Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação

Deciding difference logic in a Nelson-Oppen combination framework

Diego Caminha Barbosa de Oliveira

Natal / RN
Novembro de 2007

Diego Caminha Barbosa de Oliveira

Deciding difference logic in a Nelson-Oppen combination framework

Dissertação submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como parte dos requisitos para a obtenção do grau de Mestre em Sistemas e Computação (MSc.).

Prof. Dr. David Boris Paul Déharbe
Orientador

Natal / RN
Novembro de 2007

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial
Especializada do Centro de Ciências Exatas e da Terra – CCET.

Oliveira, Diego Caminha Barbosa de.
Deciding difference logic in a Nelson-Oppen combination framework / Diego Caminha Barbosa de Oliveira. – Natal, 2007.
70 f. : il.

Orientadora: Prof. Dr. David Boris Paul Déharbe
Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação.

1. Decision procedure – Dissertação. 2. Difference logic – Dissertação. 3. Nelson Oppen – Dissertação. 4. SMT – Solvers – Dissertação. 5. Formal Methods – Dissertação. I. Déharbe, David Boris Paul. II. Título

RN/UF/BSE-CCET CDU: 004.72

Resumo O método de combinação de *Nelson-Oppen* permite que vários procedimentos de decisão, cada um projetado para uma teoria específica, possam ser combinados para inferir sobre teorias mais abrangentes, através do princípio de propagação de igualdades. Provadores de teorema baseados neste modelo são beneficiados por sua característica modular e podem evoluir mais facilmente, incrementalmente.

Difference logic é uma subteoria da aritmética linear. Ela é formada por *constraints* do tipo $x - y \leq c$, onde x e y são variáveis e c é uma constante. *Difference logic* é muito comum em vários problemas, como circuitos digitais, agendamento, sistemas temporais, etc. e se apresenta predominante em vários outros casos.

Difference logic ainda se caracteriza por ser modelada usando teoria dos grafos. Isto permite que vários algoritmos eficientes e conhecidos da teoria de grafos possam ser utilizados. Um procedimento de decisão para *difference logic* é capaz de induzir sobre milhares de *constraints*.

Um procedimento de decisão para a teoria de *difference logic* tem como objetivo principal informar se um conjunto de *constraints* de *difference logic* é satisfatível (as variáveis podem assumir valores que tornam o conjunto consistente) ou não. Além disso, para funcionar em um modelo de combinação baseado em *Nelson-Oppen*, o procedimento de decisão precisa ter outras funcionalidades, como geração de igualdade de variáveis, prova de inconsistência, premissas, etc.

Este trabalho apresenta um procedimento de decisão para a teoria de *difference logic* dentro de uma arquitetura baseada no método de combinação de *Nelson-Oppen*.

O trabalho foi realizado integrando-se ao provador haRVey, de onde foi possível observar o seu funcionamento. Detalhes de implementação e testes

experimentais são relatados.

Contents

1 Introduction **1**

2 Techniques for SMT solving **5**

2.1 Introduction 5

2.2 Nelson and Oppen Combination Framework 7

2.3 Sat-Solver with DPLL 9

2.4 Conclusion 12

3 Difference Logic **13**

3.1 Introduction 13

3.2 Properties and Graph Representation 14

3.2.1 Dependency 16

3.2.2 Strongest Constraint 16

3.2.3 Unsatisfiability 17

3.2.4 Equality Between Variables 18

3.3 Conclusion 20

4 Decision Procedure for Difference Logic **21**

4.1 Requirements 21

4.1.1 Conflict Set Generation 22

4.1.2 Equality Generation 23

4.1.3 Premisses 25
4.1.4 Incrementability 26
4.2 Algorithms for the Difference Logic Decision Procedure 28
4.2.1 Satisfiability Checking 29
4.2.2 Incremental Satisfiability Checking 30
4.2.3 Conflict Set Construction 34
4.2.4 Equality Generation 34
4.2.5 Premisses Information 40
4.3 Conclusion 42

5 Implementation details **44**
5.1 Benchmark 44
5.2 Handling Constraints 46
5.3 Building the Graph 47
5.4 Conclusion 50

6 Experimental results **52**

7 Conclusion **60**

Chapter 1

Introduction

The construction of software that works perfectly is always desired. But building software 100% free of failures is most of the times a hard task. Many techniques may be applied during software construction to make they work as well as possible. Good software engineering, architecture and coding can help to achieve this goal, but usually they are not enough. Tests are the most common way to try to ensure that a software works as expected, but it might be hard (or even impossible) to test it exhaustively. Formal methods try to prove formally models of a software, so that, if it is proved, we can be totally sure that the model is correct with respect to the proven properties. But as any of the other techniques, they have their limitations.

It is not all sort of software that have the real need to be entirely correct. For many applications the extra effort to ensure that it works perfectly is not even worth. That is because using techniques to prove the correctness may take a long time and may also not be easy to use. Sometimes, software can be released when it seems to work correctly, and later, when a problem comes out, it is fixed and a new version is released. That is usually what happens and it is not a serious concern if these hypothetical problems do not

cause a big loss when it comes to lives or money.

The real necessity of software that works correctly comes from critical applications. This is where software deals with human lives, money, credibility, that cannot be changed, etc. Software that controls airplanes, space rockets, air traffic, metro, trains, and so on, deals with lives. Making sure it is correct is essential, as lives cannot be brought back. System that deals with money, like bank and commercial software, can spend an extra effort during the construction of the software to make sure they will not lose money later. Operating Systems need to have the credibility that they will work correctly, as they are the base for many other software. Bug free software may also be a good propaganda for companies that produce such software, as they may gain the trust of their users. As a last example, embedded systems or hardware, that once out in the market cannot be changed, need extra attention too.

Testing is a mandatory phase when making a software. It is a good starting point and may detect many simple and common mistakes as well as deeper problems such as wrong algorithms. A simple way of completely testing the system and be sure it works correctly is to test all possible input configurations and to see if they return the expected results. But testing every input configuration is only possible for very few special cases as the number of configurations usually grows exponentially with the size of the input. So testing them all would take unreasonable time even for relatively small inputs. Also, in some situations, testing all the cases is impossible as they might be infinite. Therefore, for most software, testing techniques must be applied, using only a fraction of the whole input set of configurations. It will probably detect some errors, but it will not be possible to know if a software if free of failures, as it cannot test all input configurations.

Formal methods come to help proving the correctness of software. There are many techniques that aim to prove that an implementation of an algorithm works, based on formal theories. But proving that any code on any programming language gives always the expected result is not possible, as there is no formal theory that handles all the aspects of an arbitrary programming language. So, normally, important aspects of the software are translated to a model that can be then checked. After that, the proof may be done with the help of tools that can be interactive or automated.

Formal methods are in constant evolution, as researchers create new methodologies, model languages and more robust provers. Although currently not attractive to all, it is well used in industry in many cases and it is also a promising area, as one of its main goals is to be suitable for more users and situations.

The aspects to be proven usually have many theories involved of different complexity levels: theory of lists, arrays, functions, numbers, sets, etc. Making a tool that can infer about a combination of them might not be an easy task. A common way of doing it is to have a framework that combines decision procedures that can infer about single theories, e.g., the Nelson and Oppen combination framework [14]. That also allows a prover to be more adaptive and to progressively incorporate new decision procedures that will make it possible to prove more facts.

In number theory, a very common branch is linear arithmetic. Proving for instance that $x + x = 1$ is valid in the real field, but not in the integer field, is one of its concerns. Many arithmetic information can be extracted from applications and most of it is linear.

An interesting fraction of linear arithmetic is *difference logic*. It is based on constraints of the form $x - y \leq c$, where x and y are variables and c

is a numerical constant. Although very simple, difference logic can express important practical problems like timed systems, scheduling problems and paths in digital circuits (see for example [15]). It also appears in a huge proportion inside linear arithmetic problems. Difference logic can be entirely modelled with graph theory. That allows the use of fast known algorithm making trackable the resolution of large problem instances.

This work aims at the construction of a decision procedure for difference logic and its integration to a theorem prover (haRVey [8, 11]) based on the Nelson and Oppen combination framework. The aim of the decision procedure is to collect the information necessary to prove the difference logic constraints giving the correct status of satisfiability along with other important information necessary to work well inside the combination framework.

Following in this thesis, there will be information about: theorem proving, where we will describe Nelson and Oppen combination framework and Sat-Solvers with DPLL [9, 10], focusing their importance to understand well the requirements; difference logic, where we will explore the interesting properties always using the graph analogy, for later introducing the algorithms necessary for the decision procedure; finally some implementation details will show the minor differences from the theory. The aim is to work correctly for the benchmark from SMT-LIB [17].

Chapter 2

Techniques for SMT solving

In this chapter we give a quick introduction to theorem proving and describe the Nelson and Oppen combination framework [14] and the SAT-Solver with DPLL [9, 10]. They are important as a basis to understand well following chapters, particularly the requirements and some of the algorithms.

2.1 Introduction

The complexity and functionality of software and hardware systems are growing and so is the error probability when making them. Money, time and even lives can be lost because of failures, and therefore, avoiding them is a big concern in critical systems.

Software engineering comes to help developing systems with methodological techniques, so big projects can be made despite their complexity. One of its concerns is to make systems correct. Besides testing, formal methods are being used to certify the correctness of software and hardware systems. Formal methods are languages based on mathematics, techniques and tools used to specify and verify systems. They do not guarantee, a priori, the cor-

rectness of a system. However, they improve the understanding about it and help to reveal inconsistencies, ambiguities and incompleteness that otherwise would not be noticed.

In formal verification, two approaches are more spreadly used [4]: model checking and theorem proving. They are used to prove or disprove a system property. Both are based on formal specification.

A formal specification is a mathematical model of a system or part of it. It helps to remove the ambiguities that a specification in a natural language would normally have. It needs to be precise and correct, so its validation would reflect the validation of the modelled system. Examples of formal specification models are: finite state machines, labelled transition systems, Petri nets, timed automata, hybrid automata, process algebra, formal semantics of programming languages such as operational semantics, denotational semantics, axiomatic semantics and Hoare logic.

Model checking consists of exhaustive searches of a (finite) formal model. Using some techniques to reduce the computing time, it explores all states and transitions of the model.

Theorem proving will searches for the proofs of the modelled properties of a system, using axioms and inference rules. The proof may even be produced by hand, but even for small specifications this can become a hard task. Theorem provers are used to help this process. They may be interactive or automated [19].

2.2 Nelson and Oppen Combination Framework

Nelson and Oppen combination framework [14] allows a theorem prover to handle larger (combined) theories that otherwise would be hard or impossible to deal with. It will combine decision procedures where each of them only needs to worry about its specific theory and generate the equalities with the set of constraints that was given.

A single decision procedure that handles combined theories might be possible to make. However, it will suffer from modularity and it will probably have difficulties to grow and incorporate new theories, as adding a new theory might make it totally different.

Nelson and Oppen combination framework has as strong property its modularity. A decision procedure in this framework has to worry only about its designed theory. If a theorem prover wants to raise its capacity and handle a new theory, it is just necessary to design a new decision procedure for the theory and incorporate it to the theorem prover.

The main idea of Nelson and Oppen combination framework is the propagation of equalities between decision procedures. Every time a new equality is generated by a decision procedure it will be propagated to all the other decision procedures which, with more information, can try to infer about more things. The process continues until unsatisfiability is detected by one of the decision procedures or until no more equalities are found.

To explain how it works, consider the following formula:

$$x \leq y \wedge y \leq car(cons(x, l)) \wedge P(f(x) - f(y)) \wedge \neg P(0)$$

The formula contains information from three theories. There is arithmetic with relational operator $\leq$, subtraction function $-$ and number 0; the

uninterpreted function f and predicate P; and list functions *car* (that returns the first element of a list) and *cons* (that adds a new element to the head of a list and returns the new list).

