UNIVERSITÉ JOSEPH FOURIER – GRENOBLE 1
SCIENCES, TECHNOLOGIE ET MÉDECINE

THÈSE

pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

en Informatique

préparée au laboratoire VERIMAG

présentée et soutenue publiquement par

Moez MAHFOUDH

le 27 Mai 2003

Sur la Vérification de la Satisfaction pour la Logique des Différences

Directeur de thèse :
Oded MALER

Membres du jury :

Anatoli IOUDITSKI,	Président
Karem A. SAKALLAH,	Rapporteur
Henrik Reif ANDERSEN,	Rapporteur
Oded MALER,	Directeur de thèse
Alessandro CIMATTI,	Examinateur
Eugène ASARIN,	Examinateur
Peter NIEBERT,	Examinateur

À ma mère,
à mon père,
à ma sœur ...

Remerciements

Je tiens à remercier Anatoli Iouditski, Professeur à l'Université Joseph Fourier, pour avoir présidé le jury de cette thèse. Je remercie aussi messieurs Karem A. Sakallah, Professeur à l'Université du Michigan, Henrik Reif Andersen, Professeur à IT University of Copenhagen, et Alessandro Cimatti, chercheur à l'Instituto per la Ricerca Scientifica e Tecnologica à Trento, Italie, pour avoir accepté de juger ce travail avec beaucoup d'intérêt et pour avoir participé au jury de cette thèse.

Je remercie aussi Eugène Asarin, Professeur à l'Université Joseph Fourier et chercheur au laboratoire VERIMAG, pour avoir toujours été disponible pour des discussions, souvent longues, concernant mes recherches, et pour avoir fait partie du jury.

De même, je remercie Peter Niebert, Maître de conférence à l'Université de Provence, pour sa participation au jury. Il a été la source dinspiration première de ce travail : Il a jeté ses bases alors qu'il était encore à VERIMAG et a continué à suivre son avancement après son départ et ce malgré l'éloignement géographique.

Je voudrais aussi exprimer ma gratitude à Oded Maler, Directeur de recherche au CNRS et chercheur au laboratoire VERIMAG, qui m'a permis de faire cette thèse. J'ai apprécié son encadrement libéral et l'autonomie qu'il m'a accordée pour faire mes recherches. J'aurai toujours le souvenir de nos discussions où nous confrontions nos visions, souvent diamétralement opposées, sur l'informatique, les sciences, le monde et la vie. *Merci Oded, merci pour tout.*

Je tiens aussi à remercier tous les chercheurs de VERIMAG et plus spécialement le directeur du laboratoire, Joseph Sifakis, ainsi que Yassine Lakhnech, Marius Bozga, Sergio Yovine, Stavros Tripakis, Saddek Bensalem, et Nicolas Halbwachs, pour leurs aides administratives ou scientifiques.

Un grand merci aussi au personnel administratif de VERIMAG et plus spécifiquement à Jeanne-Marie Colle pour ses conseils et son aide lors de mon installation à Grenoble.

Je remercie Riadh Robbana, Maître assistant à l'Université de Carthage, qui m'a permis d'établir le lien initial avec VERIMAG lorsque j'étais encore élève-ingénieur à l'Ecole Polytechnique de Tunisie.

Je remercie Yasmina, Thao, Gordon, Christos, Radu, (ex)membres de VERIMAG, pour leur amitié sincère et nos longues discussions dans les cafés de la ville où se mêlaient culture, cinéma et gamineries.

Je remercie Chaker, Ramzi, Cyril, David, Alexandre et Karim, étudiants à VERIMAG, avec qui je partageais mes pauses et des discussions sur la vie et la

météo. Merci aussi à Marcello pour les pauses café et son aide très appréciable lors la préparation de ma soutenance de thèse.

Merci à mes amis de Grenoble, spécialement Walid Dkhil et Walid Hasni, pour leur soutien et pour nos sorties en ville où nous tentions de nous changer les idées en oubliant le stress des études et des thèses.

Merci à mes amis et anciens de l'Ecole Polytechnique de Tunisie, et plus particulièrement merci à Firaz, Olfa, Anis, Amin, Hakim, Fadhel, qui ont su me rendre la vie plus agréable durant leurs séjours à Grenoble.

Merci aussi à mes amis de toujours, Badii, Karim, et Helmi, que j'ai connus sur les bancs du collège (et même bien avant) et qui, bien que vivant à Tunis, m'ont toujours fait sentir que je n'étais pas si éloigné d'eux avec leurs coup de fils, leurs emails et leurs messages. *Merci pour votre amitié et pour vos encouragements.*

Finalement, je remercie ma mère, Nebiha, mon père, Mohamed, ma sœur, Asma, et tous les membres de ma famille, pour leur amour, leur soutien indéfectible et leur aide. *Ce travail a été fait en pensant à vous, il vous est naturellement dédié. Merci pour tout. Que Dieu vous bénisse.*

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Related Work . . . 2
- 1.3 Thesis Outline . . . 3

2 Timed Formalisms **5**
- 2.1 Difference Logic . . . 5
- 2.2 Satisfiability . . . 6
- 2.3 Conjunctive Normal Form . . . 7
- 2.4 Translating from DL to MX-CNF . . . 8
  - 2.4.1 Trivial translation . . . 8
  - 2.4.2 Tseitin's translation . . . 9
  - 2.4.3 Wilson's translation . . . 12
- 2.5 Conjunctions of Difference Constraints . . . 14
  - 2.5.1 Convex timed polyhedra . . . 14
  - 2.5.2 Bounds . . . 14
  - 2.5.3 Representing convex timed polyhedra using DBMs . . . 16
  - 2.5.4 Canonical representation of DBMs . . . 17
  - 2.5.5 Conjunction of DBMs . . . 21

3 SAT Solvers **23**
- 3.1 Overview . . . 23
- 3.2 History . . . 24
  - 3.2.1 Before 1960 . . . 24
  - 3.2.2 From 1960 to 1990 . . . 24
  - 3.2.3 From 1990 . . . 25
- 3.3 Basic Algorithm . . . 25
  - 3.3.1 Description . . . 25
  - 3.3.2 Improvements to the basic algorithm . . . 26
- 3.4 Binary constraint propagation . . . 27
  - 3.4.1 Unit resolution . . . 27
  - 3.4.2 Pure literals resolution . . . 27
  - 3.4.3 Equivalent literals detection . . . 27
- 3.5 Branching heuristics . . . 27
  - 3.5.1 Maximizing satisfied clauses heuristics . . . 27
  - 3.5.2 Literal count heuristics . . . 30

3.5.3 Variable state independent heuristics 30
3.6 Conflict Directed Backtracking and Learning 31
3.6.1 Implication graph 32
3.6.2 Conflict analysis 34
3.6.3 Conflict-free learning 37
3.6.4 Backtracking 37
3.6.5 Clause recording 38
3.6.6 Restarts 38
3.7 Alternative Techniques 38
3.7.1 Resolution *à la* Davis-Putnam 38
3.7.2 Local search 40
3.7.3 Stålmarck's method 41

4 A Mixed SAT Solver 45
4.1 Overview 45
4.1.1 Solving context 45
4.1.2 Assignments 47
4.1.3 Clause rewriting 47
4.2 Simplification Rules 50
4.2.1 Boolean unit resolution 50
4.2.2 Boolean literals equivalence detection 51
4.2.3 Pure Boolean literals resolution 51
4.2.4 DBM updating 52
4.2.5 DBM analysis 52
4.3 Davis-Putnam Rules 53
4.3.1 Restricted Boolean Davis-Putnam rule application 53
4.3.2 Restricted numerical Davis-Putnam rule application . . . 54
4.4 DPLL Branching 55
4.5 Conflict Analysis 61
4.5.1 Extension of solving contexts 61
4.5.2 Changes in operations 61
4.5.3 Conflict detection and analysis 63
4.6 MX-Solver Implementation 64
4.6.1 Technical overview 64
4.6.2 Data structures 64
4.6.3 Features 66
4.6.4 Design guidelines 67

5 Timed Systems 69
5.1 Timed Automata 69
5.1.1 Flat timed automata 69
5.1.2 Translation into DL 71
5.1.3 Composition 74
5.2 Asynchronous Digital Circuits 77
5.2.1 Used model 77
5.2.2 Conversion to timed automata 80
5.3 Non-Preemptive Job-Shops 81

5.3.1 Introductory example . . . 81
5.3.2 Expression of job-shop problems . . . 82
5.3.3 Translation into DL . . . 83

6 Experimental Results **85**
6.1 Study Methodology . . . 85
6.2 Job-Shop problems . . . 86
6.2.1 Overview . . . 86
6.2.2 Instances . . . 86
6.2.3 Results . . . 89
6.3 Timed-Automata problems . . . 90
6.3.1 Overview . . . 90
6.3.2 Instances . . . 90
6.3.3 Results . . . 90
6.4 Asynchronous circuits problems . . . 92
6.4.1 Overview . . . 92
6.4.2 Instances . . . 93
6.4.3 Results . . . 93
6.5 Conclusions . . . 94

7 Conclusions **97**
7.1 Contribution . . . 97
7.2 Future Research Directions . . . 97
7.2.1 New Algorithms and Methods . . . 98
7.2.2 The MX Toolbox . . . 98

A MX Toolbox **99**
A.1 Overview . . . 99
A.2 To DL Translators . . . 99
A.2.1 Timed automata to DL . . . 99
A.2.2 Asynchronous digital circuits to DL . . . 101
A.2.3 Job-shop to DL . . . 102
A.3 DL to MX-CNF Translator . . . 102
A.4 The Mixed SAT Solver . . . 103
A.5 DL file format . . . 104
A.6 Credits . . . 105

B SCC Decomposition Algorithm **107**
B.1 Definitions . . . 107
B.2 Depth-First search . . . 107
B.3 SCC Algorithm . . . 109

List of Figures

2.1 Graph associated with Π . . . 18
2.2 Graph associated with Π' . . . 18
2.3 Example of the intersection of two convex timed polyhedra . . . 21

3.1 Implication graph . . . 33
3.2 Implication graph with a conflict cut . . . 35
3.3 Other possible cuts . . . 36

4.1 Reduction algorithm . . . 46
4.2 Example of a graph built to detect Boolean equivalences . . . 52
4.3 Data Structures in MX-Solver . . . 65

5.1 A timed gate model . . . 78
5.2 An input signal i and some corresponding delayed output signals $\{o_1, \ldots, o_4\} \in \Delta_{[2,4]}$ . . . 79
5.3 A timed automaton model of the delay element . . . 80
5.4 A sample manufacturing process . . . 81
5.5 Variables associated with the start time of tasks . . . 82

6.1 The relation between w_d and solving time for ft06 using $C7$ . . . 87
6.2 The relation between w_d and solving time for abz5 using $C7$ . . . 88
6.3 The relation between w_d and solving time for la25 using $C7$ . . . 89
6.4 A sample timed automaton . . . 91
6.5 Asynchronous 2-bit adder . . . 92

A.1 MX Toolbox architecture . . . 100

B.1 Step-by-step DFS example . . . 110
B.2 SCC example: $DFS(G)$ . . . 112
B.3 SCC example: G^T . . . 112
B.4 SCC example: $DFS(G^T)$ . . . 113

Chapter 1

Introduction

1.1 Motivation

Formal verification is a collection of techniques for ensuring that a discrete system (hardware or software) performs its intended function and does not exhibit undesired behaviors. The adjective "formal" is intended to distinguish verification from techniques known as testing, debugging or simulation that usually cover only a fraction of the admissible behaviors of the system. All verification techniques are based on an abstract mathematical model of the system to be verified, typically in a form of interacting automata.

Bounded model checking is one of the several verification methods that have been developed. It is based on writing a logical formula that characterizes the set of all possible executions of the system under consideration the formula's length is bounded by a given constant and which violates a desired property. The existence of an assignment of Boolean values to the variables which makes the formula true implies the existence of such an execution.

The problem of checking whether such an assignment exists is called the Boolean satisfiability problem (SAT), and is considered to be the "generic" hard computational problem for which no sub-exponential algorithm is known. Due to its challenging nature, SAT has been the subject of ongoing research and development leading to algorithms and tools with impressive performance. Such "SAT solvers" constitute the key ingredient in the bounded model checking methodology.

In recent years attempts have been made to extend the scope of verification technology to capture timing aspects in order to take into account phenomena such as response times, delays, durations, deadlines etc. A commonly accepted model for describing such "timed" systems is the timed automaton, an automaton augmented with auxiliary real variables called clocks that measure the time elapsed since certain transitions and hence impose timing constraints on the behaviors of the automaton. The goal of this thesis is to contribute to the applicability of bounded model-checking techniques to timed automata and other models of timed systems.

The first step in extending bounded model checking to timed systems is to define an appropriate logic which, in this case, is propositional logic extended

with timing constraints concerning the difference between two real variables. This extended logic will be called *difference logic* in this thesis.

There are two basic approaches to apply bounded model checking to timed systems. The first tests the satisfaction of the formula expressing the bounded length executions by using a general purpose constraint solver that can treat both logical and numerical constraints. The second consists in transforming that formula to a purely propositional one either by using a binary encoding of clocks, or by representing timing constraints and their interdependencies using Booleans. In both cases, the resulting formula is submitted to a Boolean SAT solver to check its satisfiability. Deduced assignments on Booleans that are originally equivalent to constraints are used to determine if the solution is not *"spurious"*, i.e. if the found Boolean solution is numerically consistent, otherwise the solution is abandoned, and the search for other Boolean assignments is continued.

This thesis explores an intermediate approach consisting in creating an appropriate solver that can directly handle formulae in difference logic. This solver does not operate on arbitrary numerical constraints and hence can take advantage of the special properties of difference constraints. Thus, the major contribution of this work is the development of mixed satisfiability checking techniques that process both Booleans and timing constraints. These techniques are inspired by those already used in Boolean SAT solvers but most of them are extended in order to take into consideration the mixed nature of the formulae.

1.2 Related Work

The term "bounded model checking" for discrete finite-state systems was first put forward in [BCC+99a, BCC+99b]. In fact, the idea had been already presented in [SS90]. Several researchers investigated the extension of SAT-based bounded model checking from finite-state systems to systems with unbounded variables representing clocks. A solver for a logic similar to difference logic was used for artificial intelligence temporal planning problems in [ACG99].

In [ACKS02, ABC+02], the authors develop an extended SAT solver to verify timed automata against temporal logic specifications. Although this work has the same motivation as this thesis, the approach it adopts is quite different: the interaction between the Boolean and the numerical part is much more limited and the numerical constraints are submitted to a general purpose solver to determine their satisfiability.

A similar approach, but with more general numerical constraints is considered in [MRS02] where numerical constraints are encoded as Booleans and decision methods are used in order to get rid of spurious solutions.

A more sophisticated approach has been proposed in [SBS02] in the context of *separation logic* (another name of difference logic). Constraints are encoded by Boolean variables along with their interdependencies. This way, the modified problem can be submitted to a propositional solver without worrying about the above mentioned spurious solutions. As pointed in [SBS02], encoding all the

dependencies might lead to an exponential blow-up of the size of the formula.

In [PWZ02], a binary encoding of clocks is used to convert the problem into a Boolean SAT instance in order to check the satisfiability of TCTL formulae. The potential application of bounded model checking to the duration calculus is described in [Fra02]. In general, the idea of applying bounded model checking to timed systems is very popular these days, and there are probably many other papers on that topic that are currently being written.

A survey of the related work on Boolean satisfiability techniques, on which our solver is based, appears in Chapter 3.

1.3 Thesis Outline

Chapter 2 presents the definitions and the notations of mathematical objects necessary for subsequent discussions. In particular, *difference logic* is defined along with *conjunctive normal form* for mixed clauses. Algorithms that convert formulae from difference logic to the mixed conjunctive normal form are detailed. *Difference bound matrices* are also introduced as a convenient way to represent conjunctions of timing constraints.

Chapter 3 is concerned with the state-of-the-art in the Boolean SAT solving field. It starts with a brief historical overview and presents the common architecture of Boolean SAT solvers based on a depth-first exploration of the space of assignments. The chapter contains also a survey of branching heuristics as well as a synthesis of the major useful techniques that can be found in contemporary efficient solvers.

In Chapter 4 we describe a generic mixed SAT solver. We extend methods and algorithms used for Boolean SAT to take into account numerical difference constraints, and develop new techniques specific to these constraints. The chapter concludes with a description of an implementation of such a solver.

Chapter 5 presents three classes of timed systems, namely automata, asynchronous digital circuits, and non-preemptive job-shop scheduling problems, and shows how it is possible to describe their behavior using difference logic in order to verify them with a mixed SAT solver.

Chapter 6 provides an empirical study of the efficiency of the mixed SAT solving techniques based on experimentation carried out on some examples belonging to the three classes of problems presented earlier. This gives a rough idea on how much the key solving techniques contribute to the performance of the solver.

The concluding chapter summarizes the contributions of this thesis and suggests future research directions.

Chapter 2

Timed Formalisms

In this chapter, we introduce difference logic and focus on how to translate formulae expressed in that logic to a special conjunctive normal form. Difference logic consists of classical propositional logic extended with difference constraints, i.e. inequalities of the form $(x - y \prec c)$ where $\prec \in \{<, \leq\}$, x and y are numerical variables, and c is a constant. The use of difference constraints is natural when analyzing problems where time and delays are expressed.

2.1 Difference Logic

This logic has already been used under different names elsewhere [ABK+97, LPWY99, MLA+99, BM00, SBS02] and is probably part of the folklore. We chose to baptize it in this thesis as *Difference Logic* or *DL*.

Formulae of difference logic are formed by joining sub-formulae with such connectives as $\wedge$ (*and*), $\vee$ (*or*), and by prefixing $\neg$ (*not*). The formulae may be defined recursively as the smallest class containing:

- Atomic formulae consisting of Booleans or constraints whose valuations are either *true* or $false$;
- Expressions of the form $\neg\phi_1$, $\phi_1 \wedge \phi_2$, and $\phi_1 \vee \phi_2$.

A more formal definition of DL follows.

Definition 1 (DL Syntax) *Let $\mathcal{B} = \{b_1, b_2, \dots\}$ be a set of Boolean variables and $\mathcal{X} = \{x_1, x_2, \dots\}$ be a set of numerical variables. The difference logic over $\mathcal{B}$ and $\mathcal{X}$ is called $DL(\mathcal{X}, \mathcal{B})$ and given by the following grammar:*

$$\phi \ ::= \ b \mid (x - y < c) \mid (x - y \leq c) \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi$$

where $b \in \mathcal{B}$, $x, y \in \mathcal{X}$ and $c \in \mathbb{D}$ is a constant. The domain $\mathbb{D}$ is either the integers $\mathbb{Z}$ or the real numbers $\mathbb{R}$.

Remark 1 *Other Boolean connectives can also be expressed in DL:*

- *Equivalence* $\Leftrightarrow$*:* $\phi_1 \Leftrightarrow \phi_2 \equiv (\phi_1 \vee \neg\phi_2) \wedge (\neg\phi_1 \vee \phi_2)$
- *Implication* $\Rightarrow$*:* $\phi_1 \Rightarrow \phi_2 \equiv (\neg\phi_1 \vee \phi_2)$
- *Exclusive or* $\oplus$*:* $\phi_1 \oplus \phi_2 \equiv (\phi_1 \wedge \neg\phi_2) \vee (\neg\phi_1 \wedge \phi_2)$

Remark 2 *If* $\mathbb{D} = \mathbb{Z}$ *, the DL syntax can be reduced to:*

$$\phi \ ::= \ b \mid (x - y \leq c) \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi$$

because $(x - y < c)$ *is equivalent to* $(x - y \leq c - 1)$ *over the integer domain.*

Definition 2 (Valuation) *Consider a* $\mathcal{B}$*-valuation* $v_{\mathcal{B}}$ *and an* $\mathcal{X}$*-valuation* $v_{\mathcal{X}}$ *defined by:*

$$v_{\mathcal{B}} : \mathcal{B} \to \mathbb{B} \qquad v_{\mathcal{X}} : \mathcal{X} \to \mathbb{D}$$

where $\mathbb{B} = \{true, false\}$*. An* $(\mathcal{X}, \mathcal{B})$-valuation *is a function*

$$v : DL(\mathcal{X}, \mathcal{B}) \to \mathbb{B}$$

such that:

$$\begin{array}{rcl} v(b) & = & v_{\mathcal{B}}(b) \\ v(x - y < c) & = & \left\{ \begin{array}{ll} true & \textit{iff } v_{\mathcal{X}}(x) - v_{\mathcal{X}}(y) < c \\ false & \textit{otherwise} \end{array} \right. \\ v(x - y \leq c) & = & \left\{ \begin{array}{ll} true & \textit{iff } v_{\mathcal{X}}(x) - v_{\mathcal{X}}(y) \leq c \\ false & \textit{otherwise} \end{array} \right. \\ v(\neg\phi) & = & \neg v(\phi) \\ v(\phi_1 \wedge \phi_2) & = & v(\phi_1) \wedge v(\phi_2) \\ v(\phi_1 \vee \phi_2) & = & v(\phi_1) \vee v(\phi_2) \end{array}$$

where $b \in \mathcal{B}$*,* $x, y \in \mathcal{X}$*,* $c \in \mathbb{D}$*, and* $\phi, \phi_1, \phi_2 \in DL(\mathcal{X}, \mathcal{B})$*.*

2.2 Satisfiability

Definition 3 (Satisfiability) *A formula* $\phi \in DL(\mathcal{X}, \mathcal{B})$ *is satisfiable* iff *there exists an* $(\mathcal{X}, \mathcal{B})$*-valuation* v *such that* $v(\phi) = true$*. A formula* ϕ *is unsatisfiable* iff *for every* $(\mathcal{X}, \mathcal{B})$*-valuation* v*,* $v(\phi) = false$*.*

Proposition 1 *The satisfiability problem for* $DL(\mathcal{X}, \mathcal{B})$ *is NP-complete.*

Proof: NP-hardness is an immediate consequence of the Boolean case as described in Cook's theorem [GJ79]. For NP-easiness, a non-deterministic algorithm works by guessing which atomic formulae (Boolean variables and constraints) appearing in the formula are true and which are not. Then, a polynomial time test has to check that this assignment renders the entire formula true (linear time in the size of the formula) and that the corresponding set of constraints on the real numbers is in fact satisfiable. The satisfiability of

a conjunction of difference constraints (a special case of linear programming) can be solved in polynomial (cubic) time using a variant of the Floyd-Warshall algorithm[1] [CLR+01]. ■

2.3 Conjunctive Normal Form

The *conjunctive normal form (CNF)* is well suited to analyze the satisfiability of a formula. This explains why this form has been extensively used in the classical satisfiability problems.

Definition 4 (Literal) *A literal can be either Boolean or numerical. A Boolean literal is a formula of the form b or $\neg b$ with $b \in \mathcal{B}$. A numerical literal is a formula of the form $(x - y \prec c)$ with $x, y \in \mathcal{X}$, $\prec \in \{<, \leq\}$, and $c \in \mathbb{D}$.*

Remark 3 *In the rest of this document, we may refer to Boolean literals as* Booleans *and to numerical literals as* constraints.

Definition 5 (Clause) *A clause is a finite disjunction of literals. An n-clause is a disjunction of at most n literals.*

Remark 4 *A 0-clause is the empty clause and its valuation is $false$. A* unit clause *is a clause containing only one literal.*

Intuitively, a mixed clause denotes a clause that can contain both Boolean and numerical literals. However, in this work, we will define mixed clauses more restrictively:

Definition 6 (Mixed clause) *A mixed clause is a clause that contains at most one numerical literal.*

Clauses have a number of interesting properties that deserve attention. First, checking whether a clause is satisfied or not is immediate: if it contains at least a literal whose valuation is *true*, it is satisfied. Second, given two clauses C_1 and C_2, it is easy to know if C_1 is *weaker* than C_2, i.e. if C_2 implies C_1, by checking that all the literals of C_2 occur in C_1. In such a case, we say that C_2 *absorbs* C_1. Based upon that property, it is straightforward to deduce that equivalent clauses contain the same literals.

Definition 7 (Conjunctive normal form) *A formula F is in the conjunctive normal form (CNF)* iff *it is a conjunction of clauses, i.e.:*

$$\begin{aligned} F &= \bigwedge_{i=1}^{m} C_i \\ &= \bigwedge_{i=1}^{m} (\bigvee_{j=1}^{n_i} L_{ij}) \end{aligned}$$

where each L_{ij} denotes a literal.

[1] A discussion of this algorithm is on page 20.

Remark 5 *F is said to be n-CNF if all its clauses are n-clauses, i.e.:*

$$\forall\ i \in \{1 \dots m\},\ n_i \leq n$$

Definition 8 (Mixed conjunctive normal form) *A formula is in the mixed conjunctive normal form (MX-CNF)* iff *it is a conjunction of mixed clauses.*

2.4 Translating from DL to MX-CNF

We present, in the increasing order of efficiency, three algorithms to perform the translation from difference logic to the mixed conjunctive normal form. Only the second and the third are really usable in practice due to their interesting properties and to the relative compactness of the MX-CNF translations they produce.

2.4.1 Trivial translation

Algorithm 1 (Trivial translation) *The translation is done using the elementary properties of logical connectives. It is based on pattern matching and rewriting, until no more of the following rules can be applied:*

$$\neg(\phi_1 \wedge \phi_2) \rightarrow \neg\phi_1 \vee \neg\phi_2$$

$$\neg(\phi_1 \vee \phi_2) \rightarrow \neg\phi_1 \wedge \neg\phi_2$$

$$\phi_1 \vee (\phi_2 \wedge \phi_3) \rightarrow (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$$

$$(\phi_1 \vee \phi_2) \wedge \phi_3 \rightarrow (\phi_1 \vee \phi_3) \wedge (\phi_2 \vee \phi_3)$$

$$\neg(\neg\phi) \rightarrow \phi$$

The main drawback of the trivial conversion is that it may result in an exponential growth of the size of the formula as illustrated in the following example.

Example: Using the trivial translation, the formula:

$$(x_1 \wedge y_1) \vee \dots \vee (x_n \wedge y_n)$$

is converted to a conjunction of 2^n binary clauses:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} (x_i \vee y_j)$$

□

2.4.2 Tseitin's translation

By adding new variables, translation to MX-CNF can be done with linear complexity. This simple but efficient idea is credited to Tseitin [Tse68].

The translation is based on the introduction of Boolean variables that are equivalent to the sub-formulae encountered while parsing the formula to convert.

Example: Consider the formula:

$$(a \wedge \neg b) \vee (x - y \leq 5) \tag{2.1}$$

Tseitin's translation starts by creating a new Boolean variable ϕ_0, such that:

$$\phi_0 \Leftrightarrow (a \wedge \neg b) \vee (x - y \leq 5) \tag{2.2}$$

Another new variable ϕ_1 is created and is set equivalent to the left sub-formula of ϕ_0:

$$\phi_1 \Leftrightarrow (a \wedge \neg b) \tag{2.3}$$

As a result, ϕ_0 can be written as:

$$\phi_0 \Leftrightarrow \phi_1 \vee (x - y \leq 5) \tag{2.4}$$

Checking the satisfiability of (2.1) is equivalent to checking the satisfiability of the following conjunction:

$$\begin{array}{lrcl} & \phi_0 & & \\ \wedge & \big(\phi_0 & \Leftrightarrow & \phi_1 \vee (x - y \leq 5)\big) \\ \wedge & \big(\phi_1 & \Leftrightarrow & (a \wedge \neg b)\big) \end{array}$$

Using elementary Boolean rules, the above problem can be immediately translated to MX-CNF clauses. □

Algorithm 2 (Tseitin's translation)

global k

tseitin(F)
begin
    $k := 0$
    create variable ϕ_0
    return$(\phi_0 \wedge$ **convert**$(F, \phi_0))$
end

convert(F, ϕ)
begin

```
        if F has the form G ∨ H
                then
                        begin
                                l1 := literal(G)
                                l2 := literal(H)
                                return (φ ∨ ¬l1) ∧ (φ ∨ ¬l2) ∧ (¬φ ∨ l1 ∨ l2)
                                                ∧convert(G, l1) ∧ convert(H, l2)
                        end
        else if F has the form G ∧ H
                then
                        begin
                                l1 := literal(G)
                                l2 := literal(H)
                                return (¬φ ∨ l1) ∧ (¬φ ∨ l2) ∧ (φ ∨ ¬l1 ∨ ¬l2)
                                                ∧convert(G, l1) ∧ convert(H, l2)
                        end
        else if F has the form ¬ G
                then
                        return convert(G, φ)
        else
                return true
end

literal(F)
begin
        if F is a Boolean or is a constraint
                then
                        return(F)
        else if F has the form ¬ G
                then
                        return(¬literal(G))
        else
                begin
                        k := k + 1
                        create variable φ_k
                        return(φ_k)
                end
end
```

Remark 6 (Constraints in the translation) *When applying our adapted version of Tseitin's algorithm, care must be taken when adding clauses to the MX-CNF formula. In fact, as MX-CNF clauses must contain at most one constraint, the algorithm must modify clauses with two constraints to conform with the MX-CNF syntax.*

As the two-constraint clauses case happens only with clauses of the form $b \vee c_1 \vee c_2$ *where* b *is a Boolean literal and* c_1 *and* c_2 *are constraints, we can*

create a new Boolean variable $\beta \equiv c_2$ *and replace the clause* $b \vee c_1 \vee c_2$ *by:*

$$\begin{array}{cccccc} & b & \vee & c_1 & \vee & \beta \\ \wedge & \beta & \vee & \neg c_2 & & \\ \wedge & \neg\beta & \vee & c_2 & & \end{array}$$

Example: Consider again the formula:

$$(a \wedge \neg b) \vee (x - y \leq 5)$$

According to the algorithm, we first add a new variable ϕ_0 such that:

$$\phi_0 \Leftrightarrow (a \vee \neg b) \vee (x - y \leq 5)$$

and then add a single unit clause to the MX-CNF formula (which is currently empty):

$$\phi_0$$

The second step consists in creating an extra new variable equivalent to the left sub-formula. The right sub-formula is a literal so it requires no renaming.

$$\phi_1 \Leftrightarrow (a \wedge \neg b)$$

Then, the three following clauses are added to the MX-CNF formula:

$$\begin{array}{cccccc} & \phi_0 & \vee & \neg\phi_1 & & \\ \wedge & \phi_0 & \vee & (x - y > 5) & & \\ \wedge & \neg\phi_0 & \vee & \phi_1 & \vee & (x - y \leq 5) \end{array}$$

The right sub-formula must then be converted using the same mechanism. But as it contains only literals, only these three clauses need to be added to the MX-CNF formula:

$$\begin{array}{cccccc} & \neg\phi_1 & \vee & a & & \\ \wedge & \neg\phi_1 & \vee & \neg b & & \\ \wedge & \phi_1 & \vee & \neg a & \vee & b \end{array}$$

The final result of the translation is:

$$\begin{array}{cccccc} & \phi_0 & & & & \\ \wedge & \phi_0 & \vee & \neg\phi_1 & & \\ \wedge & \phi_0 & \vee & (x - y > 5) & & \\ \wedge & \neg\phi_0 & \vee & \phi_1 & \vee & (x - y \leq 5) \\ \wedge & \neg\phi_1 & \vee & a & & \\ \wedge & \neg\phi_1 & \vee & \neg b & & \\ \wedge & \phi_1 & \vee & \neg a & \vee & b \end{array}$$

□

Remark 7 *Tseitin's translation generates MX-3-CNF clauses only. Hence, the final formula contains only the following forms:*

$$\begin{array}{l} b \\ b \ \vee \ b \\ b \ \vee \ b \ \vee \ b \\ c \\ b \ \vee \ c \\ b \ \vee \ b \ \vee \ c \end{array}$$

where b is a Boolean literal and c is a constraint.

2.4.3 Wilson's translation

A more compact translation was proposed by Wilson in [Wil90]. It is based on replacing a disjunction of the form $F \vee G$ with the conjunction:

$$\begin{array}{rccc} & \phi_0 & \vee & \phi_1 \\ \wedge & \neg\phi_0 & \vee & F \\ \wedge & \neg\phi_1 & \vee & G \end{array}$$

where ϕ_0 and ϕ_1 are auxiliary variables. The original disjunction and the translated conjunction are equi-satisfiable. In fact, $\neg\phi_0 \vee F$ and $\neg\phi_1 \vee G$ encode respectively that $\phi_0 \Rightarrow F$ and $\phi_1 \Rightarrow G$.

