If ψ_w is unfeasible, so are $\psi_{w.1}$ and $\psi_{w.0}$, which can be cut. Otherwise, ψ_w is a conjuction of atoms which defines a polyhedron P_w in $\mathbb{Q}^m$ or $\mathbb{Z}^m$. In P_w, all $a_1, \ldots, a_{n-1}$ have fixed truth values, so has g, and so has f by (5). If this value is **true**, then the path $w.0$ must be unfeasible and can be cut. The same argument applies to $w.1$ if the truth value is **false**. The same reasoning can be applied to all occurrences of a_n, and a_n has been eliminated. ∎

This suggest the following algorithm :
- Construct a BDD for the given formula
- For each atom in the formula :
  - Bring all occurrences of the selected atom to the last position
  - Cut all unfeasible paths from the root to a leaf throught an occurence of the selected atom
- Extract the simplified formula from the BDD

Practically, cutting the link from a node to one of its sons is done by having its father points directly to its other son.

2.2 Moving Atoms Around

The present task is to insure that each atom in turn occupies the last position in the BDD ordering. In [9], the authors study the problem of finding the best order in term of the BDD size, and propose to move atoms down one at a time until a local minimum is found. The corresponding programming is quite complex, as one has to take into account all possible relations between an occurrence of the moving atom and its two sons. The method which is used here consists in repeatedly moving the last atom to the top position, using the following algorithm.

Let B be the BDD to be modified, and let a be the atom to be brought to the top.

- Build two copies B' and B'' of B. In B', each a node is replaced by its left son. In B'', each a node is replaced by its right son.
- Build the BDD for $\mathbf{ite}(a, B', B'')$.

Clearly, if the initial BDD is ordered and if a is the last atom, then the new BDD will also be ordered, with a as the first atom. After n iterations, each atom will have occupied the last position once (unless it is removed early by the algorithm).

2.3 Heuristics and Extensions

2.3.1 Special cases

It is worthwhile to detect cases in which the simplification process can be accelerated. Firstly, if the given formula is a tautology, or is unsatisfiable, it simplifies respectively to **true** or **false**. This can be efficiently checked using any SMT solver.

Second, if the formula is a conjunction, the reader may check that the proposed algorithm reduces to negating each atom in turn and testing satisfiability. This can be implemented directly, without recourse to BDDs, with a much lower overhead. It may happen that a formula does not look like a conjunction, but is equivalent to one due to boolean cancellation. This can be detected by using an SMT solver based on semantic tableaux.

2.3.2 Semantic Tableaux

A semantic tableau [14, 6] is a systematic attempt to build a model of a formula, i.e. to find a valuation that makes the formula true. A tableau is a tree whose nodes are labeled by formulas (not to be confused with BDDs : the semantics is quite different). One starts with a one-node tree wearing the formula to be tested, and extend it by the following algorithm :

- Select a node with an unused formula,
  - If the formula is of the form $\neg\neg\phi$, extend each branch issuing from the selected node by a node wearing ϕ.
  - If the formula is of the form $\phi\wedge\psi$, extend each branch by two consecutive nodes wearing ϕ and ψ.
  - If the formula is of the form $\neg(\phi\wedge\psi)$, split each outgoing branch in two, one extended by $\neg\phi$ and the other by $\neg\psi$.
- A branch is boolean closed if it contains a formula and its negation.

– A branch is affine closed if the conjunction of its atoms is unfeasible. Since the atoms are affine, this can be checked using any Linear Program solver.

The development rules can be summarized by the following diagrams :

$$\frac{\neg\neg\phi}{\phi} \qquad \frac{\phi \wedge \psi}{\begin{array}{c}\phi\\ \psi\end{array}} \qquad \frac{\neg(\phi \wedge \psi)}{\neg\phi \mid \neg\psi}$$

There are many variations on this basic scheme. One may easily introduce rules for other boolean connectives, or work with signed formulas (see [6] for an exhaustive discussion). For instance, the signed rules for **ite** are :

$$\frac{T : \mathbf{ite}(\phi, \chi, \psi)}{\begin{array}{c|c} T : \phi & F : \phi \\ T : \chi & T : \psi \end{array}} \qquad \frac{F : \mathbf{ite}(\phi, \chi, \psi)}{\begin{array}{c|c} T : \phi & F : \phi \\ F : \chi & F : \psi \end{array}}$$

The point is that if all branches of a tableau are closed, then the formula at its root is unsatisfiable. Furthermore, if a tableau has open branches then the conjunction of the atoms in each open branch is a clause in the disjunctive normal form of its root formula. As a particular case, a tableau with just one open branch indicates a conjunctive formula, whose simplification can be accelerated as indicated above.