Evaluating the formula and dispatching, according to the Nelson and Oppen scheme, the appropriated constraints to their corresponding decision procedures, we would get what is shown in Table 2.1. Every literal that has information from more than one theory is split so that each part can be dispatched directly to a decision procedure. This is done by introducing fresh variables and equalities. For example, $y \leq car(cons(x, l))$ becomes $y \leq v_1 \wedge v_1 = car(cons(x, l))$.

level	**DP A**	**DP UF**	**DP L**
0	$x \leq y$ $y \leq v_1$ $v_2 = v_3 - v_4$ $v_5 = 0$	$\underline{P(v_2) = true}$ $\underline{P(v_5) = false}$ $v_3 = f(x)$ $v_4 = f(y)$	$v_1 = car(cons(x, l))$ $v_1 = x$ **(detected)**
1	$v_1 = x$ $x = y$ **(detected)**	$v_1 = x$	
2		$x = y$ $v_3 = v_4$ **(detected)**	$x = y$
3	$v_3 = v_4$ $v_2 = v_5$ **(detected)**		$v_3 = v_4$
4		$\underline{v_2 = v_5}$ $\underline{\text{unsatisfiable}}$	$v_2 = v_5$

Table 2.1: Checking satisfiability with Nelson and Oppen method.

There are 3 decision procedures, the first one for arithmetic (DP A), the second one for uninterpreted function and predicate (DP UF) and the third one for list theory (DP L). We have the following sequence of events through time:

0 - At a first moment, all the decision procedures (DPs) are executed and they cannot detect unsatisfiability. Then $v_1 = x$ is detected (*cons* adds

the element x to the beginning of list l and *car* returns the first element of the resulting list, i.e., x) by *DP L* and propagated to the others DP.

1 - With a new constraint received, both *DP A* and *DP UF* can be executed again. The set of constraints is almost the same, except for the new equality received that should also be considered now. After checking the satisfiability, the status still does not change and remains satisfiable, but a new equality is detected by *DP A* $((v_1 = x) \Rightarrow (x \leq y \wedge y \leq v_1 \Leftrightarrow x \leq y \wedge y \leq x \Leftrightarrow x = y))$.

2 - Once more, two DPs have a new constraint and can check for satisfiability one more time. The status continues the same, and a new equality is detected by *DP UF* $((x = y \wedge f(x) = v_3 \wedge f(y) = v_4) \Rightarrow (v_3 = v_4))$.

3 - Once again, new constraints are received and the status does not change. A new equality is detected by *DP A* $((v_3 = v4 \Rightarrow v_3 - v_4 = 0)$ and $(v_2 = 0 \wedge v_5 = 0 \Rightarrow v_2 = v_5))$.

4 - Finally, with the new equality $v_2 = v_5$, DP UF is able to detect the unsatisfiability due to a contradiction in $v_2 = v_5 \wedge P(v_2) = true \wedge P(v_5) = false$.

2.3 Sat-Solver with DPLL

Given a formula, if there are only conjunctions, we can know if the formula is satisfiable by just making one check. We assume that all the literals are *true* and check if there is no contradiction. For such a case a Sat-Solver is not even necessary.

But if the formula has disjunctions, the number of checks will grow exponentially. To see this effect consider the following formula, where $A..F$ are

constraints in some theory:

$$(A \vee B) \wedge (C \vee D) \wedge (E \vee F)$$

To check if it is satisfiable using brute force, we might need to do 8 consistency checks. When we do a check, we assume some literals as *true* and check for their consistency. If one of them is consistent the formula is satisfiable. Otherwise, if all the checks have contradictions the problem is unsatisfiable. The brute force checks would be:

$$A \wedge C \wedge E$$
$$A \wedge C \wedge F$$
$$A \wedge D \wedge E$$
$$A \wedge D \wedge F$$
$$B \wedge C \wedge E$$
$$B \wedge C \wedge F$$
$$B \wedge D \wedge E$$
$$B \wedge D \wedge F$$

This is the Boolean satisfiability problem (SAT) and was the first known NP-complete problem, as proved by Stephen Cook [5].

When there are repeated information, a Sat-Solver with smart techniques and heuristics becomes more interesting than just a brute force solver. Consider now this formula:

$$(A \vee B) \wedge (C \vee F) \wedge (A \vee F)$$

A smart Sat-Solver with only 3 checks, with less literals, could see if the formula is satisfiable:

$$A \wedge C$$

$$A \wedge F$$

$$B \wedge F$$

A very used technique in Sat-Solvers is DPLL [9, 10]. It is a complete, backtracking-based algorithm for checking the satisfiability of propositional logic formulas in CNF (Conjunctive Normal Form - a formula is in CNF if it is a conjunction of clauses, where a clause is a disjunction of literals).

The basic algorithm is shown in Algorithm 1. It recursively chooses a literal, assigning a truth value to it, simplifies the formula and then checks if the simplified formula is satisfiable; if this is the case, the original formula is satisfiable; otherwise, the same recursive check is done assuming the opposite truth value.

input : $\phi : Formula$
output: $isSatisfiable : Boolean; \phi' : Formula$

1 **if** ϕ *is a consistent set of literals* **then**
2 | **return** true;
3 **end**
4 **if** ϕ *contains an empty clause* **then**
5 | **return** false;
6 **end**
7 **foreach** *unit clause* l *in* ϕ **do**
8 | ϕ = `UnitPropagate`(l, ϕ);
9 **end**
10 **foreach** *literal* l *that occurs pure in* ϕ **do**
11 | ϕ = `PureLiteralAssign`(l, ϕ);
12 **end**
13 l = `ChooseLiteral`(ϕ);
14 **return** `DPLL`($\phi \wedge l$) or `DPLL`($\phi \wedge \neg l$);

Algorithm 1: DPLL

The simplification step essentially removes all clauses which become true under the assignment from the formula, and all literals that become false

from the remaining clauses.

The unit propagation checks if the clause is a *unit clause*, i.e., it contains only a single unassigned literal. This clause can become true by just assigning the only value possible to make the literal true.

The pure literal assign will check if the literal occurs with only one polarity in the formula (pure). Pure literals can always be assigned in a way that makes all clauses containing them true, so the clauses can be deleted as they do not constrain anymore.

The formula is unsatisfiable if the clause becomes empty, i.e., there were assignments that made the clause *false*.

Choosing the literal may change the search tree of the satisfiability check. Some heuristics can be used for this.

This is the basic DPLL. Many extensions were and are been developed, each of them may be good for a specific scenario. They may change the algorithm a bit using new heuristics, but a high degree of similarity is preserved.

2.4 Conclusion

The information here presented gives the basis to understand the requirements of the decision procedure that is proposed later in this thesis. Nelson and Oppen is strongly necessary to understand these requirements. DPLL is the technique that the Sat-Solver of haRVey is based on. But from the viewpoint of the decision procedure, its knowledge is mainly necessary to justify some requirements.

Chapter 3

Difference Logic

In this chapter, we describe difference logic. We give the details of how it can be understand using graph theory and show some properties necessary for a good comprehension of the algorithms for the decision procedure presented later.

3.1 Introduction

Difference Logic (DL) is the part of arithmetic that handles constraints of the type $x - y \leq c$, where x and y are variables and c is a numerical constant that can be real, integer or rational. It appears in lots of different and important practical problems such as timed systems, scheduling problems, paths in digital circuits, see, e.g., [15]. They also are the predominant kind of constraint in problems involving arithmetic.

Difference logic is a well studied subject and it can be fully modelled with graph theory, using many fast known algorithms that this theory provides. Therefore, solvers that are aware of these facts and make use of these algorithms may accomplish a good performance.

Difference logic can be found in books, see, e.g., [6]. Its study is strongly related to graphs and algorithms for satisfiability check, but usually are superficial. Here, we go deeper, describing in details interesting properties necessary to our decision procedure and illustrating them.

3.2 Properties and Graph Representation

The classic difference logic problem deals only with $x - y \leq c$. It can be interpreted in graph theory as an edge from y to x with cost (or weight) c, see Figure 3.1, for both real and integer theory. That can be read as node (or vertex) x should be at most node $y + c$.

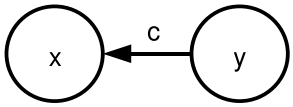

Figure 3.1: Representation of the constraint $x - y \leq c$ using graphs.

There are other constraints that can be easily translated and integrated to this graph model. Table 3.1 gives a few of the common constraints that can be translated to DL.

constraint	translated to
$x - y \geq c$	$y - x \leq -c$
$x - y = c$	$x - y \leq c$ and $y - x \leq -c$
$x \leq y$	$x - y \leq 0$
$x \geq y$	$y - x \leq 0$
$x = y$	$x - y \leq 0$ and $y - x \leq 0$

Table 3.1: Table of constraints

Strict inequalities can also be handled with minor changes. We can always change a constraint like $x - y < c$ to $x - y \leq c - \delta$. The value of δ depends

on the type of numerical variables presented in the constraint. If they are all integers, δ is precisely 1, e.g., we can change $x - y < 1$ to $x - y \leq 0$ without any loss of precision. But if there are rationals or reals in the constraint, δ has to be a very small value, small enough to not change the result of evaluating the constraints.

It is a hard task to determine the value of δ in this case. However, we can always say that the value of δ is something very small like $\frac{1}{\infty}$. Representing this value in the computer is not possible, but we do not actually need it. We can say that δ is $\frac{1}{\infty}$, do the calculations using the symbol δ as a variable and see its real value when necessary.

Instead of evaluating a number c as itself, we think of it as a pair (c, k) equivalent to $c + k\delta$. The operations on the pair are like the ones we use for equations $(c + k\delta)$, where c and k are known and δ is a variable. Only a few operations will be necessary:

- $(c_1, k_1) + (c_2, k_2) \equiv (c_1 + c_2, k_1 + k_2)$
- $(c_1, k_1) - (c_2, k_2) \equiv (c_1 - c_2, k_1 - k_2)$
- $c' \times (c, k) \equiv (c' \times c, c' \times k)$

Additionally, we can also compare two pairs because we know the value of δ. We know that:

- $(c_1, k_1) \leq (c_2, k_2) \equiv (c_1 < c_2) \vee (c_1 = c_2 \wedge k_1 \leq k_2)$

After mounting a graph with the entire set of difference logic constraints we can observe many interesting facts. The following subsections will describe the interesting ones for us.

3.2.1 Dependency

If two variables x and y don not depend on each other, there will not be a path from x to y neither a path from y to x. In graph theory, a path from x to y is a set of edges that connect x to y.

A variable x may depend on another variable y either directly or indirectly. In the first case, there will be an edge from x to y and thus a path from x to y. In the second case, there will be a path from x to y that represents a combination of the constraints in the path and the length (or sum of the edges cost) indicates the relationship between the variables.

For instance, take two constraints $x - y \leq -1$ and $y - z \leq -2$. The combination of them ($x - z \leq -3$) show the indirect dependency between x and z. The graph representation for this example can be seen in Figure 3.2.

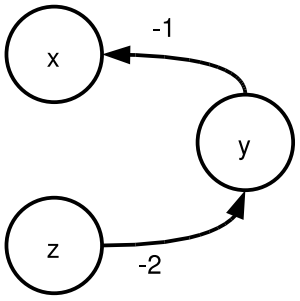

Figure 3.2: Direct ($x - y \leq -1$ and $y - z \leq -2$) and indirect ($x - z \leq -3$) dependency of variables.

3.2.2 Strongest Constraint

The *Strongest Constraint* is related to a pair of variables. As the name suggests, it is the strongest constraint that can be created (or extracted) from a set of constraints, related to a pair of variables. If we find that

$y - x \leq c_1$ is the strongest constraint related to y and x, that means that we cannot extract any other constraint $y - x \leq c_2$ where $c_2 < c_1$.