Example: With Wilson's translation, the formula, used in a previous example:

$$(x_1 \wedge y_1) \vee \cdots \vee (x_n \wedge y_n)$$

is converted to the conjunction:

$$(\phi_0 \vee \cdots \vee \phi_{n-1}) \wedge \bigwedge_{i=1}^{n} \Big((\neg\phi_{i-1} \vee x_i) \wedge (\neg\phi_{i-1} \vee y_i) \Big)$$

The advantage of this translation is obvious as it yields a compact formula. □

Algorithm 3 (Wilson's translation)

global k

wilson(F)
begin
    $k := 0$
    return convert(F)
end

convert(F)
begin

if F **is a clause**
  then
    return F
else if F **has the form** $G \wedge H$
  then
    return convert$(G) \wedge$ **convert**(H)
else if F **has the form** $\neg(G \vee H)$
  then
    return convert$(\neg G \wedge \neg H)$
else if F **has the form** $\neg(G \wedge H)$
  then
    return convert$(\neg G \vee \neg H)$
else if F **has the form** $\lambda \vee (G \wedge H)$ **where** λ **is a literal**
  then
    begin
      $k := k + 1$
      create variable ϕ_{k-1}
      return $(\phi_{k-1} \vee \lambda) \wedge$ **convert**$(\neg\phi_{k-1} \vee G)$
        $\wedge$**convert**$(\neg\phi_{k-1} \vee H)$
    end
else if F **has the form** $\lambda \vee \neg(G \vee H)$ **where** λ **is a literal**
  then
    return convert$(\lambda \vee (\neg G \wedge \neg H)$
else if F **has the form** $\lambda \vee G \vee H$ **where** λ **is a literal**
  then
    begin
      $k := k + 2$
      create variable ϕ_{k-1}
      create variable ϕ_{k-2}
      return $(\phi_{k-1} \vee \phi_{k-2} \vee \lambda) \wedge$ **convert**$(\neg\phi_{k-2} \vee G)$
        $\wedge$**convert**$(\neg\phi_{k-1} \vee H)$
    end
else
  begin
    write F **as** $G \vee H$
    $k := k + 2$
    create variable ϕ_{k-1}
    create variable ϕ_{k-2}
    return $(\phi_{k-1} \vee \phi_{k-2}) \wedge$ **convert**$(\neg\phi_{k-2} \vee G)$
      $\wedge$**convert**$(\neg\phi_{k-1} \vee H)$
  end
end

Example: Again, assume the formula:

$$(a \wedge \neg b) \vee (x - y \leq 5)$$

Using Wilson's algorithm, it is translated into:

$$\begin{array}{cccc} & \phi_0 & \vee & (x-y \leq 5) \\ \wedge & \neg\phi_0 & \vee & a \\ \wedge & \neg\phi_0 & \vee & \neg b \end{array}$$

□

2.5 Conjunctions of Difference Constraints

This section presents some possible representations of conjunctions of difference constraints. It starts by the geometrical one, namely convex timed polyhedra, and focuses after on *Difference Bound Matrices* (or *DBMs*) which were first introduced by Bellman in [Bel57] and used for timing verification by Dill in [Dil89].

2.5.1 Convex timed polyhedra

A convex timed polyhedron [BM00] is a polyhedron resulting from the finite intersection of half-spaces.

More formally, a convex timed $\mathcal{X}$-polyhedron Π is defined as follows:

$$\Pi = \Pi_1 \cap \cdots \cap \Pi_{m'} \cap \Pi_{m'+1} \cap \ldots \cap \Pi_m$$

where $\mathcal{X} = \{x_1, \ldots, x_n\}$ and $\Pi_1, \ldots, \Pi_m$ are hyper-planes of dimension $(n-1)$.

Each of those hyper-planes is defined by the equation:

$$\Pi_i : \begin{cases} x_{f(i)} - x_{s(i)} \prec_i c_i & \text{, if } i \in \{1, \ldots, m'\} \\ x_{f(i)} \prec_i c_i & \text{, if } i \in \{m'+1, \ldots, m\} \end{cases}$$

where:

$$\forall i \in \{1, \ldots, m\},\ f(i) \in \{1, \ldots, n\},\ \prec_i \in \{<, \leq\},\ \text{and } c_i \in \mathbb{D}$$

and

$$\forall i \in \{1, \ldots, m'\},\ s(i) \in \{1, \ldots, n\},\ \text{and } f(i) \neq s(i)$$

Hence, the convex timed polyhedron Π represents a conjunction of difference constraints:

$$\Pi : \bigwedge_{i=1}^{m'} (x_{f(i)} - x_{s(i)} \prec_i c_i) \wedge \bigwedge_{i=m'+1}^{m} (x_{f(i)} \prec_i c_i)$$

2.5.2 Bounds

A *bound* is a pair $(c, \prec)$ where $c \in \mathbb{D} \cup \{\infty\}$ and $\prec \in \{<, \leq\}$. The standard order $<$ on $\mathbb{D}$ is extended to the elements of $\mathbb{D} \cup \{\infty\}$ by letting $c < \infty$, $\forall\ c \in \mathbb{D}$.

Definition 9 (The $<$ order) *A total order on bounds can be defined as follows:*

$$(c,\prec) < (c',\prec') \text{ iff } c < c' \text{ or } \begin{cases} c = c' \\ \text{and} \\ \prec = < \\ \text{and} \\ \prec' = \leq \end{cases}$$

Therefore, we can also define:

$$(c,\prec) \leq (c',\prec') \text{ iff } (c,\prec) < (c',\prec') \text{ or } \begin{cases} c = c' \\ \text{and} \\ \prec = \prec' \end{cases}$$

Remark 8 *The bounds $(\infty,<)$ and $(\infty,\leq)$ are said to be* trivial.

Definition 10 (Minimum, $\oplus$ operation) *The minimum of two bounds $(c,\prec)$ and $(c',\prec')$ is denoted $(c,\prec)\oplus(c',\prec')$ and is defined by:*

$$(c,\prec)\oplus(c',\prec') = \begin{cases} (c,\prec) & \text{if } (c,\prec)\leq(c',\prec') \\ (c',\prec') & \text{otherwise} \end{cases}$$

Definition 11 (Sum, $\otimes$ operation) *The sum of two bounds $(c,\prec)$ and $(c',\prec')$ is defined by:*

$$(c,\prec)\otimes(c',\prec') = (c+c',\prec'')$$

where

$$\prec'' = \begin{cases} < & \text{if } \prec = < \text{ or } \prec' = < \\ \leq & \text{otherwise} \end{cases}$$

Remark 9 *The standard $+$ operation is extended over $\mathbb{D}\cup\{\infty\}$ for each $c\in\mathbb{D}$ as follows:*

$$\begin{aligned} c+\infty &= \infty \\ c+(-\infty) &= -\infty \\ \infty+\infty &= \infty \\ \infty+(-\infty) &= \infty \\ (-\infty)+(-\infty) &= -\infty \end{aligned}$$

Remark 10 *The set of bounds provided with the minimum operator $\oplus$ and the sum operator $\otimes$ defines a (min,+)-linear system [Gau99] such that:*

- $(\infty,<)$ *is the* neutral element *for the $\oplus$ operation. In fact, for any bound b, $b\oplus(\infty,<) = b$.*
- $(0,\leq)$ *is the* neutral element *for the $\otimes$ operation.*
- $(\infty,<)$ *is the* absorbing element *for the $\otimes$ operation. In fact, for any bound b, $b\otimes(\infty,<) = (\infty,<)$.*

2.5.3 Representing convex timed polyhedra using DBMs

Consider the convex timed polyhedron given by the conjunction of constraints:

$$(x - y \leq 5) \wedge (z > 2) \wedge (y - z > 8)$$

The DBM associated with this polyhedron is:

$$\left(\begin{array}{c|cccc} & \mathbf{0} & x & y & z \\ \hline \mathbf{0} & (0,\leq) & (\infty,<) & (\infty,<) & (-2,<) \\ x & (\infty,<) & (0,\leq) & (5,\leq) & (\infty,<) \\ y & (\infty,<) & (\infty,<) & (0,\leq) & (\infty,<) \\ z & (\infty,<) & (\infty,<) & (-8,<) & (0,\leq) \end{array}\right)$$

With this example, we can understand the way a DBM is built from a set of constraints: Each item of the matrix represents the bound of the difference between two variables. A special variable **0** is introduced and is used to encode bounds on single variables (such as $z > 2$).

More formally, a *difference bound matrix* (DBM) of dimension n is a $(n+1)$-square matrix M whose elements are bounds of the difference between variables from the set $\mathcal{X} \cup \{\mathbf{0}\}$, where $\mathcal{X}$ is a set of n variables ranging over $\mathbb{D}$ and **0** is the zero variable.

Each element of a DBM M is indexed by two variables $x, y \in \mathcal{X} \cup \{\mathbf{0}\}$. It is referred to as M_{xy}.

If we consider a convex timed $\mathcal{X}$-polyhedron $\Pi = \Pi_1 \cap \cdots \cap \Pi_m$ where $\Pi_1, \ldots, \Pi_m$ are hyper-planes of dimension $(n-1)$, we can build its associated DBM of dimension n using the following algorithm:

Algorithm 4 (A convex timed polyhedron → DBM)

ConvexTimedPolyhedronToDBM(Π)
begin
  Initialize all elements of M **to** $(\infty, <)$

  for $k := 1$ **to** m
    begin
      if $\Pi_k = (x_i - x_j \prec c)$
        then
          $M_{x_i x_j} := (c, \prec)$
      else if $\Pi_k = (x_i - x_j \succ c)$
        then
          $M_{x_j x_i} := (-c, \prec)$
      else if $\Pi_k = (x_i \prec c)$
        then
          $M_{x_i \mathbf{0}} := (c, \prec)$
      else if $\Pi_k = (x_i \succ c)$
        then
          $M_{\mathbf{0} x_i} := (-c, \prec)$

end

return(M)
end

Inversely, any DBM M of dimension n defines a convex timed polyhedron $[[M]]$ on the set of variables $\mathcal{X} = \{x_1, \ldots, x_n\}$ such that:

$$\begin{array}{rcl} [[M]] & = & \bigcap_{1\leq i\leq n} \Big((x_i \ \ll_{M_{x_i\mathbf{0}}} \ b_{M_{x_i\mathbf{0}}}) \cap (-x_i \ \ll_{M_{\mathbf{0}x_i}} \ b_{M_{\mathbf{0}x_i}})\Big) \\ & \cap & \bigcap_{1\leq i\neq j\leq n}(x_i - x_j \ll_{M_{x_ix_j}} b_{M_{x_ix_j}}) \end{array}$$

where $\ll_{(c,\prec)} = \prec$ and $b_{(c,\prec)} = c$. $[[M]]$ is the *convex timed polyhedron associated* with M.

2.5.4 Canonical representation of DBMs

A combination of constraints that defines a convex timed polyhedron can lead to the deduction of other *implicit constraints*.

Consider the convex timed polyhedron Π defined by the constraints:

$$(x \geq 1) \wedge (x \leq 4) \wedge (y \geq 2) \wedge (y \leq 33) \wedge (y - x \leq 1) \wedge (x - y \leq 0)$$

From constraints $(x \leq 4)$ and $(y - x \leq 1)$, we deduce $(y \leq 5)$. We can easily check that the inequality $(y \leq 5)$ holds for each point (x, y) in Π. Thus, it is obvious that Π' given by the conjunction of constraints

$$(x \geq 1) \wedge (x \leq 4) \wedge (y \geq 2) \wedge (y \leq 5) \wedge (y - x \leq 1) \wedge (x - y \leq 0)$$

is equivalent to Π.

This simple example shows that there is no unique set of constraints whose conjunction defines a given convex timed polyhedron. Hence, there is no bijection between DBMs and convex timed polyhedra.

Such a bijection, if it were available, would have solved many issues, including the detection of equality between two DBMs.

Intuitively, we can guess that testing if two DBMs M_1 and M_2 are equivalent consists in testing if the sets of all their original constraints with all the implicit constraints are the same.

In order to generate all the constraints that can be deduced from a DBM M, we will not consider it as a geometrical object for a while. We will rather consider it as an adjacency matrix of a directed graph whose vertices are elements of $\mathcal{X} \cup \{\mathbf{0}\}$. An edge in that graph that connects x to y has M_{xy} as weight.

The DBMs associated with Π and Π' are the adjacency matrices of the two graphs in Figure (2.1) and (2.2).

The graph associated with Π' can be obtained from the one associated with Π by replacing the weight of the edge connecting y to $\mathbf{0}$ by the weight of the path linking y to $\mathbf{0}$ that goes through x. In other words, the weight of any edge

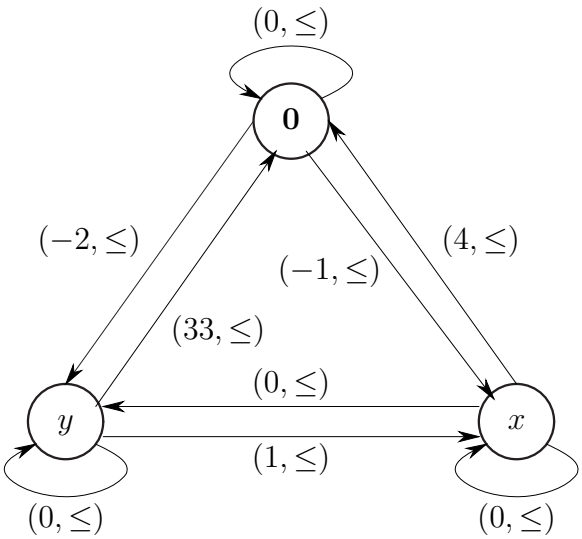

Figure 2.1: Graph associated with Π

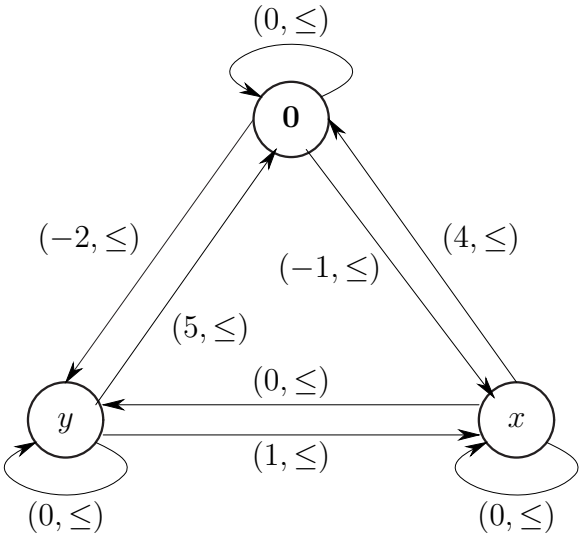

Figure 2.2: Graph associated with Π'

connecting x and y can be replaced by a smaller constraint that results from the computation of the weight of the shortest path whose extremities are x and y.

Definition 12 (Weight of a path) *Consider a DBM M and a path P in the graph whose adjacency matrix is M, such that:*

$$P = x_1 \rightarrow \ldots \rightarrow x_k$$

The weight of P is :

$$weight(P) = M_{x_1x_2} \otimes M_{x_2x_3} \otimes \ldots \otimes M_{x_{k-1}x_k}$$

Lemma 1 *Consider a DBM M and $Z \in [[M]]$. For each $x, y \in \mathcal{X} \cup \{\mathbf{0}\}$ and for each path P connecting x and y :*

$$Z_x - Z_y \leq b_{weight(P)}$$

Proof: Suppose that $P = x \rightarrow x_1 \rightarrow \ldots \rightarrow x_k \rightarrow y$. As $Z \in [[M]]$, we have :

$$\begin{array}{rcl} Z_x - Z_{x_1} & \leq & b_{M_{xx_1}} \\ Z_{x_1} - Z_{x_2} & \leq & b_{M_{x_1x_2}} \\ & \vdots & \\ Z_{x_{k-1}} - Z_{x_k} & \leq & b_{M_{x_{k-1}x_k}} \\ Z_{x_k} - Z_y & \leq & b_{M_{x_ky}} \end{array}$$

By summing the inequalities, we obtain:

$$Z_x - Z_y \leq b_{M_{xx_1}} + b_{M_{x_1x_2}} + \ldots + b_{M_{x_{k-1}x_k}} + b_{M_{x_ky}}$$

As

$$\begin{array}{rcl} b_{M_{xx_1}} + b_{M_{x_1x_2}} + \ldots + b_{M_{x_{k-1}x_k}} + b_{M_{x_ky}} & = & b_{M_{xx_1}} \otimes M_{x_1x_2} \otimes \ldots \otimes M_{x_{k-1}x_k} \otimes M_{x_ky} \\ & = & b_{weight(P)} \end{array}$$

We conclude that:

$$Z_x - Z_y \leq b_{weight(P)}$$

■

Lemma 2 *Let M be a DBM and P a cycle in the graph whose adjacency matrix is M. If the weight of P is negative then $[[M]]$ is empty.*

Proof: Let $P = x_1 \rightarrow \ldots \rightarrow x_k$ such that $b_{weight(P)} < 0$. Suppose now that there exists $Z \in [[M]]$, then:

$$\begin{array}{rcl} Z_{x_1} - Z_{x_2} & \leq & b_{M_{x_1x_2}} \\ & \vdots & \\ Z_{x_{k-1}} - Z_{x_k} & \leq & b_{M_{x_{k-1}x_k}} \\ Z_{x_k} - Z_{x_1} & \leq & b_{M_{x_kx_1}} \end{array}$$

Summing the inequalities gives:

$$\begin{array}{rcl} 0 & \leq & b_{M_{x_1x_2}} + \ldots + b_{M_{x_{k-1}x_k}} + b_{M_{x_kx_1}} \\ & \leq & M_{x_1x_2} \otimes \ldots \otimes M_{x_{k-1}x_k} \otimes M_{x_kx_1} \\ & \leq & b_{weight(P)} \end{array}$$

This contradicts the supposition $b_{weight(P)} < 0$. Therefore:

$$[[M]] = \varnothing$$

■

Thus, by replacing the weight of each edge $x \rightarrow y$ in the graph with the minimum of the weights of the paths connecting x to y, we obtain a graph whose adjacency matrix does not enclose all implicit constraints.

That operation can be achieved by applying the Floyd-Warshall algorithm [AHU74]. In fact, it assigns to each edge the weight of the shortest path that connects its vertices.

Definition 13 (Floyd-Warshall of a DBM) *For each DBM M, $\mathbf{fw}(M)$ is the DBM resulting from the application of the Floyd-Warshall algorithm to the matrix M.*

Algorithm 5 (Floyd-Warshall)

FloydWarshall(M)
begin
  for $i := 1$ **to** n
    for $j := 1$ **to** n
      for $k := 1$ **to** n
        $M_{x_ix_j} := M_{x_ix_j} \oplus (M_{x_ix_k} \otimes M_{x_kx_j})$
end

Remark 11 *Let M be a DBM and $M' = \mathbf{fw}(M)$. According to Lemma 2, if there exists $x \in \mathcal{X} \cup \{\mathbf{0}\}$ such that $M'_{xx} < (0, \leq)$ then $[[M]] = \varnothing$.*

Definition 14 (The empty DBM) *The DBM $\mathbf{O}$ defined by:*

$$\forall x, y \in \mathcal{X} \cup \{\mathbf{0}\},\ \mathbf{O}_{xy} = (-\infty, <)$$

is the empty DBM.

Definition 15 (Canonical form of a DBM) *For each DBM M, we define its canonical form $\mathbf{cf}(M)$ as follows:*

$$\mathbf{cf}(M) = \begin{cases} \mathbf{O} & \text{if there exists } x \in \mathcal{X} \cup \{\mathbf{0}\} \text{ such that } \mathbf{fw}(M)_{xx} < (0, \leq) \\ \mathbf{fw}(M) & \text{otherwise} \end{cases}$$

Testing if two DBMs are associated with the same convex timed polyhedron is equivalent to testing the equality of their canonical forms.

2.5.5 Conjunction of DBMs

Several operations can be applied to DBMs such as conjunction, assignment and time passage. In this section, we focus on the conjunction as it is the only operation that we will use later.

Example: Consider the convex timed polyhedra Π and Π' defined by:

$$\Pi = (x \geq 1) \wedge (x \leq 4) \wedge (y \geq 2) \wedge (y - x \leq 1)$$

$$\Pi' = (x \geq 0) \wedge (x < 6) \wedge (y < 6) \wedge (y \geq 0) \wedge (x - y < 1)$$

Let M and M' be the DBMs associated with, respectively, Π and Π'.

$$M = \left(\begin{array}{c|ccc} & \mathbf{0} & x & y \\ \hline \mathbf{0} & (0,\leq) & (-1,\leq) & (-2,\leq) \\ x & (4,\leq) & (0,\leq) & (\infty,<) \\ y & (\infty,<) & (1,\leq) & (0,\leq) \end{array}\right)$$

$$M' = \left(\begin{array}{c|ccc} & \mathbf{0} & x & y \\ \hline \mathbf{0} & (0,\leq) & (0,\leq) & (0,\leq) \\ x & (6,<) & (0,\leq) & (1,<) \\ y & (6,<) & (\infty,<) & (0,<) \end{array}\right)$$

Computing the DBM $M \oplus M'$ gives:

$$M \oplus M' = \left(\begin{array}{c|ccc} & \mathbf{0} & x & y \\ \hline \mathbf{0} & (0,\leq) & (-1,\leq) & (-2,\leq) \\ x & (4,\leq) & (0,\leq) & (1,<) \\ y & (6,<) & (1,\leq) & (0,<) \end{array}\right)$$

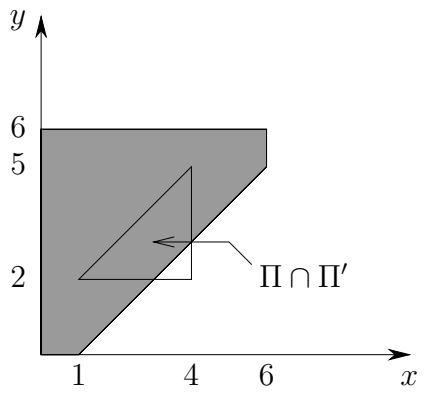

Figure 2.3: Example of the intersection of two convex timed polyhedra

The DBM $M \oplus M'$ corresponds to the convex timed polyhedron:

$$(x \geq 1) \wedge (x \leq 4) \wedge (y \geq 2) \wedge (y < 6) \wedge (y - x \leq 1) \wedge (x - y < 1)$$

which is equal to $\Pi \cap \Pi'$

□

Lemma 3 *Let* Π *and* Π' *be two convex timed polyhedra whose associated DBMs are* M *and* M'*. The convex timed polyhedron* $\Pi \cap \Pi'$ *is associated with* $M \oplus M'$*,i.e.,*

$$\Pi \cap \Pi' = [[M \oplus M']]$$

Proof:

$$\begin{array}{rcl}
\Pi \cap \Pi' & = & \bigcap_{1\leq i\leq n}\Big((x_i \ll_{M_{x_i\mathbf{0}}} b_{M_{x_i\mathbf{0}}}) \cap (-x_i \ll_{M_{\mathbf{0}x_i}} b_{M_{\mathbf{0}x_i}})\Big) \\
& \cap & \bigcap_{1\leq i\neq j\leq n}(x_i - x_j \ll_{M_{x_ix_j}} b_{M_{x_ix_j}}) \\
& \cap & \bigcap_{1\leq i\leq n}\Big((x_i \ll_{M'_{x_i\mathbf{0}}} b_{M'_{x_i\mathbf{0}}}) \cap (-x_i \ll_{M'_{\mathbf{0}x_i}} b_{M'_{\mathbf{0}x_i}})\Big) \\
& \cap & \bigcap_{1\leq i\neq j\leq n}(x_i - x_j \ll_{M'_{x_ix_j}} b_{M'_{x_ix_j}}) \\
& = & \bigcap_{1\leq i\leq n}\Big((x_i \ll_{M_{x_i\mathbf{0}}} b_{M_{x_i\mathbf{0}}}) \cap (x_i \ll_{M'_{\mathbf{0}x_i}} b_{M'_{\mathbf{0}x_i}})\Big) \\
& \cap & \bigcap_{1\leq i\leq n}\Big((-x_i \ll_{M_{x_i\mathbf{0}}} b_{M_{x_i\mathbf{0}}}) \cap (-x_i \ll_{M'_{\mathbf{0}x_i}} b_{M'_{\mathbf{0}x_i}})\Big) \\
& \cap & \bigcap_{1\leq i\neq j\leq n}\Big((x_i - x_j \ll_{M_{x_ix_j}} b_{M_{x_ix_j}}) \cap (x_i - x_j \ll_{M'_{x_ix_j}} b_{M'_{x_ix_j}})\Big) \\
& = & \bigcap_{1\leq i\leq n}(x_i \ll_{M\oplus M'_{x_i\mathbf{0}}} b_{M\oplus M'_{x_i\mathbf{0}}}) \\
& \cap & \bigcap_{1\leq i\leq n}(-x_i \ll_{M\oplus M'_{x_i\mathbf{0}}} b_{M\oplus M'_{x_i\mathbf{0}}}) \\
& \cap & \bigcap_{1\leq i\neq j\leq n}(x_i - x_j \ll_{M\oplus M'_{x_ix_j}} b_{M\oplus M'_{x_ix_j}}) \\
& = & [[M \oplus M']]
\end{array}$$

■

Chapter 3

SAT Solvers

The Boolean satisfiability problem is common to many fields including, but not restricted to, logic verification, timing analysis and automatic test pattern generation. Though its theoretical aspects are widely studied, it is still the focus of many researchers in order to develop practical techniques that can improve its solving time. The Boolean satisfiability problem is an NP-complete problem and no algorithms with worst-case complexity than exponential are known [GJ79].

During the last decades, many SAT solving algorithms, all based on the same framework, have been proposed. In this chapter, we do a synthesis of most of them and try to formalize the way such an algorithm is built.

The outline of this chapter is as follows: After an overview of available methods used to solve SAT problems, we report important historical milestones related to SAT solvers. We next describe the basic algorithm shared by most of them. We give also a complete survey of techniques commonly used to improve their performance.

3.1 Overview

Below is a non-exhaustive list of complete methods for determining the satisfiability of a given CNF formula ϕ; these methods are listed in no particular order:

- *Enumerating all possible truth values and checking each of them to see whether it satisfies ϕ.* This approach is not practical as it requires processing time that grows exponentially with the size of the formula.

- *Performing a backtracking search algorithm through the possible truth assignments of ϕ to show that it is satisfiable.*This method is by far the most used and it is the one on which we will focus in the rest of this thesis.

- *Checking directly if ϕ is a contradiction by completely simplifying it and testing if the resulting formula is empty.* This method is not even known to be in NP [Gal86].

- *Using the resolution method to show that ϕ is unsatisfiable.* This method is attributed to Robinson [Rob65], but it was first proposed by Blake in 1937 [CS88]. Important complexity studies have been carried out during the late 70's and the 80's and include the works of Galil [Gal77], Haken [Hak85], Urquhart [Urq87], and Chvátal and Szemerédi [CS88].
- *Showing that the complement of ϕ* [1] *is not valid using a theorem prover.* Until the 80's, interest in SAT was motivated by the possibility of using a SAT solver as the main piece of a theorem prover for first-order logic [DP60].
- *Using binary decision diagrams (BDD)* [Bry92].
- Other methods including term-rewriting [DHJP83, Hsi85] and production systems [Sie87].

3.2 History

The history of SAT solvers can be divided into three periods that can be put in parallel with the creation and evolution of computers. Until 1960, SAT solving algorithms were just theoretically described but with no real interest as they could not be used in practice. In 1960, Davis and Putnam published an algorithm which inaugurated a new era. The algorithm has been used extensively to solve many kinds of problems using computers (especially from artificial intelligence and operating research). In the 90's, the interest in SAT solvers became more effective. Computers of the 90's were able to handle large SAT instances, which widened the scope of SAT solver applications to include circuit design, scheduling, and model checking.

3.2.1 Before 1960

Boolean satisfiability was always diluted with other types of logical problems. Löwenheim is probably the first mathematician who identified SAT problems and discovered the first search algorithm around 1910 [CS88].

3.2.2 From 1960 to 1990

In 1960, Davis and Putnam rediscovered the algorithm already designed by Löwenheim [DP60, CS88]. It became widely known as the Davis-Putnam procedure, or simply DP. Two years later, Davis contributed with Logemann and Loveland to the creation of a new SAT search algorithm [DLL62] usually referred to as DPL or DPLL.

DPL is generally mistaken for DP although the two methods are radically different: DP is based on a *variable elimination rule* that usually transforms a Boolean satisfiability problem into a larger one. On the contrary, DPLL uses a *splitting rule* that replaces a problem with two smaller sub-problems. That is why DPLL is the most commonly-implemented algorithm.

[1] The complement is a disjunctive normal form expression.

DP is not the preferred algorithm for four main reasons:

- It increases the length and the number of clauses;
- It generates many duplicate and subsumed clauses;
- It rarely leads to the generation of unit clauses;
- The variable elimination rule is harder to implement than the splitting rule.

Another property makes DPLL a better choice: the splitting rule is well suited to prove satisfiability whereas DP's variable elimination rule renders unsatisfiability proving easy.

Cook's seminal paper [Coo71], where he introduces the NP-completeness notion, uses SAT as the canonical NP-complete problem.

3.2.3 From 1990

Until 1990, researchers focused on studying and improving the theory that underlies SAT solvers. During that period, important results have been proved but the rare solvers were very rudimentary as they remained close to the immediate transcription of the Davis-Putnam search algorithm (DPLL/DLL) to a programming language.

In the beginning of the 90's, computers became powerful enough to solve medium sized SAT instances using simple implementations. As a consequence, some researchers started studying how to improve SAT solvers *in practice*. Benchmarks and worldwide competition has been established and the quest for the *fastest ever* solver was born.

Many SAT solvers were created during that rush period such as C-SAT [DABC93], 2cl [GT93], NTAB [CA93], GSAT [GM94], POSIT [Fre95], rel_sat [BS97], SATO [Zha97], SATZ [LA97], GRASP [SS99], HeerHugo [GW99, GW00], cnfs [DD01], (z)Chaff [MMZ$^+$01], and BerkMin [GN02].

3.3 Basic Algorithm

3.3.1 Description

Almost all the SAT solvers are based on the DPLL algorithm. DPLL is traditionally written in a recursive manner. It requires two parameters:

- ϕ: The formula to solve in the conjunctive normal form;
- A: A set of assignments. An assignment of a variable b is denoted:

$$b = true/false$$

Remark 12 *In this chapter, we will use the term* literal *to designate a Boolean literal. We will also extend the assignments to literals. Assume a literal l*

containing a boolean variable b. Saying that l is assigned a value v, or l is set to v, is the same as assigning to the variable b the value v if $l = b$, and is the same as setting b to $\neg v$ if $l = \neg b$.

To determine the satisfiability of a given formula ϕ, the set A must be initially empty, i.e., the algorithm is invoked as:

$$\textbf{DPLL}(\phi, \varnothing)$$

Algorithm 6 (Recursive DPLL)

DPLL(ϕ, A)
begin
$A' =$ **deduction**$(\phi, A) \cup A$

if is-satisfied(ϕ, A')
then
return(*satisfiable*)
else if is-conflicting(ϕ, A')
then
return(*conflicting*)

$v =$ **choose-free-variable**(ϕ, A')

if DPLL$(\phi, A' \cup \{v = true\}) = satisfiable$
then
return(*satisfiable*)
else
return(**DPLL**$(\phi, A' \cup \{v = false\})$)
end

The algorithm introduces the function **deduction**(ϕ, A) which returns the necessary set of assignments that can be deduced from the set of assignments A applied to ϕ.

The DPLL algorithm ends when the formula is found to be satisfied or unsatisfied with the current set of assignments A. Otherwise, DPLL chooses an unassigned variable v and branches on it. The first branch has v set to $true$ and the second has v set to $false$. DPLL is applied recursively on each branch.

While the recursive version of DPLL can be used without any problem, implementors often tend to favor its iterative version to lower memory usage and to improve the performance. In fact, the recursion implies multiple pushes and pops of the whole formula on the stack of the solver.

3.3.2 Improvements to the basic algorithm

In this section, we describe the main ways to enhance the basic solving framework provided by DPLL. Although conceptually interesting, not all of the presented methods are efficient in the sense that they do not speed up the basic

algorithm. It is also important to emphasize that these enhancements depend a lot on the class of the problems to solve and on their size.

Improving DPLL can be achieved at two distinct levels:

- The **deduction** function: It simplifies the formula and returns variable assignments that can be deduced from it. The algorithm is commonly called *binary constraint propagation* or BCP;
- The **choose-free-variable** function: Its efficiency is related to the heuristic used to pick a variable from the set of free variables.

3.4 Binary constraint propagation

This technique consists of applying a series of transformations to simplify and deduce new assignments until the stabilization of the formula, i.e. until no more simplifications or deductions can be done.

3.4.1 Unit resolution

For each clause consisting of a single literal l in the formula, l is set to *true*.