2.3.3 BDD Construction

Assume now that a tableau has been constructed for the subject formula. An OBBD can be constructed from the collection of its open branches. The method is based on the following tautology :

$$(a \wedge B) \vee (\neg a \wedge C) \vee D \equiv \mathbf{ite}(a, B \vee D, C \vee D).$$

This can be interpreted as follows : let a be the first atom, $a \wedge B$ be the disjunction of all branches containing a, $\neg a \wedge C$ the disjunction of all branches containing $\neg a$, and D the disjunction of all branches containing neither a nor $\neg a$. Then the associated OBDD has a as its root node, the left son being an OBDD for $B \vee D$, and the right son an OBDD for $C \vee D$. Incidentally, one sees that D is duplicated in this construction, and this is the reason why OBDDs may have a size exponential in the size of the input formula. One possible heuristics consists in selecting an a such that D is of smallest size. The implementation of this optimization is left for future work.

2.3.4 Putting Everything Together

All of this can be summarized by the following algorithm :

1. Build a tableau T_1 for the formula to be simplified, ϕ
2. If T_1 is closed, return **false**
3. Build a tableau T_0 for $\neg\phi$
4. If T_0 is closed, return **true**
5. Select the simplest of T_1 and T_0, T, i.e. the tableau with the minimum number of open branches

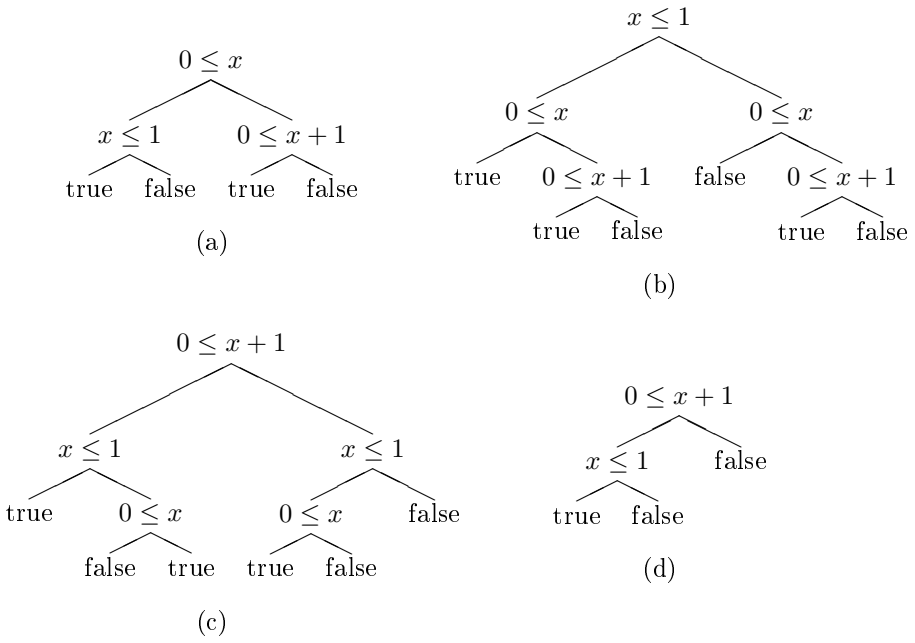

FIGURE 2 – Simplification of **ite**$(x \geq 0, x \leq 1, x \geq -1)$

6. If T has only one open branch, simplify it as in Sect. 2.3.1
7. If not, build an OBDD B for T as in Sect. 2.3.3
8. Repeat n times, where n is the number of atoms in ϕ
   - Cut unfeasible paths in B
   - Move the bottom atom of B to the top as in Sect. 2.2
9. Extract a formula ψ from B
10. Return ψ if T_1 was selected in step 5, and $\neg\psi$ if not.

As an example, Fig. 2 presents the successive stages in the simplification of the formula **ite**$(x \geq 0, x \leq 1, x \geq -1)$. Fig. 2(a) corresponds to the order $0 \leq x, x \leq 1, x \geq -1$. In (b), atom $x \geq -1$ has been moved to the bottom of the BDD. In both cases, there are no unfeasible paths. In (c), $x \geq 0$ is now at the bottom. The path from the root to the true son of the leftmost occurence of $0 \leq x$ is unfeasible and can be removed. Similarly, the path going to the true son of the other occurrence of $0 \leq x$ can be removed. Fig. 2(d) is the simplified OBDD, which represents $x \leq 1 \wedge x \geq -1$.