If there is a path from x to y, the shortest path len (the shortest path in our case is related to the sum of the edges cost and not to the number of the edges) from x to y gives the strongest constraint, related to x and y ($y - x \leq len$), that it is possible to create from the given facts. It means that y must be at most $x + len$.

For instance, the following set of constraints are represented by the graph in Figure 3.3: $y - x \leq -1$, $z - x \leq -2$, $w - y \leq -3$, $w - z \leq 0$ and $y - z \leq 0$. The strongest constraint that we can build between x and w is $w - x \leq -5$, built from the shortest path from x to w (that goes through z and y).

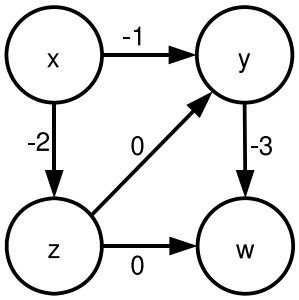

Figure 3.3: The strongest constraint related to two variables (vertices) can be build from the shortest path between them. In the figure, we have three strongest constraints ($w - x \leq -5$, $w - z \leq -3$ and $y - x \leq -2$) that are derived from the shortest path and are not given in the set of original constraints.

3.2.3 Unsatisfiability

A set of constraints is unsatisfiable if there is a contradiction, e.g., $x - y \leq -1 \wedge x - y \geq 2$. If there is no combination of constraints that can make the problem unsatisfiable, the problem is satisfiable.

Theorem 1 *There is a negative cycle in the graph if and only if the problem is unsatisfiable.*

Proof. If there is a negative cycle in the graph it means that there are two vertices x and y in the cycle where the shortest path from x to y costs $-\infty$ and the shortest path from y to x is also $-\infty$. That is possible because since the shortest path is related to the length (sum of the weights), and not the number of edges, we can take the same edge many times. So, we have $x - y \leq -\infty$ and $y - x \leq -\infty$, that implies: $x - y \leq -\infty$ and $x - y \geq \infty$, and thus a contradiction.

For the other half, if the problem containing only difference constraints is unsatisfiable it is because we have something such as $x - y \leq c_1 \wedge x - y \geq c_2$, where $c_1 < c_2$. The first literal implies that we have a path from y to x with length c_1 and the second one (can be changed to $y - x \leq -c2$ implies that we have a path from x to y with length $-c_2$. So, we have a cycle (x, y, x) with length $c1 + (-c2)$. We know that $c_1 < c_2 \Rightarrow c_1 - c2 < 0$. Therefore the length of the cycle is negative.

After looking for negative cycles, we do not need to check for satisfiability. In case no negative cycle is detected the problem is satisfiable.

Figure 3.4 shows a scenario where there is a negative cycle. The proof for the unsatisfiability (or conflict set) can be constructed from the edges in the negative cycle, the combination of these constraints will lead to a contradiction like $x - y \leq -\infty$ and $x - y \geq \infty$.

3.2.4 Equality Between Variables

Theorem 2 *Two variables x and y are equal if and only if the shortest path from x to y costs 0 and the shortest path from y to x also costs 0.*

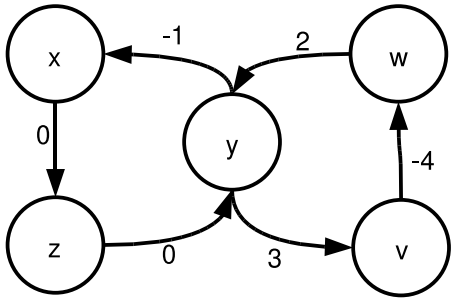

Figure 3.4: Graph with a negative cycle. The set of constraints representing it is unsatisfiable. The proof is the edges from the negative cycle: $z - x \leq 0$, $y - z \leq 0$ and $x - y \leq -1$.

Proof. That is easy to see, with the knowledge that we presented so far, we know that the constraints representing these two shortest paths are $x - y \leq 0$ and $y - x \leq 0$ (which is the same as $x - y \geq 0$). Thus, from $x - y \leq 0$ and $x - y \geq 0$ we have $x - y = 0$ (or $x = y$).

For the other half of the theorem, if we have two variables that are equal, x and y, we have: $x = y \Rightarrow x - y = 0 \Rightarrow x - y \leq 0 \wedge x - y \geq 0 \Rightarrow x - y \leq 0 \wedge y - x \leq 0$. These last two represents the shortest paths with length 0. They cannot be stronger because otherwise the problem would be unsatisfiable.

Figure 3.5 shows an example where an equality is found using shortest path information.

$x - y \leq 0 \wedge y - x \leq 0 \Rightarrow x = y$ is exactly the way back from understanding a constraint like $x = y$ showed in Table 3.1. The difference is that this information might be gotten indirectly and not as given constraints.

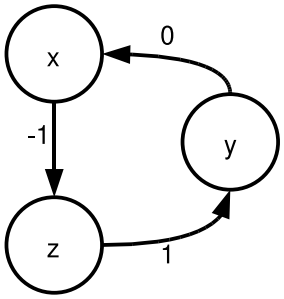

Figure 3.5: Graph made from 3 constraints: $z - x \leq -1$, $y - z \leq 1$ and $x - y \leq 0$. An equality can be found in this graph: x = y.

3.3 Conclusion

We presented difference logic and its graph model. We showed some interesting properties that allow us to present, in Chapter 4, algorithms for the decision procedure. We know, along other things, that we can check for satisfiability using algorithms for negative cycle detection. Additionally, the theorem of equality between variables will give us enough knowledge to present an algorithm for finding equalities.

Chapter 4

Decision Procedure for Difference Logic

A decision procedure is a method to solve a decision problem, where the expected answer is *yes* or *no*. In our case, a decision procedure will decide if a set of formulas is satisfiable or not, i.e, if there exists an interpretation of variables, functions and predicates that makes true every formula in the set.

For a decision procedure to work well in a Nelson and Oppen combination framework and produce some worthy results, it should take some requirements in consideration. In this chapter, we will explain these requirements and show how the algorithms for implementing them work.

The work here was implemented in the theorem prover haRVey [8, 11]. Everything in the following sections is related to this implementation.

4.1 Requirements

A few things are necessary to the decision procedure beyond returning the status of satisfiability. They are requirements for the correct integration with

the Nelson and Oppen combination framework in a theorem prover.

There are a few basic requirements that are necessary. The decision procedure needs to be correct, i.e., it should always return the correct status of satisfiability. It should terminate, i.e., it cannot run for an undeterminable amount of time.

The other requirements will be presented in the following subsections. The desirable requirements are conflict set generation, premisses, incrementability. They can be considered accessory because the prover can still give the correct status of satisfiability without them, but they are useful for improving efficiency. The mandatory requirement is equality generation, that is fundamental for the Nelson and Oppen combination framework.

4.1.1 Conflict Set Generation

Given a set of constraints, if the decision procedure detects that the problem is unsatisfiable then a proof is necessary. We call proof any subset of the constraints that is unsatisfiable. It can optionally contain the explanation of how rules were applied to get the status. It is useful for the user to understand if there is something wrong, so he/she can reformulate the problem when necessary. It also can be used to conclude that the result is as expected.

The proof may also be useful when used by the theorem prover to reduce the time of execution. If it is detected that an already proved unsatisfiable subset of constraints is in the current problem the prover can stop running the problem, because it is known that it is unsatisfiable. This can be realized by the SAT solver by adding a so called conflict clause, corresponding to the negation of the literals found in the proof.

As a simple example, given the set of constraints:

$$x - y \leq 0; y - z \leq 1; z - x \leq -1; y - x \leq -1$$

The decision procedure should be able to detect that the problem is unsatisfiable due to:

$$x - y \leq 0; y - x \leq -1$$

4.1.2 Equality Generation

One of the main assumptions of the Nelson and Oppen combination framework is the equality propagation between decision procedures. Every decision procedure in the framework should be able to detect equalities, so these equalities may be propagated to the other decision procedures. That will allow the theorem prover to infer about combined theories.

Without the equality propagation, the isolated decision procedures may not have enough information to find satisfiability of some combined theory. For example, given the formula:

$$x - y \leq 0 \wedge y - z \leq 1 \wedge z - x \leq -1 \wedge x - w \leq -1 \wedge f(x) \neq f(y)$$

The Nelson and Oppen scheme could split the formula and send constraints to two decision procedures, one to deal with the arithmetic theory and another one to deal with uninterpreted functions. A possible scenario is shown in Table 4.1.

The Table 4.1 shows that, in a first moment, working alone, the decision procedures cannot detect the unsatisfiability. However, an equality is detected and propagated to the other decision procedure. This new information allows the decision procedure for uninterpreted function to detect the unsatisfiability.

level	**DP for arithmetic**	**DP for uninterpreted functions**
0	$x - y \leq 0$ $y - z \leq 1$ $z - x \leq -1$ $x - w \leq -1$ $x = y$ *(detected)*	$\underline{f(x) \neq f(y)}$
1		$\underline{x = y \text{ } (new)}$ $\underline{unsatisfiable}$

Table 4.1: Example of equality generation.

Integer Case

For non-convex theories, such as integer difference logic, it is also necessary to generate disjunction of equalities (see Nelson and Oppen [14]). To illustrate the necessity, consider the following set of constraints as example, where the variables can only have integer values.

$$\neg(x - y = 1)$$
$$\neg(x - y = 2)$$
$$x - y \geq 1$$
$$x - y \leq 2$$

The first two constraints are not handled by the decision procedure for DL because they are not part of the theory. They will be handled by another DP. The decision procedure for DL should be able to generate $x-y = 1 \vee x-y = 2$. If such disjunction is not generated, the contradiction $(\neg(x - y = 1) \wedge \neg(x - y = 2) \wedge (x - y = 1 \vee x - y = 2))$ will not be possible to be found by the other DP.

4.1.3 Premisses

When unsatisfiability is detected, the proof may contain constraints that are not in the original problem. That is because a decision procedure may have constraints generated internally by the theorem prover, like, for example, the detected and propagated equalities.

For constructing a consistent proof, the original constraints are necessary. It is possible to track them by knowing the premisses of a constraint. The premisses of a constraint are the constraints used to generate it.

For example, from the following formula, we extract the information presented in Table 4.2:

$$x - y \leq 0 \wedge y - z \leq 1 \wedge z - x \leq 0 \wedge y - z \leq 0 \wedge f(x) \neq f(y) \wedge G(x, y) = f(z)$$

level	**DP for arithmetic**	**DP for UF**
0	$x - y \leq 0$ $y - z \leq 1$ $z - x \leq 0$ $y - x \leq 0$ $x = y$ *(detected; premisses* $= \{x - y \leq 0, y - x \leq 0\}$*)*	$\underline{f(x) \neq f(y)}$ $g(x, y) = f(z)$
1		$\underline{x = y \text{ } (new)}$ $\underline{unsatisfiable}$

Table 4.2: Example of premisses.

In the decision procedure for uninterpreted functions (U.F.), the problem is detected to be unsatisfiable due to $f(x) \neq f(y)$ and $x = y$. But the information $x = y$ is not present in the original formula, so giving it in the proof might not be interesting.

The decision procedure that generates the new constraint should also be able to return its premisses when asked. Therefore, in the previous example,

when the unsatisfiability is detected using premisses information, we can construct as proof:

$$x - y \leq 0 \wedge y - x \leq 0 \wedge f(x) \neq f(y)$$

4.1.4 Incrementability

When testing a set of constraints, a decision procedure may detect that a subset of the constraints is unsatisfiable. In that case, it is not necessary to continue executing the decision procedure since the entire problem is also unsatisfiable.

Therefore, a decision procedure that works incrementally (checking the satisfiability each time a new constraint is received) may be more efficient. But, for that, a good incremental algorithm is necessary. It should remember the previous status and when a new constraint comes, it should use previous calculated information to reduce the computation of the new check.