3.4.2 Pure literals resolution

If the formula contains a *pure literal*, i.e. a literal l such that $\neg l$ does not occur in the formula, l is set to *true*.

3.4.3 Equivalent literals detection

Binary clauses in the formula are used to build a directed graph. For each clause $l_1 \vee l_2$, an edge joining l_1 to $\neg l_2$ and another joining $\neg l_1$ to l_2 are added to the graph. Cycles in that graph are made of equivalent literals.

3.5 Branching heuristics

When no more deductions are possible, the DPLL algorithm chooses a free variable and assigns to it a value. It is well established that different branching heuristics produce search trees whose sizes may vary significantly for the same algorithm. Thus, it is obvious that choosing a *good* variable is crucial as it may have a direct impact on the performance of the solver.

3.5.1 Maximizing satisfied clauses heuristics

The goal of early branching heuristics was to find a variable which, when assigned, will maximize the number of the satisfied clauses. These heuristics were based on evaluation functions that take into account statistics relative to the clauses in the formula: clause length, number of occurrences of each variable, etc.

MOM's heuristic

The MOM's heuristic, or *Maximum Occurrences on Minimum-sized clauses*, fits in that class of heuristics. It is certainly the most widely used SAT heuristic as it is simple, easy to implement and relatively problem-independent. The MOM's heuristic finds the variable that occurs the most in clauses of minimum size, hence its name [Pre93].

Formally, the MOM's heuristic scans the list of clauses of a given SAT problem and counts for each $v \in V$:

- $o_n(v)$: the number of unresolved clauses of size n that contain positive occurrences of v;
- $o_n(\neg v)$: the number of unresolved clauses of size n that contain negative occurrences of v.

where V denotes the set of the variables used in the problem.

Then, for each variable v, $o(v)$ and $o(\neg v)$ are computed as follows:

$$\begin{array}{rcl} o(v) & = & min\{o_n(v) \mid o_n(v) > 0 \text{ and } n \in \mathbb{N}^*\} \\ o(\neg v) & = & min\{o_n(\neg v) \mid o_n(\neg v) > 0 \text{ and } n \in \mathbb{N}^*\} \end{array}$$

The variable chosen by the heuristic is v^* such that:

$$\begin{cases} o(v^*) = max\{o(v) \mid v \in V\} \\ \text{and} \\ o(\neg v^*) = max\{o(\neg v) \mid v \in V\} \end{cases}$$

We can guess that this heuristic behaves well because it chooses variables that are the most constrained in the formula. Branching on them maximizes the effect of BCP and shortens the search sub-tree. But, a close look at the MOM's heuristic reveals two main disadvantages:

- It heavily depends on the number of short clauses;
- It does not provide good results at the first levels of the search tree. This can drastically harm the performance of the solver as the first choices of branching variables affect the size of the search tree, especially when the SAT instance is satisfiable.

Variations of this heuristic have been studied and implemented:

- Zabih and McAllester's version considers only negative occurrences of the variables [ZM88]. Formally, v^* satisfies:

$$o(\neg v^*) = max\{o(\neg v) \mid v \in V\}$$

- Freeman's version used in POSIT [Fre95] chooses the variable v^* such that:

$$o(v^*) + o(\neg v^*) = max\{o(v) + o(\neg v) \mid v \in V\}$$

Bohm's heuristic

Bohm's heuristic [BKB92] selects a variable that maximizes the vector H with respect to the lexicographical order where H is:

$$H = \big(H_1(v), H_2(v), \ldots, H_n(v)\big)$$

Each $H_i(v)$ is computed as follows:

$$H_i(v) = \alpha \; max\big\{o_i(v), o_i(\neg v)\big\} + \beta \; min\big\{o_i(v), o_i(\neg v)\big\}$$

The constants used in this heuristic offer a way to choose what to prefer:

- Satisfying the small clauses when assigning $true$ to the selected variable;
- Or reducing the size of small clauses when setting the selected variable to $false$.

The values of α and β are chosen empirically. Suggested values in [BKB92] are:

$$\begin{cases} \alpha &= 1 \\ \beta &= 2 \end{cases}$$

It is possible to think about changing these values dynamically while exploring the search tree according to information collected during runtime.

Jeroslow and Wang's heuristics

Jeroslow and Wang proposed two heuristics in [JW90]. Both are based on the definition of $J(l)$ for a given literal l:

$$J(l) = \sum_{l \in C \wedge C \in \phi} 2^{-|C|}$$

These heuristics count the positive and negative occurrences of each variable. Next, rather than considering only the shortest clauses, as it is the case with the MOM's heuristics, each occurrence is weighted by the size of the clause in which it appears.

The one-sided Jeroslow-Wang heuristic chooses the variable v^* with the largest $max\{J(v^*), J(\neg v^*)\}$. The two-sided version selects the variable v^* such that:

$$J(v^*) + J(\neg v^*) = max\{J(v) + J(\neg v) \mid v \in V\}$$

In the first branch, v^* is set to $true$ if $J(v^*) \geq J(\neg v^*)$. Otherwise, it is assigned $false$.

Later, in 1993, Van Gelder and Tsuji, and Dubois *et al.* designed sophisticated versions belonging to this class of heuristics [Fre95].

The heuristics that try to maximize the number of satisfied clauses work well on some classes of SAT instances but make the solver behave very badly on others. The main reason is that using the statistics can be useful for randomly generated instances but generally fail to give relevant information about structured problems.

3.5.2 Literal count heuristics

Literal count heuristics have been developed in order to reduce the time a solver spends computing heuristics. In fact, the major drawback of the maximizing satisfied clauses heuristics is their *excessive* runtime. Literal count heuristics take into account only the number of occurrences of a given variable in unresolved clauses at each step of the search algorithm.

This class of heuristics is based on counting $c(v)$ (respectively $c(\neg v)$) that represents the number of unresolved clauses in which a variable v appears positively (respectively negatively.) The advantage of literal count heuristics becomes obvious here since these two figures are computed during the search without being dependent on the current state of the solver, which consists essentially of variable assignments.

Below we present some of the most popular literal count heuristics. Most of the heuristics belonging to this class are variations of these.

Dynamic Largest Individual Sum (DLIS)

This heuristic selects the variable with the largest individual value of $c(v)$ or $c(\neg v)$. Formally, the selected branching variable v^* satisfies:

$$max\{c(v^*), c(\neg v^*)\} = max\big\{max\{c(v), c(\neg v)\} \mid v \in V\big\}$$

Variable v^* is assigned the value $true$ if $c(v^*) \geq c(\neg v^*)$ and $false$ otherwise.

Dynamic Largest Combined Sum (DLCS)

In this heuristic, $c(v)$ and $c(\neg v)$ are considered combined. We choose the variable v^* such that:

$$c(v^*) + c(\neg v^*) = max\{c(v) + c(\neg v) \mid v \in V\}$$

As with DLIS, v^* is set to $true$ if $c(v^*) \geq c(\neg v^*)$ and to $false$ if $c(v^*) < c(\neg v^*)$.

Randomly Dynamic Largest Combined Sum (RDLCS)

RDLCS is a modified version of DLCS. Because this latter can lead to bad branching variable choices, RDLCS tries to randomly assign $true$ or $false$ to the selected v^* instead of comparing $c(v^*)$ and $c(\neg v^*)$.

Randomization has proved to provide a good compromise by avoiding too many bad decisions for some classes of SAT problems. It can also be used with the DLIS heuristic.

3.5.3 Variable state independent heuristics

Although sharing the same background with literal count heuristics, variable state independent heuristics form a distinct class. In fact, a score is assigned to each phase of a given variable v (positive and negative).

Initially, the score of a variable v is set to a linear combination of $c(v)$ and $c(\neg v)$. As the search progresses, and periodically, all the scores are divided by

the same constant. The score of a variable v is increased whenever it satisfies a given property $\mathcal{P}(v)$. The selected variable to branch on is the one with the highest sum of its positive and negative scores.

By construction, variable state independent heuristics tend to privilege the choice of a free variable that lately satisfied the property $\mathcal{P}$. The focus on the *recent* is guaranteed by the periodic decay of the scores.

Variable state independent heuristics are quite competitive when compared with other *good* heuristics. Besides, these heuristics are cheap in terms of computation time. Being state independent (scores do not depend on the assignments of the variables) is the main explanation of their performance.

Variable State Independent Decaying Sum (VSIDS)

Since most modern SAT solvers implement a learning mechanism (see Section 3.6) which is likely to add clauses to the problem while running, VSIDS increases the score of a variable whenever it is included in a learned clause. Formally, the score of a variable v is incremented when the following property is satisfied:

$$\mathcal{P}(v) \ : \ v \text{ occurs in a learned clause}$$

VSIDS has been introduced in Chaff [MMZ+01].

BerkMin's VSIDS

VSIDS has been pushed a little bit further in the SAT solver BerkMin [GN02]. In that version, activity of the variables is captured by their participation in a conflict instead of being captured by their occurrence in learned clauses as in VSIDS. In fact, BerkMin's version is identical to VSIDS except that it does not increase the score of a variable that occurs in a learned clause. Such an increase is rather done when a variable is identified as being involved in a conflict. BerkMin also restricts its choice to the variable with the highest combined score that occurs in the last added clause that is still unresolved. According to published results, this version of VSIDS seems to be more powerful than regular VSIDS on some SAT instances.

3.6 Conflict Directed Backtracking and Learning

When the solver finds a conflicting clause, it goes back to undo assignments that caused that conflict. This is called backtracking. The solver also performs a conflict analysis: it finds a reason for that conflict and tries to resolve it. This analysis prunes the search space by excluding a subspace where a conflict is guaranteed to occur [SS99].

The basic DPLL algorithm performs a simple conflict analysis. Each decision variable has a flag that indicates whether it was tried in both phases or not. When a conflict is detected, the algorithm looks for the furthest decision variable from the root that is not yet flipped, discards all the assignments that have been deduced in the search sub-tree starting at that variable, sets the flag, and tries the other phase for that decision variable. The solver is said to do a

chronological backtracking in that case because it always backtracks to the last decision it made.

Chronological backtracking has proven to be good for some randomly generated SAT problems. However it fails to be efficient on structured instances that describe real world problems. The need for a better solution led to the development of sophisticated conflict analysis procedures. The results of this advanced analysis allow the solver to backtrack to an earlier decision and not to the last one necessarily. The most obvious effect is that the solver prunes bigger subspaces of the search space and runs significantly faster. This kind of backtracking is called *non-chronological backtracking*.

3.6.1 Implication graph

The implication graph is a *directed acyclic graph*, or *DAG*, built using the implication relationships between assignments that were deduced during the SAT solving search [SS99]. Each of its vertices represents an assignment of a variable and is denoted $x = v$ where x is a variable and v is either $true$ or $false$. Let $\lambda(x)$ represent the decision level at which the assignment was made. By extension, we will sometimes write a variable assignment $x = v@\lambda(x)$, i.e. x was set to v at level $\lambda(x)$.

Incident edges of a vertex are the reasons that lead to its related assignment. Each assignment has a decision level associated with it. That level is the depth of the SAT search tree at which a value has been assigned to the variable.

Vertices that have directed edges to a given vertex are called its *antecedent vertices*. A variable whose value has not been deduced but assigned during DPLL branching is called a *decision variable*. Decision variables have no antecedent vertices.

Definition 16 (Antecedent sub-graph of a vertex) *Given an implication graph $\mathcal{I}$ and a vertex $z \in \mathcal{I}$, we define the antecedent sub-graph of z as the antecedent vertices of z and their connecting edges. It is denoted $A_{\mathcal{I}}(z)$.*

A conflict is detected whenever two vertices in the graph represent two contradictory assignments for the same variable. That variable is said to be the *conflicting variable*.

Example: Suppose that at the sixth level of a solving process, the following assignments are available:

$$\begin{cases} y_1 = false@1 \\ y_2 = false@3 \\ y_3 = false@2 \\ y_4 = false@1 \\ y_5 = false@4 \\ y_6 = true@5 \end{cases}$$

Consider also that the SAT instance includes the following clauses:

$$\begin{cases} C_0 = \neg x_2 \vee \neg x_0 \\ C_1 = x_3 \vee y_2 \vee y_3 \vee \neg x_0 \\ C_2 = \neg x_1 \vee y_1 \vee \neg x_0 \\ C_3 = x_4 \vee x_2 \vee \neg x_3 \vee x_1 \\ C_4 = \neg x_5 \vee y_4 \vee \neg x_4 \\ C_5 = x_6 \vee x_5 \\ C_6 = \neg x_7 \vee x_5 \\ C_7 = x_8 \vee x_5 \\ C_8 = x_9 \vee y_5 \vee \neg x_6 \vee x_7 \\ C_9 = \neg x_9 \vee \neg y_6 \vee \neg x_8 \vee x_7 \end{cases}$$

The implication graph that results from the assignment of $true$ to x_0 at decision level 6 is represented in Figure (3.1).

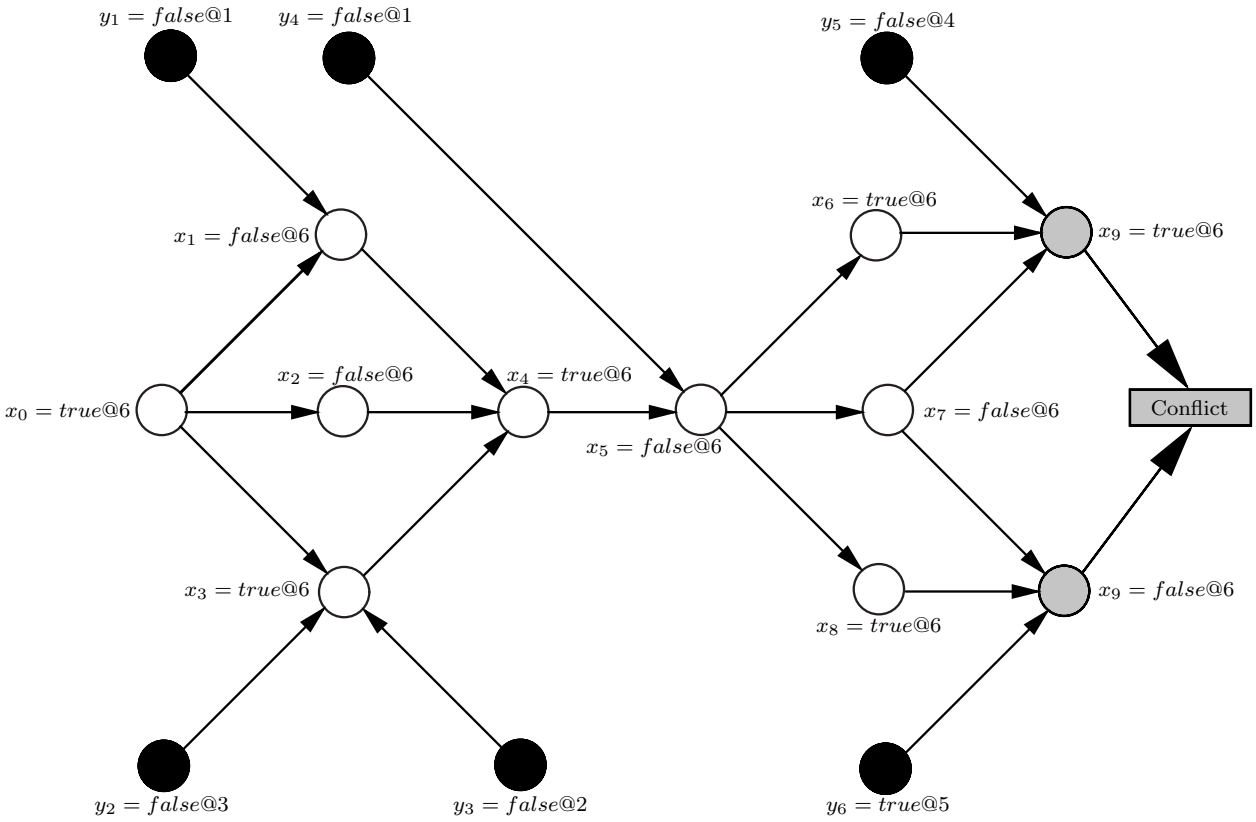

Figure 3.1: Implication graph

□

Since conflict analysis, as its name suggests, is concerned with the assignments that lead to a conflict, the implication graph is not required in its totality. Only a limited number of the antecedent vertices of the two conflicting assignments and their connecting edges are relevant for the analysis.

The key idea behind learning and conflict-directed backtracking is to identify the main reasons that imply a conflict. For that purpose, we introduce the following definitions:

Definition 17 *Given an implication graph $\mathcal{I}$ and two vertices $a, b \in \mathcal{I}$,*

$$a \text{ dominates } b \iff \text{any path from } d \text{ to } b \text{ goes through } a$$

where d is the decision variable whose decision level is the same as the decision level of a.

Definition 18 (UIP) *A unique implication point (UIP) is a vertex that dominates both vertices that represent the contradictory assignments for a conflicting variable.*

Remark 13 *A decision variable is always a UIP.*

A UIP can also be seen as a sufficient reason that leads to a given conflict. Usually, one or more UIPs are associated with a conflict.

Example: In the above example, there are 3 UIPs, namely:

$$x_0 = true@6 \qquad x_4 = true@6$$

$$x_5 = false@6$$

□

3.6.2 Conflict analysis

Advanced conflict analysis techniques use the implication graph to extract the reasons that lead to conflicts. Knowing these reasons, it is possible to build clauses called *conflict clauses* and to add them to the problem in order to prevent the search process from encountering the same conflicts in the future. This operation is called *learning*.

Remark 14 *The conflict clauses must not be confused with the conflicting clauses, which are the clauses that cause a conflict.*

A conflict clause is directly deduced from the reasons that are responsible for a given conflict. It simply states that a certain combination of assignments is not allowed as it leads to a contradiction for the value of the conflicting variable.

Given an implication graph $\mathcal{I}$ in which a conflict occurs, let us denote the conflicting variable k. Since the analysis requires only the part of the graph that contains the implications that produce the conflict, the first step is to extract from $\mathcal{I}$ that relevant part that is denoted $\alpha_{\mathcal{I}}(k)$ and formally defined as follows:

$$\alpha_{\mathcal{I}}(k) = A_{\mathcal{I}}(k = true) \cup A_{\mathcal{I}}(k = false)$$

The second step consists in partitioning the extracted sub-graph $\alpha_{\mathcal{I}}(k)$ into two parts: One part contains all the conflicting assignments and the other holds all the decisions that end up in a conflict. The first is known as the *conflict side* and the second as the *reason side*. Such a bipartition is called a *cut*.

The most evident cut in the implication graph already presented in Figure (3.1) is depicted in Figure (3.2). It delimits a conflict side containing only the conflicting assignments. Figure (3.3) shows other possible cuts.

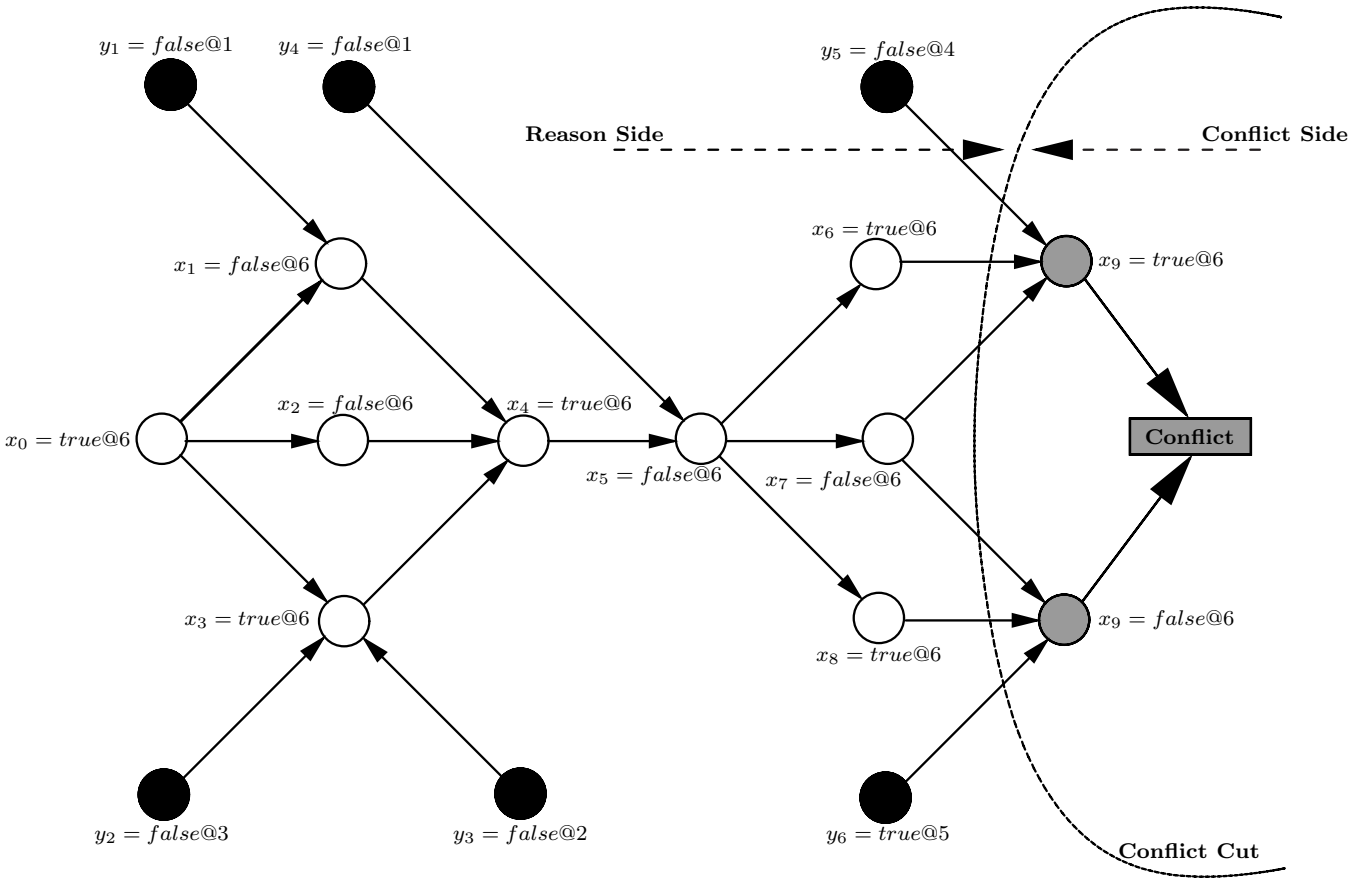

Figure 3.2: Implication graph with a conflict cut

The third step is to find a set $\mathcal{V}$ of vertices such that one or more of their outgoing edges cross the cut line (the boundary between the two sides of the partition). With $\mathcal{V}$, it is fairly easy to generate the conflict clause associated to the conflict involving k. Suppose that $\mathcal{V} = \{x_1 = v_1, \ldots, x_n = v_n\}$, then the reason for that conflict is:

$$x_1^{v_1} \wedge \ldots \wedge x_n^{v_n}$$

where

$$x^v = \begin{cases} x & \text{if } v = true \\ \neg x & \text{otherwise} \end{cases}$$

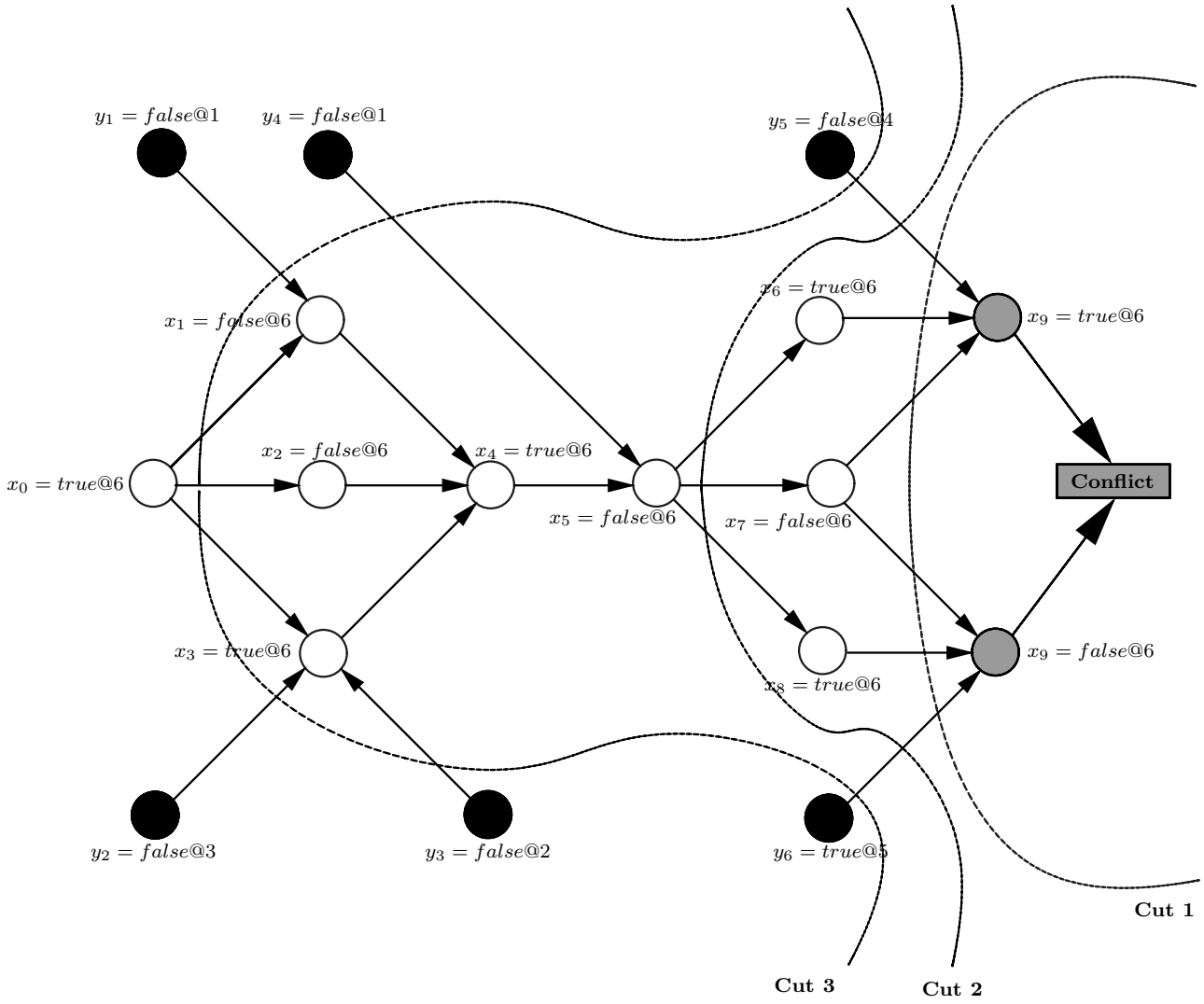

Figure 3.3: Other possible cuts

Thus, the conflict clause, which is the negation of the conflict's reason, is:

$$x_1^{\neg v_1} \vee \ldots \vee x_n^{\neg v_n}$$

Each cut results in a different conflict clause and represents a specific learning strategy. Solving times can vary enormously between two different strategies. In fact, adding the conflict clause to the clauses of the SAT instance narrows the search space. Thus, the shorter that conflict clause is, the narrower the search space becomes. Shorter conflict clauses are obtained when cuts pass by UIPs such that the resulting conflict clauses are built using those UIPs.

Example: Figure (3.3) represents three possible cuts. The conflict clauses resulting from these cuts are:

$$\begin{array}{lcl} \text{Cut 1} & \rightarrow & y_5 \vee \neg x_6 \vee x_7 \vee \neg x_8 \vee \neg y_6 \\ \text{Cut 2} & \rightarrow & y_5 \vee x_5 \vee \neg y_6 \\ \text{Cut 3} & \rightarrow & y_5 \vee y_4 \vee y_1 \vee \neg x_0 \vee y_2 \vee y_3 \vee \neg y_6 \end{array}$$

Cut 2 gives the smallest conflict clause, compared with Cut 1 and Cut 3. In fact, it passes by the UIP $x_5 = false$. Cut 3 passes also by an UIP, which is in that case also a decision variable, but it results in a long conflict clause. □

3.6.3 Conflict-free learning

Adding conflict clauses after performing a conflict analysis is the primary form of learning. But cuts are not restricted to implication graphs with conflicts. It is possible to think about cuts that are conflict-free and that involve only implications between variable assignments.

Such cuts can generate clauses that are already in the problem. They can also generate new clauses that restrict the search space a little more. This kind of learning is possible only if the implication graph (or at least the part that interests the solver) is not a tree. In fact, there must be a confluence in the implication graph to have interesting cuts that can result in new learned clauses.

3.6.4 Backtracking

When a conflict is encountered and when a conflict clause is derived, the solver backtracks to an earlier search level. This level is the highest decision level at which the assignments in $\mathcal{V}$ were made. Formally, the solver backtracks to the level β such that:

$$\beta = \max\{\lambda(x) | (x = v) \in \mathcal{V}\}$$

At decision level β, the derived conflict clause is used either to create a new conflict clause or to backtrack again to a higher level.

3.6.5 Clause recording

During the conflict resolution and the learning steps, new clauses are added to the original SAT instance. For performance and space considerations, that growth in the size of the clauses database must be limited.

Many techniques can be applied to pick the added clauses that can be removed to guarantee a bounded size. The most efficient consists in keeping only clauses of size less or equal to a given integer m. That results in the automatic deletion of large unresolved clauses.

Deletion can also be directed by the relevance of the learned clause. For example, it is good practice to delete large unresolved clauses with at least m' free literals, where m' is an integer fixed in advance.

3.6.6 Restarts

Learning helps to prune the search space when solving a SAT instance by adding new clauses to the initial problem. Those added clauses can be seen as the knowledge about the problem that is accumulated during the search process. The most interesting property of such knowledge is its independence from the current position in the search tree: it still holds even if the position changes, hence the idea of using restarts.

The restart technique consists of stopping the search process when a certain search parameter (such as the solving time or the current search depth) goes beyond a certain limit value. Some solvers even do that on a random basis. The search is stopped and immediately restarted. However, some precautions must be taken in order to not retrace a previous search.

These precautions simply consist of choosing a free variable that has not yet been used for branching at the first level. The learned clauses ensure also that the search is different since they prune its space.

Frequent restarts are good in helping the solver avoid getting stuck searching for a solution in a *difficult* subspace because of earlier bad decisions. Restarts do not even harm unsatisfiable problems identification.

3.7 Alternative Techniques

Several alternative techniques have been studied during the last decade. They can be used to completely replace the DPLL branching part. They can also be applied with a limited scope inside the classic DPLL algorithm. This latter alternative is still the most used in current Boolean SAT solvers.

3.7.1 Resolution *à la* Davis-Putnam

Resolution is the basis of the original Davis-Putnam algorithm. It is better understood when applied on formulae in the *disjunctive normal form* (DNF). A DNF formula is a finite disjunction of conjunctions of literals.

Definition 19 (Disjunctive normal form) *A formula ϕ is in the disjunctive normal form (DNF)* iff *it can be written as a finite disjunction of conjunctions of literals, i.e.,*

$$\begin{aligned}\phi &= \bigvee_{i=1}^{m} D_i \\ &= \bigvee_{i=1}^{m} (\bigwedge_{j=1}^{n_i} L_{ij})\end{aligned}$$

Given a DNF formula ϕ, we call D the set of its conjunctions D_i.

$$D = \{D_1 \dots D_m\}$$

Resolution consists of choosing a variable v used in ϕ and rewriting that formula as follows:

$$\phi = \phi[v = true] \vee \phi[v = false]$$

Notice that the above equality always holds for every choice of v.

Practically, resolution consists of partitioning ϕ into 3 sub-formulae:

$$\phi_v = \bigvee_{\delta \in D \wedge v \in \delta} \delta$$

$$\phi_{\neg v} = \bigvee_{\delta \in D \wedge \neg v \in \delta} \delta$$

$$\phi_* = \bigvee_{\delta \in D \wedge v \notin \delta \wedge \neg v \notin \delta} \delta$$

Let ϕ_v^R (respectively $\phi_{\neg v}^R$) be the result of the removal of all v (respectively $\neg v$) occurrences from ϕ_v (respectively $\phi_{\neg v}$.) and let $\phi^R = \phi_v^R \vee \phi_{\neg v}^R \vee \phi_*$.