2.4 Quasts

A quast is similar to a BDD, with the difference that there may be many different leaves, and that the atoms are not necessarily ordered. Quasts are naturally constructed by parametric linear programming. In this case, leaves are affine expressions, and the quast represents a piecewise affine function. Other applications are to value-based dependence analysis [10] and code generation [3]. The aim of quast simplification is to create an equivalent sequential conditional (in VHDL parlance), where each leaf has only one occurence, and where the

predicates are as simple as possible. One possibility is to extract from the quast a reaching condition for each leaf, and to simplify it. One can do better by noticing that when building the predicate for the k-th leaf, one knows that all preceding predicates have tested false, and hence that the corresponding leaves are don't cares. Don't cares (here noted $\perp$) are then eliminated by rules like :

$$\mathbf{ite}(x, y, \perp) \Rightarrow y.$$

Consider the following example, drawn from a high-level synthesis application [1] :

$$\begin{array}{ll} \mathbf{ite} & (c_2 + c_3 + 1 \leq L, \\ & \mathbf{ite}(c_2 + c_4 + 1 \leq L, S_1, S_2) \\ & \mathbf{ite}(c_2 + c_4 + 1 \leq L, S_1, S_3) \end{array}$$

The formula associated to S_1 is

$$\mathbf{ite}(c_2{+}c_3{+}1 \leq L, \mathbf{ite}(c_2 + c_4 + 1 \leq L, \mathbf{true}, \mathbf{false}), \mathbf{ite}(c_2 + c4 + 1 \leq L, \mathbf{true}, \mathbf{false})),$$

and simplifies to $c_2 + c_4 + 1 \leq L$. Next, the formula for S_2 is :

$$\mathbf{ite}(c_2{+}c_3{+}1 \leq L, \mathbf{ite}(c_2 + c_4 + 1 \leq L, \perp, \mathbf{true}), \mathbf{ite}(c_2 + c4 + 1 \leq L, \perp, \mathbf{false})),$$

and simplifies to $c_2 + c_3 + 1 \leq L$. Lastly, the formula for S_3 is :

$$\mathbf{ite}(c_2 + c_3 + 1 \leq L, \mathbf{ite}(c_2 + c_4 + 1 \leq L, \perp, \perp), \mathbf{ite}(c_2 + c4 + 1 \leq L, \perp, \mathbf{true})),$$

and simplifies to **true**. The final result is :

$$\begin{array}{ll} \mathbf{ite} & (c_2 + c_4 + 1 \leq L, \\ & S_1, \\ & \mathbf{ite}(c_2 + c_3 + 1 \leq L, S_2, S_3) \end{array}$$

It is clear that the precise shape of the result will depend on the leaf order, but this point must be taken care of from outside the simplifier.

3 Experimental Results

3.1 Implementation

The algorithm of Sect. 2.3.4 has been implemented as a library of Java classes. The basic SMT solver is a home-made implementation of the Semantic Tableau method, using the signed formula variant. Rules for many logical connectives (implication, equivalence, **if then else**) have been added to supplement the basic $\wedge$ and $\neg$. This tool is used for satisfiability checking and for DNF construction.

The feasibility of a conjunctive affine formula is decided by the all-integer linear programming tool PIP [1]. PIP can handle both integer and rational variables at the turn of a switch. Basically, PIP handles only the non-strict comparator $\geq$. A strict comparison $x > 0$ is replaced by $x \geq \epsilon$ where ϵ is a new positive parameter. In that case, PIP returns the solution as a piecewise affine function of ϵ, in which one has only to make ϵ tend to zero.

1. `www.piplib.org`

name	size	before	after	time
Small				
g1g2.dat	3	2	1	0.012
g2l3.dat	5	3	2	0.015
split-centered.dat	4	3	2	0.030
dillig.dat	40	5	2	0.10
scholl2.dat	7	4	3	0.018
Quasts				
matmul-3.quast2.dat	10	4	4	0.039
poly-1.quast1.dat	142	9	3	0.14
poly-1.quast2.dat	62	8	2	0.062
poly-1.quast3.dat	131	8	4	0.16
poly-1.quast4.dat	33	6	3	0.051
poly-3.quast1.dat	18	4	1	0.023
poly-5.quast1.dat	8	4	2	0.012
poly-6.quast1.dat	11	4	2	0.033
choles-Q2.dat	2918	10	1	0.22
choles-Q4.dat	5291	11	3	0.46
choles-Q5.dat	5033	10	6	0.41
choles-Q6.dat	1196	9	1	0.12
thomr-C23.dat	4105	11	6	0.54
thomr-SX3.dat	2298	13	6	0.53
thomr-WC3.dat	3275	16	3	0.56
thomr-Z12.dat	2298	13	1	0.45
thomr-Z15.dat	1814	15	1	0.28