Moreover, even after testing the entire set of constraints originally assigned by the Nelson and Oppen scheme, new constraints may come. These new constraints, such as the variable equalities, produced by other decision procedures, have to be incorporated and a new check for satisfiability is necessary. The set of constraints to test is the same, except for the new one. For a large problem, this might happen a considerable number of times. So, using an incremental algorithm for checking the consistency of the new set is also very desirable for this situation.

Another importance of using incrementability comes when testing formulas leads to several checking paths. When disjunction is present in a formula, a SAT-solver, using techniques such as DPLL [9, 10], may have to test repeatedly some sets of constraints to verify the satisfiability of the original formula. Each of these sets has some common subset of constraints. Smart

techniques will keep track of paths and occasionally, when testing another path is necessary, backtrack to a common point in the path and do not redo subproblems over and over again.

As a consequence of having to come back in the path, i.e., remove some constraints from the set and later add others, the incremental algorithm also should be **backtrackable**. It should be able to remove the last constraint added and then reestablish the previous state efficiently, when the removed constraint was not in the set.

As an example of this situation and the necessity of having an incremental and backtrackable algorithm, consider the following formula:

$$(x-y \leq 0 \vee x-y \leq -1) \wedge (y-z \leq 0 \vee y-z \leq -1) \wedge (z-x \leq -1 \vee z-x \leq -2)$$

For testing the formula in CNF (Conjunctive Normal Form - a formula is in CNF if it is a conjunction of clauses, where a clause is a disjunction of literals), a SAT-solver will set values (true or false) to the literals to make each clause true. In this example, where all the literals are different from each other, in the worst case scenario it may be necessary to explore $2 \times 2 \times 2 (= 8)$ paths to check different sets of constraints. In Figure 4.1, there is a simulation of a scenario where a SAT-solver based on DPLL using a decision procedure for arithmetic checks the formula.

In an incremental and backtrackable algorithm, for every new constraint added, a satisfiability check is done. If there is a contradiction in the current set, it is not necessary to continue and a backtrack to a common point is done. From this point, the process continues and the SAT-solver resumes assigning appropriated values to the literal to find out if the original formula is satisfiable.

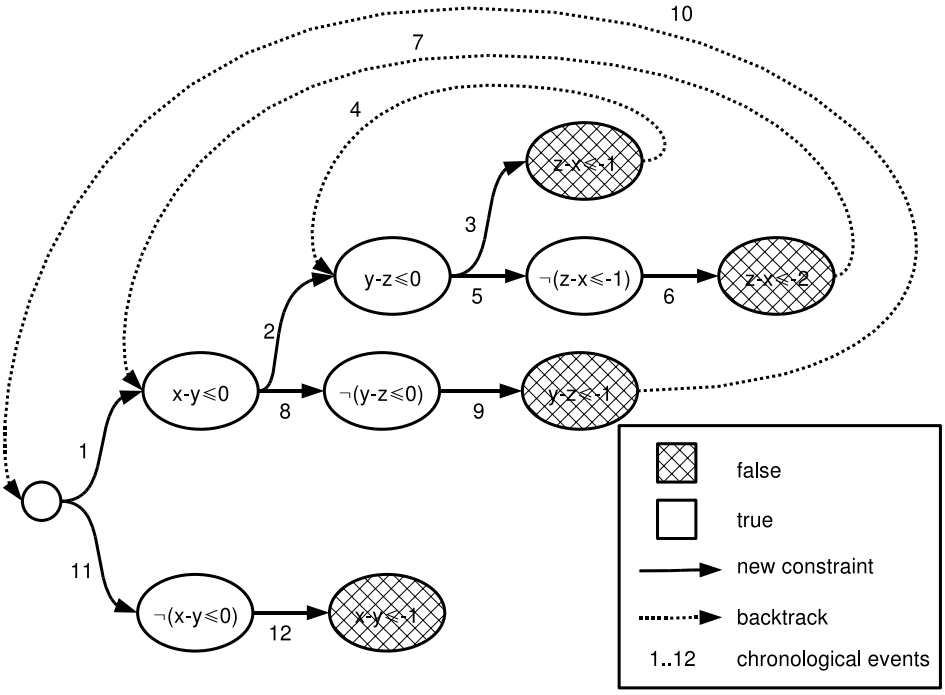

Figure 4.1: Paths to check (from the previous formula). In this case, it was necessary to take 4 paths to conclude that the formula is unsatisfiable. After events 9 and 12 contradictions are early found so there is no necessity to continue checking the branch.

4.2 Algorithms for the Difference Logic Decision Procedure

With these requirements in mind, the following subsections will explain how algorithms for a difference logic decision procedure work. Many different algorithms can be used for the same purpose. We decided to describe those that work well for our requirements.

4.2.1 Satisfiability Checking

As shown in the Section 3.2.3, a set of constraints is unsatisfiable if, and only if, the graph made from the constraints has a negative cycle. The classical algorithm to check if a graph has a negative cycle is the Bellman-Ford algorithm [3, 12]. Algorithm 2 shows its pseudocode. Although it does not match our requirements for incrementability, it is a good reference algorithm.

```
   input  : G = (V, E) : Graph, source : Vertex
   output: hasNegativeCycle : Boolean
   // Initialize graph
 1 foreach vertex v in V do
 2 |   v.distance = INFINITY;
 3 |   v.predecessor = NULL;
 4 end
 5 V[source].distance = 0;
   // Relax edges repeatedly
 6 for i = 1 to |V| do
 7 |   foreach edge e in E do
 8 |   |   u = e.source;
 9 |   |   v = e.destination;
10 |   |   if v.distance > u.distance + e.weight then
11 |   |   |   v.distance = u.distance + e.weight;
12 |   |   |   v.predecessor = u;
   |   |   |   // Check for negative-weight cycles
13 |   |   |   if i == |V| then
14 |   |   |   |   return hasNegativeCycle = true;
15 |   |   |   end
16 |   |   end
17 |   end
18 end
19 return hasNegativeCycle = false;
```

Algorithm 2: BellmanFord

The Bellman-Ford algorithm computes single-source shortest path in a

graph and also looks for negative cycles. It is a non incremental algorithm and runs in $O(VE)$, where V is the number of vertices and E is the number of edges.

The idea of Bellman-Ford algorithm is based on relaxing the estimated distance between the *source* and all the other vertices. After the cycle i from the *For* loop, $v.distance$ will contain the shortest path from *source* to v using at most i edges. After the $|V| - 1$ cycle, the distances from *source* to the vertices should be the shortest path. The exception is when there is a negative cycle reachable from *source*. In such cases, the shortest path can always be reduced by going through the cycle an infinite number of times.

4.2.2 Incremental Satisfiability Checking

To be worth using, an incremental algorithm should be more efficient than a static (non incremental) one. In this subsection we present an incremental satisfiability check algorithm developed by us, Algorithm 3. It can be implemented using a heap with complexity $O(V + E'logV')$, where V is the number of vertices, V' is the number of vertices that have their distance changed in the algorithm and E' is the number of outgoing edges of the V' vertices.

The idea of the algorithm is to search in the graph for the vertices that may have their distances changed due to the last added edge. The distance is the length of the shortest path from an arbitrary source vertex. For this purpose and for simplicity, an *artificial vertex* can be created to be the source vertex. It will connect all the other vertices with an unidirectional edge with weight 0. The original system will be unsatisfiable if, and only if, the slightly modified one (with the artificial vertex) is.

The algorithm starts by checking if the new edge will improve the distance to the destination vertex. If it does, the search starts. The search is done by

```
   input  : G = (V, E) : Graph, e : Edge
   output: hasNegativeCycle : Boolean

   // Initialize Δ
 1 foreach vertex v in V do
 2 |  Δ(v) = 0;
 3 end

   // Initial check
 4 u = e.source, v = e.destination;
 5 Δ[v] = (u.distance + e.weight) − v.distance;
 6 if Δ[v] < 0 then
 7 |  Q = NewHeap() ;
 8 |  Q.InsertOrImprove(v, Δ(v), u);

   |  // Improving Search
 9 |  while Q.Empty()== false do
10 |  |  v, delta, u = Q.RemoveMin();
11 |  |  v.distance = v.distance + Δ[v];
12 |  |  v.predecessor = u;
13 |  |  foreach outgoing edge i of v do
14 |  |  |  t = i.destination;
15 |  |  |  if (v.distance + i.weight) − t.distance < Δ[t] then
16 |  |  |  |  if t == e.source then
17 |  |  |  |  |  return hasNegativeCycle = true;
18 |  |  |  |  else
19 |  |  |  |  |  Q.InsertOrImprove(t, Δ[t], v);
20 |  |  |  |  end
21 |  |  |  end
22 |  |  end
23 |  end
24 end
25 return hasNegativeCycle = false;
```

Algorithm 3: IncrementalNegativeCycleDetection

greedily picking the vertex in the heap that will have its distance improved the most. The improvement is the difference between the new and old value of distance. It is assumed that, when a vertex v is improved, its neighbors will have their improvement by at most the same as v. So, when a greedily picked

vertex has its distance improved, it is done by the path that will mostly improve the distance. Therefore, each vertex should have their distance improved at most once. The exception is when there is a negative cycle.

A negative cycle happens if and only if the *e.source* distance improves. Because if *e.source* improves that means that *e.destination* will improve again, so a cycle is present.

Figure 4.2 shows an example of an arbitrary set of constraints. The graph representation is shown with the distance information of each variable in the squares. A new constraint is added, $a - f \leq -2$. From that, we show the behavior of the Algorithm 3 in the Table 4.3.

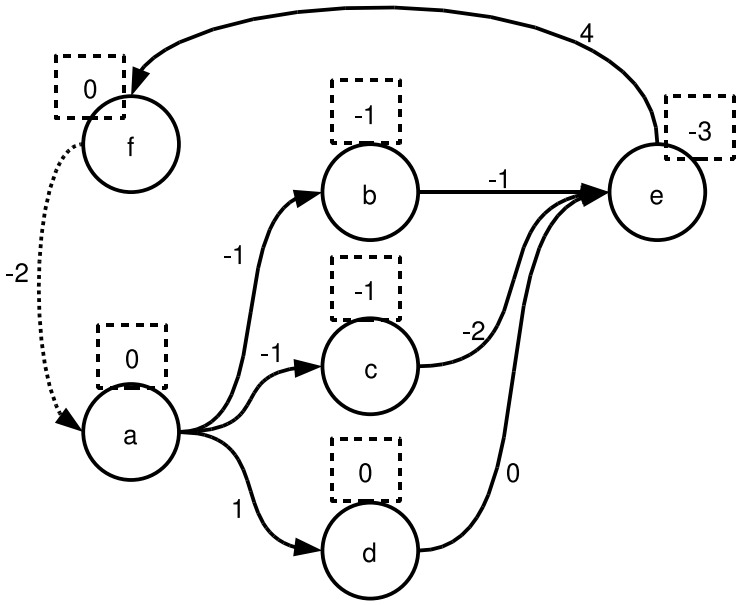

Figure 4.2: Graph representation of an arbitrary example, state before constraint insertion.

At each step, the chosen vertex from the heap will have its distance improved (only once) by the shortest path. When the source vertex f is reached for improvement (time 4), a negative cycle is detected. Figure 4.3

shows the final state of this example.

Time	Current vertex	Heap (vertex, improvement)
0	f	$(a, -2)$
1	a	$(b, -2)$ $(c, -2)$ $(d, -1)$
2	b	$(c, -2)$ $(d, -1)$ $(e, -1)$
3	c	$(e, -2)$ $(d, -1)$
4	e	$(d, -1)$ $(f, -1)$ // *negative cycle*

Table 4.3: Behavior of the Algorithm 3 running on Figure 4.2.