ϕ^R is the result of *resolving ϕ on v*. It is also said to be the *resolvent of ϕ* with v being the variable *resolved on.*

Example: Suppose that ϕ is defined by:

$$\phi = (a \wedge b) \vee (a \wedge c) \vee (\neg a \wedge d) \vee (\neg a \wedge \neg c \wedge e) \vee (e \wedge f)$$

Resolving ϕ on a gives:

$$b \vee c \vee d \vee (\neg c \wedge e) \vee (e \wedge f)$$

□

Resolution is suitable for DNF formulae as it reduces their sizes. It is possible, though, to use it on formulae expressed in CNF but it leads to the growth of their size in most cases.

Example: Assume the CNF formula:

$$\phi = (a \vee b) \wedge (a \vee c) \wedge (a \vee d) \wedge (\neg a \vee e) \wedge (\neg a \vee f)$$

Converting ϕ to DNF gives:

$$\begin{aligned} \phi &= \big(a \vee (b \wedge c \wedge d)\big) \wedge \big(\neg a \vee (e \wedge f)\big) \\ &= (a \wedge e \wedge f) \vee (\neg a \wedge b \wedge c \wedge d) \vee (b \wedge c \wedge d \wedge e \wedge f) \end{aligned}$$

Let ϕ^R be the result of resolving ϕ on a:

$$\begin{aligned} \phi^R &= (e \wedge f) \vee (b \wedge c \wedge d) \vee (b \wedge c \wedge d \wedge e \wedge f) \\ &= (b \wedge c \wedge d) \vee (e \wedge f) \end{aligned}$$

Transforming ϕ^R back to CNF gives:

$$\phi^R = (b \vee e) \wedge (b \vee f) \wedge (c \vee e) \wedge (c \vee f) \wedge (d \vee e) \wedge (d \vee f)$$

ϕ has 5 clauses and applying resolution increases their number to 6. □

Resolution should not be used in all cases as it is inefficient on CNF formulae. The only situation where it does not increase the size of the formula is when there exists a literal l that occurs positively in only one clause and negatively in one or more other clauses.

Let C_l be the only clause where l occurs positively such that:

$$C_l = l \vee l_1 \vee \ldots l_k$$

Resolving the formula on the variable contained in the literal l is equivalent to substituting all the occurrences of l with $\neg(l_1 \vee \ldots l_k)$. In other words, it is equivalent to removing the clause C_l and replacing negative occurrences of l by $l_1 \vee \ldots l_k$. It is obvious that this operation decreases the number of the clauses of the formula.

3.7.2 Local search

Local search belongs to the family of stochastic methods. These methods have no way to prove that a SAT instance is unsatisfiable. However, they can quickly find solutions for some hard satisfiable instances.

A local search algorithm starts by assigning, for each free variable, a random value. Then it searches for the variable that can satisfy the largest number of unsatisfied clauses and flips its value. That step is performed at most k times, where k is a given integer. The algorithm stops when the entire formula is satisfied.

The local search algorithm iteration count must be limited. In fact, there is no guarantee that it can give a solution after a bounded number of successive

applications. In practical implementations, the algorithm is repeated k' times, k' being an already fixed integer. If the formula is still not satisfied, there is no means to show that it is unsatisfiable. Hence, the use of local search can just be considered as a complement to other complete solving methods. It can be applied with some restrictions when satisfiability is suspected. If satisfiability cannot be found using local search, the solver must proceed using other techniques.

3.7.3 Stålmarck's method

Stålmarck's method[2] does not share many properties with other SAT solving algorithms. It is *particular* because it does not rely on a CNF representation. Besides, it uses a *breadth-first* search by opposition to DPLL which is based on a *depth-first* search.

In the sequel, an overview of the original Stålmarck's method is given. A more recent version has been described by Harrison in [Har96] and is believed to be very close to the algorithm used in the commercial tool developed by Prover AB, a company founded by Stålmarck. Lately, Groote and Warners developed HeerHugo, a solver largely inspired from the original Stålmarck's method [GW99, GW00].

The following transformations are repeatedly applied on the SAT instance until stabilization:

$$\begin{aligned} \phi_1 \wedge \phi_2 &\equiv \neg(\phi_1 \rightarrow \neg\phi_2) \\ \phi_1 \vee \phi_2 &\equiv \neg\phi_1 \rightarrow \phi_2 \\ \neg\neg\phi_1 &\equiv \phi_1 \\ \neg\phi_1 &\equiv \phi_1 \rightarrow false \end{aligned}$$

Next, the formula is translated into a special form. During that step, new auxiliary variables that are equivalent to sub-formulae are introduced. At the end of the process, we obtain a set of special clauses of the form:

$$a \Leftrightarrow (b \rightarrow c)$$

The above form is denoted as a triplet (a, b, c). $true$ and $false$ are considered as special cases of Boolean variables. They are respectively written 1 and 0 in the triplet notation.

Example: Assume the formula:

$$\phi \equiv p \rightarrow (p \vee q)$$

By applying the transformation rules, ϕ is equivalent to:

$$\begin{aligned} \phi &\equiv p \rightarrow (p \vee q) \\ &\equiv p \rightarrow (\neg p \rightarrow q)) \\ &\equiv p \rightarrow ((p \rightarrow false) \rightarrow q) \end{aligned}$$

After introducing intermediate variables, the transformed formula becomes:

$$\phi \equiv t_0$$

[2]Patented for commercial use [Stå94].

where:

$$\begin{array}{lcl} t_0 & \Leftrightarrow & (p \rightarrow t_1) \\ t_1 & \Leftrightarrow & (t_2 \rightarrow q) \\ t_2 & \Leftrightarrow & (p \rightarrow false) \end{array}$$

Using the triplet notation, ϕ is reduced to:

$$\begin{array}{c} (t_0, p, t_1) \\ (t_1, t_2, q) \\ (t_2, p, 0) \end{array}$$

□

Once the translation is made, the algorithm proves the formula by supposing that its value is $false$ and by trying to derive a contradiction. For that purpose, it uses a set of rules. Each rule has a *triggering triplet* which, when matched, results in the assignment of one or two Boolean variables.

The first rule is the following:

$$(r1)\ \frac{(0, b, c)}{\begin{array}{cc} b/1 & c/0 \end{array}}$$

When a triplet matches the pattern $(0, b, c)$, the second variable b is set to $true$ and the third c is set to $false$.

Applying a rule on an element of a set of triplets leads to a new set in which the assigned variables are substituted with their deduced values. The algorithm is stopped when the new set contains a *terminal* triplet. By definition, *terminal* triplets express an impossible equivalence. Hence, the presence of such a triplet in the new set indicates a contradiction. There are only three terminal triplets:

$$\begin{array}{c} (1, 1, 0) \\ (0, a, 1) \\ (0, 0, a) \end{array}$$

For instance, $(1, 1, 0)$ is a terminal triplet because $true \Leftrightarrow (true \rightarrow false)$ is a contradiction.

The remaining rules are:

$$(r2)\ \frac{(a, b, 1)}{a/1} \quad (r3)\ \frac{(a, 0, c)}{a/1}$$

$$(r4)\ \frac{(a, 1, c)}{a/c} \quad (r5)\ \frac{(a, b, 0)}{a/\neg b}$$

$$(r6)\ \frac{(a, a, c)}{\begin{array}{cc} a/1 & c/1 \end{array}} \quad (r7)\ \frac{(a, b, b)}{a/1}$$

Example: As a continuation to the previous example, assume that ϕ is $false$, i.e. $t_0 \equiv false$. That results in the following set of triplets to solve where only t_0 was substituted with 0:

$$\begin{array}{c} (0, p, t_1) \\ (t_1, t_2, q) \\ (t_2, p, 0) \end{array}$$

Applying the listed rules to that set of triplets related to ϕ gives:

$$\begin{matrix}(0,p,t_1)\\(t_1,t_2,q)\\(t_2,p,0)\end{matrix} \rightarrow (r1) \rightarrow \begin{matrix}p/1\\t_1/0\\(t_1,t_2,q)\\(t_2,p,0)\end{matrix} \longrightarrow \begin{matrix}p/1\\t_1/0\\(0,t_2,q)\\(t_2,1,0)\end{matrix}$$

The application of rule $(r1)$ yields a new set of triplets and a couple of assignments.

$$\begin{matrix}p/1\\t_1/0\\(0,t_2,q)\\(t_2,1,0)\end{matrix} \rightarrow (r1) \rightarrow \begin{matrix}p/1\\t_1/0\\t_2/1\\q/0\\(t_2,1,0)\end{matrix} \longrightarrow \begin{matrix}p/1\\t_1/0\\t_2/1\\q/0\\(1,1,0)\end{matrix}$$

Another application of the same rule results in a terminal triplet. As a conclusion t_0 cannot be $false$ and the formula is satisfiable.

□

Deduction of a new set of triplets using the seven simple rules is called *0-saturation*. Usually, 0-saturation is not enough to prove a formula as sometimes a contradiction cannot be reached after its application. In such a situation, the solving process must proceed with the *dilemma rule*.

The dilemma rule consists of a special case split over a chosen variable. Suppose that the 0-saturation gives a set of triplets S and that the variable v is chosen. The sets $S \cup \{v/0\}$ and $S \cup \{v/1\}$ are 0-saturated respectively to get new triplet sets S_0 and S_1. Then, Σ is computed, where:

$$\Sigma = \begin{cases} S_0 & \text{if } S_1 \text{ contains a terminal triplet} \\ S_1 & \text{if } S_0 \text{ contains a terminal triplet} \\ S_0 \cap S_1 & \text{otherwise} \end{cases}$$

Σ is a super set of the original triplet set S, i.e. $S \subseteq \Sigma$. In fact, the clauses represented by the triplets it contains hold for every value of v. Therefore, the case split was used to gain new information about the problem.

By extension, we can now define *1-saturation* as the operation consisting in applying the dilemma rule to each free variable. If 1-saturation does not lead to a solution, *2-saturation* is carried out. 2-saturation is similar to 1-saturation except that during the dilemma rule application, case splits are done over pairs of variables. For each pair v and w of variables, the following sets of triplets are 0-saturated:

$$S \cup \{v/0, w/0\} \ ,$$

$$S \cup \{v/1, w/0\} \ ,$$

$$S \cup \{v/1, w/1\} \ ,$$

$$\text{and } S \cup \{v/0, w/1\}$$

0-saturation results respectively in the sets S_{00}, S_{01}, S_{11}, and S_{10}. Σ is computed as:

$$\Sigma = \bigcap_{\sigma \in S_*} \sigma$$

where

$$S_* = \{S_{ij} \mid i, j \in \{0, 1\} \text{ and } S_{ij} \text{ does not contain a terminal triplet}\}$$

Similarly, we can define *n-saturation* for any integer n based on case splits over n-tuples of variables. Stålmarck's method uses $(n + 1)$-saturation if n-saturation is unsuccessful in solving the problem.

The major advantage of this method is that case splits are made on both the original variables of the problem and the generated variables that are equivalent to sub-formulae. That results in a fast propagation of the assignments of the variables in the *formula tree*. Thanks to the use of the breadth-first search, Stålmarck's method keeps the number of case splits as low as possible and avoids exponential growth of the search space.

Based upon his method, Stålmarck defined a new classification of Boolean SAT instances according to their hardness. An instance is said to be n-hard when it cannot be solved by $(n - 1)$-saturation. It is said to be n-easy if it can be proved by n-saturation.

It is interesting to note that many instances that are generated from *real world* problems turn out to be 1-easy. According to Harrison [Har96], the commercial implementation of Stålmarck's method seems to take into account that fact: It performs at most 1-saturation. If no result is found, it tries to find a falsifying set of assignments.

Chapter 4

A Mixed SAT Solver

This chapter is concerned with the core of the work of this thesis. It describes a *generic* mixed SAT solver and presents all the algorithms it can use, most of which are adaptations or extensions of methods already found in its Boolean counterparts. Due to the specificity of the mixed SAT instances, some other techniques are introduced in order to handle and process constraints.

4.1 Overview

The skeleton of the mixed SAT solver general algorithm is based on the DPLL scheme, widely implemented in Boolean SAT solvers and discussed in Chapter 3. It consists of two major parts:

- The reduction part: It applies several techniques to reduce the size of the SAT instance while discovering assignments and making deductions. An outline of the reduction algorithm is sketched in Figure (4.1).
- The branching part: Although conceptually similar to what Boolean SAT solvers use for branching, it entails many specific details and uses revamped algorithms that make it unique.

The reduction techniques belong to two distinct families:

- The simplification rules: These handle both Boolean and constraint simplification. Many methods are used for that purpose. Boolean methods have already been introduced in Chapter 3 while techniques specific to constraints will be presented in the sequel.
- The Davis-Putnam rules: They consist of transforming the formula while narrowing the search space without affecting the satisfiability of the solved problem.

4.1.1 Solving context

During the solution process, the current state is held in the *current solving context* which contains:

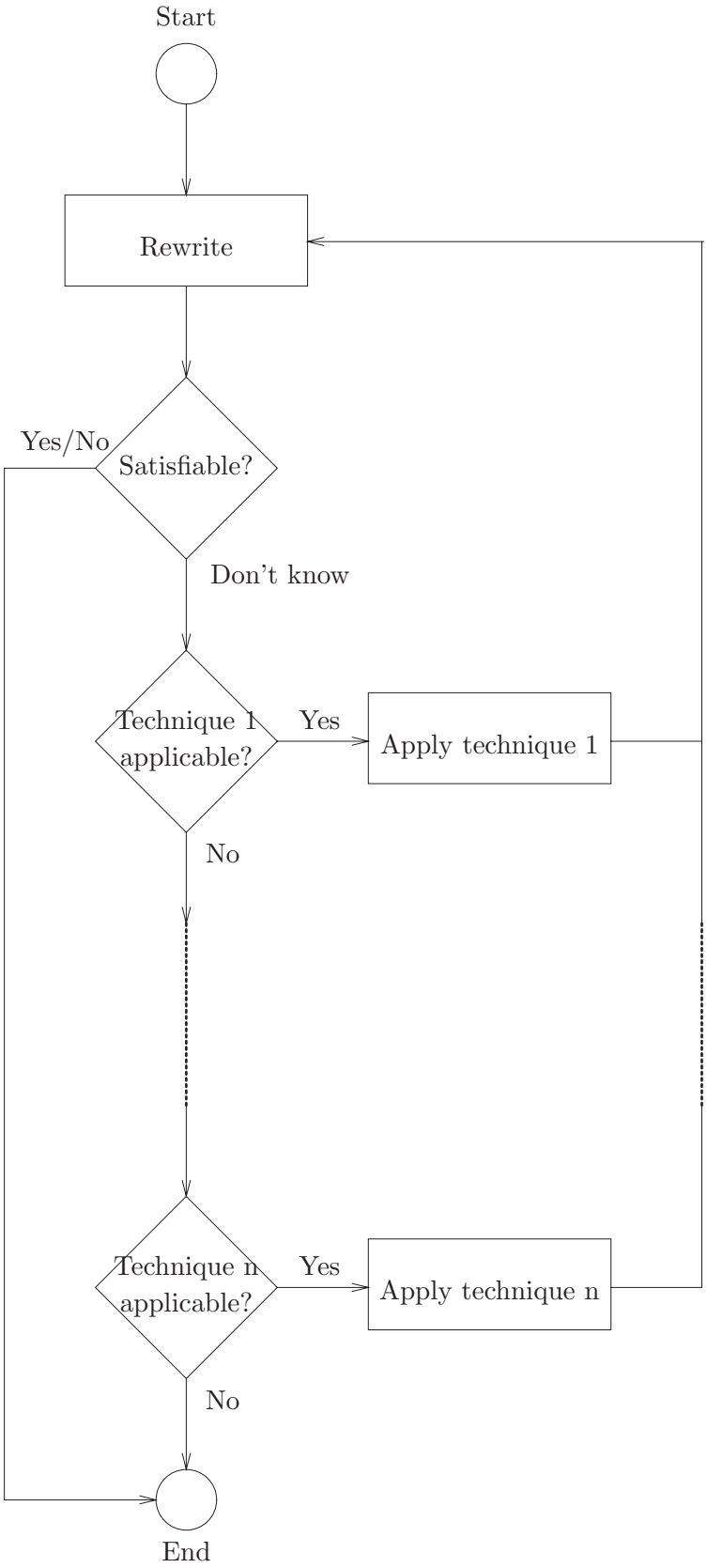

Figure 4.1: Reduction algorithm

- A working copy of the set of clauses.
- A DBM that represents in a compact structure the constraints that exist between numerical variables that occur in the SAT instance.
- A set of variables assignments. These assignments are *absolute* when the value is constant (e.g. $a = true$, $a = false$, or $x = 55$), or *relative* when it includes a reference to one or more variables (e.g. $a = b$, $a = \neg c$, $a = b \vee c$, $x = y + 5$, or $x = min\{z + 2, y - 3\}$).

Each solving context produced during the solution process is fully equivalent to the original SAT instance. Thus the use of the solving context preserves the satisfiability (or the unsatisfiability) of the original SAT instance.

Remark 15 *In the algorithms presented in the sequel, a solving context is denoted as a tuple* $(\mathcal{C}, M, \mathcal{A})$ *where* $\mathcal{C}$ *is a set of clauses,* M *is a DBM, and* $\mathcal{A}$ *is a set of variable assignments.*

4.1.2 Assignments

There are four types of assignments for Boolean variables:

- Constant: the variable is set to $true$ or $false$.
- Literal: the variable is assigned the value of a Boolean or a numerical literal, e.g. $a = b$, $a = \neg b$, or $a = (x - y < 5)$.
- Disjunction: the variable is assigned the value of a clause, e.g.:

$$a = b \vee (x - y < 5)$$

- Conjunction: the variable is assigned the negated value of a clause, e.g.:

$$a = \neg b \wedge (y - x \leq -5)$$

Disjunction and conjunction assignments have at most one numerical literal.

Numerical variables admit three kinds of assignments:

- Constant: the variable is set to a constant value, e.g. $x = 55$.
- Bias: the variable is assigned the value of another numerical variable plus or minus a constant value, e.g. $x = y + 5$ or $x = y - 5$.
- MinMax: the variable is assigned the minimum or the maximum value of a set of biased assignments, e.g. $x = min\{y + 5, z - 4, w + 3\}$ or $x = max\{w + 5, y - 3\}$.

4.1.3 Clause rewriting

Rewriting the set of clauses belonging to the current solving context is an operation that involves assignments deduced so far as well as the DBM of that context. It modifies the set of clauses and eliminates references to variables that are assigned a value whether absolute or relative.

Boolean literal rewriting

Each clause C that contains a Boolean literal denoting an assigned Boolean variable b is rewritten according to the assignment type of b as follows:

- Constant: If $l = b$ and $b = true$ or if $l = \neg b$ and $b = false$, C can be removed. Otherwise, only the literal l is removed from C.
- Literal: b inside l is substituted with λ where λ stands for the literal value assigned to b.
- Disjunction: Assume $b = \lambda_1 \vee \ldots \vee \lambda_k$. If $l = b$, the disjunction $\lambda_1 \vee \ldots \vee \lambda_k$ is or-ed with the clause where l occurs and the reference to l is removed from the latter. In the other case, i.e. when $l = \neg b$, the clause C is replaced by the following conjunction of clauses:

  $$\bigwedge_{i=1}^{k} (C' \vee \neg\lambda_i)$$

  where $C = C' \vee l$, i.e. C' is a copy of C from which references to the literal l were deleted.
- Conjunction: Assume $b = \lambda_1 \wedge \ldots \wedge \lambda_k$. If $l = \neg b$, the disjunction $\neg\lambda_1 \vee \ldots \vee \neg\lambda_k$ is or-ed with the clause where l occurs and the reference to l is removed from that latter. In the other case, i.e. when $l = b$, the clause C is replaced by the conjunction of clauses:

  $$\bigwedge_{i=1}^{k} (C' \vee \lambda_i)$$

  where $C = C' \vee l$, i.e. C' is a copy of C from which references to the literal l were deleted.

Example: Consider the following set of clauses:

$$\begin{array}{lcccccl} & a & \vee & b & & & (1) \\ \wedge & \neg a & \vee & c & \vee & (x - y < 5) & (2) \\ \wedge & \neg b & \vee & d & & & (3) \\ \wedge & e & \vee & (z - w \leq 12) & & & (4) \\ \wedge & \neg e & \vee & d & & & (5) \end{array}$$

Assume also that we have the following assignments:

$$\begin{array}{rcl} a & = & false \\ b & = & \neg c \\ e & = & c \vee f \end{array}$$

Using the assignment of a, rewriting clause (1) gives the unit clause b while clause (2) is removed. Clause (3) is transformed to $c \vee d$. Now, using the

assignment of variable e, the fourth clause becomes $c \vee f \vee (z - w \leq 12)$ and clause (5) is replaced with the conjunction:

$$\begin{array}{rlll} & \neg c & \vee & d \\ \wedge & \neg f & \vee & d \end{array}$$

□

Numerical literal rewriting

Each clause C containing a numerical literal $(x - y \prec c)$ such that x (respectively y) is already assigned a value is rewritten according to the assignment type of the latter as follows:

- Constant: Assume that the value is set to $v \in \mathbb{D}$. Then the numerical literal becomes $(\mathbf{0} - y \prec c - v)$ (respectively $(x - \mathbf{0} \prec c + v)$).
- Bias: If the assignment's value is $z + v$ where z is a variable and $v \in \mathbb{D}$, the numerical literal is substituted with the constraint $(z - y \prec c - v)$ (respectively $(x - z \prec c + v)$).
- MinMax: This type of assignment is not used when rewriting clauses due to its complexity and to its dependency on the information available when it is deduced, i.e. during the application of the Davis-Putnam rules (refer to Section 4.3.2 for a detailed discussion.)

In addition to the above rewriting rules, each numerical literal $(x - y \prec c)$ is tested against the DBM M associated with the current solving context. If $(c, \prec) > M_{xy}$, the clause C is removed from the set of clauses as the value of the numerical literal is $true$. If $(-c, \prec) > M_{yx}$, that literal is removed from C since its value is $false$.

Example: Suppose that the current solving context consists of the following set of clauses:

$$\begin{array}{rlllll} & a & \vee & b & \vee \quad (x - y < 5) & (1) \\ \wedge & \neg a & \vee & (u - v \leq 6) & & (2) \\ \wedge & c & \vee & (z - v \leq 12) & & (3) \\ \wedge & \neg d & \vee & (x - z < 7) & & (4) \end{array}$$

Assume also that we have the following assignments:

$$\begin{array}{rcl} y & = & 6 \\ u & = & x - 3 \end{array}$$

Suppose also that the DBM M associated with the current solving context is:

$$M = \left(\begin{array}{c|cccccc} & \mathbf{0} & x & y & z & u & v \\ \hline \mathbf{0} & (0,\leq) & (\infty,<) & (\infty,<) & (\infty,<) & (\infty,<) & (\infty,<) \\ x & (\infty,<) & (0,\leq) & (\infty,<) & (5,<) & (\infty,<) & (\infty,<) \\ y & (\infty,<) & (\infty,<) & (0,\leq) & (\infty,<) & (\infty,<) & (\infty,<) \\ z & (\infty,<) & (\infty,<) & (\infty,<) & (0,\leq) & (8,\leq) & (\infty,<) \\ u & (\infty,<) & (\infty,<) & (\infty,<) & (\infty,<) & (0,\leq) & (\infty,<) \\ v & (\infty,<) & (\infty,<) & (\infty,<) & (\infty,<) & (\infty,<) & (0,\leq) \end{array}\right)$$

When rewritten using the assignment of y, clause (1) becomes :

$$a \vee b \vee (x - \mathbf{0} < 11)$$

Using the assignment of u, clause (2) is transformed to:

$$\neg a \vee (x - v \leq 9)$$

.

Now, using the DBM M, the constraint $(z - v \leq 12)$ in clause (3) is unchanged since $M_{zv} = (\infty, <)$, which means that z and v have no relation. On the contrary, as $M_{xz} = (5, <)$, clause (4) can be removed since the constraint it contains is *true*. In fact, $(5, <) < (7, <)$, i.e.:

$$(x - z < 5) \Rightarrow (x - z < 7)$$

. □

4.2 Simplification Rules

4.2.1 Boolean unit resolution

This rule consists of setting the single Boolean literal in each unit clause to *true*.

Example: Consider the following set of clauses:

$$\begin{array}{cccccc} & a & \vee & d & \vee & c \\ \wedge & \neg a & & & & \\ \wedge & b & & & & \end{array}$$

Applying the Boolean unit resolution leads to setting both $\neg a$ and b to *true*. Thus, a is assigned the value $false$ and b is assigned the value *true*. □

4.2.2 Boolean literals equivalence detection

A directed graph is built using the Boolean binary clauses of the considered set of clauses, i.e. clauses that contain exactly two Boolean literals. For each clause of the form $l_1 \vee l_2$, where l_1 and l_2 are Boolean literals, two directed edges are added to the graph: the first is from vertex $\neg l_1$ to vertex l_2, and the second is from l_1 to $\neg l_2$. Of course, vertices l_1, $\neg l_1$, l_2, and $\neg l_2$ are added to the graph if they are not already there. These edges represent the implications $\neg l_1 \rightarrow l_2$ and $l_1 \rightarrow \neg l_2$.

Next, the *SCC* (or *Strongly Connected Components*) decomposition algorithm is applied to the freshly built directed graph. Since it represents implications between literals, detecting the strongly connected components identifies the cycles, and thus provides the literals that belong to the same equivalence class. The number of found cycles is always even. In fact, for each detected cycle formed by vertices $l_1, \ldots, l_m$ there exists another cycle in the graph that represents the same equivalence class and whose vertices are $\neg l_1, \ldots, \neg l_m$. This is due to the symmetry of the graph building procedure.

Remark 16 *For a complete description of the SCC decomposition algorithm, refer to Appendix B.*

Example: Consider the following set of clauses:

$$\begin{array}{cccccc} & a & \vee & b & \vee & (x - y \leq 8) \\ \wedge & \neg a & \vee & \neg b & & \\ \wedge & b & \vee & c & & \\ \wedge & a & \vee & \neg c & & \\ \wedge & b & \vee & d & & \end{array}$$

Figure (4.2) shows the implication graph built using the binary clauses $\neg a \vee \neg b$, $b \vee c$, $a \vee \neg c$, and $b \vee d$. Applying the SCC decomposition algorithm on that graph leads to the detection of two cycles: the first is formed by vertices a, $\neg b$, and c; and the second is formed by vertices $\neg a$, b, and $\neg c$.

From the first cycle, an equivalence between a, $\neg b$, and c is deduced and it is safe to set $c = a$ and $b = \neg a$. The second cycle represents an equivalence between $\neg a$, b, and $\neg c$. Obviously, it is the same equivalence as discovered earlier with the first cycle. □

4.2.3 Pure Boolean literals resolution

For each pure Boolean literal l in the set of clauses, l is set to *true*.

Example: Consider the following set of clauses:

$$\begin{array}{cccccc} & a & \vee & b & \vee & c \\ \wedge & \neg a & \vee & b & & \\ \wedge & b & \vee & c & & \\ \wedge & \neg d & \vee & \neg c & & \\ \wedge & b & \vee & \neg d & & \end{array}$$

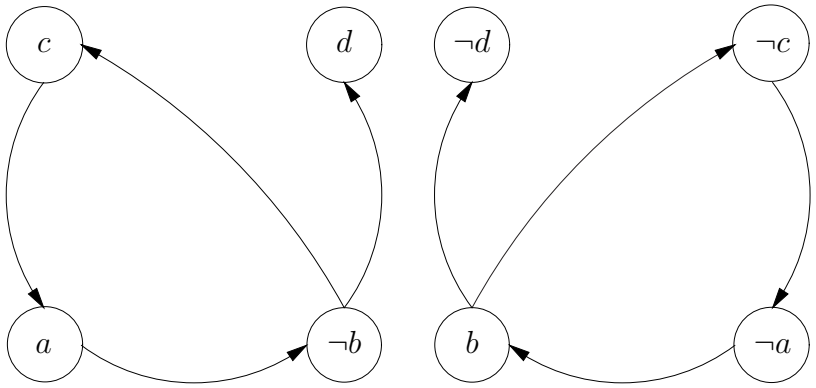

Figure 4.2: Example of a graph built to detect Boolean equivalences

b and $\neg d$ are pure Boolean literals since their negations, $\neg b$ and d, do not occur in the set of clauses. When applying the pure Boolean literals resolution, b is set to $true$ and d to $false$. □

4.2.4 DBM updating

Each unit numerical clause (i.e. each clause that contains one numerical literal $(x - y \prec c)$ and no Boolean literals) is used to update the DBM M associated with the current solving context: The polyhedron $[[M]]$ associated with M is intersected with that constraint $(x - y \prec c)$ and the DBM of the current solving context is replaced with the canonical form of the DBM associated with $[[M]] \cap (x - y \prec c)$, i.e.:

$$M := \mathbf{cf}\big([[M]] \cap (x - y \prec c)\big)$$

If that canonical form is empty, the solving process is stopped and the mixed SAT instance represented by the current solving context is known to be unsatisfiable.

4.2.5 DBM analysis

The DBM M associated with the current solving context is scanned in order to find variables x and y such that:

$$\begin{cases} b_{M_{xy}} = -b_{M_{yx}} \\ \ll_{M_{xy}} = \leq \\ \ll_{M_{yx}} = \leq \end{cases}$$

When such x and y are found, $x = y + b_{M_{xy}}$ holds and that assignment can be safely made. In fact:

$$\begin{cases} x - y \leq b_{M_{xy}} \\ y - x \leq b_{M_{yx}} \end{cases} \Leftrightarrow \begin{cases} x - y \leq b_{M_{xy}} \\ y - x \leq -b_{M_{xy}} \end{cases} \Leftrightarrow \begin{cases} x - y \leq b_{M_{xy}} \\ x - y \geq b_{M_{xy}} \end{cases} \Leftrightarrow x - y = b_{M_{xy}}$$

4.3 Davis-Putnam Rules

The Davis-Putnam rules go beyond the simplification steps presented above. In fact, their application narrows the search space without affecting the satisfiability or the unsatisfiability of the considered set of clauses. They replace the formula corresponding to the current list of clauses by a stronger but equi-satisfiable one.

4.3.1 Restricted Boolean Davis-Putnam rule application

This rule is based on a restricted version of Boolean resolution as described in [DP60]. It is applied to each literal l that has a unique negative occurrence in the set of clauses that can be written as:

$$\bigwedge_{i=1}^{k} C_i \wedge \bigwedge_{i=1}^{k'} (C'_i \vee l) \wedge (C'' \vee \neg l) \tag{4.1}$$

C_i, C'_i, and C'' are clauses or sub-clauses that contain neither l nor $\neg l$. When such a literal l is found, it can be assigned the value of the sub-clause C''. After rewriting the set of clauses, we obtain the formula:

$$\bigwedge_{i=1}^{k} C_i \wedge \bigwedge_{i=1}^{k'} (C'_i \vee C'') \tag{4.2}$$

which has one less variable. Note that equation (4.2) is not equivalent to (4.1). However, both are equi-satisfiable.

Applying this rule is not straightforward, and provisions must be taken in order to avoid the production of malformed mixed clauses, i.e. clauses that have more than one numerical literal. Therefore, the rule is not applied when such a problem can occur after assigning a value to the literal l and rewriting the rest of the clauses. More formally, this rule is used only when:

$$max\{\nu(C'_i) | 1 \leq i \leq k'\} + \nu(C'') \leq 1$$

where $\nu(Z)$ denotes the number of numerical literals in the clause Z.

Example: Consider the following set of clauses:

$$\begin{array}{cccccc} & a & \vee & b & \vee & (x - y < 5) \\ \wedge & \neg a & \vee & c & & \\ \wedge & \neg a & \vee & d & & \end{array}$$

The literal $\neg a$ is a candidate to the application of the restricted Boolean Davis-Putnam rule. Thus we can set:

$$\neg a = b \vee (x - y < 5)$$

In other words, the variable a is assigned the conjunction $\neg b \wedge (y - x \leq -5)$. Rewriting the above set of clauses gives:

$$\begin{array}{cccccc} & b & \vee & (x - y < 5) & \vee & c \\ \wedge & b & \vee & (x - y < 5) & \vee & d \end{array}$$

□

4.3.2 Restricted numerical Davis-Putnam rule application

This rule is an adaptation of the restricted Boolean Davis-Putnam rule described above for numerical constraints. It can also be considered as a special case of the Fourier-Motzkin variable elimination [DE73]. The idea is illustrated using the following example:

Example: Assume the following set of clauses:

$$\begin{array}{rcccc} & a & \vee & b & \vee & (x-y \leq 5) \\ \wedge & (y-z \leq 7) & & & & \\ \wedge & (y-w \leq 2) & & & & \end{array}$$

From both clauses where y occurs positively, we can deduce:

$$\left\{ \begin{array}{l} y-z \leq 7 \\ y-w \leq 2 \end{array} \right. \Leftrightarrow \left\{ \begin{array}{l} y \leq z+7 \\ y \leq w+2 \end{array} \right. \Leftrightarrow y \leq min\{z+7, w+2\}$$

Thus, y can be set to $min\{z+7, w+2\}$ and substituted in $(x-y \leq 5)$ to obtain:

$$\begin{array}{ll} \left\{ \begin{array}{l} x-y \leq 5 \\ y = min\{z+7, w+2\} \end{array} \right. & \Leftrightarrow \quad x \leq 5 + min\{z+7, w+2\} \\ & \Leftrightarrow \quad \left\{ \begin{array}{l} x \leq 5+z+7 \\ x \leq 5+w+2 \end{array} \right. \\ & \Leftrightarrow \quad \left\{ \begin{array}{l} x-z \leq 12 \\ x-w \leq 7 \end{array} \right. \end{array}$$

Therefore, the above set of clauses becomes:

$$\begin{array}{rcccc} & a & \vee & b & \vee & (x-z \leq 12) \\ \wedge & a & \vee & b & \vee & (x-w \leq 7) \end{array}$$

□

This example explains roughly how that rule works. In our case, unit numerical clauses are already integrated in the DBM M associated with the current solving context. So we first select a numerical variable x occurring *a few times* positively in M. Let $(x-y_1 \leq c_1), \ldots, (x-y_k \leq c_k)$ be the constraints deduced from M where occurrences of x are positive.

