Encoding and Solving Problems in Effectively Propositional Logic

A thesis submitted to The University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2007

Juan Antonio Navarro-Pérez

School of Computer Science

Contents

Word count: 72,939.

List of Tables

List of Figures

List of Algorithms

Abstract

Solving problems by translating them to propositional satisfiability checking has been found recently as a very successful approach in many application domains. Highly optimised satisfiability solvers have been used to find solutions of difficult problems. Unfortunately, this approach does not always scale when either parameters or the size of the problem description start to grow. Translations of real-world problems tend to create large formulae, whose size is dominated by slightly different copies of subformulae which had to be replicated many times.

This thesis investigates the alternative of using the language of *effectively propositional logic*, which is a decidable fragment of first-order logic, as a formalism to describe problems from applications in a much more succinct and natural way. At the same time, this would allow to face scalability issues by the use of reasoning techniques that work at a higher level of abstraction.

After a brief overview of existing propositional and effectively propositional solving techniques, some new ideas are discussed on how the encoding of problems affects the difficulty of solving them. Then, supporting the thesis hypothesis, a pair of case studies are developed where problems from planning and model checking are encoded using effectively propositional formulae. Incidentally, the products of these case studies provide a rich and diverse set of benchmarks for implementors of reasoning tools. The thesis also explores possible approaches to efficiently solve the generated formulae, either by instantiation methods or directly by reasoning at the effectively propositional level.

The work presented should be of value to communities solving problems in planning and verification, who are provided with an alternative method to apply in their domain, and to implementors of effectively propositional systems, to whom a set of relevant benchmarks is provided. The encoding techniques may also be of interest to a wider audience trying to apply automated deduction methods in other research areas.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

Foremost, I would like to thank my supervisor, Prof. Andrei Voronkov, who shared with me a lot of his expertise and research insight. He quickly became for me the role model of a successful researcher in the field. I also like to express my gratitude to Dr.-Ing. Renate A. Schmidt, whose thoughtful advise often served to give me a sense of direction during my PhD studies. And I am deeply grateful to the National Council of Science and Technology in México (CONACYT) for the trust and support that they gave me in order to study in the UK.

It is difficult to overstate my appreciation to Prof. Mauricio Osorio, who first brought me into the world of research and with whom I began to learn about mathematical logic and theoretical computer science. Not only a great mentor and colleague, he has also been a cornerstone in my professional development.

I wish to thank everybody with whom I have shared experiences in life. From the people who first persuaded and got me interested into the study of mathematics, specially those who also played a significant role in my life, to those which with the gift of their company made my days more enjoyable and worth living.

I am tempted to individually thank all of my friends which, from my childhood until graduate school, have joined me in the discovery of what is life about and how to make the best of it. However, because the list might be too long and by fear of leaving someone out, I will simply say *thank you very much to you all.* Some of you are, anyway, quite lucky: thank you Luis and Enrique (los Novatos), Ariadna, Joachim, Gaby and Héctor for being some of my very best friends.

I cannot finish without saying how grateful I am with my family: grandparents, uncles, aunts, cousins and nephews all have given me a loving environment where to develop. Particular thanks, of course, to Daniel and Rodrigo my brothers and best mates. Lastly, and most importantly, I wish to thank my parents, Maria Eugenia and Juan Antonio. They have always supported and encouraged me to do my best in all matters of life. To them I dedicate this thesis.

The author

Juan Antonio Navarro-Pérez studied at Universidad de las Américas, Puebla, in México where he got an undergraduate degree in mathematics. There he began his research work together with Prof. Mauricio Osorio on the foundations of logic programming and, in particular, the paradigm of answer set programming.

He got a master's degree in computer science, also at Universidad de las Américas, while still collaborating with Prof. Osorio's research group. Their publications in this area mostly consider the study of relations between intuitionistic, multivalued and modal logics in the context of the answer set semantics.

This document summarises his research work, carried out while studying a PhD degree at The University of Manchester under the supervision of Prof. Andrei Voronkov, in the fields of logic and automated reasoning. The main topic of his research is the description of problems from applications using the language of effectively propositional logic, a finite domain fragment of first-order logic.

Chapter 1

Introduction

The idea of designing *intelligent machines*, capable of automatically solving our problems, is highly appealing and fascinating. A great amount of research has been spent since the middle of the last century when scientists started developing the theory and implementing some early systems to automate all sorts of tasks which would, otherwise, be tedious or impractical for us humans to do.

The computer is one of the major breakthroughs in the development of a *universal machine*, capable of systematically working out solutions of problems that naturally arise from the most diverse applications, both in the contexts of science and industry. In order to do so, however, one must first describe these problems in a clear and unambiguous way, suitable for the computer to manage and process.

In this setting, a very significant issue is how a problem, which was originally found in a *real-world* situation, can be easily abstracted in such a way that it becomes amenable for mechanical and even autonomous manipulation. Formal languages have been devised for such purpose which allow, while removing any possible ambiguity, to describe problems originating from different application domains. Specifically designed algorithms and computer programs can then be used to operate on these formal descriptions and produce an output which, when translated back to the context of our original problem, represents the solutions that we were looking for.

The language of mathematical logic, being one of the most well researched formal languages, is also one of the first natural candidates to describe and represent problems for computers to solve. The main advantages of mathematical logic are that, in the spirit of mathematics itself, it aims to be both unambiguous, with expressions having a clearly defined meaning, and general, so that it is capable of modelling a broad range of diverse applications.

1.1 Solving problems by translation to logic

The language of mathematical logic allows one to write *formulae*, syntactic expressions built from logical symbols, which are used to represent relations between basic atomic elements in the domain of some logic. A real-world problem can be formally specified by encoding it as a logical formula in such a way that solutions of the problem correspond to *models* of the formula. In this context, a model is an assignment of *truth values* which are given to some atomic elements of the formula while trying to turn it into a *true* sentence.

Take for instance a problem in logistics. A delivery company needs to transport a number of packages between various locations using a set of trucks and aeroplanes. We can encode a description of this problem as a set of logical constraints (e.g. where the packages originally are, where do they need to be sent, how packages are loaded and unloaded from vehicles, and how vehicles are driven between locations), enabling us to use a computer program that automatically searches for a logical model which *satisfies* all of these constraints. Problems stated in this way are known as *satisfiability problems*.