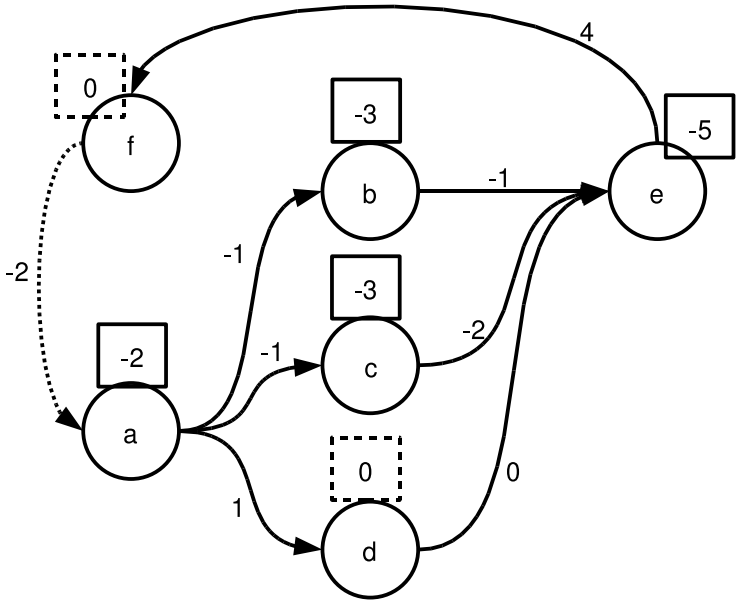

Figure 4.3: Graph representation of an arbitrary example, state when a negative cycle is detected.

4.2.3 Conflict Set Construction

When unsatisfiability is detected, a proof is necessary. The proof is the set of constraints that generated a conflict (the conflict set). A minimal conflict set can be obtained by the constraints that form the negative cycle.

The algorithm for checking the unsatisfiability keeps the predecessor of the vertices while updating the shortest path distances and doing the search for a negative cycle. So, if a negative cycle is detected, it is only necessary to go through the cycle, by using the predecessor, and collect the constraints associated to the edges. A pseudo-code for this is showed in Algorithm 4.

```
   input  : G = (V, E) : Graph, e : Edge
   output: s : Set of constraints
1  v = e.source;
2  s.Insert(e.constraints);
3  while v ≠ e.destination do
4  |   s.Insert(Edge(v.predecessor, v).constraints);
5  |   v = v.predecessor;
6  end
7  return s;
```

Algorithm 4: ConflictSet

When a new edge is added, more than one negative cycle may arise. See for instance, Figure 4.4. In this case, it is possible to return more than one minimal conflict set. But the algorithm will return the first negative cycle found.

4.2.4 Equality Generation

It was shown in Section 3.2.4 that, in our graph model, two variables x and y are equal if, and only if, the lengths of the shortest path between x and y and between y and x are 0.

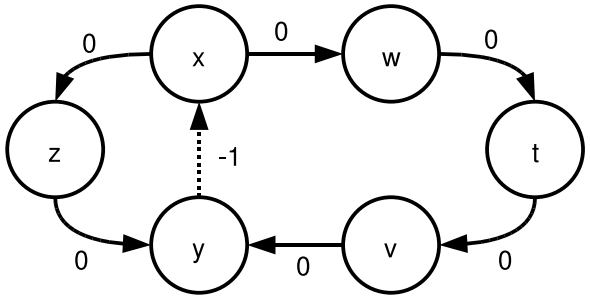

Figure 4.4: In this graph, representing a set of constraints, two negative cycles are found after the edge $(y, x, -1)$ is added. The cycles are $\{y, x, z, y\}$ and $\{y, x, w, t, v, y\}$. Any of them can be used to construct the conflict set.

Algorithm 5 (inspired by [13]) shows how to generate equalities without explicitly calculating the shortest path between each pair of vertices in the graph. It assumes that no negative cycle is present.

It is not an incremental algorithm, because it redoes every calculation each time it is called, i.e., each time a new constraint is added. It can be implemented with complexity $O(V * logV + E)$, where E is the number of edges in G and V is the number of vertices in G.

The algorithm starts by mounting a graph G' from the edges where *slack* is equal to zero. An edge e has *slack(e)* zero when e is part of a shortest path between the *artificial* variable and the destination of e ($e.destination$). We define *slack(e)* by:

$$slack(e) = e.source.distance - e.destination.distance + e.weight$$

Any cycle $C = [v_1, v_2, ..., v_n]$ in G' will have its length equal to zero. To see that, let us first make some definitions:

- $v_1..v_n$ are vertices
- v_1 is equal to v_n

```
   input  : G = (V, E) : Graph
   output: s : Set of equalities
   // Mount graph G' from E' and its vertices
 1 foreach edge e from E do
 2 |   if e.source.distance - e.source.distance + e.weight == 0 then
 3 |   |   E'.Insert (e);
 4 |   end
 5 end
 6 G' = Graph (E');
   // Look for equalities in each SCC
 7 SCCs = G'.FindStronglyConnectedCompenents();
 8 foreach scc in SCCs do
 9 |   sort vertices of scc by distance;
10 |   for i = 1 to |scc| - 1 do
11 |   |   if v_i.distance == v_{i+1}.distance then
12 |   |   |   eq = Equality(v_i, v_{i+1});
13 |   |   |   if NotGeneratedYet(eq) then
14 |   |   |   |   s.Insert (eq);
15 |   |   |   |   eq.SetPremisses(FindPremisses(G', v_i, v_{i+1}));
16 |   |   |   end
17 |   |   end
18 |   end
19 end
20 return s;
```

Algorithm 5: GenerateEqualities

- $w(v_i, v_{i+1})$ is the weight of the edge between v_i and v_{i+1}. It is a shortcut for $Edge(v_i, v_{i+1}).weight$

- $d(v_i)$ is the length of the shortest path between a vertex (the *artificial* one) and v_i. It is a shortcut for $v_i.distance$

The length of a cycle C is the sum of the edge weights in the cycle. More formally we have:

$$Length(C) = w(v_1, v_2) + w(v_2, v_3) + ... + w(v_{n-1}, v_1);$$

Adding the terms $+d(v_i)$ and $-d(v_i)$ will not change the result of the length, because they cancel each other. So, doing this we have:

$$Length(C) = d(v_1) - d(v_1) + d(v_2) - d(v_2) + ... + d(v_{n-1}) - d(v_{n-1}) + w(v_1, v_2) + w(v_2, v_3) + ... + w(v_{n-1}, v_1);$$

Now, we just rearrange the terms to see more precisely the next step. It will be like this:

$$Length(C) = [d(v_1) - d(v_2) + w(v_1, v_2)] + [d(v_2) - d(v_3) + w(v_2, v_3)] + ... + [d(v_{n-1}) - d(v_1) + w(v_{n-1}, v_1)];$$

We can see that $[d(v_i) - d(v_{i+1}) + w(v_i, v_{i+1})]$ is exactly the *slack* defined before. We know that in G', all the *slacks* are 0 because of the way G' was mounted. So evaluating the expression we have:

$$Length(C) = 0$$

The next step in the algorithm is to get the Strongly Connected Components (SCCs). We call a graph a strongly connected component (SCC) if for every pair of vertices u and v there is a path from u to v and also from v to u. The SCCs of a graph are the maximal strongly connected subgraphs of the graph.

Any two vertices in a SCC of G' will be in a cycle of length zero and the path between them is also the shortest path in the original graph G. That is the primary condition for two variables to be equal. They need to be in a cycle of length zero. Based on that, every pair of vertices in each SCC are potential candidates to be equal.

For checking if two variables v_1 and v_n are equal without having to calculate the shortest path between every pair of variables, we can use the

distance information that we keep when looking for negative cycles. The shortest paths between them have to be zero, so mathematically we have:

$$0 = w(v_1, v_2) + w(v_2, v_3) + ... + w(v_{n-1}, v_n);$$
$$0 = w(v_n, v_{n-1}) + w(v_{n-1}, v_{n-2}) + ... + w(v_2, v_1);$$

We know by the definition of *slack* that: $slack(v_i, v_{i+1}) = d(v_i) - d(v_{i+1}) + w(v_i, v_{i+1})$. Knowing that the *slacks* in G' are zero, we have: $w(v_i, v_{i+1}) = -d(v_i) + d(v_{i+1})$. Thus, developing the previous expressions:

$$0 = -d(v_1) + d(v_2) - d(v_2) + ... + d(v_{n-1}) - d(v_{n-1}) + d(v_n);$$
$$0 = -d(v_n) + d(v_{n-1}) - d(v_{n-1}) + ... + d(v_2) - d(v_2) + d(v_1);$$

Simplifying everything:

$$0 = -d(v_1) + d(v_n);$$
$$0 = -d(v_n) + d(v_1);$$

Therefore, in any SCC of G', two variables v_1 and v_n are equal if they have the same distance value, i.e., $d(v_1) = d(v_n)$.

Summarizing, Algorithm 5 will look in each SCC of G' for every pair of vertices. If they have the same distance, an equality is found. If this equality is not new, it is added to the set of new equalities generated to be returned. Additionally, the set of premisses is associated to it.

Algorithm 6 shows a pseudo-code for finding the strongly connected components (SCCs) of a graph. Its complexity is linear in the number of edges [20].

It runs a series of depth first searches (DFS) using a graph G starting from each not visited vertex. Later, another series of DFS is done, but now using the transpose set of edges E^T and choosing the vertex in decreasing

```
   input  : G = (V, E) : Graph
   output: s : Set of SCC
   // Mount G^T by creating new edges changing the source
      and destination of E
 1 foreach edge e in E do
 2 |  E^T.Insert(Edge(e.destination, e.source));
 3 end
 4 G^T = Graph(V, E^T);
   // First DFS
 5 foreach vertex v of V do
 6 |  if v.visited == false then
 7 |  |  v.visited = true;
 8 |  |  DFS(G, v);
 9 |  end
10 end
   // Second DFS, by decreasing finish time of the first
      DFS
11 Sort V by the decreasing time of v.timeFinished;
12 foreach vertex v of V do
13 |  if v.visited == false then
14 |  |  v.visited = true;
15 |  |  DFS(G^T, v);
16 |  |  s.Insert(each vertex visited now);
17 |  end
18 end
19 return s;
```

Algorithm 6: FindStronglyConnectedComponents

order of "finished visit" time of the first DFS. In this second series of DFS, all the vertices reachable on each DFS belong to the same SCC.

Algorithm 7 shows how a depth-first search (DFS) works. The DFS is a search algorithm that explores each branch as deep as possible before backtracking, see, e.g., [6]. Starting in a vertex v, it will mark as visited every vertex reachable from v. Additionally, when a vertex u cannot continue the search, i.e., all its neighbors were already visited, u will have a "finished

visit" time set.

```
   input : G = (V, E) : Graph, v : Vertex
1  foreach outgoing edge e of v do
2  |  if e.destination.visited == false then
3  |  |  e.destination.visited = true;
4  |  |  DFS(G, e.destination);
5  |  end
6  end
7  v.timeFinished = NextTime();
```

Algorithm 7: DFS

4.2.5 Premisses Information

Given an equality between two variables, the premisses of this equality are the constraints that generated the equality. We know that two variables u and v are equal if the length of the shortest paths from u to v and from v to u are 0.

Then, for constructing the set of premisses, it is only necessary to extract the constraints related to each edge in the shortest path between u and v and between v and u. Algorithm 8 shows a pseudo-code for this.

```
   input : G = (V, E) : Graph, u : Vertex, v : Vertex
   output: s : Set of Constraints
   // Update the predecessor
1  BFS(G, u, v);
2  BFS(G, v, u);
   // Get the set of premisses
3  s = GetConstraints(G, u, v) + GetConstraints(G, v, u);
4  return s;
```

Algorithm 8: FindPremisses

It does two breadth-first searches (BFS). First starting from u and second

starting from v. That will save in G the predecessors corresponding to (one of) the shortest paths between u and v and between v and u. Later, this information is used to go through the paths and collects the constraints associated to each edge in the paths.