Next, x is set to $min\{y_1+c_1, \ldots, y_k+c_k\}$. Each constraint $(z-x \prec c)$ in the set of clauses, in which x occurs negatively, is replaced by the conjunction:

$$(z-y_1 \prec c-c_1) \wedge \ldots \wedge (z-y_k \prec c-c_k)$$

M is also updated in order to remove references to x:

- For each numerical variable $z \in \mathcal{X} \cup \{\mathbf{0}\} \backslash \{x\}$ such that $M_{zx} \neq (\infty, <)$, and for each $1 \leq i \leq k$, M_{zy_i} is set to $M_{zy_i} \oplus (-c_i, \leq)$, and M_{zx} is reset to $(\infty, <)$.

- For each $z \in \mathcal{X} \cup \{\mathbf{0}\} \backslash \{x\}$, M_{xz} is reset to $(\infty, <)$.

Analogously, we can do the same for a numerical variable y that occurs *a few times* negatively in M. Let $(x_1 - y \leq c_1), \ldots, (x_k - y \leq c_k)$ be the constraints deduced from M where occurrences of y are negative. y is set to $max\{x_1 - c_1, \ldots, x_k - c_k\}$. Each constraint $(y - z \prec c)$ in the set of clauses, where y occurs negatively, is substituted by the conjunction:

$$(x_1 - z \prec c + c_1) \wedge \ldots \wedge (x_k - z \prec c + c_k)$$

M is also updated in order to remove references to y:

- For each numerical variable $z \in \mathcal{X} \cup \{\mathbf{0}\} \backslash \{y\}$ such that $M_{yz} \neq (\infty, <)$, and for each $1 \leq i \leq k$, $M_{x_i z}$ is set to $M_{x_i z} \oplus (c_i, \leq)$, and M_{yz} is reset to $(\infty, <)$.

- For each $z \in \mathcal{X} \cup \{\mathbf{0}\} \backslash \{y\}$, M_{zy} is reset to $(\infty, <)$.

Remark 17 *This rule does not affect the satisfiability of a given SAT instance. In fact, for a numerical variable x that occurs positively as described above, any valuation v satisfying that instance must satisfy $v(x) \leq v(y_i) + c_i$. The valuation $v' = v[x := min\{y_1 + c_1, \ldots, y_k + c_k\}]$ where only the value of x is changed such that $v'(x) \leq v(x)$ also satisfies the instance. In fact, it satisfies the constraints $x - y_i \leq c_i$ for all $i \in \{1, \ldots, k\}$. It satisfies also all the constraints where x occurs negatively.*

A similar reasoning can be made for numerical variables that occur a few times negatively.

Remark 18 *This rule is applied when a numerical variable occurs* a few times *positively or negatively. This is a vague quantifier but it is up to the implementor of the solving algorithm to define how much* a few *is. This depends on numerous factors especially on how much gain in performance the elimination of a numerical variable can provide.*

4.4 DPLL Branching

The global algorithm of the mixed SAT solver is based on DPLL as described in Chapter 3. The solver reduces the current context. If it is found to be unsatisfiable or satisfiable, it stops. Otherwise, it will choose a free variable, i.e. a variable v that is not already assigned, and branch on it. The first branch is with $v = true$ and the second with $v = false$. In each branch, the same algorithm is applied recursively.

Reducing the current context consists of successive applications of the simplification steps and the restricted Davis-Putnam rules. Figure (4.1) depicts how the reduction is achieved.

Algorithm 7 (Recursive DPLL)

MXSolve$(C, M, \mathcal{A})$
begin
$(C, M, \mathcal{A}) :=$ **reduce**$(C, M, \mathcal{A})$

if is-unsatisfiable$(C, M, \mathcal{A})$
then
return $(C, M, \mathcal{A})$

if no-free-variable$(C, M, \mathcal{A})$
then
return $(C, M, \mathcal{A})$

$v :=$ **choose-free-variable**$(C, M, \mathcal{A})$
$(C', M', \mathcal{A}') :=$ **MXSolve**$(C, M, \mathcal{A} \cup \{v = \mathit{true}\})$

if is-satisfiable$(C', M', \mathcal{A}')$
then
return $(C', M', \mathcal{A}')$
else
return MXSolve$(C, M, \mathcal{A} \cup \{v = \mathit{false}\})$
end

Remark 19 *Since* **MXSolve** *does not return explicitly the satisfiability of a given context, it should be invoked from another function that tests the returned context using* **is-unsatisfiable** *or* **is-satisfiable**.

While the above recursive DPLL algorithm is good for didactic purposes, it turns out to be practically inefficient because recursion adds a processing overhead at each nesting level. Besides, advanced features such as non-chronological backtracking[1] and restarts are hard to integrate into such a recursive algorithm. For all these reasons, a non-recursive version of the algorithm has been designed (See Algorithm 8).

Recursion is avoided by using an array of contexts and by augmenting each context with its *state* indicator and its current *branching variable* if any. The state of a context gives information about what the algorithm is currently doing:

- The state is 0 when the solver is reducing the current context.
- The state is 1 when the solver is processing the left branch after it has chosen a free variable for branching on.
- The state is 2 when the solver is processing the right branch.

In the following algorithm, i represents the current branching level. Each context at level i is made of:

- A set of clauses C_i;

[1] Backtracking by more than one level at a time

- A DBM M_i;
- A set of assignments $\mathcal{A}_i$;
- A state indicator s_i;
- A branching variable v_i;

Algorithm 8 (Non-Recursive DPLL)

```
MXSolve(C, M, A)
begin
    (C_1, M_1, A_1) := (C, M, A)
    s_1 := 0
    i := 1

    while (i > 0)
    begin
        (C_i, M_i, A_i) := reduce(C_i, M_i, A_i)

        if is-unsatisfiable(C_i, M_i, A_i)
        then
            begin
                i := i - 1
                continue
            end

        if is-satisfiable(C_i, M_i, A_i)
        then
            return (C_i, M_i, A_i)

        if (s_i = 0)
        then
            begin
                if no-free-variable(C_i, M_i, A_i)
                then
                    begin
                        i := i - 1
                        continue
                    end
                else
                    begin
                        v_i := choose-free-variable(C_i, M_i, A_i)
                        s_i := 1
                    end
            end
```

```
            if (s_i = 1)
            then
                value := true

            if (s_i = 2)
            then
                value := false

            if (s_i = 3)
            then
                begin
                    i := i − 1
                    continue
                end

            if (s_i ∈ {1, 2})
            then
                begin
                    s_i := s_i + 1
                    (C_{i+1}, M_{i+1}, A_{i+1}) := (C_i, M_i, A_i ∪ {v_i = value})
                    s_{i+1} := 0
                    i := i + 1
                end

            end
        end

        return (C_1, M_1, A_1)
end
```

Remark 20 *In the above algorithm, a new keyword* **continue** *is used. It stops the execution of the current* **while** *loop and restarts it.*

The non-recursive algorithm makes backtracking as easy as setting the level variable i and restarting the loop (by using **continue**). This permits adding conflict analysis and restarts to the algorithm with minimum changes as can be seen in Algorithm 9 where the new or modified lines are marked with the ◄ symbol.

Algorithm 9 (Non-Recursive DPLL with conflict analysis and restarts)

```
MXSolve(C, M, A)
begin
    (C_1, M_1, A_1) := (C, M, A)
    s_1 := 0
    i := 1
```

```
while (i > 0)
begin
    (C_i, M_i, A_i) := reduce(C_i, M_i, A_i)

    if is-unsatisfiable(C_i, M_i, A_i)
    then
        begin
            (C, M, A, i) := conflict-analysis(C, M, A, i) ◄
            continue
        end

    if is-satisfiable(C_i, M_i, A_i)
    then
        return (C_i, M_i, A_i)

    if (s_i = 0)
    then
        begin
            if restart(C_i, M_i, A_i, i) ◄
            then ◄
                begin ◄
                    i := 1 ◄
                    continue ◄
                end ◄

            if no-free-variable(C_i, M_i, A_i)
            then
                begin
                    i := i − 1
                    continue
                end
            else
                begin
                    v_i := choose-free-variable(C_i, M_i, A_i)
                    s_i := 1
                end
        end

    if (s_i = 1)
    then
        value := true

    if (s_i = 2)
    then
        value := false
```

```
            if (s_i = 3)
            then
                begin
                    i := i − 1
                    continue
                end

            if (s_i ∈ {1, 2})
            then
                begin
                    s_i := s_i + 1
                    (C_{i+1}, M_{i+1}, A_{i+1}) := (C_i, M_i, A_i ∪ {v_i = value})
                    s_{i+1} := 0
                    i := i + 1
                end

            end
        end

        return (C_1, M_1, A_1)
end
```

The updated version of the non-recursive DPLL with conflict analysis and restarts uses two functions that can be customized at will by the implementor of the algorithm:

- **conflict-analysis**$(C, M, \mathcal{A}, i)$: It does the conflict analysis on an unsatisfiable context at level i. It operates on the entire array of contexts $(C, M, \mathcal{A})$ and updates all the contexts with the newly learned clauses. In fact, each learned clause is valid regardless of the current branching configuration. **conflict-analysis** returns the new level to which the algorithm must backtrack.

- **restart**$(C_i, M_i, \mathcal{A}_i, i)$: It decides whether to restart the solving at level 1 or continue. This decision is based upon the current context and the current branching level.

Other functions, that are already used in the previous versions of the algorithm, can also be subject to customizations:

- **reduce**$(C_i, M_i, \mathcal{A}_i)$: It returns a copy of the current context to which the simplification steps and the restricted Davis-Putnam rules have been applied.

- **choose-free-variable**$(C_i, M_i, \mathcal{A}_i)$: It selects a free variable to branch on, and it can do so using various heuristics.

To summarize, Algorithm 9 can be considered as a template for a mixed solving algorithm with at least four parameters, namely the implementations

of **reduce**, **choose-free-variable**, **conflict-analysis**, and **restart**. The customization can go further since the implementor has a wide choice of what methods to use in these functions.

4.5 Conflict Analysis

Conflict analysis used in the mixed solver is a generalization of the conflict analysis technique used in Boolean SAT solvers and already described in Chapter 3.

4.5.1 Extension of solving contexts

In order to use conflict analysis, clauses, DBM bounds, and assignments must be extended. The extension consists in augmenting each of these context elements with a set of variables that represent the reasons that resulted in that element change or setting.

An extended clause is a pair $Z^* = (Z, R)$ where Z is a clause and $R = \{v_1, \ldots, v_k\}$ is the set of variables whose assignments lead to the current form of the clause. Likewise, an extended DBM M^* is a DBM whose entries are extended bounds, i.e pairs $M^*_{xy} = (M_{xy}, R')$ where R' is the set of variables whose assignments lead to the inclusion of M_{xy} in the DBM. Finally, an extended assignment is a pair $A^* = (A, R'')$ where R'' is the set of variables whose assignments lead to A. We will use the following notations in the rest of the section:

$$\begin{array}{rcll} clause(Z^*) & = & Z & \qquad reason(Z^*) = R \\ bound(M^*_{xy}) & = & M_{xy} & \qquad reason(M^*_{xy}) = R' \\ assignment(A^*) & = & A & \qquad reason(A^*) = R'' \end{array}$$

From now on, we assume solving contexts to be of the form $(C^*, M^*, \mathcal{A}^*)$ where C^* is a set of extended clauses, M^* is an extended DBM, and $\mathcal{A}^*$ is a set of extended assignments.

4.5.2 Changes in operations

The extension of the components of the solving contexts changes slightly the operations that can be applied to them as explained below.

Assignments

Assume that the extended clauses involved in a given simplification step or in an application of a Davis-Putnam rule are $\{C^*_1, \ldots, C^*_m\} \subset C^*$. A clause is said to be involved in a step when it is used to decide if that step can be applied. If that step results in an assignment $v = V$, the extended assignment $(v = V, R)$ is appended to $\mathcal{A}^*$, where:

$$R = \bigcup_{j=0}^{m} reason(C^*_j)$$

Clause rewriting

For each extended clause C_i^* in the current solving context that is rewritten based on an assignment $v = V$, the variable v is added to its set of reason variables, i.e.:

$$reason(C_i^*) := reason(C_i^*) \cup \{v\}$$

During a rewrite, if a numerical literal $(x - y \prec c)$ in an extended clause C_i^* is found to be *true* because $(c, \prec) > bound(M_{xy}^*)$, then:

$$reason(C_i^*) := reason(C_i^*) \cup reason(M_{xy}^*)$$

Likewise, if it is found to be $false$ because $(-c, \prec) > bound(M_{yx}^*)$, then

$$reason(C_i^*) := reason(C_i^*) \cup reason(M_{yx}^*)$$

DBM updating

If an extended clause C_i^* contains only a numerical literal, that literal is candidate to be included in the DBM M^*. In such a case, the updated bound of the DBM has its reason set to $reason(C_i^*)$. If adding that constraint to the DBM results in an empty matrix, a conflict is detected

During the update of an extended DBM, reasons associated with the bounds must be also set. For that purpose, the Floyd-Warshall algorithm already described on page 20 is extended in order to record and update the set of reason variables related to each bound of the DBM.

Algorithm 10 (Extended Floyd-Warshall)

ExtendedFloydWarshall(M^*)
begin
  for $i := 1$ **to** n
    for $j := 1$ **to** n
      for $k := 1$ **to** n
      begin
        $b := bound(M_{x_i x_j}^*) \oplus \big(bound(M_{x_i x_k}^*) \otimes bound(M_{x_k x_j}^*)\big)$
        if $\big(b \neq bound(M_{x_i x_j}^*)\big)$
        begin
          $reason(M_{x_i x_j}^*) := reason(M_{x_i x_k}^*) \cup reason(M_{x_k x_j}^*)$
          $bound(M_{x_i x_j}^*) := b$
        end
      end
end

DPLL branching

When the solver selects a variable v to branch on, it adds a new assignment of v to $\mathcal{A}^*$ with an empty set of reason variables.

4.5.3 Conflict detection and analysis

Now, consider an extended clause C_i^* in the current solving context. If rewriting the clause of C_i^* gives the $false$ clause, a conflict is deduced and the solver uses $R = reason(C_i^*)$ to build the implication graph.

Another type of conflicts arises when the canonization of a DBM gives an empty matrix. In such a case, there is at least one numerical variable x that satisfies $\mathbf{fw}(M)_{xx} < 0$, i.e. there is at least one negative cycle after the application of the Floyd-Warshall algorithm (see Lemma 2 on page 19). The following set of variables R is used to build the implication graph:

$$R = \bigcup_{(x,y)\in Z} reason(M_{xy}^*)$$

where:

$$Z = \left\{(x,y) \in (\mathcal{X} \cup \{\mathbf{0}\})^2 | \mathbf{fw}(M)_{xx} < 0\right\}$$

In both cases, invoking **ImplicationGraph**(R) generates the implication graph that ends in a conflict. The following algorithm uses $reason(r)$ that denotes, by extension, the set of reason variables associated with the assignment of the variable r.

Algorithm 11 (Implication graph) [2]

ImplicationGraph(R)
begin
    $V(G) := \varnothing$
    $E(G) := \varnothing$

    for each $r \in R$
        $G := G \cup$ **ImplicationGraphOfVariable**$\big(reason(r), r\big)$

    return G
end

ImplicationGraphOfVariable(R, v)
begin
    $V(G) := \{v\}$
    $E(G) := \varnothing$

    for each $r \in R$
    begin
        $V(G) := V(G) \cup \{r\}$
        $E(G) := E(G) \cup \{\langle r, v\rangle\}$
    end

[2] The algorithms use the directed graph notations presented in Appendix B.

for each $r \in R$
$G := G \cup$ **ImplicationGraphOfVariable**$\big(reason(r), r\big)$

return G
end

Notice that the vertices of the graph are made of variables and not of assignments. This is not a limitation since a full assignment can be easily retrieved from a variable name by doing a lookup in the set of assignments of the current solving context.

Building the full implication graph is not required as only a portion of it is relevant to perform the conflict analysis. Thus, **ImplicationGraphOfVariable** can be modified in order to return an empty implication graph when the variable v passed as a parameter does not satisfy a given property. For example, such a property can test the branching level at which the variable was assigned a value.

That graph is then analyzed to extract a learned clause that will be added to the SAT instance to prevent the occurrence of the same conflict in the future.

4.6 MX-Solver Implementation

4.6.1 Technical overview

The current MX-Solver implementation is written in ANSI C. It consists of about 6500 lines of code. It has been successfully tested on many 32-bit and 64-bit platforms powered by Windows or popular UNIX flavors (such as Linux and Solaris).

Since speed and stability are the major concerns, MX-Solver does not rely on external libraries: The implementation is self-contained and only uses some I/O and memory allocation routines from the standard C library. Besides, extensive code reuse inside the implementation has been practiced in order to reduce the size of the resulting executable code which is around 55 KB. From a technical point of view, these consented *restrictions* are required in order to get the most from the solver. In fact, the smaller the executable code is, the longer it will reside in the processor cache, and the faster will be its execution.

A second version of the solver, called MX3-Solver, has been implemented as well. It is restricted to formulae in MX-3-CNF, i.e. every clause has at most three literals and at most one of them can be numerical. MX3-Solver shares 75% of its code with MX-Solver. In the sequel, we will refer to both versions by MX-Solver except for cases where we mention their differences.

4.6.2 Data structures

MX-Solver relies on several data structures that store the information and the attributes related to the objects it manipulates. Figure (4.3) shows the dependencies between them.

In the sequel, we discuss some of the main data structures used in MX-Solver and describe, at some level of detail, how they are implemented.

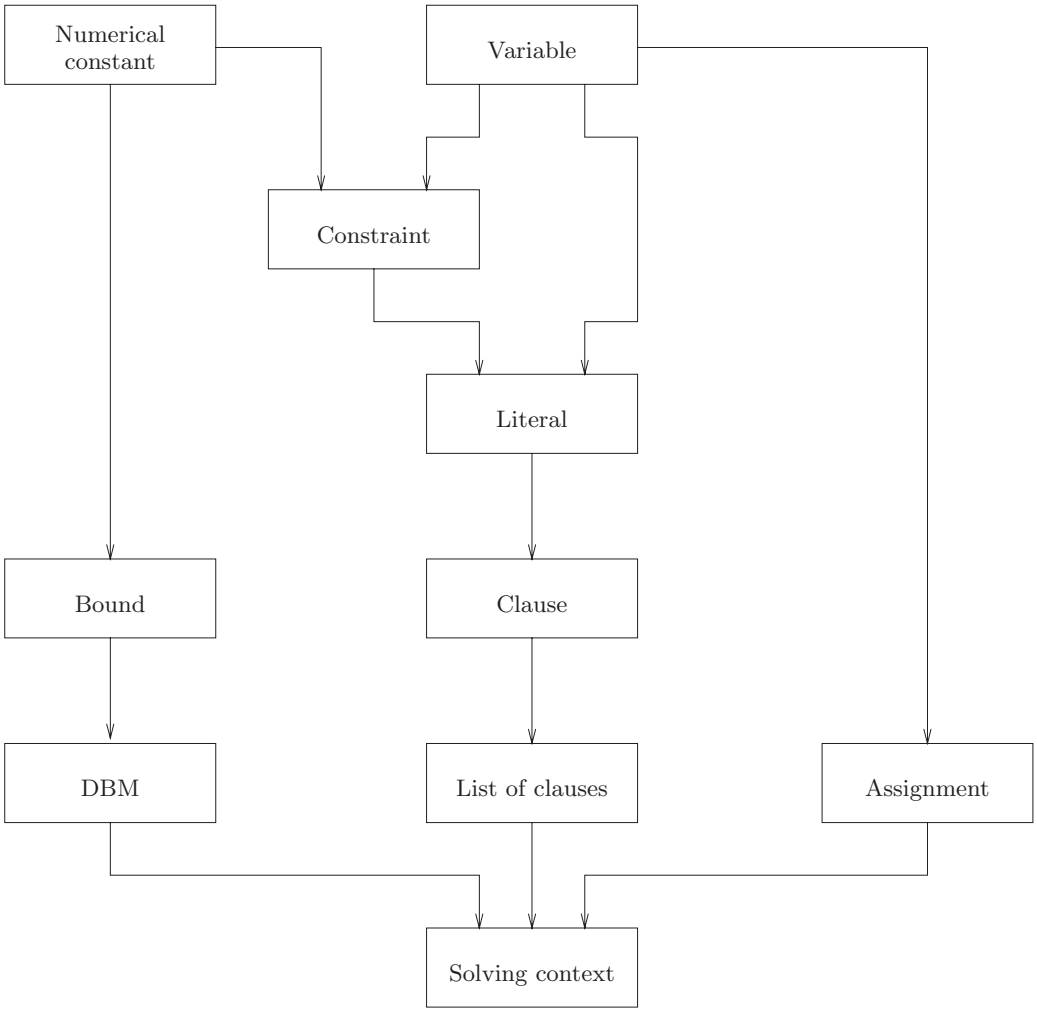

Figure 4.3: Data Structures in MX-Solver

Clause

In MX3-Solver, clauses are MX-3-CNF. Each clause is stored as a static array of three literals. On the other hand, MX-Solver does not set any limitation on the size of the clauses. Consequently, they are implemented as a dynamic array of literals. But due to performance reasons, this simple way of describing clauses has been abandoned at an early stage in favor of a more complex definition. Thus, a clause is made of static array that has room for three literals. It has also the ability to contain more literals in an additional dynamic array.

List of clauses

A list of clauses consists of seven dynamic arrays, each of them is dedicated to store a specific type of clauses. The first six arrays contain respectively clauses of the form b, $b \vee b$, $b \vee b \vee b$, c, $b \vee c$, and $b \vee b \vee c$, where b represents a boolean literal and c a constraint. These enumerated types represent all the possible kinds of MX-3-CNF clauses. Clauses that do not fit in one these forms go in the seventh array. This definition of the list of clauses makes browsing easy, especially if the search is restricted to one or a few types of clauses.

DBM

The set $\mathbb{D}$ in the MX-Solver is the set of integers $\mathbb{Z}$. This results in the simplification of bounds: a bound is reduced to a single integer. In fact, a bound $(z, <)$ is equivalent to $(z - 1, \leq)$ (See also Remark 2 on page 6). Thus, all bounds have the form $(z, \leq)$ and only the integer z is needed.

A DBM can then be defined as a $(n + 1)$-square matrix of integers, where n is the number of numerical variables referenced in the mixed SAT instance. The first row and the first column of each DBM refer always to the special $\mathbf{0}$ variable.

Solving context

A solving context is a structure containing mainly:

- A list of clauses;
- A DBM;
- An array of assignments. This array has a fixed size as the number of the variables referenced in the mixed SAT instance is known in advance.
- An array of statistics on variables. Useful information about each variable, such that the number of its positive and negative occurrences in the current list of clauses, is kept in that array and is frequently updated.

4.6.3 Features

MX-Solver starts by reading a mixed CNF formula from an input file. This formula is used to initialize the main solving context. Next, the solver processes that context in order to find if it is satisfiable.

MX-Solver relies on many of the ideas already presented in Chapter 4. The current implementation is based on the DPLL algorithm and features all the reduction rules described in Chapter 4.

Rule	MX-Solver	MX-3-Solver
Unit resolution	√	√
Pure literals resolution	√	√
Equivalence detection	√	√
DBM updating	√	√
DBM Analysis	√	√
Restricted Boolean DP	√	
Restricted numerical DP	√	√

MX-3-Solver does not implement the restricted Davis-Putnam rules as they can produce clauses that are not MX-3-CNF.

Both solvers are fully configurable at the command line: the user can restrict the set of reduction rules that should be applied. He/she can also set a maximum branching depth for the DPLL algorithm. More information about using the solvers is available in Appendix A.

4.6.4 Design guidelines

Implementing a solver results in the accumulation of a non-negligible *know how*. In the sequel, general programming advice, based on my experience with MX-Solver design, is given.

- Improve the locality of the data: Modern processors use memory caches as an intermediate layer to make memory accesses faster. But if the program does frequent random accesses outside the cache scope, the performance becomes poor as the processor spends most of its time flushing and loading the cache. As a consequence, it is highly recommended to have contiguous data especially when executing algorithms that require frequent accesses to the memory.

- Minimize the dependency of the critical code on external libraries: This could be rephrased into "Improve the locality of the code". In fact, for the same caching consideration mentioned above, the more the code is self-contained, the faster it is. Self-containment means here that the code does not rely, or rely a little, on other libraries and system calls. For that reason, static linking, i.e. incorporation of library routines in the generated executable file, should be preferred over dynamic linking.

- Avoid as much as possible conditional statements in computing intensive portions of the code: In such portions, conditional structures dramatically slow the execution. This is due to the fact that conditions prevent the processor from using its code pipelines and its unordered execution features, and thus participate in lowering temporarily its performance.

- Avoid recursions: Although elegant, recursive procedures have generally a large overhead due to the stack pushes and pops they require. Besides, stack overflow error handlers are rarely written by programmers, which leads to a *fragile* code.

- Forget common sense and use a profiler: When it comes to improving the performance of the solver, the best is to use a profiler to locate the functions that are *really* time consuming. This advice is not so futile as it seems to be. In fact, programmers tend to rely on their common sense or their subjective convictions (related to how they perceive the complexity of an algorithm) to designate functions that need improvements. But those functions are seldom responsible for the poor performance of the program. Usually, small and frequently called functions are the main speed bottlenecks. Profiling the program on a wide range of sample executions can help identifying them.

The choice of the C language to write MX-Solver was motivated by the above advice and remarks I have made during the implementation of its first prototypes. In fact, its compiled code is highly optimized, it allows full control over the data structures, and it is backed by powerful development tools such as debuggers and profilers. Moreover, the use of C makes MX-Solver portable and *virtually* compatible with any modern computing platform.

Chapter 5

Timed Systems

Timed systems are systems whose behavior and dynamics are dependent on time. In this chapter, we focus on three classes of timed system models, namely, timed automata, asynchronous digital circuits, and non-preemptive job-shops.

For each class, we show how systems belonging to it are modeled and how possible behaviors of such models can be expressed using difference logic formulae.

5.1 Timed Automata

Timed automata are automata augmented with clocks [AD94]. They are useful to model systems with timing constraints and where states depend on time. A timed automaton behavior is made of an alternation of discrete states (as in a classical automaton) and time passage. During time passage, all clock values grow at a uniform rate. Conditions on clock values enable transitions or disable staying in state. Besides clock values can be reset during transitions.

5.1.1 Flat timed automata

In the sequel, we give a bunch of basic definitions related to flat timed automata.

Definition 20 (Time constraints) *A* time constraint *is a constraint of the form* $(x - y \prec c)$ *or* $x \prec c'$ *where* x *and* y *are clocks,* $\prec \in \{<, \leq\}$, $c \in \mathbb{D}$, *and* $c' \in \mathbb{D}^+$. $\mathbb{D}^+$ *is defined as follows:*

$$\mathbb{D}^+ = \{x \in \mathbb{D} \mid x \geq 0\}$$

Definition 21 (Timed automaton) *A* timed automaton *is a tuple* $\mathcal{A}$ *such that:*

$$\mathcal{A} = (\mathcal{Q}, \mathcal{X}, q_0, \Delta, \mathcal{S})$$

where:

- $\mathcal{Q}$ *is a finite set of* discrete states*;*
- $\mathcal{X}$ *is a finite set of clocks whose values range over* $\mathbb{D}$*;*

- q_0 *is the* initial state*;*
- Δ *is a finite set of transitions of the form* (q, Π, R, q') *where* $q, q' \in \mathcal{Q}$ *are the source and the target discrete states,* Π *is the guard of the related transition made of a conjunction of clock constraints on* $\mathcal{X}$ *defining a convex* $\mathcal{X}$*-polyhedron, and* R *is a subset of* $\mathcal{X}$ *that contains clocks to be reset when the transition is made.* Δ *is also called the* transition relation*;*
- $\mathcal{S}$ *is a function that associates to each discrete state* q *a conjunction of clock constraints on* $\mathcal{X}$ *defining a convex* $\mathcal{X}$*-polyhedron.* $S(q)$ *defines the staying conditions for* q*.*

Definition 22 (Clock valuation) *A* clock valuation *is a function:*

$$\mathbf{v} : \mathcal{X} \to \mathbb{D}^+$$

The set of clock valuations is denoted $\mathcal{C}$*.*

Definition 23 (Clock reset function) *A reset function on a subset of clocks* R *is defined by:*

$$\begin{array}{rccl} Reset_R : & \mathcal{C} & \to & \mathcal{C} \\ & \mathbf{v}(x) & \mapsto & \left\{ \begin{array}{ll} 0 & \text{if } x \in R \\ \mathbf{v}(x) & \text{otherwise} \end{array} \right. \end{array}$$

In other words, a clock reset function sets to zero all the clocks in R *and does not modify the others.*

Definition 24 (State) *A* state *is a pair* $(q, \mathbf{v})$ *where* $q \in \mathcal{Q}$ *and* $\mathbf{v}$ *is a clock valuation.*

Remark 21 (Initial state) *The initial state of the timed automaton* $\mathcal{A}$ *is* $(q_0, \mathbf{0})$ *where* $\mathbf{0}$ *is the clock valuation:*

$$\begin{array}{rccl} \mathbf{0} : & \mathcal{X} & \to & \mathbb{D}^+ \\ & x & \mapsto & 0 \end{array}$$

Definition 25 (Transition) *A* transition *is one of the following:*

- *A* discrete transition
$$(q, \mathbf{v}) \xrightarrow{\delta} (q', \mathbf{v}')$$
where $\delta = (q, \Pi, R, q') \in \Delta$*, such that* $\mathbf{v}$ *satisfies* Π *and* $\mathbf{v}' = Reset_R(\mathbf{v})$*.*
- *A* time transition
$$(q, \mathbf{v}) \xrightarrow{\theta} (q, \mathbf{v}')$$
where the delay $\theta \geq 0$ *and* $\mathbf{v}' = \mathbf{v} + \theta \cdot \mathbf{1}$*,* $\mathbf{1}$ *is the clock valuation defined as follows:*
$$\begin{array}{rccl} \mathbf{1} : & \mathcal{X} & \to & \mathbb{D}^+ \\ & x & \mapsto & 1 \end{array}$$
In such a time transition, $\mathbf{v}'$ *satisfies* $S(q)$*.*

Remark 22

- *The concatenation of two time transitions* $(q, \mathbf{v}) \xrightarrow{\theta} (q, \mathbf{v} + \theta \cdot \mathbf{1})$ *and* $(q, \mathbf{v}) \xrightarrow{\theta'} (q, \mathbf{v} + \theta' \cdot \mathbf{1})$ *is the time transition* $(q, \mathbf{v}) \xrightarrow{\theta+\theta'} \big(q, \mathbf{v} + (\theta + \theta') \cdot \mathbf{1}\big)$.
- *Inversely, a time transition can be divided into* k *time transitions*

$$(q, \mathbf{v}) \xrightarrow{\theta_1} (q, \mathbf{v} + \theta_1 \cdot \mathbf{1}) \xrightarrow{\theta_2} \ldots \xrightarrow{\theta_k} (q, \mathbf{v} + \theta_k \cdot \mathbf{1})$$

  such that $\sum_{i=1}^{k} \theta_i = \theta$. *If* $\mathbb{D} = \mathbb{Z}$, $k \leq \theta$. *If* $\mathbb{D} = \mathbb{R}$, *due to the dense nature of the reals,* k *can be unbounded, i.e.* $k \geq 1$.
- *An* idle time transition *is a time transition with a zero delay, i.e.:*

$$(q, \mathbf{v}) \xrightarrow{0} (q, \mathbf{v})$$

Definition 26 (Finite run) *A* finite run *of a timed automaton is a finite sequence of transitions:*

$$(q_1, \mathbf{v_1}) \xrightarrow{z_1} (q_2, \mathbf{v_2}) \xrightarrow{z_2} \ldots \xrightarrow{z_k} (q_k, \mathbf{v_k})$$

A finite run with no two consecutive time transitions is called a minimal run.