TABLE 1 – Experimental results

The BDD package is also home-made. It uses the standard `unique` hash-table method [9] for detecting isomorphic diagrams.

Several frontends to the library have been implemented. They differ in their input syntaxes, among which are a C-like form, the SMTLib `smt2` syntax, and the Mathematica syntax. There is also a quast simplifier with an ad-hoc syntax.

3.2 Benchmarks

The method has been tested on many benchmarks of various origins. The experimental results are given in Table 1 and 2. All source files can be found at URL `http://www.ens-lyon.fr/LIP/COMPSYS/Tools/Simple`. The size of a formula is the sum of the number of connectives and the number of atoms. The "before" and "after" columns give the number of atoms before and after simplification. These columns must be interpreted with care, as the method may introduce atom negations, or split equalities. The time for simplification is given in seconds, and was measured on an Intel Centrino 2 clocked at 2.5 GHz with 2 GBytes of memory, running Ubuntu 10.10 and the Sun 1.5 JVM. On repeating the experiments, the time variability was found to be quite large. The reported value is the best of a serie of 8 experiments, and has been rounded to two significant figures.

name	size	before	after	time
formula-026.m	19	10	8	0.60
formula-142.m	19	10	9	0.65
formula-151.m	25	13	11	1.9
formula-192.m	25	13	10	1.3
formula-107.m	27	14	13	7.1
formula-069.m	27	14	12	11
formula-147.m	29	15	11	37
formula-011.m	29	15	14	11
formula-124.m	31	16	14	28
formula-115.m	31	16	14	81
formula-227.m	33	17	11	8.6
formula-263.m	33	17	13	40
formula-094.m	33	17	12	67
formula-228.m	33	17	0	0.12

TABLE 2 – Random Formulas

The method has first been tested on a set of small examples, including those which were used to illustrate this article. The following example from Dillig et. al. [8] :

$$op = 0 \vee (op \neq 0 \wedge op = 1) \vee (op \neq 0 \wedge op \neq 1 \wedge op = 2) \\ \vee (op \neq 0 \wedge op \neq 1 \wedge op \neq 2 \wedge y \neq 0) \\ \vee (op \neq 0 \wedge op \neq 1 \wedge op \neq 2 \wedge op \neq 3)$$

is interesting in that its negation is a conjunction which can be simplified trivially. An example from Scholl et. al. [13] :

$$x \geq 0 \wedge y \geq 0 \wedge x + y > 0 \wedge x + 2y > 0$$

shows that simplification is not a convergent process : it has two distinct outcomes, none of which can be simplified further.

The method was then used to simplify quasts obtained from the code generation backend of a high level synthesis tool [1] and of an automatic parallelization tool [11]. While all other examples were solved in rationals, the quasts examples, whose variables are loop counters or array subscripts, were simplified in positive integers. Some of the formulas were large and had many atoms, but simplification was found to be easy and very effective.

Lastly, the method was applied to the random formulas generated by David Monniaux for testing a new quantifier elimination algorithm [12] (see Table 2). Since these formulas [2] are in prenex normal form, it was easy to remove their quantifier prefix and to attempt simplifying their body. Not surprisingly, these examples were much more difficult to solve and did not simplify often. It is likely that, on random examples, the algorithm runs at or near its worst case complexity. This is borne out by the observation that problems of equivalent size that have many simplifications run faster that those that do not : removing an atom divides the worst case complexity by two.

2. also available on `http://www-verimag.imag.fr/~monniaux/download/`

4 Conclusion

4.1 Contributions

The simplification method presented here is complete, and can be implemented easily using known techniques. It leverages and combines several approaches : semantics tableaux for fast detection of tautologies and unsatisfiable formulas, OBDD for boolean simplification, and a new technique, path cutting, for the final effort. It would be easy to build a parallel version of the algorithm, since the evaluation of a path in the OBDD is independent of all other paths.