An equality can have more than one set of minimal premisses (see Figure 4.5 for an example). Using BFS to find the shortest path (number of edges) will allow to find the *minimum* set of premisses.

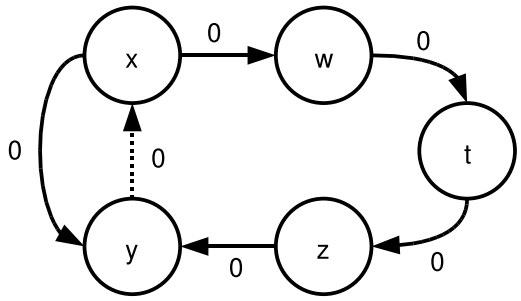

Figure 4.5: In this graph, representing a set of constraints, an equality can be found after the edge $(y, x, 0)$ is added. Two sets of premisses can be returned as a proof for $x = y$ (there are two shortest paths with length 0): $\{x{-}y \leq 0, y{-}x \leq 0\}$ and $\{x{-}y \leq 0, w{-}x \leq 0, t{-}w \leq 0, z{-}t \leq 0, y{-}z \leq 0\}$.

The BFS is a search algorithm that can be used to find the shortest path (related to the shortest number of edges), see, e.g., [6]. It executes the search by visiting all neighbors first and then visiting the neighbors of the neighbors and so on, until it finds the goal. For this, it uses a queue (FIFO - First In First Out). Algorithm 9 shows a pseudo-code. The goal is to find the destination vertex.

Algorithm 10 shows a pseudo-code for getting the constraints in a path. It is similar to the Algorithm 4 that gets the conflict set. It just follows the path using the predecessor information that was previously recorded and collect the constraints from the edges in the path.

```
   input : G = (V, E) : Graph, source : Vertex, destination : Vertex
 1 source.predecessor = NULL;
 2 Q.Add (source);
 3 while Q.Empty() == false do
 4 |   u = Q.Remove();
 5 |   foreach outgoing edge e of u do
 6 |   |   if e.destination.visited == false then
 7 |   |   |   e.destination.visited = true;
 8 |   |   |   e.destination.predecessor = u;
 9 |   |   |   if e.destination == destination then
10 |   |   |   |   return;
11 |   |   |   end
12 |   |   |   Q.Add(e.destination);
13 |   |   end
14 |   end
15 end
```

Algorithm 9: BFS

```
   input : G = (V, E) : Graph, source : Vertex, destination : Vertex
   output: s : Set of constraints
 1 while source ≠ destination do
 2 |   pred = destination.predecessor;
 3 |   s.Insert(Edge(pred, destination).constraints);
 4 |   destination = pred;
 5 end
 6 return s;
```

Algorithm 10: GetConstraints

4.3 Conclusion

This chapter described the requirements to build a decision procedure for difference logic and presented some algorithms for it. There are some desirable and necessary requirements that comes specially from Nelson and Oppen framework necessities and observations. For the algorithms, some known ones were presented, few adaptations were made, and a new incremental al-

gorithm for satisfiability check was described. Generating the disjunction of equalities for the integer case is still missing.

Chapter 5

Implementation details

The Chapter 4 explained what is necessary to write a decision procedure (DP) for difference logic (DL). An implementation integrated to the theorem prover haRVey [8, 11] was done following that specification. However, there are a few extra details that only comes when implementing the decision procedure.

This chapter will explain exactly what kinds of constraints the implementation handle and how they are handled. It also describes the minor differences presented in this implementation, as well as its behavior inside haRVey and some use cases.

5.1 Benchmark

Arithmetic is a vast theory. Even for the segment of difference logic, we have to define precisely what kinds of constraint we should handle taking in consideration the capacity of the decision procedure.

The primary goal is to handle the benchmarks from SMT-LIB [17], the Satisfiability Modulo Theories Library. Its major goal is to establish a li-

brary of benchmarks for satisfiability of formulas with respect to background theories for which specialized decision procedures exist. The constraints that appear in these benchmarks can be summarized in the Table 5.1.

The constraints are rewritten for simplicity and efficiency. We work with only one predicate ($\leq$) instead of four ($\leq, \geq, <, >$). That allows the SAT solver to infer about things like $a \leq b \wedge a > b$, that would be rewritten to $a \leq b \wedge \neg(a \leq b)$, and detect the contradiction without the necessity of calling the DP.

constraint	rewrite to
$x - y \leq c$	$x \leq y + c$
$x - y \geq c$	$y + c \leq x$
$x - y < c$	$\neg(y + c \leq x)$
$x - y > c$	$\neg(x \leq y + c)$
$x - y = c$	$x = y + c$
$x - y \neq c$	$\neg(x = y + c)$
$x \leq y$	$x \leq y$
$x \geq y$	$y \leq x$
$x < y$	$\neg(y \leq x)$
$x > y$	$\neg(x \leq y)$
$x = y$	$x = y$
$x \neq y$	$\neg(x = y)$

Table 5.1: Table of constraints and their rewrite

haRVey, using the DP for DL, should be able to give the correct answer to any set containing those constraints. The negation of equality constraints will not be directly received by the DP. That is because haRVey follows the combination framework of Nelson and Oppen, and it will filter the constraints and give only some of them to the DP. But another DP will handle it and the combination should work correctly.

5.2 Handling Constraints

Additionally to the benchmark, the DP will have to handle data created internally by haRVey. We call them clues. They are constraints or terms of a constraint. There are four kinds of clues that the DP for DL may receive:

abstract Any term of a constraint that has a meaning for difference logic, i.e., numbers and sums. Examples: $0; x + 2$.

predicate Any constraint that has the predicate $\leq$. Examples: $x \leq y + 1; \neg(z \leq x)$.

merge A equality between clues that will merge two class representants. Examples: $x = y; z = y + 2$.

replace The same as merge, but one of the clues are not known by the DP.

Every clue has a class representant that is also a clue. All the clues with the same class representant are equal. A clue c has its class representant changed when an equality is found between c and another clue by some DP.

Figure 5.1 shows a scheme for handling the clues received by the DP for DL. The idea is to keep a table that associate a number to a clue whenever there is a direct or indirect relationship between them. The second step is to update the graph and to use the algorithms presented in Chapter 4. Also, when we can infer about something, we save the premisses to be able to construct the proof later. This is omitted in the diagram for simplicity.

Only the top most symbols of the clues are checked. Everything else is treated as a variable. Therefore, if constraints from other theories are received, like for example, $car(cons(w, l)) \leq f(z) + 1$, they will be treated as if they were $X \leq Y + 1$, i.e, $car(cons(w, l))$ and $f(z)$ are treated as they were single variables.

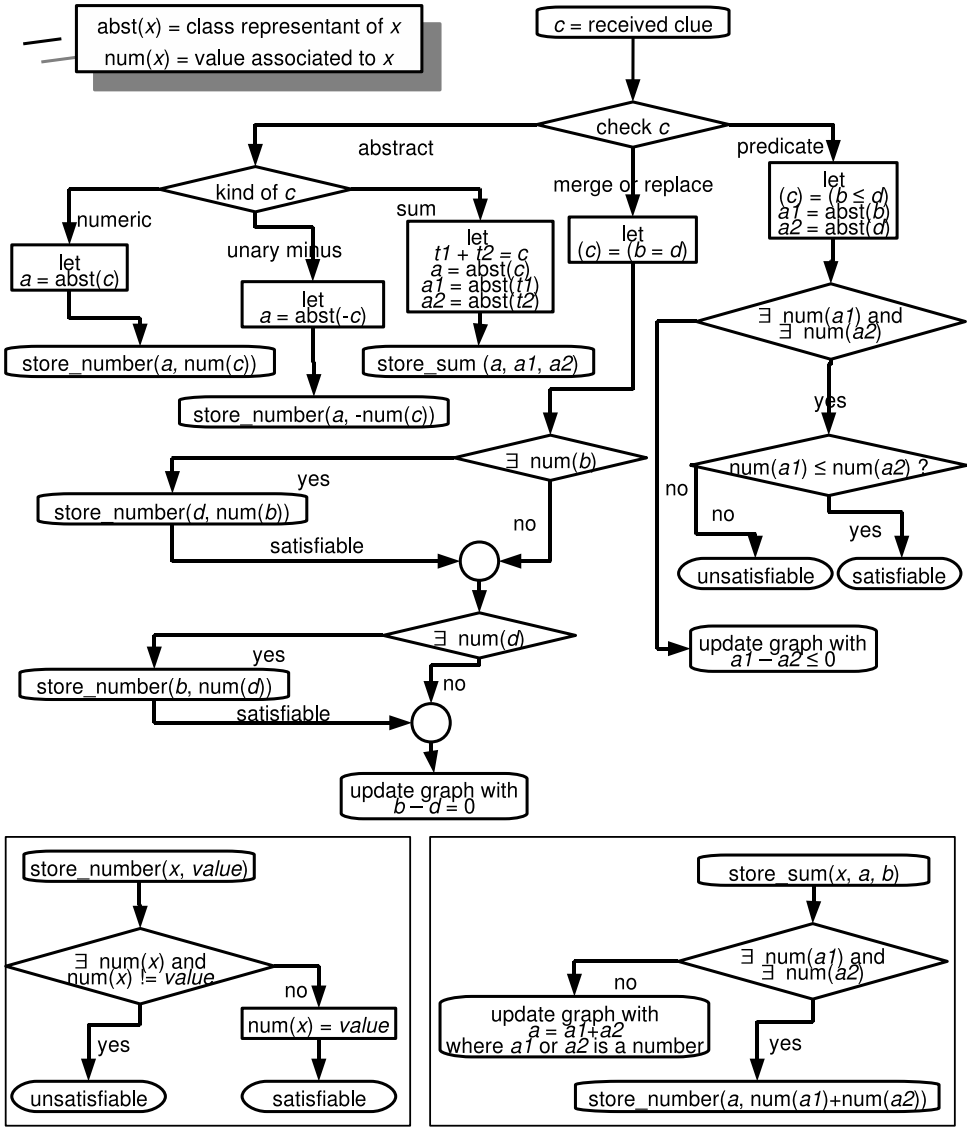

Figure 5.1: Decision diagram showing how to handle the clues.

5.3 Building the Graph

All the information received in the clues will be transmitted to the graph that will run the main algorithms. The numbers will be in the graph as the weights of the edges and the rest (the unknown) will be the vertices.

Let us take as example the set of constraints: $x \leq y + 1; x = y; y \leq x + (-1)$. From them, the DP for DL will receive a set of clues. They are shown in Table 5.2

time	clue	type
1	1	abstract
2	$y + 1$	abstract
3	$x \leq y + 1$	predicate
4	$x = y$	merge
5	-1	abstract
6	$x + (-1)$	abstract
7	$y \leq x + (-1)$	predicate

Table 5.2: Example of clues received by the DP for DL.

At each step, we follow the diagram in Figure 5.1. If the graph is changed, a satisfiability check is done. If the problem is detected unsatisfiable, we can stop the process.

At the first moment, when the clue "1" is received, the information will not go immediately to the graph. When the sum "$y + 1$" appears, the graph will start being built. If we follow the diagram in Figure 5.1, we will "update the graph with $a = a1 + a2$". a is the representant of $y + 1$, that is $y + 1$ itself, $a1$ is the representant of y, y itself, and $a2$ is the number 1. So the graph will be updated with something like, $(y + 1) - (y) = 1$, or, as we saw before, $(y + 1) - (y) \leq 1 \wedge (y) - (y + 1) \leq -1$. We can see the graph after receiving $y + 1$ in Figure 5.2.

Adding to the graph the information $(y+1) = (y)+(1)$, may look redundant, but it provides an easier way to generate the *important* equalities. If we have only two constraints like $a = b+1$ and $b = c+1$, it is not important to the other decision procedures to know that $a = c + 2$, that is because they cannot infer about arithmetic (sum in this case), so $c + 2$ is something

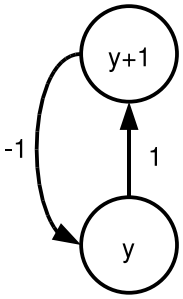

Figure 5.2: Graph after receiving the second clue $y + 1$.

unnecessary as it will not help. It is only important to generate equality between information that are already known by the others DPs.