5.1.2 Translation into DL

Consider a timed automata $\mathcal{A} = (\mathcal{Q}, \mathcal{X}, q_0, \Delta, \mathcal{S})$ that is being translated into difference logic.

State encoding

To encode states, each element of $\mathcal{Q}$ must be encoded using Booleans. In the sequel, we will use a set of Booleans $\mathcal{B}$ to encode elements of $\mathcal{Q}$. $\Phi_q(\mathcal{B})$ is the formula over those Boolean variables denoting state q. We will also use respectively $\mathcal{B}'$ and $\mathcal{X}'$ to represent the values of states and clock variable after a step is made.

The most compact encoding, i.e. the encoding that requires the least number of Booleans, consists in associating each state with a formula containing the conversion of its rank in $\mathcal{Q}$ to a binary basis.

Assume that $\mathcal{Q} = \{q_0, \ldots, q_m\}$. In that case, we define $\mathcal{B} = \{b_1, \ldots, b_k\}$ such that $k = \text{ceiling}(\ln(m+1)/\ln 2)$, where:

$$\forall n \in \mathbb{Z},\ \forall x \in]n-1, n],\ \text{ceiling}(x) = n$$

The formula denoting state q_i is:

$$\Phi_{q_i}(\mathcal{B}) = \bigwedge_{j=1}^{k} {b_j}^{\beta(i,j)}$$

where:

$$i = \sum_{j=1}^{k} \beta(i,j).2^{j-1}$$

and:

$$b_j{}^v = \begin{cases} b_j & \text{if } v = 1 \\ \neg b_j & \text{otherwise} \end{cases}$$

Example: Let $\mathcal{Q} = \{q_0, q_1, q_2, q_3, q_4\}$. The set of Booleans used to encode the states has 3 as cardinal and is $\mathcal{B} = \{b_1, b_2, b_3\}$. The encoded states are:

$$\Phi_{q_0}(\mathcal{B}) = \neg b_1 \wedge \neg b_2 \wedge \neg b_3 \qquad \Phi_{q_1}(\mathcal{B}) = b_1 \wedge \neg b_2 \wedge \neg b_3$$

$$\Phi_{q_2}(\mathcal{B}) = \neg b_1 \wedge b_2 \wedge \neg b_3 \qquad \Phi_{q_3}(\mathcal{B}) = b_1 \wedge b_2 \wedge \neg b_3$$

$$\Phi_{q_4}(\mathcal{B}) = \neg b_1 \wedge \neg b_2 \wedge b_3$$

□

Reset formula

The formula that expresses the effect of resetting clocks in R is:

$$\Phi_R(\mathcal{X}, \mathcal{X}') = \bigwedge_{x_i \in R} (x_i' = 0) \wedge \bigwedge_{x_i \notin R} (x_i' = x_i)$$

Discrete transition translation

A discrete transition $\delta = (q, \Pi, R, q')$ translates into:

$$\Psi_\delta(\mathcal{B}, \mathcal{X}, \mathcal{B}', \mathcal{X}') = \Phi_q(\mathcal{B}) \wedge \Phi_\Pi(\mathcal{X}) \wedge \Phi_R(\mathcal{X}, \mathcal{X}') \wedge \Phi_{q'}(\mathcal{B}')$$

where $\Phi_\Pi(\mathcal{X})$ is the formula of the $\mathcal{X}$-polyhedron Π.

Time passage formula

The time passage formula is the following:

$$\Phi_\tau(\mathcal{X}, \mathcal{X}') = (\exists t \geq 0) \wedge \bigwedge_{x_i \in \mathcal{X}} (x_i' - x_i = t)$$

Time transition translation

The formula expressing a time transition at state q is:

$$\Psi_q(\mathcal{B}, \mathcal{X}, \mathcal{B}', \mathcal{X}') = \Phi_q(\mathcal{B}) \wedge \Phi_\tau(\mathcal{X}, \mathcal{X}') \wedge \Phi_{S(q)}(\mathcal{X}') \wedge \Phi_q(\mathcal{B}')$$

where $\Phi_{S(q)}(\mathcal{X}')$ is the formula of the $\mathcal{X}'$-polyhedron $S(q)$.

Finite run translation

The formula associated with a valid transition is:

$$\Psi(\mathcal{B}, \mathcal{X}, \mathcal{B}', \mathcal{X}') = \bigvee_{q \in \mathcal{Q}} \Psi_q(\mathcal{B}, \mathcal{X}, \mathcal{B}', \mathcal{X}') \vee \bigvee_{\delta \in \Delta} \Psi_\delta(\mathcal{B}, \mathcal{X}, \mathcal{B}', \mathcal{X}')$$

Thus, the formula expressing a valid run of length k is:

$$\Psi_k = \Psi(\mathcal{B}^0, \mathcal{X}^0, \mathcal{B}^1, \mathcal{X}^1) \wedge \Psi(\mathcal{B}^1, \mathcal{X}^1, \mathcal{B}^2, \mathcal{X}^2) \wedge \ldots \wedge \Psi(\mathcal{B}^{k-1}, \mathcal{X}^{k-1}, \mathcal{B}^k, \mathcal{X}^k)$$

Notice that Ψ_k is also valid for runs of length less than k. This is due to idling.

Conformance with DL

The time passage formula expressed above is not DL conformant. Elimination of t in Φ_τ gives:

$$\Phi_\tau(\mathcal{X}, \mathcal{X}') = \bigwedge_i \bigwedge_{i \neq j} \big((x_i' - x_i = x_j' - x_j) \wedge (x_i' - x_i \geq 0)\big)$$

Φ_τ is still beyond the scope of DL. But this can be *easily* avoided by using *extended states*.

Definition 27 (Extended state) *States can be extended to include the absolute time since the beginning of the finite run. Therefore, a discrete transition becomes:*

$$(q, \mathbf{v}, T) \xrightarrow{\delta} (q', \mathbf{v}', T)$$

And a time transition is written:

$$(q, \mathbf{v}, T) \xrightarrow{\theta} (q, \mathbf{v} + \theta \cdot \mathbf{1}, T + \theta)$$

This extension can be seen as an addition of a clock that is never reset and which valuation is T.

For each clock $x \in \mathcal{X}$, the date variable $\xi = T - x$ represents the last time when x was reset, and the pair (ξ, T) can replace the clock x with no loss of expressivity.

When applying that isomorphic transformation on the clocks in $\mathcal{X}$, i.e. when substituting each clock x with $T - \xi$, changes occur in the formulae expressed so far:

- Time passage only affects T. It does not affect variables in $\Xi(T)$, where:

$$\Xi(T) = \{T - x \mid x \in \mathcal{X}\}$$

- A reset of a clock x is equivalent to setting its associated date variable ξ to T.

- Polyhedra associated with the guards and the staying conditions must be evaluated on elements of $\Xi(T)$.

The reset formula becomes:

$$\Phi_R\big(\Xi(T),\Xi'(T),T\big) = \bigwedge_i(\xi_i' = T) \wedge \bigwedge_i(\xi_i' = \xi_i)$$

where $\Xi(T) = \{\xi_1,\ldots,\xi_n\}$.

The time passage formula can be written:

$$\Phi_\tau\big(\Xi(T),T,\Xi'(T),T'\big) = (\exists t \geq 0) \wedge (T' - T = t) \wedge \bigwedge_i(\xi_i' = \xi_i)$$

It becomes after eliminating t:

$$\Phi_\tau\big(\Xi(T),T,\Xi'(T'),T'\big) = (T' - T \geq 0) \wedge \bigwedge_i(\xi_i' = \xi_i)$$

A discrete transition is transformed into:

$$\begin{array}{rcll} \Psi_\delta\big(\mathcal{B},\Xi(T),\mathcal{B}',\Xi'(T),T\big) & = & \Phi_q(\mathcal{B}) \wedge \Phi_\Pi\big(\Xi(T),T\big) \\ & & \wedge & \Phi_R\big(\Xi(T),\Xi'(T),T\big) \wedge \Phi_{q'}(\mathcal{B}') \end{array}$$

A time transition at state q becomes:

$$\begin{array}{rcll} \Psi_q\big(\mathcal{B},\Xi(T),T,\mathcal{B}',\Xi'(T'),T'\big) & = & \Phi_q(\mathcal{B}) \wedge \Phi_\tau\big(\Xi(T),T,\Xi'(T'),T'\big) \\ & & \wedge & \Phi_{S(q)}\big(\Xi'(T')\big) \wedge \Phi_q(\mathcal{B}') \end{array}$$

Thus, a valid transition satisfies the formula:

$$\begin{array}{rcll} \Psi\big(\mathcal{B},\Xi(T),T,\mathcal{B}',\Xi'(T'),T'\big) & = & \bigvee_{q\in\mathcal{Q}} \Psi_q\big(\mathcal{B},\Xi(T),\mathcal{B}',\Xi'(T),T\big) \\ & & \vee & \bigvee_{\delta\in\Delta} \Psi_\delta(\mathcal{B},\Xi(T),T,\mathcal{B}',\Xi'(T'),T'\big) \end{array}$$

Finally, the formula expressing a valid run of length k becomes:

$$\begin{array}{rcll} \Psi_k & = & \Psi\big(\mathcal{B}^0,\Xi^0(T^0),T^0,\mathcal{B}^1,\Xi^1(T^1),T^1\big) \\ & & \wedge & \Psi\big(\mathcal{B}^1,\Xi^1(T^1),T^1,\mathcal{B}^2,\Xi^2(T^2),T^2\big) \\ & & \wedge & \ldots \\ & & \wedge & \Psi\big(\mathcal{B}^{k-1},\Xi^{k-1}(T^{k-1}),T^{k-1},\mathcal{B}^k,\Xi^k(T^k),T^k\big) \end{array}$$

That new version of Ψ_k is fully in conformance with DL. Such a goal was achieved by adding k new variables T^0,..., and T^k, each of them represents the time at which a transition was taken.

5.1.3 Composition

In this sub-section, we describe the translation to DL of a product of interacting timed automata and focus on the composition operator based on communication by variables. In such a composition an automaton may observe the states of other automata, that is, it may use their values in its transition guards and staying conditions. Hence, in an updated definition of timed automata that

takes interaction into account, both the staying conditions and the guards are extended to be a conjunction of Boolean variables that encode the states of external automata with the clock constraints on $\mathcal{X}$.

Two composition variants are presented in the sequel. In the first, all the composed automata share the same global time-scale whereas in the second, each automaton has its own.

In the rest of that sub-section, consider n timed automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ such that for each $i \in 1 \leq i \leq n$, $\mathcal{A}_i = (\mathcal{Q}_i, \mathcal{X}_i, q_{0,i}, \Delta_i, \mathcal{S}_i)$, where $i \in \{1, \ldots, n\}$. Assume also the states of each automaton $\mathcal{A}_i$ are encoded using a set $\mathcal{B}_i$ of Boolean variables such that $\forall i \neq j,\ \mathcal{B}_i \cap \mathcal{B}_j = \varnothing$. Let $\mathcal{B} = \bigcup \mathcal{B}_i$ and $\mathcal{X} = \bigcup \mathcal{X}_i$.

Global time scale composition

The set of automata that should be observed while taking a transition $\delta_i \in \Delta_i$ with a guard Π_i is:

$$J_{\delta_i} = \{\mathcal{A}_j | \exists q \in \mathcal{Q}_j, q \text{ appears in } \Pi_i\}$$

The set of automata to be observed during time passage in a state $q_i \in \mathcal{Q}_i$ is:

$$J_{q_i} = \{\mathcal{A}_j | \exists q \in \mathcal{Q}_j, q \text{ appears in } S(q_i)\}$$

The set of automata that might be influenced by transitions in automaton $\mathcal{A}_i$ during their time passage is:

$$J_i = \{\mathcal{A}_j | \exists q_j \in \mathcal{Q}_j, \mathcal{A}_i \in J_{q_j}\}$$

A local discrete step of an automaton $\mathcal{A}_i$ is:

$$(q_i, \mathbf{v}_i, T) \xrightarrow{\delta_i} (q_i', \mathbf{v}_i', T)$$

such that the clocks of $\mathcal{A}_i$ and the states of automata in J_{δ_i} satisfy the guard Π_i of the transition $\delta_i \in \Delta_i$ at time T.

A time step of $\mathcal{A}_i$ is:

$$(q_i, \mathbf{v}_i, T) \xrightarrow{\theta} (q_i, \mathbf{v}_i + \theta \cdot \mathbf{1}, T + \theta)$$

such that the staying condition for state q_i denoted $S(q_i)$ is satisfied by the clocks of $\mathcal{A}_i$ and by states of automata in J_{q_i} during the time interval $[T, T+\theta)$.

A global discrete step is of the form:

$$\begin{gathered}\big((q_1, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots, q_n), (\mathbf{v}_1, \ldots, \mathbf{v}_{i-1}, \mathbf{v}_i, \mathbf{v}_{i+1}, \ldots, \mathbf{v}_n), T\big) \\ \xrightarrow{\delta_i} \\ \big((q_1, \ldots, q_{i-1}, q_i', q_{i+1}, \ldots, q_n), (\mathbf{v}_1, \ldots, \mathbf{v}_{i-1}, \mathbf{v}_i', \mathbf{v}_{i+1}, \ldots, \mathbf{v}_n), T\big)\end{gathered}$$

where $(q_i, \mathbf{v}_i, T) \xrightarrow{\delta_i} (q_i', \mathbf{v}_i', T)$ is a discrete step of automaton $\mathcal{A}_i$.

Likewise, a global time step is of the form:

$$\begin{gathered}\big((q_1, \ldots, q_n), (\mathbf{v}_1, \ldots, \mathbf{v}_n), T\big) \\ \xrightarrow{\theta} \\ \big((q_1, \ldots, q_n), (\mathbf{v}_1 + \theta \cdot \mathbf{1}, \ldots, \mathbf{v}_n + \theta \cdot \mathbf{1}), T + \theta\big)\end{gathered}$$

such that for every state q_i, the staying condition $S(q_i)$ is satisfied by every $\mathbf{v}_i + \theta \cdot \mathbf{1}$ and by every $q_j \in J_i$.

This definition of global steps requires that whenever an automaton makes a discrete step at time T, any other automata that is in a time step from time T' to T'', such that $T \in [T', T'')$, must split that step into two.

Building the DL formula for the composition of automata is very similar to building the DL formula for a single automaton. Date variables replace clock variables and all the automata share the same time-stamp T, which guarantees the synchronization between them. The formula of a global step is the conjunction of step formulae constructed for all automata.

Private time-scale composition

The behaviors of loosely-coupled automata that make independent transitions often can be encoded by formulae with less variables. The idea, inspired by [BJLY98], is based on expressing the steps of each automaton using a private time-scale and on the synchronization of those time-scales whenever required, i.e. only when automata interact. Each automaton $\mathcal{A}_i$ uses its own time-stamp variables T_i such that its corresponding date variables are $\xi_i = T_i - x_i$.

The reset formula for $\mathcal{A}_i$ is:

$$\Phi_{R_i}\big(\Xi(T_i), \Xi'(T_i), T_i\big) = \bigwedge_{j=1}^{n} (\xi'_j = T_j) \wedge \bigwedge_{j=1}^{n} (\xi'_j = \xi_j)$$

The time passage formula for $\mathcal{A}_i$ is written:

$$\Phi_{\tau_i}\big(\Xi(T_i), T_i, \Xi'(T'_i), T'_i\big) = (T'_i - T_i \geq 0) \wedge \bigwedge_{j=1}^{n} (\xi'_j = \xi_j)$$

The formula for discrete transition $\delta_i \in \Delta_i$ is:

$$\begin{array}{rcl} \Psi_{\delta_i}\big(\mathcal{B}, \Xi, \mathcal{T}, \mathcal{B}', \Xi', \mathcal{T}'\big) & = & \Phi_{q_i}(\mathcal{B}_i) \wedge \Phi_{\Pi_i}\big(\mathcal{B}, \Xi(T_i), T_i\big) \\ & & \wedge \ \Phi_{R_i}\big(\Xi(T_i), \Xi'(T_i), T_i\big) \wedge \Phi_{q'_i}(\mathcal{B}'_i) \\ & & \wedge \ (T_i = T'_i) \wedge \bigwedge_{j \in J_{\delta_i} \cup J_i} (T_i = T_j) \end{array}$$

where $\mathcal{T} = \{T_1, \ldots, T_n\}$ and $\mathcal{T}' = \{T'_1, \ldots, T'_n\}$.

A time transition at state q_i is:

$$\begin{array}{rcl} \Psi_{q_i}\big(\mathcal{B}, \Xi, \mathcal{T}, \mathcal{B}', \Xi', \mathcal{T}'\big) & = & \Phi_{q_i}(\mathcal{B}_i) \wedge \Phi_{\tau_i}\big(\Xi(T_i), T_i, \Xi'(T'_i), T'_i\big) \\ & & \wedge \ \Phi_{S(q_i)}\big(\mathcal{B}, \Xi'(T'_i), T'_i\big) \wedge \Phi_{q'_i}(\mathcal{B}'_i) \\ & & \wedge \ \bigwedge_{j \in J_{q_i}} (T_i = T_j) \end{array}$$

where $\Xi = \{\Xi(T_1), \ldots, \Xi(T_n)\}$, $\Xi' = \{\Xi'(T'_1), \ldots, \Xi'(T'_n)\}$, and $\Phi_{S(q_i)}\big(\mathcal{B}, \Xi'(T'_i), T'_i\big)$ is the result of the substitution of $T'_i - \xi'_i$ and all the state variables in $S(q_i)$.

Thus, a valid step of $\mathcal{A}_i$ is represented by the formula:

$$\begin{array}{rcl} \Psi_i\big(\mathcal{B}, \Xi, \mathcal{T}, \mathcal{B}', \Xi', \mathcal{T}'\big) & = & \bigvee_{q \in \mathcal{Q}_i} \Psi_q\big(\mathcal{B}, \Xi, \mathcal{T}, \mathcal{B}', \Xi', \mathcal{T}'\big) \\ & & \vee \ \bigvee_{\delta \in \Delta_i} \Psi_\delta\big(\mathcal{B}, \Xi, \mathcal{T}, \mathcal{B}', \Xi', \mathcal{T}'\big) \end{array}$$

The formula for a global step is:

$$\Psi\big(\mathcal{B},\Xi,\mathcal{T},\mathcal{B}',\Xi',\mathcal{T}'\big)=\bigwedge_{i=1}^{n}\Psi_i\big(\mathcal{B},\Xi,\mathcal{T},\mathcal{B}',\Xi',\mathcal{T}'\big)$$

Finally, the formula expressing a valid run of length k is:

$$\begin{array}{rcl}\Psi_k & = & \Psi\big(\mathcal{B}^0,\Xi^0(T^0),T^0,\mathcal{B}^1,\Xi^1(T^1),T^1\big)\\ & \wedge & \Psi\big(\mathcal{B}^1,\Xi^1(T^1),T^1,\mathcal{B}^2,\Xi^2(T^2),T^2\big)\\ & \wedge & \ldots\\ & \wedge & \Psi\big(\mathcal{B}^{k-1},\Xi^{k-1}(T^{k-1}),T^{k-1},\mathcal{B}^k,\Xi^k(T^k),T^k\big)\end{array}$$

5.2 Asynchronous Digital Circuits

This section is concerned with modeling circuits with bi-bounded delays using timed automata [MP95, BMT99, BJMY02] and expressing them in difference logic.

5.2.1 Used model

We present a general model that expresses the behavior of asynchronous digital circuits without any limitation or restriction. In this model, all variables can influence each other. Besides, all circuit structures, including the acyclic structure, can be modeled in a natural manner.

Assume a circuit made of wires transporting k different signals. It can contain at most k Boolean gates. Each gate can have up to k signals as input but it has always a single output signal. To simplify, we can consider that such a circuit has k gates, additional gates perform the identity transformation on their single input signal. Each signal is modeled by a Boolean variable and each gate is represented by a Boolean function that takes all the Boolean variables as input and outputs a single Boolean value.

In order to add timing considerations to that model, it is extended with a delay operator that expresses the non-immediate propagation of signals inside a gate. This operator satisfies the following properties:

- Positive lower-bound: A minimal amount of time has to elapse to propagate a change from the input to the output.
- Finite upper-bound: Every persistent change in the input propagates to the output in a bounded amount of time.
- Uncertainty: The exact delay duration is not predictable. It can be unknown or vary according to some hidden or complex phenomena. The delay can only be estimated to be within an interval.
- Inertia: The delay element filters small fluctuations in the input. Only changes that persist for a minimal duration propagates to the output.

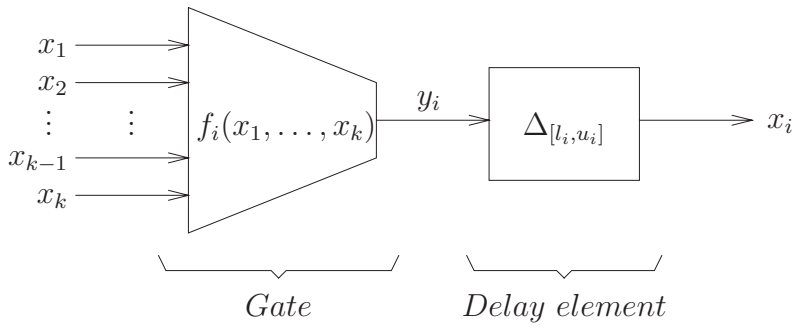

Figure 5.1: A timed gate model

Due to uncertainty, a delay element can transform an input signal to an infinity of possible output signals, all satisfying the four properties listed above. Hence, the corresponding delay operator $\Delta_{[l,u]}$ is set-valued. Figure (5.2) demonstrates that fact.

Definition 28 (Digital circuits model) *A k-variable digital circuit is:*

$$\mathcal{N} = (X, Y, F, D)$$

where:

- $X = \{x_1, \ldots, x_k\}$ *is a set of variables;*
- $Y = \{y_1, \ldots, y_k\}$ *is a set of auxiliary, or* hidden, *variables;*
- $F = \{f_1, \ldots, f_k\}$ *is a set of Boolean functions, such that for each i:*

$$f_i : \mathbb{B}^k \to \mathbb{B}$$

- $D = \big\{(l_1, u_1), \ldots, (l_k, u_k)\big\}$ *is a set of pairs of integers, such that:*

$$\forall i \in \{1, \ldots, k\},\ 0 < l_i \leq u_i$$

The semantics of a circuit is the set of all solutions of the following system of equations:

$$\begin{cases} y_1 = f_1(x_1, \ldots, x_k) \\ \vdots \\ y_k = f_1(x_1, \ldots, x_k) \\ x_1 \in \Delta_{[l_1,u_1]}(y_1) \\ \vdots \\ x_k \in \Delta_{[l_k,u_k]}(y_k) \end{cases} \tag{5.1}$$

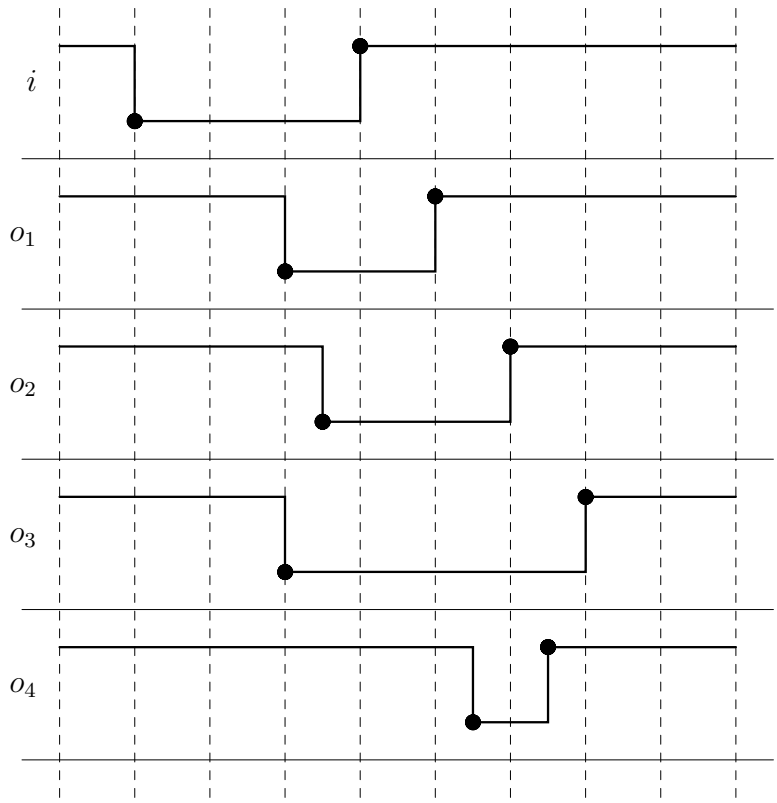

Figure 5.2: An input signal i and some corresponding delayed output signals $\{o_1, \ldots, o_4\} \in \Delta_{[2,4]}$

5.2.2 Conversion to timed automata

Every equation in (5.1) can be translated into a timed automaton whose behaviors correspond to the solutions of the considered equation. Composing all these automata produces all the possible behaviors of the digital circuit with respect to all possible choices of delays.

A Boolean gate, whose equation is $y_i = f_i(x_1, \ldots, x_k)$, is transformed into a simple one-state automaton that generates all the tuples satisfying the equation.

A delay element, whose equation is $x_i \in \Delta_{[l_i,u_i]}(y_i)$, is modeled by a four-state timed automaton with a single clock C as depicted in Figure (5.3). The rest of this paragraph explains how the automaton acts for example when the input signal goes from 0 to 1 whereas x_i was formerly set to 0: State $(0,0)$ is a *stable* state where both y_i and x_i are 0. When the input signal y_i changes to 1, an *excite* transition to state $(1,0)$ is made and the clock C is reset to 0. A transition from $(1,0)$ back to $(0,0)$ corresponds to a *regret* transition, i.e. the cancellation of the propagation of the input inside the delay element. During the stay of the automaton in state $(1,0)$, the value of C grows uniformly. If the latter crosses the lower bound l_i, the output x_i can change to 1 and the automaton can go to state $(1,1)$. But, the automaton may stay in $(1,0)$ as long as the value of C has not reached the upper bound u_i.

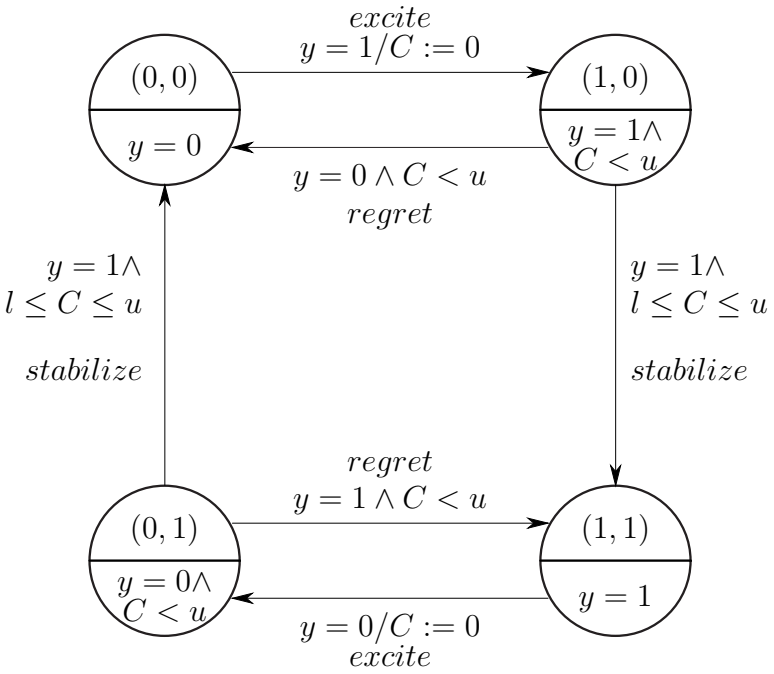

Figure 5.3: A timed automaton model of the delay element

Concretely, all the timed automata are composed using a global time-scale composition and a difference logic formula expressing the overall circuit behavior for a given run length is generated as described in Subsection 5.1.3.

5.3 Non-Preemptive Job-Shops

Job-shop problems are common to many kinds of optimization problems coming from various fields such as airplane traffic planning and manufacturing optimization. All those problems aim to find an efficient scheduling of a given number of tasks with respect to constraints on time and resources. In this section, we introduce an example of non-preemptive job-shop scheduling. Then, we give a formal description of this class of problems. Finally, we explain how to write a formula in difference logic that characterizes the feasible solutions.

5.3.1 Introductory example

A factory produces three types of products P_1, P_2, and P_3 using 3 machines M_1, M_2, and M_3. Manufacturing each product requires the use of two machines as described in Figure 5.4.

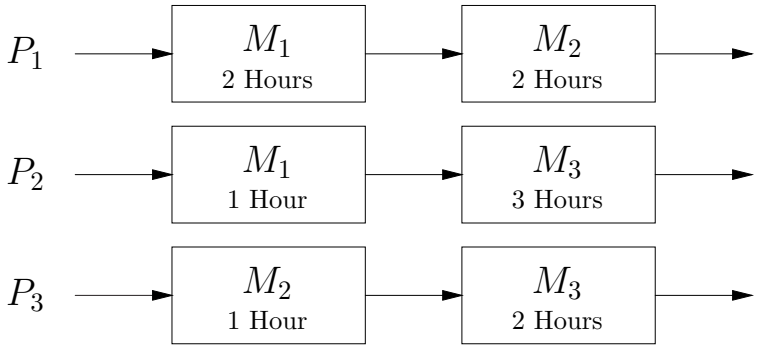

Figure 5.4: A sample manufacturing process

For obvious rentability reasons, it is necessary to find a scheduling of the manufacturing process with the minimal cycle duration and the optimal use of the machines.

In order to find such a scheduling, we introduce variables that represent the start time of each task in a given cycle (See Figure 5.5). A task consists in the manufacturing of a product on a machine. All cycles start at time 0.

The order of the operations gives a first set of timing constraints. In fact, to manufacture a product P_1, M_2 is not used until the end of the use of M_1. These constraints produce the following inequalities:

$$\begin{cases} x_1 + 2 \leq x_2 \\ y_1 + 1 \leq y_2 \\ z_1 + 1 \leq z_2 \end{cases} \tag{5.2}$$

Besides, it is impossible to use the same machine at the same time for two different products. For instance, this *mutual exclusion* condition forbids the use of M_1, while it is dedicated to P_2, to produce P_1.

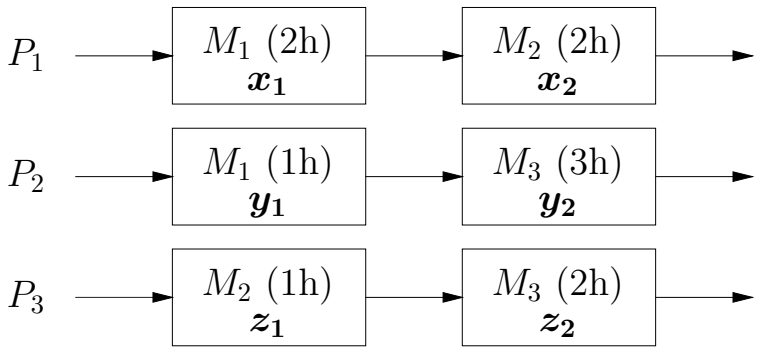

Figure 5.5: Variables associated with the start time of tasks

$$\begin{cases}]\,x_1, x_1+2\,[\;\cap\;]\,y_1, y_1+1\,[= \varnothing \text{ (Mutual exclusion for } M_1) \\ ]\,x_2, x_2+2\,[\;\cap\;]\,z_1, z_1+1\,[= \varnothing \text{ (Mutual exclusion for } M_2) \\ ]\,y_2, y_2+3\,[\;\cap\;]\,z_2, z_2+2\,[= \varnothing \text{ (Mutual exclusion for } M_3) \end{cases} \tag{5.3}$$

Solving (5.2) and (5.3) gives all the possible schedulings. In order to find the optimum one, i.e. the scheduling with the least cycle duration, we have just to introduce a variable w that represents the end of the cycle and that satisfies :

$$\begin{cases} x_2 + 2 \leq w \\ y_2 + 3 \leq w \\ z_2 + 2 \leq w \end{cases} \tag{5.4}$$

Minimizing w subject to all the above mentioned constraints gives the optimal solution to the job-shop problem.