The same problem, but seen from the opposite point of view, is closely related to *theorem proving*. Imagine that we have a circuit, some electronic component, together with a formal description of its intended functionality. In this case we can pose the problem of finding an example of a case where the component behaves incorrectly. If a solution is found, then it means that a 'bug' or some error in the implementation has been found. Otherwise, if there are no solutions, a formal proof can be produced which certifies the correctness of our component.

Satisfiability testing, and theorem proving are two standard *reasoning tasks*, which can be used as tools to find solutions of problems or prove properties in application domains which have been encoded as logical formulae. Systems are then implemented which, given a formal description of some problem, automatically search for solutions of the problem. Moreover, if no solution exists, it is then also possible to produce a formal (mathematical) proof of the fact that, indeed, no solutions exists.

Now, within mathematical logic, several languages with different trade-offs between complexity and expressibility have been defined. *First-order logic*, for example, is able to express quantified statements that hold “for every” or “at least one” element. Moreover, function symbols (e.g. the *successor* of a number) also make it possible sometimes to describe domains with an infinite amount of elements. All this expressive power comes, however, at a price. The satisfiability problem for first-order logic is undecidable. This means that it is impossible to devise an algorithm which, given *any* first-order formula as input, will eventually stop and correctly determine whether the formula has any models or not. Nevertheless, this does not rule out the possibility of having extremely optimised systems, such as VAMPIRE (Riazanov and Voronkov, 2002), which exhibit a good performance on a wide range of practical applications.

On the other hand of the spectrum, *propositional logic* is a language where the only elements allowed are basic assertions (propositions) joined together with simple logical connectives such as ***and***, ***or***, and ***not***. The satisfiability problem becomes decidable, but it is still rather challenging to solve (it is NP-complete). A great effort has also been spent into developing efficient propositional satisfiability solvers and, particularly in the last decade, the impressive achievements obtained have enabled the application of these technologies in a number of industrial strength applications.

1.2 The success of propositional translations

Not more than two decades ago, the first systems to test the satisfiability of propositional logical formulae were developed. Those early implementations were only able to solve simple toy examples while running on dedicated research workstations (Mitchell, 2005). Nowadays —as a result of advances in theory, implementation techniques as well as the increase of raw computer power— off-the-shelf satisfiability solvers, available to download from the web, are several orders of magnitude more efficient and robust. As a consequence, this has also allowed the embedding of satisfiability solving technology within more realistic applications.

It has long been known from theory that many interesting problems, those in the NP complexity class, can be solved by translating them into the problem of checking the satisfiability of a propositional logical formula. And some of the first attempts to bring this result into practise sprung from the seminal works of Kautz and Selman (1992, 1996) which tried to apply satisfiability technology to solve problems in the areas of planning, scheduling and logistics (Crawford and Baker, 1994; Giunchiglia et al., 1998; Ferraris and Giunchiglia, 2000). It was rather surprising, indeed, that a system which used a generic satisfiability solver as a black box was competitive with state-of-the-art tailor-made implementations of planning systems (Kautz, 2006).

Another area, which has greatly benefited from the recent improvements in propositional satisfiability solvers, is that of electronic design automation. Tasks such as microprocessor verification, automated test pattern generation, and proving circuit equivalence (Stephan et al., 1996; Marques Silva and Sakallah, 2000; Velev and Bryant, 2001; Goldberg et al., 2001) are typical examples where propositional satisfiability has been successfully applied. The model checking and verification communities have also developed, since the seminal work of Biere et al. (1999), tools which use a satisfiability solver as one of the main components providing search and inference services (Williams et al., 2000; Clarke et al., 2001; Sorea, 2002; Eén and Sörensson, 2003; Chaki et al., 2003; Marques Silva, 2005).

Similar ideas have also been applied in diverse fields such as natural language processing (Keselj and Cercone, 2007), knowledge representation (Lin and Zhao, 2004), security protocols (Armando and Compagna, 2002), cryptanalysis (Massacci and Marraro, 2000) and even bioinformatics (Lynce and Marques Silva, 2006). In general, almost any kind of combinatorial search problem can be addressed by using the reasoning services of propositional logic.

This approach, moreover, offers several advantages. First, after a problem has been abstracted to a logical sentence, one can better focus on developing techniques that more efficiently solve the core of the problem, without the distraction of application specific details. The use of a simple, yet general and expressive, formal language also leads to the development of simpler data structures and more efficient algorithms. Moreover, the availability of highly optimised propositional reasoning tools (e.g. Moskewicz et al., 2001; Eén and Sörensson, 2005) allows one to quickly build prototypes while importing years of research effort into completely new application domains.

The rise of these applications, in return, have also driven the interest of the research community into designing even more robust and efficient solvers. Perhaps more importantly, it has also aided to recognise *which* are the problems that we actually want to solve and *how* to better encode them using logic. A common characteristic found in such applications is for example that, often, the propositional encodings contain many duplicates of nearly identical groups of subformulae.

Scientists have started to pay more attention to these issues by exploiting the structure and symmetries that are almost inherent to problems emerging from real-world applications (Crawford et al., 1996; Sabharwal et al., 2003; Aloul et al., 2003; Lynce and Marques Silva, 2007). Proposals for solvers that are able to handle languages with a richer set of constructs, instead of the usual *normal forms*, have also been put forward (Giunchiglia and Sebastiani, 2000; Thiffault et al., 2004).

1.3 The effectively propositional alternative

A shortcoming of propositional logic is that it is not always very suitable to succinctly express patterns, symmetries and structures that naturally arise in typical applications. Problems encoded in propositional logic tend to contain many isomorphic or nearly isomorphic groups of subformulae which appear when one tries to assert, for example, that some common property is shared among many different objects in the domain being described. More importantly the size of the encodings tend to be dominated by these almost identical copies, thus preventing the approach to scale for larger or more complex applications.

And, although techniques have been devised to extract and make use of this *structure information* on propositional encodings, it is often the case that this information was explicitly available when the problem was originally described at some higher level of abstraction. The translation to propositional satisfiability often tends to blur the information about the structure and symmetries of the original problem, and bury it below large sets of seemingly meaningless formulae.

The main hypothesis of this thesis is therefore that an intermediate language, between propositional and first-order logic, is more suitable to efficiently describe and solve relevant classes of problems emerging from applications. The language proposed, known as *effectively propositional* (EPR), allows the use of variables and quantification but is limited to finite domains. Because of this restriction, the language itself has the same expressive power as propositional logic (and thus the reason of its name), but the added syntactic features allow to describe problems in a more natural and succinct way, while preserving more of their original structure.