Doing as described will add to the graph all the known (by the others DPs) information. So we can find the interesting equalities by just following Algorithm 5.

After receiving the third clue $x \leq y + 1$, following the diagram, we will "update graph with $a1 - a2 \leq 0$". $a1$ is the representant of x, that is x itself, and $a2$ is the representant of $y + 1$, that is $y + 1$ itself. So the graph will be updated with $(x) - (y + 1) \leq 0$. We can see the result in Figure 5.3.

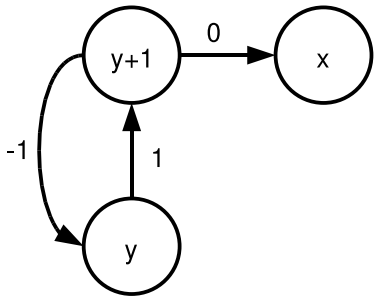

Figure 5.3: Graph after receiving the third clue $x \leq y + 1$.

The result is slighted different from the one if we had updated with $x - y \leq 1$. However, we have this information in the graph indirectly, just follow the

path between y and x. Also, by doing as described, we avoid looking deep in the term, simplifying the process.

The fourth clue $x = y$ is a merge. The representant of x will now be the same as the representant of y. So we add the information in the graph: $x - y \leq 0 \wedge y - x \leq 0$. We can see the result in Figure 5.4.

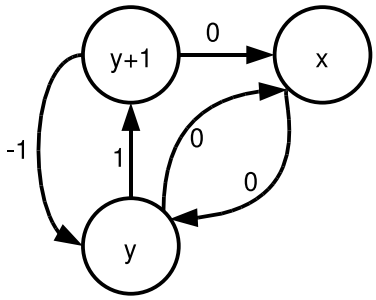

Figure 5.4: Graph after receiving the fourth clue $x = y$.

Let us assume that the representant of x changed to y, in the fourth step. The step 5 will just update the table that makes the relationship between clues and numbers.

The step 6 will be similar to the second one, the difference is that now the representant of x will point to another variable (y). We can see the result in Figure 5.5.

Finishing the example, in the step 7, we update the graph and detect a negative cycle. The final graph is shown in Figure 5.6.

5.4 Conclusion

In this chapter, we explained the details that make the implementation a bit different from theory. We described the benchmark and presented a new method to handle it. Its behavior was explained with an example showing

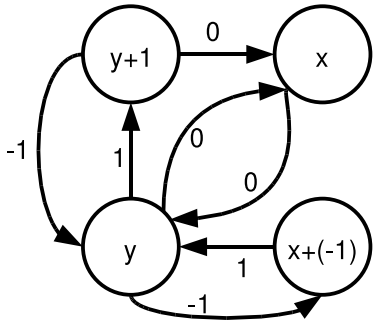

Figure 5.5: Graph after step 6.

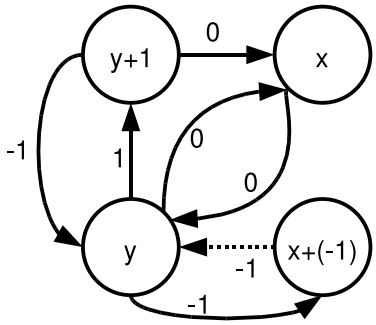

Figure 5.6: Final graph, a negative cycle is detected

the state of the graph at each step.

The automated differentiation between handling integer or real/rational constraints is still missing. They currently are done manually only.

Chapter 6

Experimental results

In this chapter, we show some experimental results of our proposed decision procedure for difference logic (DPDL) running by itself and running integrated in haRVey. We show how it scales up and compare the results with several other SMT-solvers.

For allowing to test the decision procedure itself, we created some benchmarks containing only conjunction of real difference logic constraints. The benchmark was randomly generated, variating the number of variables and constraints.

Two scenarios were created, one for testing sparse cases (where the number of variables and constraints variated independently of each other), and the second for testing dense cases (where the number of constraints were quadratic when compared to the number of variables). The numerical constant values in the formula were also random and variated from -100 to 100 in the sparse case and from -1 to 1000 in the dense case. For each (number of variable, number of constraints) configuration, ten files were generated. The time in the graphs show the total time of the ten runs. As the result of the random nature of the benchmarks, some of them were satisfiable and others

unsatisfiable.

The SMT-solvers used in our experiment were Mathsat, Barcelogic, CVC3, Z3 and Yices. We use their version available in the SMT-comp 2007 [1]. Additionally, we also tested CVCLite (cvcl).

We used 3 different configurations to test our decision procedure and haRVey. In the first one, no equality deduction is performed (it is not necessary for completeness in these benchmarks); in the second, equality deduction is executed only once, when all constraints were given; and the third configuration is when equality deduction is executed after each received constraint.

In Figure 6.1, we have the comparation of the first and second configuration for the DPDL. We can see that the difference in the running time (in miliseconds) is very small for such number of variables and constraints. A similar graph is show for comparing haRVey (Figure 6.2).

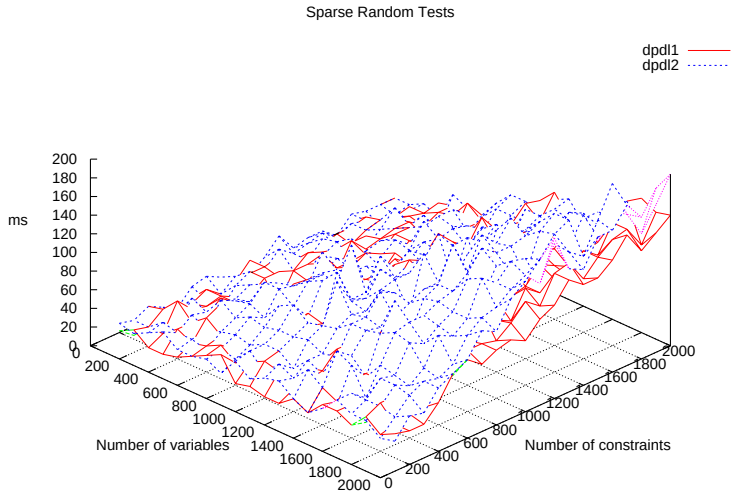

Figure 6.1: Decision procedure not propagating equalities (dpdl1) versus propagating equalities once (dpdl2)

When comparing first configuration and third configuration, the differ-

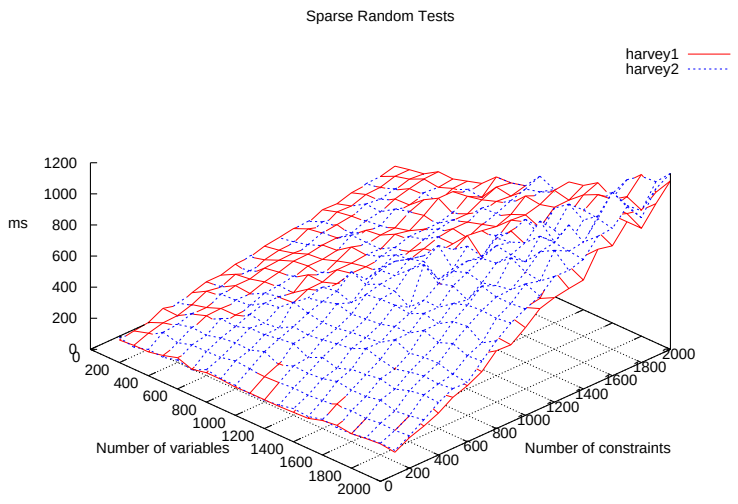

Figure 6.2: haRVey not propagating equalities (harvey1) versus propagating equalities once (harvey2)

ence is very high, see Figure 6.3. This shows that deducing equalities after each new added constraint is not good if a non incremental algorithm is been used. A similar graph is shown for comparing haRVey (Figure 6.4).

Figure 6.5 compares the running time for all the solvers and Figure 6.6 also compares the solvers but removes slower ones. DPDL and haRVey are using configuration 1 in these tests. DPDL shows a very good running time, outperforming all the solvers. The comparation is not the most fair because we compare different classes of solver (a decision procedure versus SMT-solvers), but the approximation is good enough and is the best way for providing a comparation.

When comparing the DPDL itself with haRVey (that is using the DPDL), we can see some significant difference in the running time. That is because the current implementation of haRVey does not take in consideration the nature of the theory, and instead of using only a difference logic solver, it

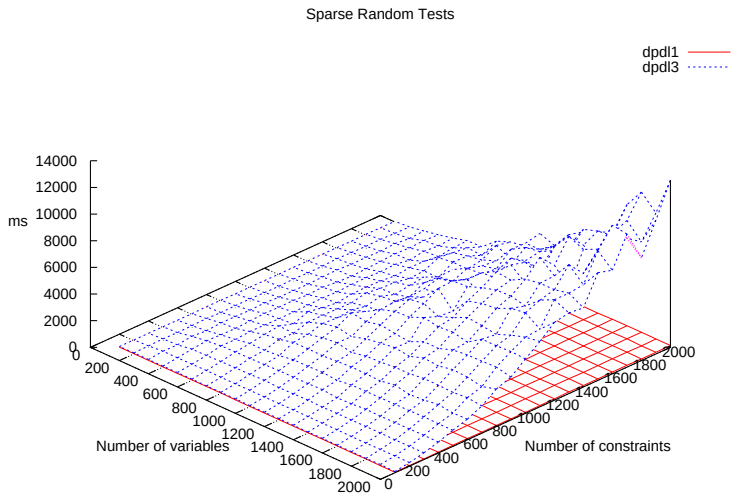

Figure 6.3: Decision procedure not propagating equalities (dpdl1) versus propagating equalities at each step (dpdl3)

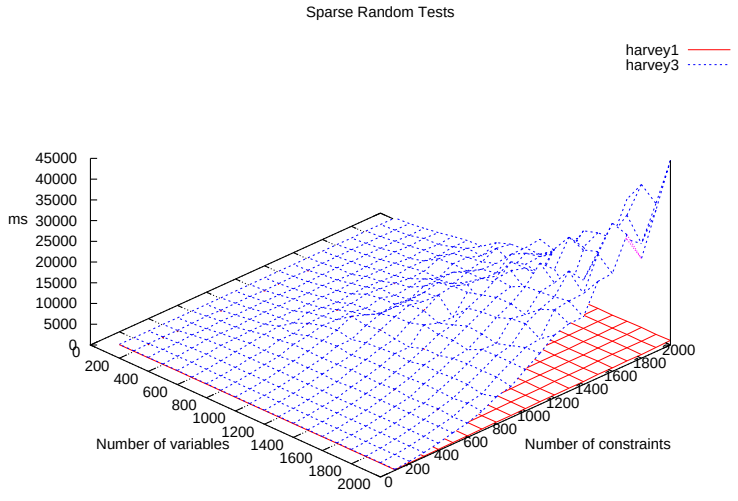

Figure 6.4: haRVey not propagating equalities (harvey1) versus propagating equalities at each step (harvey3)

also uses another decision procedure (for uninterpreted function), interfering in the process. Another important thing to mention is that the (Nelson and

Oppen) purification step in haRVey, creates more variables and constraints then necessary. This simplify the cooperation process, but it is less efficient.