5.3.2 Expression of job-shop problems

We focus on classic job-shop problems [JM99], i.e. problems with a finite number of tasks and machines where preemption is not allowed and where the goal is to find the minimum overall completion time.

Such job-shop problems can be expressed using timing constraints. In fact, they have to satisfy *absolute time*, *precedence*, and *mutual exclusion* constraints.

Absolute time

A task has constraints on its start and/or end times. Assume a task T starting at time x and lasting d units of time:

$$T \text{ starts at least at time } t \iff x \geq t$$

$$T \text{ ends at most at time } t \iff x \geq t - d$$

Precedence

The precedence expresses the order in which two tasks are executed. Assume two tasks T_1 and T_2 such that T_1 starts at x_1 and lasts d_1 time units and T_2 starts at x_2.

$$T_1 \text{ precedes } T_2 \iff x_1 + d_1 \leq x_2$$

Mutual Exclusion

The mutual exclusion expresses the impossibility of sharing the same resource by two different tasks at the same time.

Consider two tasks T_1 and T_2 requiring the same resource R. T_1 needs R for at most d_1 time units and T_2 needs R for at most d_2 time units. x_1 (respectively x_2) represents the start time of the use of R by T_1 (respectively T_2).

The mutual exclusion constraint is the following:

$$]x_1, x_1 + d_1[\quad \cap \quad]x_2, x_2 + d_2[\quad = \varnothing$$

which is equivalent to:

$$\begin{cases} x_1 + d_1 \leq x_2 \\ \vee \\ x_2 + d_2 \leq x_1 \end{cases} \iff \begin{cases} x_2 - x_1 \geq d_1 \\ \vee \\ x_1 - x_2 \geq d_2 \end{cases}$$

5.3.3 Translation into DL

Since the formulae for job-shop problems are conjunctions of absolute time, precedence, and mutual exclusion constraints, they are, by construction, in DL. In fact, they are Boolean combinations of difference constraints, with no Boolean variables.

Since solving a DL formula gives a yes/no answer about its satisfiability, the optimization problem is replaced by the decision problem: *"Is there a scheduling with a duration less or equal to w_d?"* This modification requires only adding $w \leq w_d$ as a conjunction to the formula expressing the job-shop.

Notice that this question is not restrictive as it allows to find the optimal solution using dichotomy. The initial search range of w_d can be set to $[w_{min}, w_{max}]$ where w_{min} is the maximum of all the *ideal tasks durations* and w_{max} is their sum. An ideal task duration is the duration of a task if mutual exclusion was not to be taken into account (no resource race).

Chapter 6

Experimental Results

This chapter discusses the efficiency of the techniques that can be used in a mixed SAT solver. It contains an empirical statistical study that compares the relative contribution of the key techniques to the overall performance of the mixed solver. This study is carried out on a set of sample problems belonging to three distinct classes.

6.1 Study Methodology

First, we define several significant configurations of the MX-Solver. A configuration consists of a set of reduction rules that the mixed solver uses. Since it is impractical to study all the possible combinations of the solver features (there are two versions of the solver having 7 and 6 reduction rules respectively), we need to restrict the study to a small number of configurations, called C1, C2, ..., and C7.

The following table describes the seven selected typical configurations of the solver that will be studied in the sequel. Comparison of their performance can give information about the efficiency of the available techniques. All the configurations use the MOM's heuristic.

Notice that the restricted Davis-Putnam rule is not available in configurations that use the MX3-Solver for the reasons already discussed in Section 4.6.

	C1	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
MX-Solver	√		√	√			
MX3-Solver		√			√	√	√
Unit resolution	√	√	√	√	√	√	√
Pure literals resolution	√	√	√	√	√	√	
Equivalence detection	√	√					
DBM updating	√	√	√	√	√	√	√
DBM Analysis	√	√	√	√	√	√	
Restricted Boolean DP	√	N/A	√		N/A	N/A	N/A
Restricted numerical DP	√	√		√	√		

For instance, comparing C1 and C2 can give an idea about the difference between the two versions of the solver when both are run using all the available techniques. Likewise, comparing C2 and C6 provides information about the efficiency of the restricted Davis-Putnam rule.

For each class of problem, an overview explains the studied problem(s) used to generate the mixed SAT instances. Next, the latter are described in detail. Finally, both the execution time and the maximum memory usage for solving each instance on each configuration are reported. All the experimentations were carried out on a standard PC powered by a 600 MHz Pentium III with 384 MB of RAM.

6.2 Job-Shop problems

6.2.1 Overview

This section is concerned with studying the satisfiability of the decision problem *"Is there a feasible scheduling for a given job-shop whose duration is less or equal to w_d?"*, where $w_d \in \mathbb{N}$.

For illustration purposes, we consider the job-shop *ft06* [FT63, JM99] containing 6 machines and 6 jobs and whose optimal make-span is 55. The following table shows the results obtained when solving the decision problem applied to the latter with $C7$ while varying w_d. The solving times in seconds are also listed.

w_d	**Satisfiable?**	**Solving time of C7**
0	No	0.14
50	No	0.14
51	No	0.14
52	No	1.79
53	No	7.67
54	No	21.47
55	Yes	1.31
56	Yes	0.20
57	Yes	0.92
58	Yes	1.88
59	Yes	0.57
60	Yes	0.21
100	Yes	0.25

Figure (6.1) shows graphically the relation between the solving time and w_d. The solving time grows exponentially when w_d is relatively *close* to the optimal schedule. In fact, when w_d is much smaller than the optimum, the solver quickly detects a contradiction. Likewise, when w_d is much larger than the optimal schedule, a satisfying assignment is easily found.

6.2.2 Instances

In the sequel, we consider job-shop problems bigger than *ft06*, namely *abz5* [ABZ88, JM99] and *la25* [Law84, JM99], whose properties are summarized in the following table:

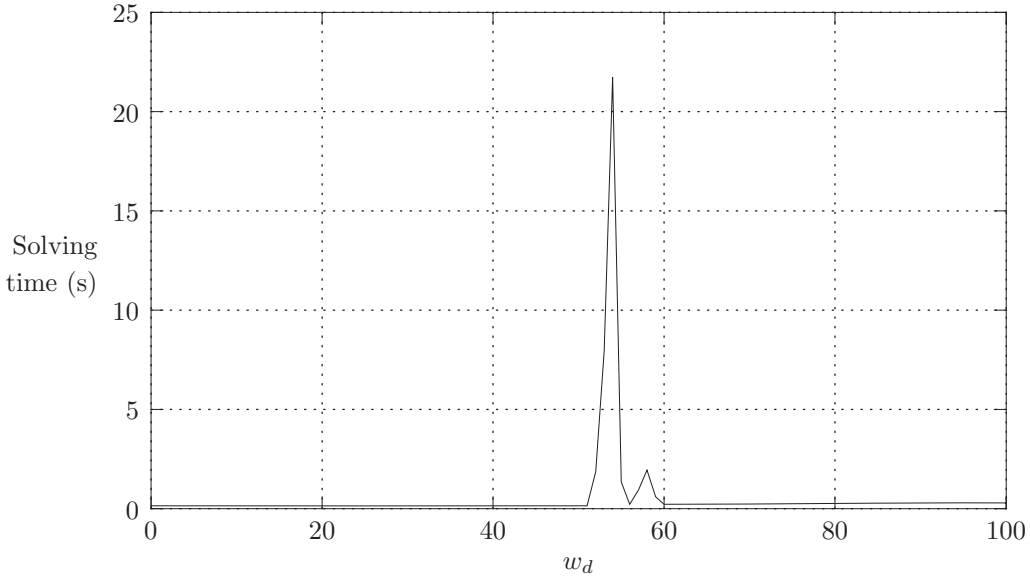

Figure 6.1: The relation between w_d and solving time for ft06 using $C7$

Problem	**# Machines**	**# Jobs**	**Optimal Make-span**
abz5	10	10	1234
la25	10	15	977

For both job-shop problems, we do some preliminary satisfiability checks using $C7$ while varying w_d. The solving times in seconds are listed in the following tables and are graphically represented in Figure (6.2) and Figure (6.3).

abz5		
w_d	**Satisfiable?**	**Solving time of C7**
0	No	4.65
950	No	4.70
975	No	4.96
980	No	261.72
990	No	1709.38
1000	No	> 1800[1]
1234	Yes	> 1800
1300	Yes	> 1800
1390	Yes	> 1800
1395	Yes	17.20
1400	Yes	17.23
1600	Yes	10.42
1800	Yes	10.86
3000	Yes	11.75

la25		
w_d	**Satisfiable?**	**Solving time of C7**
0	No	28.10
800	No	29.32
815	No	29.29
816	No	> 1800
977	Yes	> 1800
1794	Yes	> 1800
1795	Yes	73.64
1800	Yes	156.37
1900	Yes	295.35
2000	Yes	88.36
5000	Yes	84.97
10000	Yes	86.69

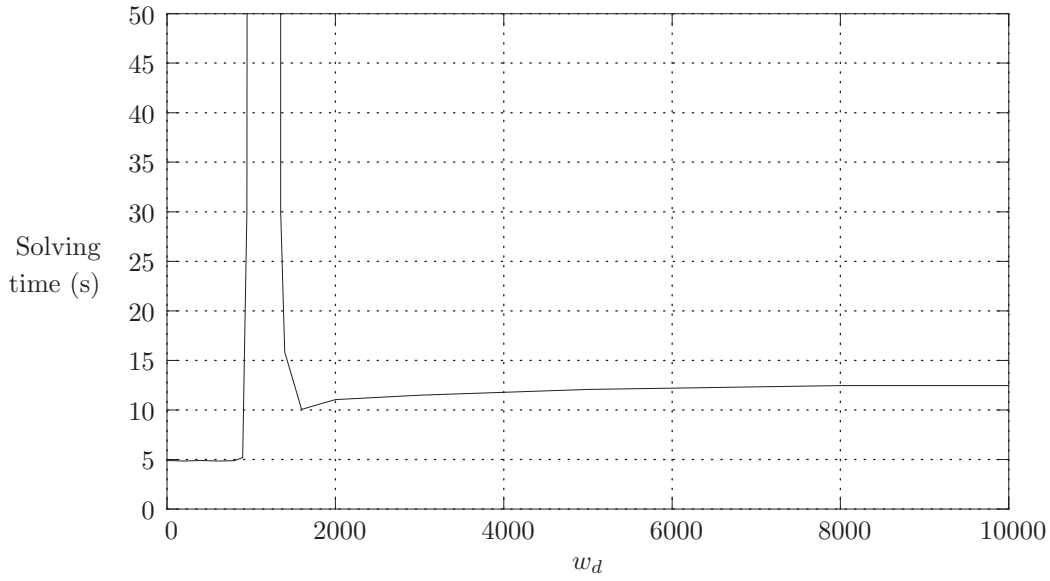

Figure 6.2: The relation between w_d and solving time for abz5 using $C7$

For each problem, we will focus on solving the decision problem applied to four values of w_d: Two of them are far from the optimum, and two others are relatively close, but without requiring a long solving time. The following table lists the studied instances and gives the number of variables and clauses found in the translation of each instance to MX-CNF.

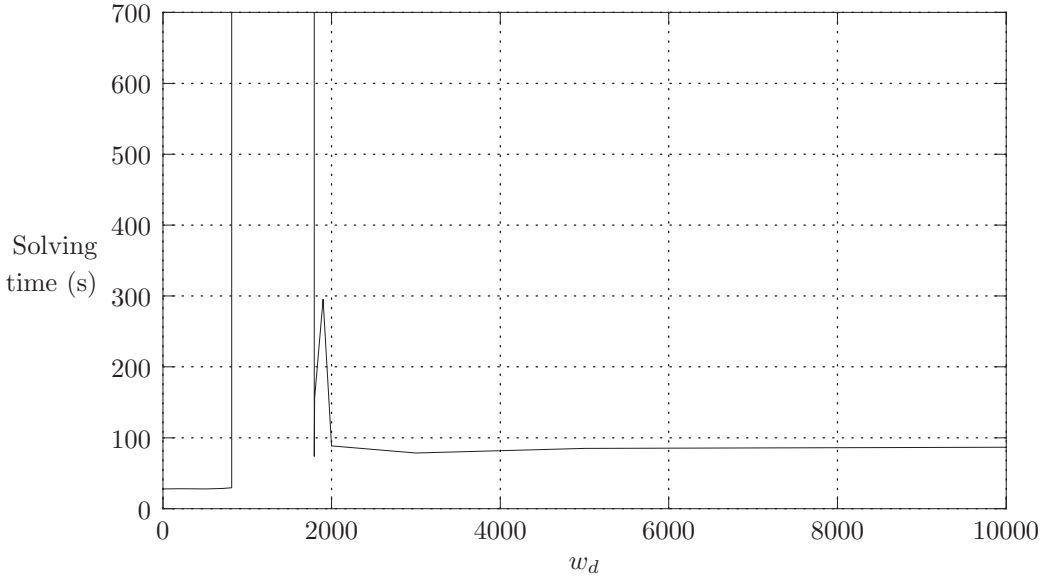

Figure 6.3: The relation between w_d and solving time for la25 using $C7$

Instance	w_d	**Bool. Var.**	**Num. Var.**	**Clauses**	**Satisfiable?**
abz5-200	200	1452	102	4353	No
abz5-980	980	1452	102	4353	No
abz5-1400	1400	1452	102	4353	Yes
abz5-5000	5000	1452	102	4353	Yes
la25-200	200	3302	152	9904	No
la25-800	800	3302	152	9904	No
la25-1800	1800	3302	152	9904	Yes
la25-5000	5000	3302	152	9904	Yes

6.2.3 Results

The following table contains the solving times in seconds of each instance on each of the seven configurations:

Instance	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
abz5-200	6.57	**4.63**	6.58	6.59	4.64	4.65	4.65
abz5-980	181.44	**172.99**	271.72	181.10	173.18	264.35	262.1
abz5-1400	18.79	16.30	19.88	18.70	**16.08**	17.45	17.22
abz5-5000	13.73	**11.23**	15.47	13.79	11.27	12.92	12.50
la25-200	41.33	30.95	41.26	37.93	**28.58**	28.92	28.56
la25-800	39.13	**30.09**	38.82	38.88	30.11	30.15	30.00
la25-1800	131.31	118.75	180.42	130.50	**118.70**	167.32	158.58
la25-5000	79.21	67.32	88.89	79.21	**67.07**	78.38	75.16
Average	63.93	56.53	82.88	63.33	**56.20**	75.51	73.59

The maximum amount of used memory in kilobytes per SAT instance and per configuration is listed below:

Instance	C1	C2	C3	C4	C5	C6	C7
abz5-200	1564	**1096**	1564	1564	**1096**	**1096**	**1096**
abz5-980	8068	6196	2696	8068	6196	**2168**	**2168**
abz5-1400	24048	19088	21548	24048	19088	**17176**	**17176**
abz5-5000	115180	92192	113364	115180	92188	**90664**	**90664**
la25-200	2056	**1704**	2056	2506	**1704**	**1704**	**1704**
la25-800	2056	**1704**	2056	2506	**1704**	**1704**	**1704**
la25-1800	333536	266140	314816	333536	266140	**250172**	**250172**
la25-5000	388128	309736	380484	388128	309736	**303660**	**303660**
Average	109329	87232	104823	109329	87231	**83543**	**83543**

As one can see, $C2$ and $C5$ behave similarly and are the fastest configurations for instances derived from the job-shop decision problems. They consume a slightly larger amount of memory than $C6$ and $C7$, and can be considered as the best compromise to solve this kind of instances.

On the other hand, $C3$, $C6$, and $C7$ are the slowest configurations. They do not use the restricted numerical Davis-Putnam rule.

6.3 Timed-Automata problems

6.3.1 Overview

This section is concerned with studying the efficiency of the solver configurations on mixed SAT instances generated from a bounded model checking problem involving a simple timed-automata with 4 states and 1 clock as depicted in Figure (6.4) and verifying if all the states of the automata are reachable.

6.3.2 Instances

The instances correspond to runs of different lengths of the automata. The following table lists the characteristics of the considered instances:

Instance	Run Length	Bool. Var.	Num. Var.	Clauses	Satisfiable?
ta-5	5	247	19	703	Yes
ta-10	10	492	34	1408	Yes
ta-50	50	2452	154	7048	Yes
ta-100	100	4902	304	14098	Yes

6.3.3 Results

As with the job-shop problems, the following table contains the solving times in seconds for the generated mixed SAT instances.

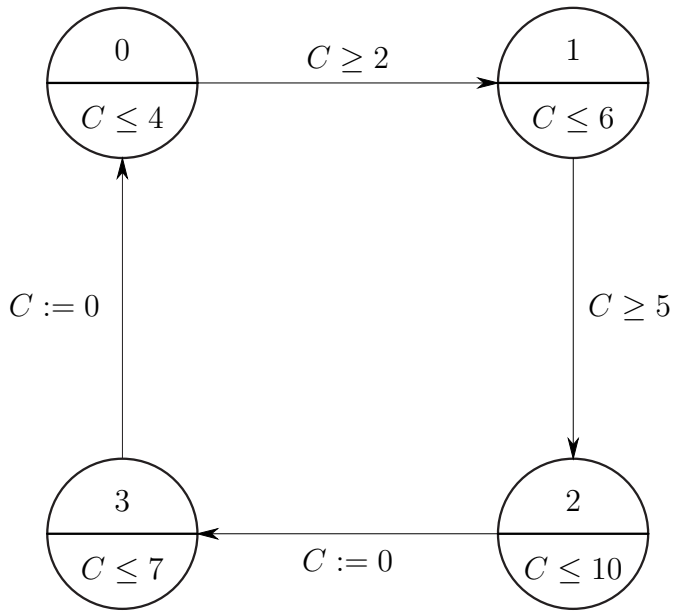

Figure 6.4: A sample timed automaton

Instance	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
ta-5	0.65	0.11	0.13	0.09	0.07	**0.06**	**0.06**
ta-10	2.01	0.49	0.68	0.43	0.29	0.29	**0.28**
ta-50	176.08	17.23	23.45	16.30	12.23	**11.68**	16.39
ta-100	1319.42	76.85	100.09	70.78	56.35	**54.85**	213.38
Average	374.54	23.67	31.08	21.90	17.23	**16.72**	57.52

The maximum memory requirements in kilobytes for solving the generated mixed SAT instances are listed below:

Instance	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
ta-5	1136	864	904	1012	780	**704**	**704**
ta-10	3976	**2432**	3132	3968	3044	2436	2436
ta-50	84204	**40700**	62192	79844	65960	48120	48120
ta-100	333024	**153892**	243224	296684	230708	158080	158080
Average	105585	**49472**	77363	95377	75123	52335	52335

The tables show that $C6$ is the best configuration for solving this type of instance. It is the fastest and uses less memory than others in almost all the cases.

Configurations using the MX-Solver have poor performance in comparison with those based on MX3-Solver. They are slow and require more memory than others, in particular, $C1$ that wastes a huge amount of time trying to detect Boolean equivalences.

The gap between the execution time of $C7$ and $C6$ seems to grow with the length of the solved instance. Hence, $C7$ takes more than 213 seconds to solve

ta-100, i.e. four times the time used by $C6$ to do the job. This shows the impact of the pure literal rule application and the DBM analysis.

6.4 Asynchronous circuits problems

6.4.1 Overview

In this section, we consider a simple 2-bit adder circuit, depicted in Figure (6.5). Each gate is labeled with an interval $[l, u]$ that represents the delay operator associated with the timed gate model whose lower bound is l and upper bound is u, i.e. each gate needs from l to u units of time to stabilize its output after its inputs were excited.

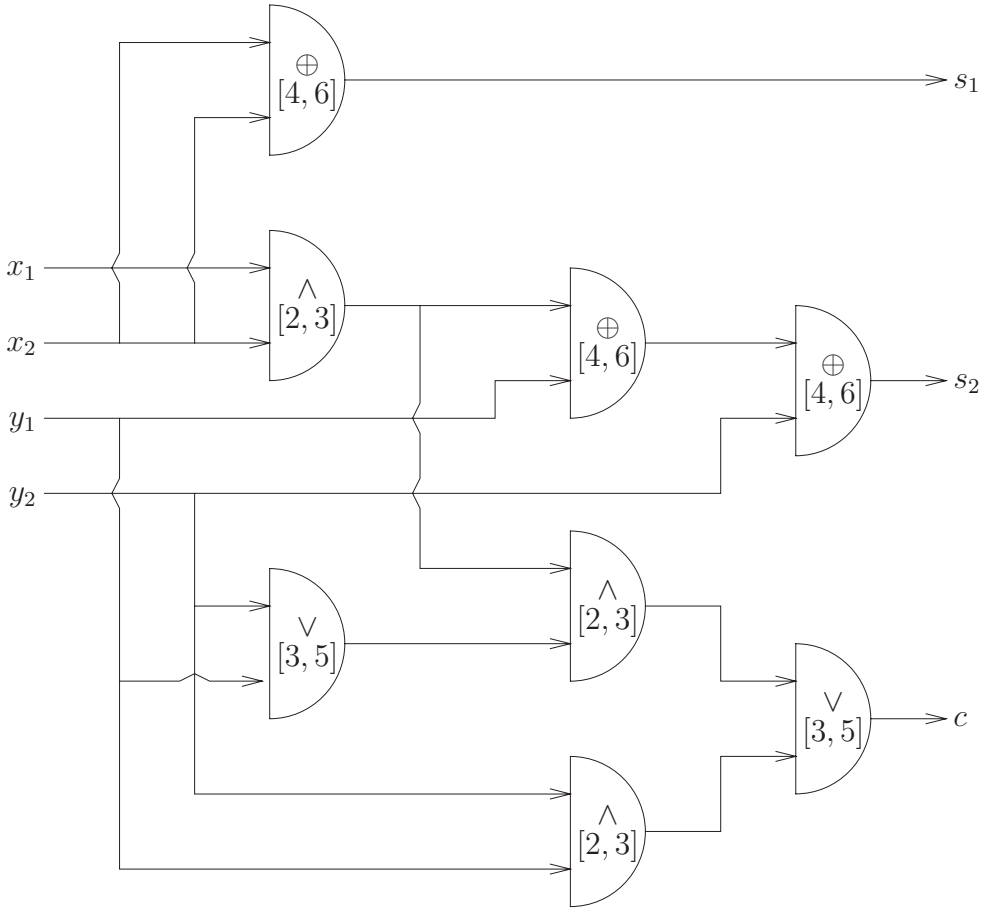

Figure 6.5: Asynchronous 2-bit adder

The studied decision problem is : *"Can all the outputs of the circuit stabilize in less than d units of time?"*, where d is a positive integer. That problem is satisfiable only when $d \geq 10$, which is the maximum of the sums of the lower bounds of all gates in all possible paths from the inputs to the outputs.

6.4.2 Instances

The generated mixed SAT instances correspond to runs of different length of automaton representing the circuit for two values of d. The characteristics of each instance are listed blow:

Instance	Run Length	d	Bool. Var.	Num. Var.	Clauses	Satisfiable?
2ba-2-5	2	5	498	13	1468	No
2ba-2-15	2	15	498	13	1468	No
2ba-3-5	3	5	733	17	2176	No
2ba-3-15	3	15	733	17	2176	No
2ba-4-5	4	5	968	21	2884	No
2ba-4-15	4	15	968	21	2884	Yes

Notice that for runs of length less than 4, the outputs cannot stabilize in less than 15 units of time.

6.4.3 Results

As before, the following table gives the execution times in seconds of the different configurations when applied to each instance:

Instance	C1	C2	C3	C4	C5	C6	C7
2ba-2-5	40.01	**9.16**	17.43	18.61	13.02	13.02	12.94
2ba-2-15	40.51	**9.34**	17.52	18.45	12.58	12.94	12.91
2ba-3-5	574.73	162.92	155.38	155.09	104.32	**104.30**	144.02
2ba-3-15	706.42	192.31	233.83	227.57	153.76	**153.71**	154.07
2ba-4-5	870.10	788.30	584.15	620.91	425.32	424.17	**424.08**
2ba-4-15	4.01	2.83	2.65	2.34	1.63	**1.57**	1.64
Average	372.63	194.14	168.49	173.82	118.43	**118.28**	124.94

The next table contains the maximum needed memory in kilobytes to solve each instance:

Instance	C1	C2	C3	C4	C5	C6	C7
2ba-2-5	37010	1460	1460	1028	**780**	**780**	**780**
2ba-2-15	37436	1460	1460	1028	**780**	**780**	**780**
2ba-3-5	23280	5808	6880	2628	**2008**	**2008**	**2008**
2ba-3-15	24532	23324	6960	3548	**2684**	**2684**	**2684**
2ba-4-5	149904	15448	14136	5156	**3880**	**3880**	**3880**
2ba-4-15	8656	7020	8492	8336	**6376**	**6376**	**6376**
Average	46803	9086	6564	3620	**2751**	**2751**	**2751**

Again, $C5$ and $C6$ are ranked the first as they have the least resolution time and the smallest memory requirements. A similar level of performance is achieved by $C7$.

Configuration $C1$ is dramatically slow and needs a large amount of memory. Equivalence detection seems to contribute to its low speed, and the same applies

to $C2$. Used in conjunction with Boolean Davis-Putnam, the reduction rules result in solving contexts where it is *too complicated* to find good branching variables, hence the abnormal memory consumption, directly related to the maximal search depth.

6.5 Conclusions

The above experiments suggest the following remarks:

- Configurations that use the MX3-Solver are faster and use less memory than those based on the MX-Solver.
- Configurations using the Boolean Davis-Putnam rule have poor performance and interfere badly with the branching heuristics. This results in a deep exploration of the search tree due to the accumulated bad choices of branching variables.
- Equivalence detection does not enhance the solver in the best cases. Some experiments even show that it becomes a real bottleneck as the solver keeps trying to detect equivalent literals on numerous binary clauses, which requires a bad processing time/discovered assignments ratio.
- The numerical Davis-Putnam rule seems to boost the solution process especially when the number of numerical variables is high. Indeed, its immediate effect is reducing the size of the solving context DBM, and hence speeding-up its normalization, an operation whose complexity is $O(n^3)$.
- The DBM analysis, whose efficiency is not as evident as that of the numerical Davis-Putnam rule, can contribute in some cases to the reduction of the DBM size. For instance, comparing $C6$ and $C7$ applied to timed automata instances shows its contribution to the overall performance of the solver.
- The efficiency of the pure literal rule is also marginal, but in some cases it can make the difference. Anyway, in the worst situation, if it does not enhance the solver, it does not affect its speed.

From these remarks, we can draw some conclusions about how to configure the solver to have it give the best performance:

- Using small fixed-size clauses saves memory and significantly reduces the solution time. Profiling the two versions of the solver showed that the MX-Solver spends more than 20% of the time managing the dynamical data-structures.
- Reduction rules that operate on numerical clauses and decrease the number of variables referenced in the DBM have a good impact on the solver. Developing new rules that achieve this goal can substantially enhance the processing times.

- Boolean reduction rules should be simple and should not require a complex detection phase before they can be applied. Thus, the simple pure literal rule is more efficient that the complex Boolean Davis-Putnam rule.

Chapter 7

Conclusions

7.1 Contribution

The main contribution of this thesis is the development of a mixed SAT solver for difference logic, in order to carry out bounded model checking for timed systems. The solver has been developed after a careful examination of the structure of existing Boolean SAT solvers and the techniques they use. The key features of our solver are:

1. Definition of a conjunctive normal form (CNF) for mixed clauses in which every clause has at most one numerical difference constraint.

2. Adaptation of the DPLL procedure for solving mixed SAT problems.

3. Using difference bound matrices (DBMs) to check the consistency of sets of unit clauses containing only numerical literals.

4. Development of new simplification rules specific to difference constraints.

5. Adaptation of conflict analysis techniques to mixed clauses.

A large part of the developed techniques have been implemented to obtain a functioning SAT solver for difference logic. Preliminary experiments were conducted on several benchmarks coming from several application domains, for which translators to difference logic were implemented. The results of these benchmark problems provide a preliminary assessment of the relative merits of the different methods used in the solver.

7.2 Future Research Directions

Needless to say, there is a lot of room for improvement in the design and implementation of the solver. We mention below some of the most promising directions divided into two parts: the first requires more conceptual work while the second is concerned with algorithms that were already developed in the thesis but have not yet been implemented.

7.2.1 New Algorithms and Methods

So far the solver was designed to be general-purpose, that is, to treat DL formulae regardless of their origin. It may turn out that knowing the origin of certain variables in the formulae (for example, state variables, synchronization variables) may help in the solution strategy. Development of such techniques involves theoretical and experimental analysis as well as inspiration from other approaches such as unbounded model-checking, constraint propagation for scheduling programs, etc.

The branching heuristics mentioned in this thesis were (mostly) simple adaptations of the heuristics used for Boolean SAT. More research is needed for developing heuristics specialized for difference logic and for specific classes of problems.

Another direction for improvements involves the trade-off between computation time and memory usage in the representation of the solving context. It is possible to omit certain information from the context and derive it when needed. For example, DBMs can be replaced by reduced constraint systems as reported in [LLPY97, LLPY02].

Although in this work we restricted ourselves to simple safety properties, the mixed SAT solver can be integrated into a more general Bounded Model Checking environment for richer properties expressed in some real-time timed logic.

7.2.2 The MX Toolbox

In addition to general code improvements, the following directions seem important:

1. On the front-end, more complete and efficient translators from timed automata and other formalisms can be developed, in particular a compositional translation with distinct time-scales as described in Chapter 5.

2. Next, the conversion to MX-CNF, currently based on Tseitin's translation can be improved by implementing Wilson's algorithm described in Chapter 2.

3. Some more of branching heuristics described in Chapter 3 can be integrated, in addition to the currently used MOM's heuristic.

4. Finally, the MX-Solver can be extended to include conflict-directed backtracking along the lines of Section 4.5. This technique improves significantly the performance of Boolean SAT solvers and we hope it can do the same for mixed SAT solvers.

Appendix A

MX Toolbox

The MX Toolbox is the concrete realization of this thesis. It consists of a set of software tools based on the algorithms presented in this document. In the sequel, we describe the MX Toolbox from a technical point of view, and provide concise instructions on how to use it.

A.1 Overview

The MX Toolbox contains a set of tools classified into three categories:

- The *To DL translators* are tools that transform several classes of problems to difference logic.
- The *DL to MX-CNF translator* is responsible for converting DL formulae into a set of clauses conforming to the mixed conjunctive normal form (MX-CNF).
- The *mixed solver* is the cornerstone of the toolbox. It reads an MX-CNF set of clauses and checks for its satisfiability.

The dependency between the different tools is depicted in Figure (A.1).

One of the major concerns about the design of the MX Toolbox was its expandability. Its architecture is flexible enough to allow using other translators. This adaptability was achieved by the use of common file formats in order to ease data exchange between tools.

A.2 To DL Translators

A To DL translator takes a file describing a problem from a given class and expressed in its proper syntax and transforms it into a DL formula. The MX Toolbox has three To DL translators.