This logic, which also corresponds to the Bernays-Schönfinkel fragment of first-order logic, has also been of recent interest for the research community in automated reasoning. The CADE ATP System Competitions (Sutcliffe and Suttner, 2006), for example, host a special category of EPR problems to assess the performance of first-order provers when dealing with this kind of formulae.

As we have mentioned, any effectively propositional formula can be translated, through a process known as *grounding*, into a plain propositional one. The size of the resulting formula, however, is often exponential in the size of the original input, thus rendering the use of standard satisfiability solvers impractical. Alternatively the use of a first-order theorem prover, which is of course possible, turned out not to be very efficient in applications with finite domains.

We expect that the use of a richer language would allow not only to describe problems more compactly, but also to reason about them more efficiently. Designing a system that is able to natively handle effectively propositional formulae may also have a number of advantages. First, because of using a more compact representation, the system can be more memory efficient and, therefore, able to process larger and more complex problems. Additionally, as being situated between two other well known and studied logical formalisms, it becomes perhaps easier to integrate techniques and methodologies from both of them.

Some theoretical foundations, in order to develop efficient reasoning tools particularly geared towards effectively propositional formulae, have already been established by authors such as Baumgartner and Tinelli (2003), as well as Ganzinger and Korovin (2003). These are logical calculi which, directly at the effectively propositional level, process and reason about the problems that one is trying to solve. Researchers have also implemented systems, such as Darwin (Fuchs, 2004) and iProver (Korovin, 2006), based on those theories; and are eager to test their capabilities when applied in realistic application domains.

Unfortunately, there is also currently a lack of benchmarks for researchers to experiment and run tests with their systems. The TPTP Problem Library (Sutcliffe and Suttner, 1998), one of the main sources of benchmarks for the theorem proving community, only contains a handful of effectively propositional formulae, a significant portion of which are just translations from randomly generated modal formulae. Moreover, the vast majority of these problems are now very easily solved by state-of-the-art systems. One of the major contributions of the research work summarised in this thesis, and in order to support our main hypothesis, consists in the generation of many fresh new benchmarks that originate from real-world application domains.

1.4 Contributions of this thesis

The research that has been carried out, and whose results are documented in this thesis, generated contributions that can be grouped in two main sections: those directly relating to propositional logic, and those which are applicable to effectively propositional logic. More details about the impact that these contributions potentially have in the research community are also discussed as part of the conclusions in Chapter 8.

The first group, on contributions with respect to propositional logic, includes:

- A method for randomly generating non-clausal formulae which have a non-trivial structure and are difficult for propositional reasoners to solve (Section 3.1).
- Some first empirical studies on how the use of different clausal form translations affect the performance of satisfiability solvers working on the translated formulae (Sections 3.1 and 3.2).
- Insights on how to develop normal form translations that make use of specialised constraints in order to better reflect the structure of problems from applications (Section 3.2).

In the second group, on contributions with respect to effectively propositional logic, we have:

- The definition of a *finite domain predicate logic*, which puts syntactic sugar on top of the effectively propositional logic to even more easily and naturally describe problems from applications (Chapter 4).
- A method to encode LTL bounded model checking problems as an effectively propositional formula, as well as some insights on how to improve and extend this method (Chapter 5).
- Two different methods, and some variants, to encode planning problems as effectively propositional formulae (Chapter 6).

- Empirical work evaluating the performance of state-of-the-art theorem proving approaches when solving effectively propositional formulae originating from these applications (Chapters 5 and 6).

- Ideas on how to efficiently implement techniques that limit the number of generated instances when using a grounding approach (Chapter 7).

- Empirical evidence suggesting that, for effectively propositional formulae, a one-shot approach is more efficient than incrementally search for finite models of an increasing size (Chapter 7).

- A formal proof on how resolution can produce exponentially shorter refutations at the effectively propositional, rather than just propositional, level (Chapter 7).

- The proposal of a *generalisation inference rule* which, when combined together with resolution, is also able to produce exponentially shorter refutations. Also a formal study on how to incorporate this inference rule with other standard techniques such as sort inference is provided (Chapter 7).

The following are also a number of contributions which are perhaps relevant to a wider range of the research community:

- An overview of the state of the art on propositional (Chapter 2) and effectively propositional logic (Chapter 4), which might serve as a helpful introductory text for students and researchers.

- Case studies on how two different applications can be encoded in effectively propositional logic were developed (Chapters 5 and 6). These also serve as a guideline, together with the finite domain predicate logic of Chapter 4, on how to design similar translations for other application domains.

- As a by product of this research, a number of effectively propositional benchmarks were generated from different applications (Chapters 5, 6 and 7). The generated benchmarks have been made available to the community through the TPTP Problem Library.

1.5 Thesis overview

In Chapter 2 we present an overview of propositional logics and describe many of the techniques that have been helpful in the design of efficient satisfiability solvers. This includes a detailed account of the Davis-Logemann-Loveland algorithm, as well as an inventory of several stochastic local search strategies.

In Chapter 3 some of our contributions in the context of encoding problems using propositional logic are discussed. This includes a proposal of a method to randomly generate formulae which are non-clausal and have a non-trivial structure. This was originally published in a paper at the Twentieth National Conference on Artificial Intelligence (Navarro Pérez and Voronkov, 2005), and a summary given at the Workshop on Automated Reasoning in 2005 (Navarro Pérez, 2005). Some ideas with respect to possible improvements on clausal form translations, the use of specialised types of constraints, as well as an empirical study on the impact that the problem encoding has on the effectiveness of satisfiability solving procedures are also discussed. An early draft proposing some of these ideas was also presented at the Workshop on Automated Reasoning in 2006 (Navarro Pérez, 2006).

In Chapter 4 we formally introduce the logic of effectively propositional formulae, and two of the main approaches to decide the satisfiability of these formulae are surveyed. The first approach is to reduce the problem to checking the satisfiability of a propositional formula, the second tries to apply inferences directly at the effectively propositional level. Finally, the chapter closes with the definition of a *finite domain predicate logic*, which is a syntactic extension of effectively propositional formulae. A simpler version of this logic was originally given, in the context of planning, at our paper to appear at the Harald Ganzinger memorial volume (Navarro Pérez and Voronkov, 2007b).