While simplification is a global problem, it has been found that applying local simplifications like $\phi \wedge \mathbf{true} \equiv \phi$ or $\phi \vee \phi \equiv \phi$ is a cheap way of greatly improving performance. In the integer case, simply replacing atoms by their strengthening (2) may reduce the running time by more than a factor of 2.

The method is agnostics to the underlying theory, provided that this theory is closed under negation – that the negation of an atom is still an atom – and that it has an elementary satisfiability tester. To pursue the metaphor, it would be possible to replace atoms by *molecules* – composite formulas – thus replacing the Linear Program Solver by a full SMT solver. However, how to group atoms into significant molecules is a difficult problem, which probably can only be solved in the light of outside knowledge.

4.2 Extensions

One should not assume that the present proposal is the ultimate in simplification. The restriction that the simplified formula uses only atoms from the original precludes some interesting simplifications. Consider for instance the formula $x = 0 \vee x \geq 1$, x an integer variable. It is obviously equivalent to $x \geq 0$, but this cannot be found by the present method. What is needed here is an oracle to propose the missing atom. It can then be introduced artificially in the original, for instance as $\mathbf{ite}(x \geq 0, x = 0 \vee x \geq 1, x = 0 \vee x \geq 1)$ which can then be simplified to the expected result, provided one is careful in ordering atoms.

Since simplification is not a convergent process, there is no guarantee at present that the result is minimal. Obtaining such a guarantee may need the construction of a full simplification tree. In the same way, while logic synthesizers may be indifferent to the boolean structure of their input, in code generation applications this is not true for ordinary compilers : a post-optimization phase or more comprehensive extraction rules (4) may be indicated.

Both in the case of code generation and of hardware synthesis, simplification occurs in contexts which may allow further optimizations, for instance in the form of strength reduction.

The present implementation has not been optimized in any way and is cluttered by many different internal representations which must be inter-converted as the algorithm proceeds. A more streamlined version is needed.

Lastly, the great variability in performance is disturbing and warrants further investigations. Whether it is due to the JVM and its garbage collector, or to the operating system and its scheduler and memory manager, or even to the processor power management module is unknown at present.

Références

[1] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. Automatic generation of fpga-specific pipelined accelerators. In Andreas Koch, Ram Krishnamurthy, John McAllister, Roger Woods, and Tarek A. El-Ghazawi, editors, *Reconfigurable Computing : Architectures, Tools and Applications - 7th International Symposium, ARC 2011, Belfast, UK*, volume 6578 of *Lecture Notes in Computer Science*, pages 53–66, 2011.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, september 2004.

[3] Pierre Boulet and Paul Feautrier. Scanning polyhedra without DO loops. In *PACT'98*, October 1998.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C35(8) :677–691, August 1986.

[5] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the ACM POPL78*, 1978.

[6] Marcello D'Agostino, Dov M. Gabbay, Reiner Hähnle, and Joachim Posegga, editors. *Handbook of Tableau Methods*. Springer, 1999.

[7] Giovanni de Micheli. *Synthesis and Optimization of Digital Circuits*. MacGraw-Hill, 1994.

[8] Isil Dillig, Thomas Dillig, and Alex Aiken. Small formulas for large programs : On-line constraint simplification in scalable static analysis. In Radhia Cousot and Mathieu Martel, editors, *SAS 2010*, LNCS 6337, pages 236–252. Springer, 2010.

[9] Rüdiger Ebendt, Görschwin Fey, and Rolf Drechsler. *Advanced BDD Optimization*. Springer, 2006.

[10] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1) :23–53, February 1991.

[11] Paul Feautrier. Scalable and structured scheduling. *Int. J. of Parallel Programming*, 34(5) :459–487, May 2006.

[12] David Monniaux. Quantifier elimination by lazy model enumeration. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19*, volume 6174 of *Lecture Notes in Computer Science*, pages 585–599, 2010.

[13] Christoph Scholl, Stefan Disch, Florian Pigorsh, and Stefan Kupferschmid. Computing optimized representations for non-convex polyhedra by detection and removal of redundant linear constraints. In Kowalewski S. and Philippou A., editors, *TACAS 2009*, LNCS 5505, pages 383–397. Springer, 2009.

[14] Raymond M. Smullyan. *First Order Logic*. Dover, 1968.

[15] F. Wang. Symbolic parametric safety analysis of linear hybrid systems with BDD-like data structures. *IEEE Trans. on Software Engineering*, 31(1) :38–52, 2005.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399