A.2.1 Timed automata to DL

`tg2dl` generates from a file in KRONOS syntax describing a timed automaton a difference logic formula characterizing a finite run of that automaton. It is based on the translation described in Section 5.1. The program is invoked using:

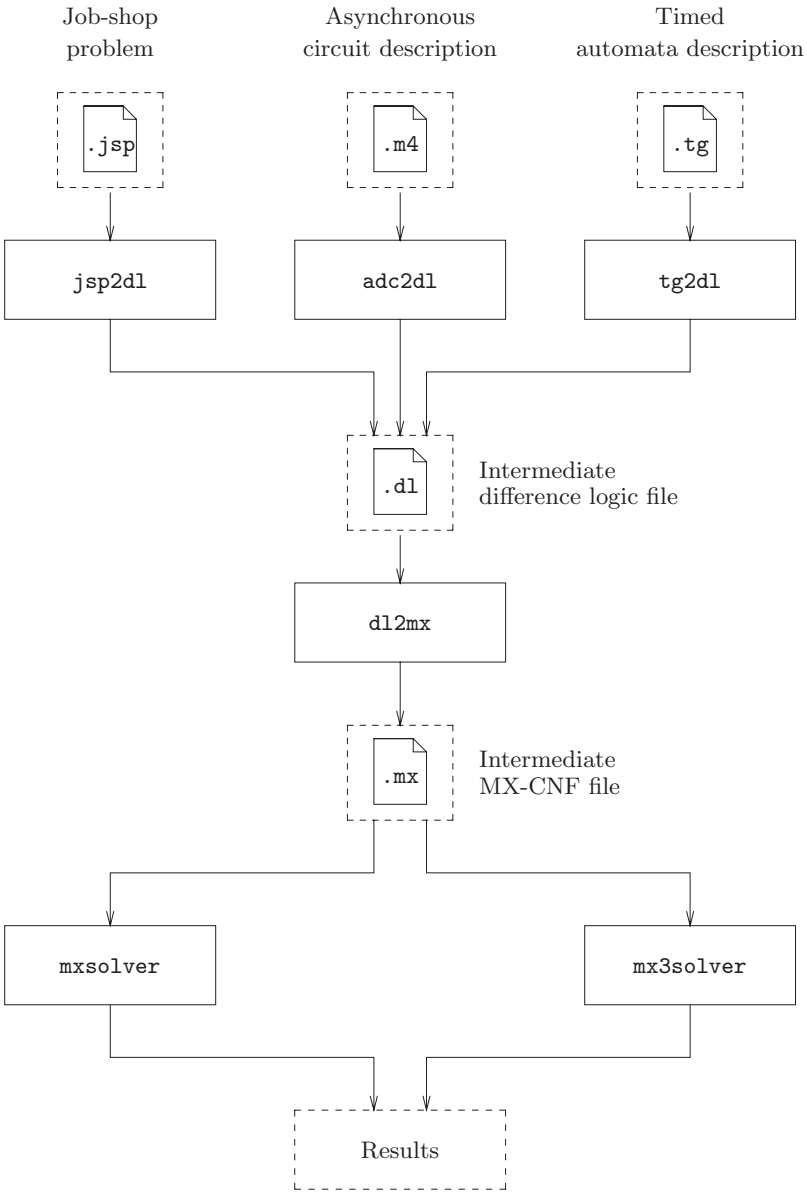

Figure A.1: MX Toolbox architecture

`tg2dl` *tg_file run_length dl_file*

Detailed information about the `.tg` file format can be found in [Oli94].

A.2.2 Asynchronous digital circuits to DL

`adc2dl` is based on an `m4` script that processes a `.adc` file containing the description of an asynchronous digital circuit. A `.acd` file consists of one or more definitions of gates with bi-bounded delays of the form:

`(` *signal* `, (` *gate_function* `),` *delay_lower_bound* `,` *delay_upper_bound* `)`

where *signal* is the name of the output signal of the gate. *gate_function* is a Boolean function on the circuit signals. The delay operator is defined by the parameters *delay_lower_bound* and *delay_upper_bound*. Successive definitions are separated by commas.

For instance, the definition `1ba.adc` of a simple 1-bit adder is:

```
(s,((x),xor,(y)),2,3),
(c,((x),&,(y)),2,3)
```

In that example, `x` and `y` are the input signals. The output signals sum `s` and carry `c` stabilize after a delay that lasts between 2 and 3 units of time after the excitement of their inputs.

Generating a DL formula corresponding to the asynchronous digital circuit definition in `1ba.adc` is achieved using:

```
adc2dl 1ba.adc 5 > 1ba-5.dl
```

For each step *i* in the generated run, the converter introduces for every *signal* a Boolean variable called `s_`*signal_i* denoting the stability of the signal in the gate output. The clock associated with the delay automaton of *signal* is named `X_`*signal_i* and the Boolean value of the signal at that step is *signal_i*. The time-stamp of that step is denoted by `T_`*i*.

Knowing these naming conventions, it is easy to produce a property file in DL that can be concatenated to the formula generated from the circuit to verify it.

For example, verifying if the 1-bit adder can stabilize in less than 4 time units for a run of length 5 when its inputs are initially enabled consists in concatenating the following DL formula chunk to `1ba-5.dl`:

```
(T_0=0)
& (~x_0 & x_1 & x_2 & x_3 & x_4 & x_5)
& (~y_0 & y_1 & y_2 & y_3 & y_4 & y_5)
& (~s_x_0 & ~s_y_0)
& (s_x_5 & s_y_5)
& (T_5<=4)
```

A.2.3 Job-shop to DL

`jsp2dl` reads a `.jsp` file containing the definition of a job-shop problem, as described in Chapter 5, and outputs a DL file that characterizes the set of feasible schedules. The program takes also a duration parameter that corresponds to w_d, the maximum length of the schedule that we try to find.

The definition of a job-shop consists in a header containing the number of the tasks followed by the number of machines, and a matrix whose rows define tasks. Each task is described by a succession of pairs, each of them contains a machine index followed by its usage duration. When a usage duration is null, the machine is not used. For example, the `plant.jsp` file below, corresponds to the job-shop presented in Figure 5.4 on page 81:

```
3  3
1  2  2  2  3  0
1  1  3  3  2  0
2  1  3  2  1  0
```

In order to obtain in `plant5.dl` the DL formula expressing the problem *"Is there a scheduling of the job-shop described in* `plant.fig` *with a duration less or equal to 5?"*, the user should run the following command:

```
jsp2dl -d 5 plant.jsp > plant5.dl
```

A.3 DL to MX-CNF Translator

The program `dl2mx` converts DL files to a mixed CNF set of clauses directly usable by the mixed solver. It starts by parsing and validating the DL formula, builds a tree representation, and simplifies it using the following simple rules:

$$\neg\neg f \rightarrow f$$

$$\begin{aligned} f \vee true &\rightarrow true \\ f \vee \neg true &\rightarrow f \\ f \vee f &\rightarrow f \\ f \vee \neg f &\rightarrow true \end{aligned}$$

$$\begin{aligned} f \wedge true &\rightarrow f \\ f \wedge \neg true &\rightarrow \neg true \\ f \wedge f &\rightarrow f \\ f \wedge \neg f &\rightarrow \neg true \end{aligned}$$

$$\begin{aligned} f \rightarrow true &\rightarrow true \\ f \rightarrow \neg true &\rightarrow \neg f \\ f \rightarrow f &\rightarrow true \\ f \rightarrow \neg f &\rightarrow \neg f \end{aligned}$$

$$\begin{aligned} f \Leftrightarrow true &\rightarrow f \\ f \Leftrightarrow \neg true &\rightarrow \neg f \\ f \Leftrightarrow f &\rightarrow true \\ f \Leftrightarrow \neg f &\rightarrow \neg true \end{aligned}$$

Then, `dl2mx` uses Tseitin's algorithm presented in Chapter 2 to do the translation. During that process, additional boolean variables may be created. To distinguish them from the original variables, their names begin with the character `#`.

The translator is invoked by the command line:

`dl2mx` *dl_file mx_file*

A.4 The Mixed SAT Solver

`mxsolver` and `mx3solver` read a `.mx` file containing the mixed SAT instance to solve. The solution process can be monitored at the will of the user. The result is dumped at the end on the screen or is saved in a file.

The solvers support a variety of command line parameters to change and control their behavior:

`mxsolver/mx3solver` [*mx_file*] [`-n` *rule*]* [`-z` *heuristic*] [`-d` *max_depth*] [`-s`] [`-q`] [`-a`] [`-r`] [`-o` *trace_file*] [`-h`]

The meaning of the parameters is as follows:

- *mx_file*: The name of the `.mx` input file containing the mixed SAT instance to solve. If not specified, input is read from `stdin`.
- `-n` *rule*: Disable the reduction rule whose acronym is *rule*. By default, all the rules are used. One or more rules can be disabled at the same time using multiple `-n` switches. Reduction rules abbreviations and availabilities are listed in the following table:

Rule	Abbreviation	`mxsolver`	`mx3solver`
Unit resolution	`ur`	√	√
Pure literals resolution	`plr`	√	√
Equivalence detection	`ed`	√	√
DBM updating	`du`	√	√
DBM Analysis	`da`	√	√
Restricted Boolean DP	`bdp`	√	
Restricted numerical DP	`ndp`	√	√

- `-z` *heuristic*: The branching heuristic to use. Two are currently implemented, namely `mom`, MOM's heuristic, and `first` which is the default heuristic that selects the first free variable.

- `-d` *max_depth*: The maximum branching depth that DPLL is allowed to reach. By default, DPLL has no limit on that parameter.
- `-s`: Print a summary on the solution process when it is over. This summary contains the satisfiability of the mixed SAT instance, the maximum amount of used memory, and the execution time.
- `-q`: Be quiet when solving and do not print a status line.
- `-a`: Dump the solution trace.
- `-r`: Dump the assignments when the solution processes ends.
- `-o` *trace_file*: The file where the solution trace should be dumped. By default, the trace is output to `stdout`.
- `-h`: Print a help screen.

A.5 DL file format

A DL file is a text file containing a *formula* whose syntax is defined by the following BNF notation:

formula ::= *variable*
| *variable relation constant*
| *variable* `-` *variable relation constant*
| *variable* `-` *constant relation variable*
| *constant* `-` *variable relation variable*
| *not formula*
| *formula and formula*
| *formula or formula*
| *formula imply formula*
| *formula equivalent formula*
| *formula xor formula*
| `(` *formula* `)`

variable ::= *alpha* (*alpha*|*digit*)$*$

constant ::= *digit*+

alpha ::= `A`−`Z`|`a`−`z`|`_`

digit ::= `0`−`9`

relation ::= `<`
| `<=`
| `>`
| `>=`
| `=`

not ::= `~`

and ::= `&`

or ::= `|`

imply ::= `->`

equivalence ::= `<->`

xor ::= `xor`

Blanks, carriage returns, and line feeds are ignored in DL files. Two variable names have special meanings:

- `true`: It refers to the special boolean variable *true*. `~true` refers to $false$.
- `zero`: It refers to the special numerical variable **0**.

A.6 Credits

- `tg2dl` has been written by Navendu JAIN[1] in 2001 based on a previous work done by the KRONOS team at VERIMAG[2].
- `jsp2dl` has also been written by Navendu JAIN in 2001 and modified by Moez MAHFOUDH[3] in 2002.
- `adc2dl` was written by Marius BOZGA[4] and has been modified and extended by Moez MAHFOUDH in 2002.
- `dl2mx`, `mxsolver`, and `mx3solver` were developed by Moez MAHFOUDH from 2001 to 2003.

[1] navendu@cse.iitd.ac.in
[2] kronos@imag.fr
[3] moez.mahfoudh@imag.fr
[4] marius.bozga@imag.fr

Appendix B

SCC Decomposition Algorithm

Strongly Connected Components or *SCC* decomposition algorithm operates on a directed graph and finds subgraphs in which every vertex can reach any other vertex by a path. It has been developed by Tarjan in 1972 [Tar72, CLR$^+$01].

B.1 Definitions

A graph G consists of two sets:

- V is a finite and non-empty set of vertices;
- E is a set of pairs of vertices, called *edges*.

G is denoted by $G = (V, E)$. We will also use $V(G)$ and $E(G)$ to represent the sets of vertices and edges of a graph G.

When every pair of vertices in a graph is ordered, the graph is said to be *directed*. In that case, we use $\langle v_1, v_2 \rangle$ to represent an ordered pair or a *directed edge*.

A path from v_i to v_j in a directed graph G is a finite sequence of directed edges $\langle v_i, v_{i+1} \rangle$, ..., $\langle v_{j-1}, v_j \rangle$ where every two successive edges share a vertex.

A graph is said to be *connected* iff $\forall v_1, v_2 \in V(G)$,there exists a path from v_1 to v_2.

A *connected component* is a connected subgraph of a graph.

A *strongly connected component* is a connected subgraph of a directed graph.

The *transpose* of graph $G = (V, E)$ is $G^T = (V, E^T)$ where

$$E^T = \{\langle v_2, v_1 \rangle \mid \langle v_1, v_2 \rangle \in E\}$$

The edge $\langle v_1, v_2 \rangle$ is said to be *incident* at v_2.

B.2 Depth-First search

The *Depth-First search* algorithm applied to a graph or *DFS* explores the edges that are incident at the most recently discovered vertex v and that have not yet been explored.

When all the edges incident at v have been explored, the search backtracks to explore edges at a higher level in the search tree. The search stops when all edges reachable from the source vertex have been explored.

The DFS algorithm takes a graph $G(V, E)$ and a source vertex s as input. To each vertex $v \in V(G)$, we associate:

- $d(v)$: The time when DFS is beginning to explore edges incident at v.
- $f(v)$: The time when DFS finished exploring edges incident at v.
- $c(v)$: The color of v that can be white, gray, or black.
- $\pi(v)$: The predecessor of v. If $\pi(v) = \perp$, v has no predecessor.

It starts by initializing the global variable *time* to 0 as well as all the predecessors, and the color of each vertex is set to white. Then it explores each white vertex.

A vertex v that is being explored has first its color set to to gray. *time* is then incremented and recorded in $d(v)$ as the exploration starting time. Each white adjacent vertex to v is explored after setting its predecessor to v. At the end of the exploration of v, its color is set to black while *time* is incremented and recorded in $f(v)$.

Algorithm 12 (DFS)

```
global time

DFS(G)
begin
    time := 0

    for each v ∈ V(G)
    begin
        π(v) := ⊥
        c(v) := white
    end

    for each v in V(G)
        if c(v) = white
        then
            DFS-Visit (G, v)
end

DFS-Visit(G, v)
begin
    c(v) := gray
    time := time + 1
    d(v) := time
```

```
        for each ⟨u1, u2⟩ ∈ E(G)
                if u1 = v and c(u2) = white
                then
                        begin
                                π(u2) := v
                                DFS-Visit (G, u2)
                        end

        c(v) := black
        time := time + 1
        f(v) := time
end
```

Example: Consider a directed graph made of four vertices s, x, y, and z. DFS starts by the source vertex s and requires 8 steps to finish the exploration (see Figure B.1). In this figure, each vertex v contains its start and finishing time written as $d(v)/f(v)$. □

Theorem 1 (Parenthesis theorem) *In any depth-first search of a graph G, only one of the following three conditions holds for any two vertices $v_1, v_2 \in V(G)$:*

1. $[d(v_1), f(v_1)] \cap [d(v_2), f(v_2)] = \varnothing$;
2. $[d(v_1), f(v_1)] \subsetneq [d(v_2), f(v_2)]$ *and v_1 is the descendant of v_2 in the depth first tree;*
3. $[d(v_1), f(v_1)] \supsetneq [d(v_2), f(v_2)]$ *and v_2 is the descendant of v_1 in the depth first tree;*

Remark 23 *We call the interval of a vertex v in a DFS the interval $[d(v), f(v)]$.*

According to the Parenthesis theorem, intervals of descendants vertices in a DFS are nesting. Moreover, for each vertex v, its immediate descendants intervals are disjoint and included in the interval of v.

B.3 SCC Algorithm

The SCC algorithm uses DFS in order to decompose a given directed graph G into a set of strongly connected components. It applies DFS to G to compute finishing time of each vertex in $V(G)$. Next, it computes the transpose of G and applies a special DFS on G^T. In that special DFS, vertices in the main loop are considered in the order of their decreasing finishing time $f(v)$ as computed in the first DFS.

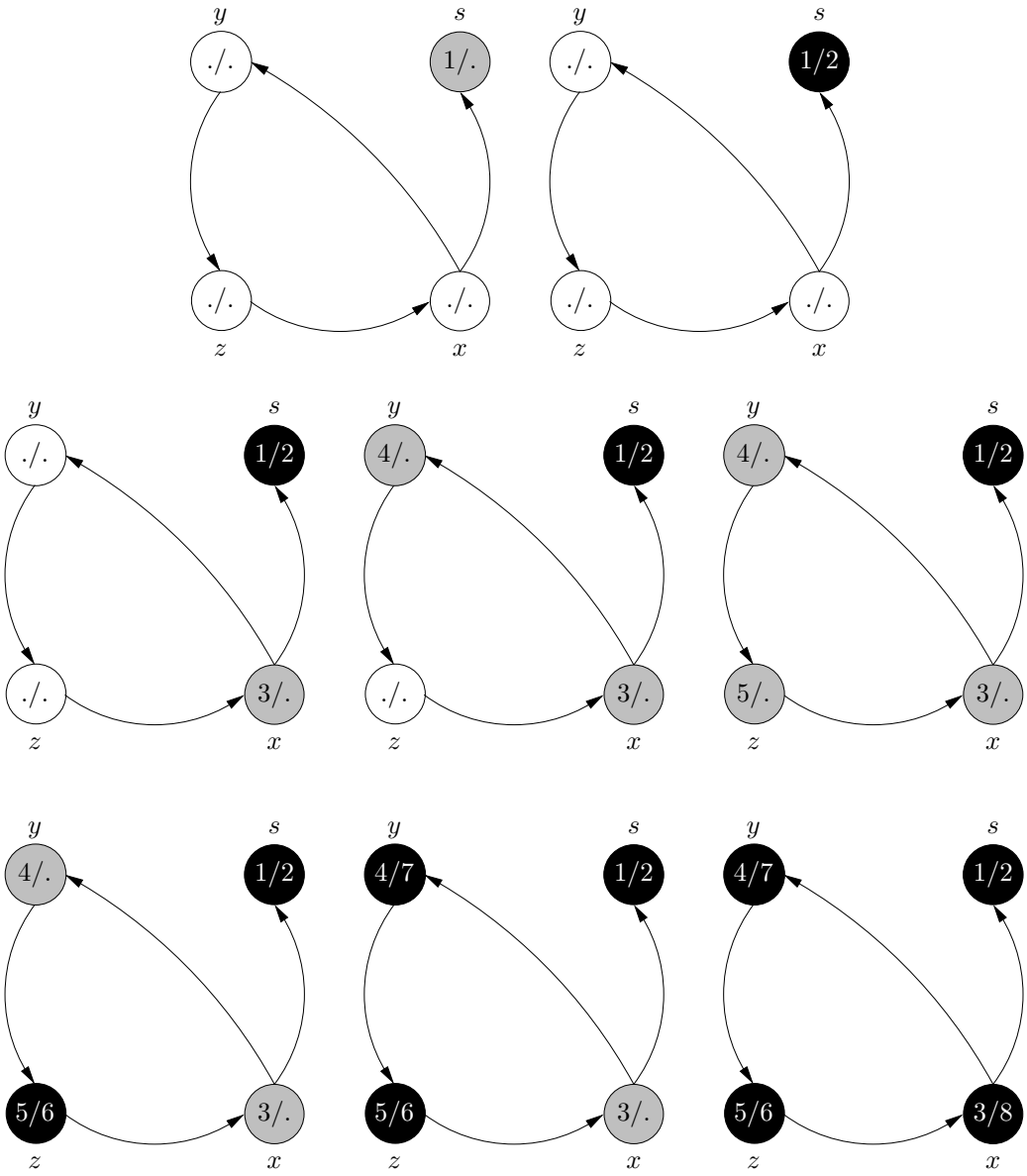

Figure B.1: Step-by-step DFS example

Using the results from the last computed DFS, we can extract the set of strongly connected components. According to the Parenthesis theorem, each SCC is made of a set of vertices $v_1, \ldots, v_k$ such that:

$$\forall i \in \{2, k\},\ [d(v_i), f(v_i)] \subsetneq [d(v_1), f(v_1)]$$

That property leads to a simple and efficient SCC extraction algorithm:

- The vertices in $V(G)$ are sorted in the order of their increasing starting time as computed by the DFS.
- That order guarantees that there exists a subdivision of the array such that each sub-array starts by a vertex v_1 and all subsequent vertices intervals are fully included in the interval of v_1. The array is then scanned sequentially to subdivide it with respect to that property.

Algorithm 13 (SCC)

SCC(G)
begin
    $S := \varnothing$

    DFS(G)
    DFS-sort-descending-f(G^T)

    $V' :=$ **sort-increasing-d**$(V(G^T))$

    for $i := 1$ **to size**(V')
        begin
            $W := \{V'_i\}$

            for $j := i + 1$ **to size**(V')
                if $d(V'_i) < d(V'_j) < f(V'_j) < f(V'_i)$
                then
                    $W := W \cup \{V'_j\}$
                else
                    break

            $S := S \cup \{W\}$
        end

    return S
end

Example: To illustrate the above algorithm, we apply it on the same graph

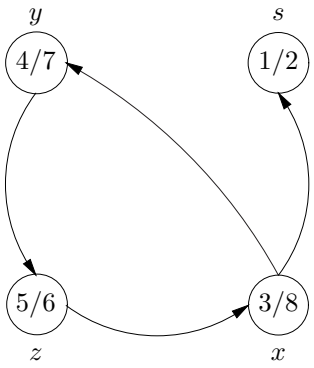

Figure B.2: SCC example: $DFS(G)$

G already used in the previous example. Figure (B.2) represents the result of DFS on G.

Figure (B.3) depicts the transpose of G. It is simply obtained by reversing the edges directions.

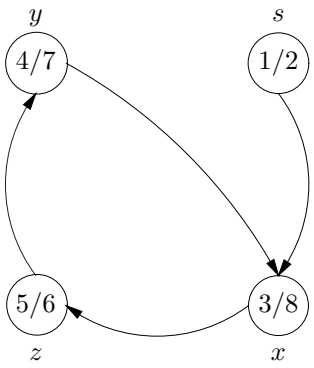

Figure B.3: SCC example: G^T

Figure (B.4) shows the result of applying DFS on G^T with respect to the order of their decreasing finishing time according to the first DFS. The main loop of that DFS is applied successively on x, y, z, and s.

Sorting the vertices in the order of their increasing start time in the last DFS gives the array x, z, y, and s. We can deduce that the first SCC is made of vertices x, z, and y. In fact, $[d(z), f(z)] \subsetneq [d(x), f(x)]$ and $[d(y), f(y)] \subsetneq [d(x), f(x)]$. The second SCC contains only the vertex s. □

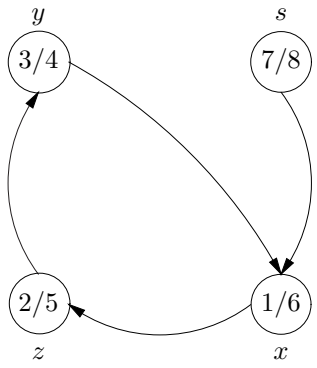

Figure B.4: SCC example: $DFS(G^T)$

Bibliography

[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicks, and R. Sebastiani. A SAT-based approach for solving formulas over boolean and linear mathematical propositions. In *Proceedings of CADE'02*, volume 2392 of *LNCS*, pages 193–208. Springer, 2002.

[ABK+97] E. Asarin, M. Bozga, A. Kerberat, O. Maler, A. Pnueli, and A. Rasse. Data structures for the verification of timed automata. In *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 1997.

[ABZ88] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job-shop scheduling. In *Management Science*, volume 34, pages 391–401, 1988.

[ACG99] A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proceedings of ECP'99*, LNCS. Springer, 1999.

[ACKS02] G. Audemard, A. Cimatti, A. Kornilowicks, and R. Sebastiani. Bounded model checking for timed systems. Technical report, IRST, Trento, 2002. Technical Report ITC-0201-05.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AHU74] A.V. Aho, J.E. Hopcroft, , and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[BCC+99a] A. Biere, A. Cimatti, E. Clarke, , and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of TACAS'99*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[BCC+99b] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of BDDs. In *Proceedings of DAC'99*, pages 317–320, 1999.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[BJLY98] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial order reductions for timed systems. In *Proceedings of Concur'98*, volume 1466 of *LNCS*, pages 485–500. Springer, 1998.

[BJMY02] M. Bozga, H. Jianmin, O. Maler, and S. Yovine. Verification of asynchronous circuits using timed automata. In *Proceedings of TPTS'02*, 2002.

[BKB92] M. Buro and H. Kleine-Büning. Report on a SAT competition. Technical report, University of Paderborn, November 1992.

[BM00] O. Bournez and O. Maler. On the representation of timed polyhedra. In *ICALP'2000*, volume 1853 of *Lecture Notes in Computer Science*, pages 793–807. Springer, 2000.

[BMT99] M. Bozga, O. Maler, and S. Tripakis. Efficient verification of timed automata using dense and discrete time semantics. In L. Pierre and T. Kropf, editors, *Proceedings of CHARME'99*, volume 1703 of *LNCS*, pages 125–141. Springer, 1999.

[Bry92] Randal E. Bryant. Ordered binary decision diagrams. *ACM Computing Surveys*, 24(3), 1992.

[BS97] Roberto J. Jr. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[CA93] James M. Crawford and Larry D. Auton. Experimental results on the crossover point. In *Proceedings of the National Conference on Artificial Intelligence*, pages 22–28, 1993.

[CLR+01] T. Cormen, C. Leiserson, R. Rivest, , and C. Stein. *Introduction to Algorithms*. MIT Press, McGraw-Hill, 2001.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[CS88] Vašek Chvátal and Endre Szemerédi. Many hard examples for resolution. *Journal of the ACM*, 35(4):759–768, 1988.

[DABC93] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. *Second DIMACS Implementation Challenge*, 1993.

[DD01] Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *IJCAI*, pages 248–253, 2001.

[DE73] G. B. Dantzig and B. C. Eaves. Fourier-Motzkin elimination and its dual. In *Journal of Combinatorial Theory*, volume 14, 1973.

[DHJP83] N. Dershowitz, J. Hsiang, N. Josephson, and D. Plaisted. Associative-commutative rewriting. In *Proceedings of IJCAI*, pages 940–944, 1983.

[Dil89] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212, 1989.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Cmmunications of the ACM*, 5:394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[Fra02] Martin Franzle. Take it NP-Easy: Bounded model construction for duration calculus. In *Proceedings of FTRTFT'02*, volume 2469 of *LNCS*, pages 245–264. Springer, 2002.

[Fre95] Jon W. Freeman. *Improvements to propositional satisfiability algorithms*. PhD thesis, University of Pennsylvania, 1995.

[FT63] H. Fisher and G.L. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. In J.F. Muth and G.L. Thompson, editors, *Industrial Scheduling*, pages 225–251. Prentice-Hall, Engelwood Cliffs, New Jersey, 1963.

[Gal77] Zvi Galil. On resolution with clauses of bounded size. *SIAM Journal on Computing*, 6(3):444–459, 1977.

[Gal86] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row, 1986.

[Gau99] Stéphane Gaubert. Introduction aux systèmes dynamiques à évènements discrets. http://amadeus.inria.fr/gaubert, 1999. Course notes in French, ENSMP, Option Automatique & DEA ATS Orsay.

[GJ79] M. Garey and D. Johnson. *Computers and Interactability*. W. H. Freeman, 1979.

[GM94] M. L. Ginsberg and D. A. McAllester. GSAT and dynamic backtracking. In P. Torasso, J. Doyle, and E. Sandewall, editors, *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*, pages 226–237. Morgan Kaufmann, 1994.

[GN02] E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT solver. In *Design Automation and Test in Europe (DATE'02)*, pages 142–149, 2002.

[GT93] A. Van Gelder and Y. K. Tsuji. Satisability testing with more reasoning and less guessing. In D. S. Johnson and M. A. Trick, editors, *Second DIMACS Implementation Challenge*. American Mathematical Society, 1993.

[GW99] J. F. Groote and J. P. Warners. The propositional formula checker HeerHugo. Technical Report SEN-R9905, Centrum voor Wiskunde en Informatica (CWI) Amsterdam, January 1999.

[GW00] J. F. Groote and J. P. Warners. The propositional formula checker HeerHugo. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT2000: Highlights of Satis ability Research in the year 2000, Frontiers in Arti cial Intelligence and Applications*. Kluwer Academic, 2000.

[Hak85] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[Har96] John Harrison. Stålmarck's algorithm as a HOL derived rule. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 221–234, Turku, Finland, 1996. Springer Verlag.

[Hsi85] J. Hsiang. Refutational theorem proving using term-rewriting systems. *Artificial Intelligence*, pages 255–300, 1985.

[JM99] A. S. Jain and S. Meeran. Deterministic job-shop scheduling: Past, present, and future. In *European Journal of Operational Research*, pages 390–434, 1999.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[LA97] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of Third International Conference on Principles and Practice of Constraint Programming (CP'97)*, LNCS, pages 341–355. Springer, 1997.

[Law84] S. Lawrence. *Supplement to Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques*. Graduate School of Industrial Administration, Carnegie-Mellon University, 1984.

[LLPY97] K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structures and state-space reduction. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, 1997.

[LLPY02] K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Compact data structure and state-space reduction for model-checking real-time systems. In *The International Journal of Time-Critical Computing Systems*, 2002.

[LPWY99] K. Larsen, J. Pearson, C. Weise, and W. Yi. Clock difference diagrams. In *Nordic Journal of Computing*, pages 271–298, 1999.

[MLA+99] J. Møller, J. Lichtenberg, H. Andersen, , and H. Hulgaard. Difference decision digrams. In *CSL'99*, 1999.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the* 39^{th} *Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[MP95] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In P.E. Camurati and H. Eveking, editors, *Proceedings of CHARME'95*, volume 987 of *LNCS*, pages 189–205. Springer, 1995.

[MRS02] L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proceedings of CADE'02*, volume 2392 of *LNCS*, pages 437–453. Springer, 2002.

[Oli94] A. Olivero. *Modélisation et Analyse des Systèmes Temporisés et Hybrides*. PhD thesis, Institut National Polytechnique de Grenoble, September 1994.

[Pre93] D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. In *Proceedings of the DIMACS Callenge II Workshop*, 1993.

[PWZ02] W. Penczek, B. Woźna, and A. Zbrzezny. Towards bounded model checking for the universal fragment of TCTL. In *Proceedings of FTRTFT'02*, volume 2469 of *LNCS*, pages 265–288. Springer, 2002.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[SBS02] S. Seshia, R. Bryant, and O. Strichman. Deciding separation formulas with SAT. In *Computer Aided Verification 2002, CAV'02*, 2002.

[Sie87] P. Siegel. *Représentation et Utilisation de la Connaissance en Calcul Propositionnel*. PhD thesis, Université Aix-Marseille II, July 1987.

[SS90] G. Stålmarck and M. Säflund. Modelling and verifying systems and software in propositional logic. In B. K. Daniels, editor, *Safety of Computer Control Systems (SAFECOMP'90)*. Pergamon Press, 1990.

[SS99] J.P. Marques Silva and K. A. Sakallah. GRASP-a search algorithm for propositional satisfiability. In *IEEE Transactions on Computers*, volume 48, pages 506–521, 1999.

[Stå94] Gunnar Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from boolean formula. US Patent 5,276,897; Canadian Patent 2,018,828; European Patent 0 403 545; Swedish Patent 467 076, 1994.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. In *SIAM Journal of Computing*, volume 1, pages 146–160, 1972.

[Tse68] G. S. Tseitin. On the complexity of derivations in the propositional calculus. In A. O. Slisenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125, 1968. Translated from Russian.

[Urq87] Alasdair Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, 1987.

[Wil90] J. M. Wilson. Compact normal forms in propositional logic and integer programming formulations. In *Computers and Operation Research*, pages 309–314, 1990.

[Zha97] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.

[ZM88] Ramin D. Zabih and David A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of AAAI-88*, pages 155–160, 1988.

RÉSUMÉ : Dans cette thèse, nous développons un solveur SAT mixte pour la logique des différences (DL) pour réaliser la vérification bornée de modèles sur des systmes temporisés.
DL est définie comme une extension de la logique propositionnelle avec des contraintes de différences de la forme $(x - y < c)$. Nous présentons des algorithmes pour convertir des formules de cette logique à la forme normale conjonctive mixte appropriée pour la résolution. Nous examinons les solveurs SAT booléens existants et les techniques qu'ils utilisent. Certaines sont étendues et intégrées dans le solveur mixte avec de nouvelles méthodes développées spécifiquement pour les contraintes de différences.
Nous nous intéressons à trois classes de systmes temporisés, nommément les automates temporisés, les circuits asynchrones digitaux, et les job-shops, et expliquons leur expression en DL. Des problèmes issus de ces classes sont générés pour évaluer la performance d'une implémentation expérimentale d'un solveur SAT mixte.

MOTS-CLÉS : Satisfaction, Logique des différences, Systèmes temporisés, Vérification formelle, Vérification bornée de modèles, Solveur.

ABSTRACT: In this thesis, we develop a mixed SAT solver for difference logic (DL) to carry out bounded model checking for timed systems. We define DL as an extension of propositional Boolean logic with difference constraints of the form (x-y¡c). We present algorithms to convert formulae in this logic to the mixed conjunctive normal form, which is appropriate for solving purposes. We examine the architecture of existing Boolean SAT solvers and the techniques they use. Some of them are extended and integrated in the mixed solver along with newly developed methods specific to difference constraints.
We focus on three classes of timed systems, namely timed automata, asynchronous digital circuits, and non-preemptive job-shops, and explain how they can be expressed in DL. Problems belonging to these classes are generated to evaluate the performance of an experimental implementation of a mixed SAT solver.

KEYWORDS: Satisfiability, Difference logic, Timed systems, Formal verification, Bounded model checking, Solver.

LABORTAOIRE D'ACCEUIL: VERIMAG, 2, avenue de Vignate, 38610 Gières, France, http://www-verimag.imag.fr