In Chapter 5 we include the first case study on using the language of effectively propositional logic to encode problems from applications. In this case we encode the problem of LTL bounded model checking, which searches for counterexamples to temporal properties that a system should always satisfy. Most of this material was published at the Conference on Automated Deduction (Navarro Pérez and Voronkov, 2007a), and also a summary at the Workshop on Automated Reasoning in 2006 (Navarro Pérez, 2007). New previously unpublished material, with an empirical evaluation of the approach presented, is now included in this chapter.

In Chapter 6 two different methods to encode planning problems as effectively propositional formulae are given. The first one, which was the main topic of our paper in the Harald Ganzinger memorial volume (Navarro Pérez and Voronkov, 2007b), involves using predicates to encode fluents as they change over time according to the actions being executed. The second alternative method involves using a single predicate to encode the whole state of the system, and then encode actions as relations between pairs of reachable states. An empirical evaluation is also performed at the end of this chapter.

In Chapter 7 we study different methods to solve the kind of problems that were generated in the two previous chapters. Some improvements on techniques, which solve effectively propositional formulae by a reduction to plain propositional logic, are proposed. A discussion of several inference rules, which work at the effectively propositional level, is also given. We formally prove, in particular, an exponential separation between the capabilities of propositional and effectively propositional resolution. And a new inference rule, which we call *generalisation* and can yield a further exponential improvement, is also proposed.

In Chapter 8 we finally present a discussion and overview of the material presented in this thesis. Its major contributions are also highlighted, and their possible impact on the research community assessed. Ideas that require further exploration, as well as interesting questions left open, are also raised.

Chapter 2

Propositional logic

The problem of propositional satisfiability (SAT) has been recognised as a key computational task in many fields of computer science. Several applications, particularly in automated reasoning and artificial intelligence, rely on the implementation of efficient procedures capable of solving this problem. As the standard example of NP-complete problems, SAT also features a central role in the theories of complexity and computation.

Given a propositional expression built from standard logical connectives, the problem is to decide whether there is a truth assignment which makes the expression evaluate to *true*. In the 1970s, after the famous NP-completeness theorem from Cook (1971), it was realised that many relevant problems in the field can be restated in terms of satisfiability. Nowadays an important set of industrial problems, notably from the domains of verification and planning, are tackled by first translating them into satisfiability checking.

Propositional satisfiability techniques have been successfully applied to perform tasks such as microprocessor verification, automated test pattern generation, and proving circuit equivalence (Marques Silva and Sakallah, 2000; Stephan et al., 1996; Velev and Bryant, 2001). Significant results have also been obtained in the areas of model checking and verification where tools for testing hardware and software components have been developed (Biere et al., 1999; Chaki et al., 2003). Applications have also arisen in the domains of planning, scheduling and logistics (Kautz and Selman, 1992, 1996) encouraging the research in methods to efficiently solve this problem.

The NP-complete complexity class has been traditionally associated with problems that are particularly challenging and difficult to solve. As a simplistic but commonly accepted belief, this kind of problems are considered as intractable in the general case. Surprisingly enough, as a consequence of a deeper understanding of the problem and the availability of more powerful computing resources, it has been possible to develop systems efficient enough to enable the use of satisfiability solving procedures in industrial applications.

In this chapter we will first define the syntax and semantics of propositional logic which lies at the core of the SAT problem. We will then also discuss some normal form translations and preprocessing techniques that have been proposed in order to further simplify formula encodings. The remainder of the chapter includes a brief overview on the state of the art of both complete and incomplete decision procedures for satisfiability checking.

A *complete* decision procedure is one that takes a formula as input and always terminates returning either *'yes'* or *'no'* as answer to whether the formula is satisfiable or not. The first such satisfiability algorithm is often attributed to Davis and Putnam (1960) for their method based in resolution. This algorithm, however, suffers from memory explosion and thus an improved version was proposed two years later by Davis, Logemann, and Loveland (1962). Using search instead of resolution, the DLL algorithm is still the basic foundation of many modern satisfiability solvers.

On the other hand, an *incomplete* procedure is one that answers *'yes'*, when a solution is found, or *'don't know'*, when the search has run long enough without finding any solution. Such procedures are usually based on stochastic local search methods that, starting with an arbitrary truth assignment, make small changes to this assignment trying to get closer to a solution. Since the algorithm does not keep track of the assignments already attempted it is not guaranteed to find a solution, nor it is able to determine the unsatisfiability of a formula. As we will see, some of the most successful implementations are variants of the WalkSat algorithm originally proposed by Selman, Kautz, and Cohen (1994).

2.1 Syntax and semantics

A *propositional formula* is a syntactic expression built from a set of *atomic predicates* —also sometimes known as *atomic propositions*, *boolean variables*, or simply *predicates*— which we denote by lowercase letters: p, q, ... The most simple propositional formula is an expression known as an *atom* which merely consists

$\wedge$	*false*	*true*
false	*false*	*false*
true	*false*	*true*

$\vee$	*false*	*true*
false	*false*	*true*
true	*true*	*true*

	$\neg$
false	*true*
true	*false*

Figure 2.1: Truth tables of propositional logic

of a single atomic predicate. More complex formulae are then built from atoms using the connectives of truth (***true***, $\top$), falsity (***false***, $\bot$), conjunction (***and***, $\wedge$), disjunction (***or***, $\vee$), and negation (***not***, $\neg$). Other connectives such as implication and equivalence are defined in terms of those already introduced, e.g. $F \rightarrow G \equiv \neg F \vee G$ and $F \leftrightarrow G \equiv (F \rightarrow G) \wedge (G \rightarrow F)$. A set of formulae will be also often referred to as a *set of constraints*. The *size* of a formula, or set of formulae, is counted as the number of symbols in it.

A *literal* is either an atom p, a literal with a *positive phase*, or a negated atom of the form $\neg p$, a literal with *negative phase*. If l is a literal, we will also use the notation $\tilde{l}$ to denote its complement, i.e. $\tilde{l} = \neg p$ if $l = p$ and $\tilde{l} = p$ if $l = \neg p$. In the propositional case, we will also often use x and y to represent arbitrary literals. A *clause* is a disjunction of literals and a *clausal formula* is a conjunction of clauses. For simplicity a clausal formula is sometimes abstracted as a set of clauses, and a clause as a set of literals.