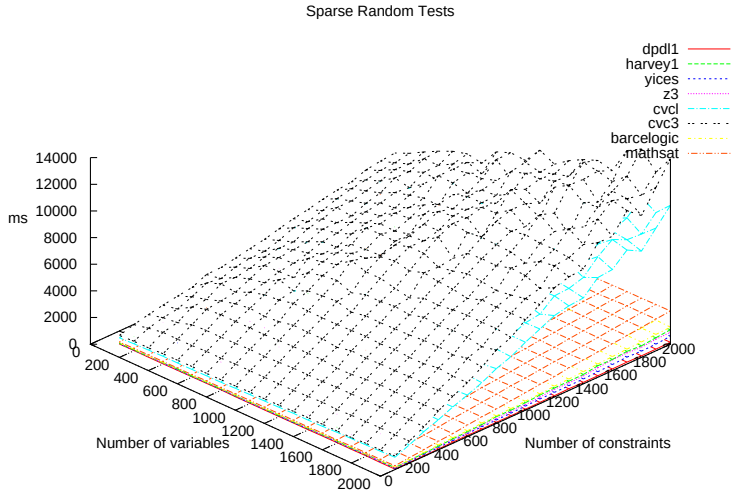

Figure 6.5: Solvers in the sparse case

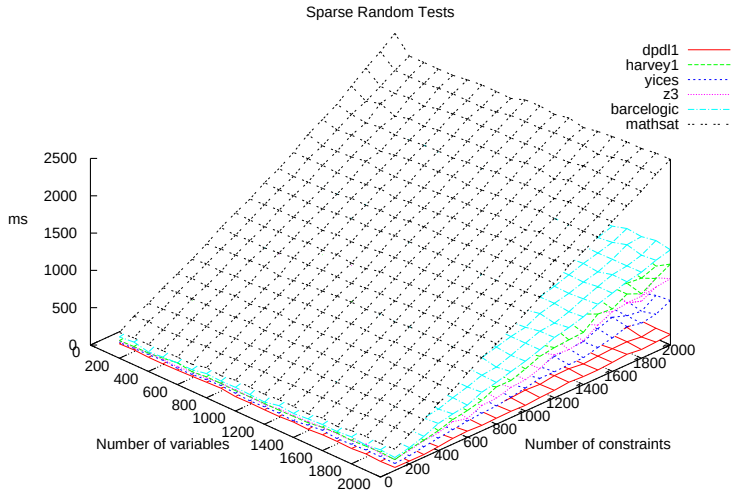

Figure 6.6: Solvers in the sparse case (2)

We can see the total time of all the solvers in the sparse case in Figure

6.7. The DPDL is about 3 times faster than the fastest SMT-solver (yices) and more than 50 times faster than CVC3.

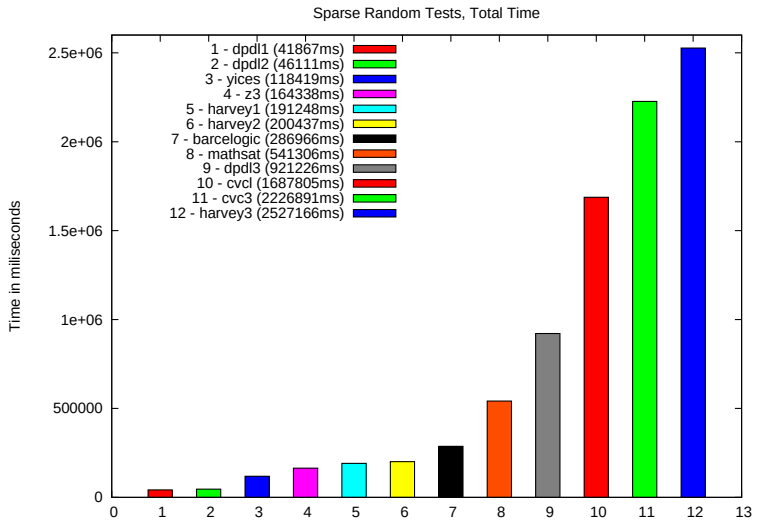

Figure 6.7: Total time of the solvers in the sparse case

For the dense case we used the fastest solvers in the sparse experiment. The graphs show the running time according to the number of variables $|V|$. For each of these cases the number of constraints $|C|$ is equal to $|V|^2$, so for instance, in the largest case (for 700 variables) the number of constraints is 490000. Figure 6.8 shows the result. Figure 6.9 is the same but removing Barcelogic.

In the dense case, the DPDL is even faster when comparing to the other solvers. It also scales quite well. We can see the total time of all the solvers in Figure 6.10. The DPDL is about 6 times faster than the fastest SMT-solver (yices) and more than 200 times faster than barcelogic.

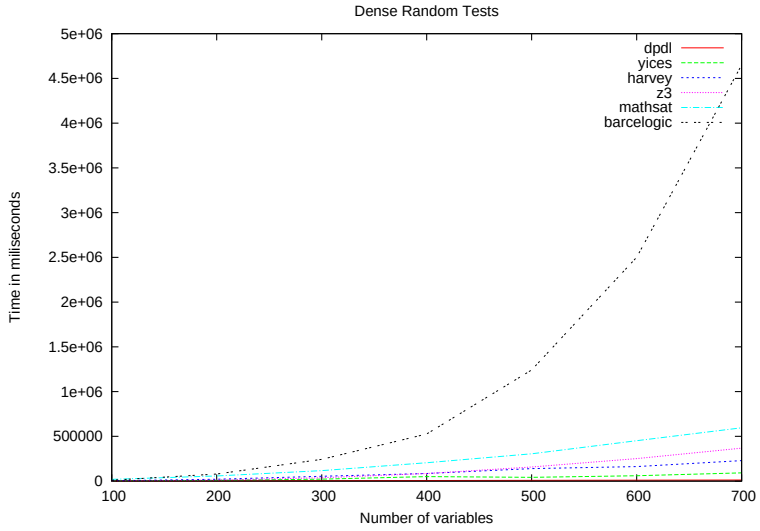

Figure 6.8: Solvers in the dense case

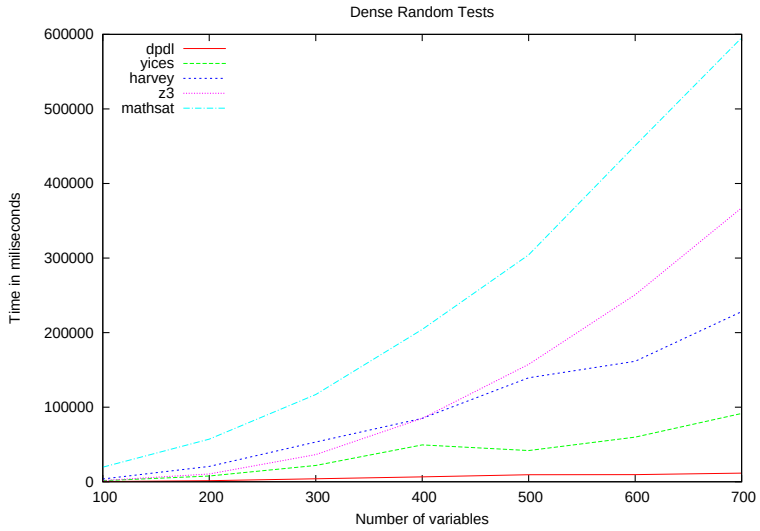

Figure 6.9: Solvers in the dense case (2)

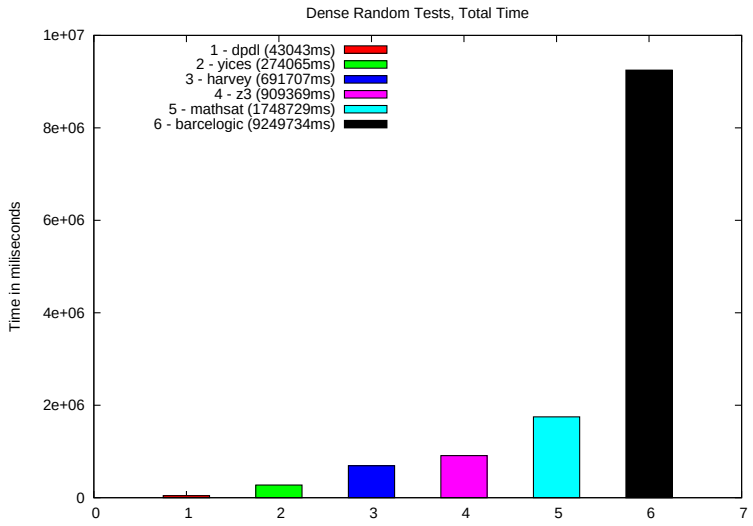

Figure 6.10: Total time of the solvers in the dense case

Chapter 7

Conclusion

This work describes a decision procedure integrated to the theorem prover haRVey that follows the Nelson and Oppen combination framework. It was presented the theory of difference logic; The Nelson and Oppen combination framework; DPLL for a Sat-solver; requirements for the decision procedure; the algorithms for the decision procedure; and the details of implementation.

Many related works can handle difference logic in different ways. Most of them have a linear arithmetic decision procedure, that can handle difference logic, but they suffer from exponential worst case complexity, see, e.g., [18, 2]. Some of them have a decision procedure for difference logic, but do not work in a combination framework. The decision procedure works direct with a Sat-Solver, see, e.g., [16, 7]. A decision procedure for rational difference logic in a Nelson and Oppen can be found in [13], but they focus only on rationals and none of their algorithms are incremental. Also, the way our approach handles the constraints is different.

The project is still under development. The module for difference logic is coded in C and has currently about 2200 lines. Its stage is unstable as many parts of haRVey are in constant evolution. Exhaustive testing is necessary.

Some points can be checked for speed improvement, such as incremental equality generation. There are also some aspects of haRVey, like backtrack of the Sat-Solver, that can be tested to see the impact in the speed. A good analysis of the effect of the messages exchanged may be also important. A way to generate the disjunction of equalities for integer difference logic still needs to be thought, as well as its consequence.

In the future, haRVey should handle linear arithmetic as well. A challenge will be make a good integration with the decision procedure for difference logic, so that the prover can take advantage of the speed of the decision procedure for difference logic.

Bibliography

[1] Satisfiability Modulo Theories Competition (SMT-COMP). `www.SMT-LIB.org`, 2007.

[2] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. A SAT-based approach for solving formulas over boolean and linear mathematical propositions. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 195–210. Springer, 2002.

[3] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[4] Clarke, Wing, and al. Formal methods: State of the art and future directions. *CSURV: Computing Surveys*, 28, 1996.

[5] Stephen Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[7] Scott Cotton, Eugene Asarin, Oded Maler, and Peter Niebert. Some progress in satisfiability checking for difference logic. In Yassine Lakhnech and Sergio Yovine, editors, *FORMATS/FTRTFT*, volume 3253 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 2004.

[8] Jean-Francois Couchot, Frédéric Dadeau, David Déharbe, Alain Giorgetti, and Silvio Ranise. Proving and debugging set-based specifications. *Electr. Notes Theor. Comput. Sci*, 95:189–208, 2004.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[10] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, April, 1960.

[11] David Déharbe and Pascal Fontaine. haRVey: combining reasoners. In *Sixth International Workshop on Automated Verification of Critical Systems*. HAL - CCSd - CNRS, 2006.

[12] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, 1962.

[13] Shuvendu K. Lahiri and Madanlal Musuvathi. An efficient nelson-oppen decision procedure for difference constraints over rationals. *Electr. Notes Theor. Comput. Sci*, 144(2):27–41, 2006.

[14] G. Nelson and Oppen D. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[15] P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, N. Jain, and O. Maler. Verification of timed automata via satisfiability checking. In *Proc. Formal Techniques in Real-Time and FaultTolerant Systems FTRTFT'02, volume 2469 of Lecture Notes in Computer Science*, pages 225–244, 2002.

[16] Robert Nieuwenhuis and Albert Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In K. Etessami and S. Rajamani, editors, *Proceedings of 17th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science. Springer, 2005. (To appear).

[17] Silvio Ranise and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2006.

[18] H. Ruess and N. Shankar. Solving linear arithmetic constraints. Technical Report CLS-SRI-04-01, SRI International, January 2004.

[19] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[20] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, June 1972.