A clause containing a single literal is known as a *unit clause*, also sometimes referred to as a *fact*. A clause without any literals at all, the *empty clause*, is also legitimate and represents a contradiction. We will often say that a propositional formula *is in clausal normal form* (CNF) or, as it is also commonly know, in *conjunctive normal form* if the formula is a clausal formula.

One way to define the semantics of propositional formulae is through the notion of truth assignments. A *truth assignment* is simply a function that maps every atomic predicate to one of the truth values of *true* or *false*. The evaluation of truth assignments is then extended to more complex expressions by using the truth tables of Figure 2.1; moreover $\bot$ always evaluates to *false*, and $\top$ to *true*.

We then say that a formula is *satisfiable* if there is a truth assignment, sometimes also called a *solution* or a *model* of the formula, that makes the formula evaluate to *true*; if there is no such assignment we say that the formula is *unsatisfiable*. Given a pair of formulae we also say that they are *equivalent* if they have exactly the same models. And they are *equisatisfiable* if one of them is satisfiable if and only if the other is.

The *propositional satisfiability problem* (SAT) asks, given a propositional formula as input, whether the formula is satisfiable or not. The *tautology problem* is the related problem that asks whether a formula is a *tautology*, i.e. it evaluates to *true* under all possible truth assignments. Notice that satisfiability checking and the tautology problem are duals: a formula is a tautology if and only if its negation is unsatisfiable. Also note that the empty clause is unsatisfiable; while a *trivial clause*, containing a complementary pair of literals x and $\tilde{x}$, is a tautology.

Alternatively, the semantics of propositional logic can be defined in terms of logical calculi and inference rules. Although many inference systems have been defined for propositional logic (see e.g. Mendelson, 1987), we will describe the inference rule of resolution which is quite relevant to the methods and techniques described in this and later chapters. In the context of propositional logic the inference rule of resolution has the form

$$\frac{C \vee p \quad \neg p \vee D}{C \vee D}$$

which allows to deduce, from a pair of clauses sharing a complementary literal, a new clause which is the union of the two after removing these literals on which the clauses were *resolved*. Alternatively, we will also sometimes use $\otimes$ as the operator for resolution and instead write

$$(C \vee p) \otimes (\neg p \vee D) = (C \vee D) \ .$$

The logical calculi of propositional resolution simply takes as input a set of clauses, applies the resolution rule among pairs of clauses in the set and incrementing it by adding the result of each such inference. This process is iterated until either: the empty clause is generated, in which case the original set of clauses is unsatisfiable; or no more resolution inferences are possible, we say then that the set has been *saturated* and, from theoretical results, it follows that the set must be satisfiable.

Additional inferences and simplifications, such as subsumption explained later in Section 2.3 are possible, but this is the most basic idea behind propositional resolution theorem proving. Also note, moreover, that this calculus only works with formulae written in clausal normal form. Nevertheless, as we are about to see, this does not impose any significant restriction in practise.

$$
\begin{array}{rclrclrcll}
\bot \wedge F & \Rightarrow & \bot & \bot \vee F & \Rightarrow & F & \neg\bot & \Rightarrow & \top & (2.1)\\
\top \wedge F & \Rightarrow & F & \top \vee F & \Rightarrow & \top & \neg\top & \Rightarrow & \bot & (2.2)\\
\neg(F \wedge G) & \Rightarrow & \neg F \vee \neg G & \neg(F \vee G) & \Rightarrow & \neg F \wedge \neg G & \neg\neg F & \Rightarrow & F & (2.3)
\end{array}
$$

Figure 2.2: Rewrite rules for negation normal form translation

2.2 Normal forms

For formulae in clausal normal form, it is very easy to solve the tautology problem (a formula is a tautology iff it contains only trivial clauses, and this can be checked in linear time). On the other hand the satisfiability problem of clausal formulae is still NP-complete (Cook, 1971). As a consequence it is very unlikely to find an efficient (polynomial) algorithm to translate arbitrary propositional formulae into an equivalent clausal form representation (since otherwise it would be possible to translate the negation of the formula into clausal form and then solve the easier tautology problem).

Using a well known translation, which works by applying distributivity laws among logical connectives, it is possible to translate an arbitrary propositional formula into an equivalent clausal form. The size of the resulting formula is, however, exponential on the size of its input. In order to more easily describe this and other clausal form translations, we will first introduce another normal form and its corresponding translation in order to simplify the treatment of negation.

Definition 2.1. A formula F is in *negation normal form*, or simply *NNF*, if the formula is either $\top$, or $\bot$, or is built from literals using the connectives of conjunction and disjunction only. The negation connective, in particular, is only allowed to appear in front of an atom to form a literal.

A formula F is also called a *negation normal form* of a formula G, if F is equivalent to G and F is in negation normal form. ■

The following algorithm allows one to translate any arbitrary propositional formula into an equivalent negation normal form. The translation moreover, is quite simple and can be performed in linear time.

Algorithm 2.1 (Negation normal form translation)**.** The rewrite rule system on formulae of Figure 2.2 gives an algorithm to translate arbitrary propositional formulae into negation normal form. Given an input formula, the algorithm simply applies the rewrite rules in a nondeterministic manner until no rule can be applied anymore.

It is easy to verify that the rewriting rule system is terminating and, moreover, that the resulting expression is indeed a negation normal form of the input. Rules (2.1–2.2) in particular, take care of evaluating the effect of the truth and falsity connectives, while rules (2.3) push negation inwards until the level of atoms is reached. Also note that, after applying this translation, if either the $\top$ or $\bot$ connectives remain in the formula, then the whole formula has been simplified to this single connective. Since the $\top$ and $\bot$ do only appear in these two trivial cases, in the following we will assume that formulae do not contain any of these connectives.

We can now introduce the standard clausal form translation, which is able to translate any formula in negation normal form to an equivalent clausal formula. This translation, however, is not very efficient and causes, in the general case, an exponential blowup in the size of the resulting formula.

Algorithm 2.2 (Standard clausal form translation)**.** Given an input formula in negation normal form, the algorithm simply applies the rewrite rule

$$\begin{array}{lcll}(A_1 \wedge \cdots \wedge A_m) \vee B_1 \vee \cdots \vee B_n & \Rightarrow & (A_1 \vee B_1 \vee \cdots \vee B_n) & \wedge \\ & & \vdots & \wedge \\ & & (A_m \vee B_1 \vee \cdots \vee B_n) \;, & \end{array}$$

in a nondeterministic manner until it cannot be further applied.

Again, it is easy to verify that the algorithm is terminating and, moreover, that the resulting formula is in clausal normal form. In practise, however, this translation is not very useful because of the exponential increase in size that it yields. Nevertheless, using an alternative translation due to Tseitin (1968), it is possible to reduce a propositional formula into an *equisatisfiable* clausal representation. Note that the resulting formula is no longer equivalent to its input, but only satisfiable if and only if the original formula is.

This translation is linear, both in time and in the size of its output, and is achieved by introducing new atoms that are used to replace nested subformulae. We will now give a simplified version of Tseitin's translation which implements the polarity optimisations of Plaisted and Greenbaum (1986), which take into account how subformulae are influenced under the scope of the negation connective, simply by assuming that the formula is already given in negation normal form.

Algorithm 2.3 (Tseitin's clausal form translation)**.** Let G be a formula in negation normal form. For each subformula F introduce a new predicate symbol p_F. Then let Δ_G be a set containing, for each subformula $F = A \wedge B$, the pair of constraints

$$\begin{aligned} &p_F \rightarrow p_A \\ &p_F \rightarrow p_B \end{aligned}$$

and, for each subformula $F = A \vee B$, the constraint

$$p_F \rightarrow p_A \vee p_B \ .$$

The algorithm produces as output the set of constraints $\Delta_G \cup \{p_G\}$.

The following theorem, which is easy to prove using the same original ideas from Tseitin (1968), establishes the correctness of this translation.

Theorem 2.1. *Let F be formula in negation normal form. The formula F and the set of constraints $\Delta_F \cup \{p_F\}$ are equisatisfiable.*

Motivated by this result, the study of the satisfiability problem and the implementation of solvers have been traditionally restricted to propositional formulae in clausal normal form. The use of a simple and uniform representation for a broad class of problems provided a favourable working framework for researchers to develop ideas and implementations. In fact, the design of an effective data structure to store and operate with formulae in a clausal form representation (Moskewicz et al., 2001) is one of the crucial elements that allows modern solvers to achieve their levels of efficiency.

Recently, however, researchers have started to understand the shortcomings of using such reduced normal forms. Foremost, problems are usually expressed in a much more natural and concise way in higher level languages: problems in scheduling and logistics are usually expressed in specialised planning languages (e.g. PDDL, STRIPS); models for verifying hardware and software systems are also described in particularly designed languages (e.g. Verilog HDL, SMV); even problems at lower electronic design levels involve circuits with a rich set of gates and components (e.g. ***xor***, bit counts and inequalities) that can, furthermore, be arbitrarily nested and wired.

When these high level descriptions are translated into clausal forms, a lot of valuable information from the original problem is lost. Problems arising from real-world applications often have a great amount of structure but, when translated to a clausal formula, complex relationships between objects and variables in the problem are just chopped down and buried into a large set of clauses. Moreover, symmetries that may be explicit in the problem description, and that could be exploited during the search for a solution, end up diluted in the translated version.

There had been many efforts to take advantage of the symmetries that are still implicitly present in clausal encodings. Crawford (1992) provided, by showing that the symmetry detection problem is equivalent to graph isomorphism, one of the first important results of symmetry in reasoning. Several techniques have also been proposed that employ this symmetry information to aid in the search for solutions (Crawford et al., 1996; Aloul et al., 2003; Sabharwal et al., 2003).

An alternative approach that has been recently explored is to design solvers that are able to directly handle more descriptive languages (such as circuit encodings) where both the symmetries and structure information are available at first hand (Thiffault et al., 2004; Sabharwal, 2005). While this line of research is still at early development stages, and the systems implemented have not reached the maturity of modern clausal solvers, the results obtained so far suggest that important progress can be made by finding adequate languages and encodings to express the kind of problems that we want are interested in solving.

2.3 Preprocessing techniques

Before applying one of the main algorithms to decide satisfiability, a formula is often preprocessed in an attempt to simplify it and possibly reduce the search space that the satisfiability checking algorithm will have to explore. The most simple preprocessing technique is to reduce a formula so that it contains no trivial clauses and that each atomic predicate appears at most once in each clause. It is also possible to immediately declare a formula as unsatisfiable if it happens to contain the empty clause.

The *pure literal rule* is another preprocessing step very commonly performed. If a literal appears in the formula in only one phase (i.e. always positive or always negative), then it is possible to assign it the truth value that will make it satisfy all the clauses where it occurs, effectively allowing us to remove all those clauses.

This rule was originally proposed by Davis et al. (1962) to be applied in the main loop of the DLL algorithm, but the cost associated with keeping track of the required literal counts was found to overcome any gains provided by the simplification (see e.g. Mitchell et al., 1992; Mitchell, 2005). One also has to note that after applying this rule, the resulting formula is no longer equivalent, but just equisatisfiable, to the original one. This is particularly important in the context of incremental satisfiability solving, where new clauses added later might invalidate previous applications of this rule.

Also proposed by Davis et al. (1962), and still the main operation during the actual solving process, is the *unit literal rule.* This rule can be applied when the formula contains a clause with a single literal: a unit clause. Since the only way to satisfy such clause is to set the adequate value to make that literal true, it is possible to remove all clauses where the literal occurs (which are already satisfied) and remove every occurrence of its complement (which are set to *false* and do not contribute to satisfy any clause).

After an application of the unit literal rule, of course, new unit clauses can be generated allowing the process to iterate and perform even further simplifications. This iterated propagation of unit clauses is know as *unit propagation* and, as detailed in a later section, is one of the core operations of the DLL algorithm itself. Another special case that might occur when performing unit propagation is that an empty clause is produced, this situation is known as a *conflict.* If a conflict occurs during the preprocessing stage then the instance is unsatisfiable.

Many other ideas for formula preprocessing have been proposed in the literature (Drake et al., 2002; Lynce and Marques Silva, 2003; Bacchus and Winter, 2003; Brafman, 2004), but only a few of them have actually been successful. One of the principal challenges, which may seem reasonable, is to achieve a good balance between the time that is spent in preprocessing and the real benefits provided by the simplification. But much more puzzling is the fact that shorter and simpler formulae are not always the ones which are easier to solve (Lynce and Marques Silva, 2001).

Sometimes having redundant clauses (i.e. clauses that follow as a logical consequence of the rest of the formula) are helpful to more quickly discover conflicts and prune the search space. It is not unusual, in fact, to find instances that become harder after being treated with a preprocessor. Moreover, techniques that "enrich" a formula by adding redundant clauses have been sometimes found useful (see e.g. Lynce and Marques Silva, 2003). The most notable example of the successful application of this idea is clause learning discussed in a later subsection.

We do will describe two of the preprocessing techniques that have shown the most promising results in practise. Bacchus and Winter (2003) propose the use of a particular form of hyper-resolution to discover binary clauses. From a single clause $(x_1 \vee \cdots \vee x_n)$ and $n-1$ binary clauses $(y \vee \tilde{x}_1), \ldots, (y \vee \tilde{x}_{n-1})$, the *HypBinRes* rule allows to derive the binary clause $(y \vee x_n)$; note that this is equivalent to $n-1$ applications of simple resolution.

Binary clauses are important because they encode simple relationships between pairs of literals, recall that the formula $x \to y$ is equivalent to the clause $(\tilde{x} \vee y)$. By implicitly computing the transitive closure of the binary implications it is possible to discover equivalences (i.e. $x \to y$ and $y \to x$) or new facts (i.e. if $x \to y$ and $\tilde{x} \to y$ then it is possible to derive the unit y). In the former, all occurrences of one of the literals can be replaced with the other; in the later, the new fact allows the application of unit propagation.

Bacchus and Winter also showed that an efficient implementation of an algorithm to compute the HypBinRes closure of clausal formulae is possible by using specialised versions of graph traversal algorithms. One of the key observations made is that HypBinRes can be simulated by asserting each literal in turns and then applying unit propagation. This follows since if a literal y is derived when the fact x is added to a given formula, what we actually prove is that $x \to y$ is a logical consequence of the formula.

In a more recent and notably successful approach, Eén and Biere (2005) proposed new preprocessing ideas based on resolution. The effectiveness of these ideas has been empirically demonstrated in the SatELite preprocessor which, in combination with the MiniSat solver (Eén and Sörensson, 2005), won all three industrial categories of the SAT 2005 competition (Le Berre and Simon, 2005). The preprocessor combines four basic reductions: predicate elimination, subsumption, self-subsumption and definitional subsumption.

The principle of *predicate elimination*, also known as *variable elimination*, dates back to the first algorithm for satisfiability from Davis and Putnam (1960). Given a clausal formula let S_p be the set of clauses that contain the atom p and $S_{\neg p}$ the clauses containing $\neg p$. It is possible to apply pairwise resolution between each clause in S_p and $S_{\neg p}$ to obtain a new set of clauses S', and then replace all the original clauses containing either p or $\neg p$ with S' thus effectively removing the predicate p from the formula.

The original algorithm of Davis and Putnam iteratively applies predicate elimination until all atomic predicates have been eliminated. But this was found to cause a dramatic increase in the size of the formula, requiring an exponential amount of space to solve the problem. However, when predicate elimination is applied in a more restricted manner, it is possible to obtain significant reductions as observed in the NiVER solver (Subbarayan and Pradhan, 2004), where predicate elimination is applied only if it produces a reduction in the number of clauses. It has been observed that, mainly in instances from real-world applications, this is often very effective since the resolution process tends to generate many trivial or subsumed clauses.

A clause C_1 is said to *subsume* another C_2, if it is the case that all the literals in C_1 also occur in C_2 (i.e. $C_1 \subseteq C_2$). The subsumed clause, C_2 in this case, is redundant and can be removed from the formula. Detection of subsumed clauses may seem very simple, but it is costly enough so that performing it inside the main loop of a SAT algorithm becomes unfeasible. As a preprocessing step subsumption can be efficiently computed using a simple signature-based algorithm (Ramakrishnan et al., 2001).

Each clause in the formula is assigned a *signature* that abstracts the set of literals that it contains. A hash function maps each literal into a number in some short domain such as the set $\{0, \ldots, 63\}$, and then the signature of a clause is a 64-bit number where the bits corresponding to literals appearing in the clause are set to 1. This allows to perform a test to quickly discard many cases of pairs of clauses where subsumption is not applicable; a complete (expensive) subset test is only required if one of the signatures is a subset of the other.

Self-subsumption occurs when a clause C_1 almost subsumes another C_2; in the sense that all the literals in C_1 except for one, say l, are in C_2 and, moreover, the literal $\tilde{l}$ also occurs in C_2. Applying resolution between C_1 and C_2 will produce a clause C_2' that contains all literals in C_2 but $\tilde{l}$. Now, notice that the resulting clause C_2' subsumes C_2 which becomes redundant. Effectively, one can simply remove the literal $\tilde{l}$ in C_2 when an instance of self-subsumption is detected. The procedure already described to detect subsumed clauses is also useful to implement self-subsumption in an efficient way (Eén and Biere, 2005).

Finally *definitional subsumption* tries to exploit the fact that many typical satisfiability instances, particularly those which come from circuit descriptions encoded with Tseitin's translation (1968), contain implicit encodings of primitive logical gates (***and***, ***or***). The groups of clauses that define each gate are extracted using graph methods (Ostrowski et al., 2002); and then the atom that represents the output of the gate is resolved using predicate elimination. Exploiting the fact that this atom is functionally dependent on the inputs of the gate, some of the resolved clauses become redundant. This increases the chances that predicate elimination does indeed reduce the total number of clauses in the formula and that the effects of the technique are beneficial (Eén and Biere, 2005).

After any of these preprocessing techniques have been applied, one now has to apply some algorithm to actually solve the core of the propositional satisfiability problem. The following two sections explore some of the most successful approaches that have been proposed in the context of, respectively, complete and incomplete decision procedures.

2.4 The Davis-Logemann-Loveland algorithm

Although one of the earliest procedures for tautology checking, which is the dual of the satisfiability problem, is the iterated consensus by Quine (1952); the credit for designing the first satisfiability algorithm is often attributed to Davis and Putnam (1960). This is a complete algorithm which takes as input a clausal formula and iteratively applies resolution, in the form of predicate elimination, until all atomic predicates have been removed from the formula. If an empty clause is produced after the elimination of an atomic predicate then the instance is unsatisfiable; otherwise the process continues until no more clauses remain in the formula in which case it is satisfiable. The problem with this approach is that resolution in general may cause an exponential growth of the formula during the solving process, thus rendering it unfeasible except for formulae with very few atomic predicates.

In order to overcome this issue; Davis, Logemann, and Loveland (1962) proposed an improvement over the previous algorithm that replaces the resolution steps with search. In its most simple presentation it takes the form of a depth first search backtracking algorithm over the space of truth assignments. On each step an atomic predicate in the formula is selected and assigned a value; the formula is simplified using this value and if it turns to be satisfiable —invoking the same procedure recursively— a solution is found and the algorithm terminates;

otherwise the other possible value for that predicate is tried in another recursive call; if both attempts fail then the formula is unsatisfiable.

Algorithm 2.4 outlines this procedure, which is commonly referred to as the DLL algorithm. The symbol $F|l$ is used to denote the loosely defined expression: the formula F after being *simplified* by adding the fact l. We will describe in a moment some possible realisations of this required simplification. Also note that the algorithm has to *decide* on which atomic predicate to branch at each recursive step, this is the matter of discussion in a later subsection on decision heuristics. The number of decisions required to solve an instance of the SAT problem is usually taken as a measure of the search space explored by the algorithm.

Algorithm 2.4 The Davis-Logemann-Loveland algorithm

```
procedure DLL($F$)
    if $F$ contains no clauses return “Satisfiable”
    if $F$ contains an empty clause return “Unsatisfiable”
    pick a literal $l$ whose atomic predicate appears in $F$          ▷ decision
    if DLL($F|l$) returns “Satisfiable” then
        return “Satisfiable”
    else
        return DLL($F|\tilde{l}$)
    end if
end procedure
```

The *search tree* for an execution of the DLL algorithm has a node for each call to the procedure, and edges from each node to the two nodes representing the performed recursive calls. The *decision level* of a node corresponds to its depth in the tree. Although it may seem that the algorithm is almost, but not quite, entirely unlike resolution; it turns out that the execution tree implicitly describes a resolution-style proof when the formula is unsatisfiable (see e.g. Beame et al., 2004). Moreover, by taking advantage of this feature, it is possible to develop systems that are able of generating proof certificates which, in turn, can be systematically verified by independent proof checkers (Zhang and Malik, 2003).

In the original description of Davis et al. (1962), it is suggested to apply the unit literal and pure literal rules to simplify the formula at each node of the execution. However, the costs associated with keeping track of the required information to detect pure literals were found to overcome any benefits provided by the simplification. In fact, the use of clever decision heuristics often diminishes the effectiveness of applying this rule.

Over the years, many other simplification and reasoning techniques have been proposed (van Gelder, 2001; Bacchus, 2002; Li, 2003; Heule and van Maaren, 2004). And although some of this approaches have been successfully applied in particular domains, the leading satisfiability solvers are still those who generally perform unit propagation only. Elaborated reasoning methods are often helpful to reduce the total search space explored, but modern implementation techniques —the matter of a later subsection on unit propagation— allow to very quickly apply unit propagation and outperform systems equipped with more powerful reasoning mechanisms. These findings suggest two important lessons: first, that efficient implementations should be based on cheap and easy to maintain methods; but, second, that the application of more expensive reasoning at appropriate times (e.g. close to the beginning of the search) is often favourable to reduce the required search space to explore.

2.4.1 Unit propagation

By profiling competitive SAT solvers it was observed that about 90% of the run time is spent in unit propagation (Moskewicz et al., 2001). One of the most significant advances in the theory and applications of satisfiability was to realise this fact and to propose specialised algorithms that efficiently perform this operation. A key point to observe is that a suitable technique should not only allow to simplify the formula, but also provide efficient means to 'undo' the simplification when the algorithm backtracks and removes some decisions.

To restate the problem in consideration, when a unit clause (l) is found in the formula —either added as a decision of the algorithm, or implied as a consequence of unit propagation itself— we have to remove: all clauses where the literal l occurs, and all occurrences of $\tilde{l}$ in other clauses. The procedure should detect when new unit clauses are produced and iterate; or immediately stop and report a conflict if an empty clause is generated. In literature this method is sometimes referred to as *boolean constraint propagation* (BCP).

Early implementations (Crawford and Auton, 1993) kept counters indicating the number of unassigned literals remaining in each clause and, for each literal, the list of clauses where it occurs. When a literal is set or unset, all the counters of clauses containing the literal (or its negation) are updated accordingly (clauses that become satisfied are assigned a special undefined value and ignored). When the counter of a clause reaches 1, the algorithm searches for the unassigned literal and adds it to a queue of literals still to propagate. When a counter reaches 0, a conflict is detected and the procedure stops. This counter-based approach is very simple to understand and implement; but it becomes extremely inefficient, particularly for large formulae, to visit all clauses in order to keep the values of the counters updated.

Zhang and Stickel (1996) were the first to observe that it was unnecessary to visit all clauses when trying to detect unary clauses. If a clause has more than two unassigned literals then a single application of unit propagation will not produce a new unit clause. In their implementation they keep a pointer to the first and the last unassigned literals of each clause. When both pointers coincide, a unit literal is detected. This technique improves from the previous one since a clause has to be visited only when one of its boundary literals gets (un)assigned. Satisfied clauses are not removed, but lazily detected when performing propagations (i.e. when a boundary literal hits a literal already assigned true).

Moskewicz et al. (2001) improve even further on this idea by noticing that it is not even necessary to maintain an order between the pointers keeping track of the two unassigned literals. The *two watched literal scheme* simply maintains the property that two different unassigned literals are watched on each clause. When a watched literal gets assigned to false, the pointer simply moves freely looking for another unassigned literal to watch; if none is found a new unit clause is detected. The improvement comes from the fact that, upon backtracking, nothing has to be done! The property that the two watched literals are unassigned is already satisfied. Algorithm 2.5 presents in a little more detail the procedure to find a new watch when the literal x gets assigned to false. Again, satisfied clauses are not removed but lazily detected and ignored.

2.4.2 Clause learning

It the last ten years there were two major breakthroughs that enabled systems to go from solving problems with few hundreds of predicates to tens of thousands. One of those breakthroughs is the two watched literal scheme, the other is clause learning. The idea originally emerged in the area of constraint satisfaction (see e.g. Kumar, 1992) and was first introduced into the context of satisfiability by Bayardo and Schrag (1997) in REL_SAT, and by Marques Silva and Sakallah (1996a) in GRASP. Clause learning also enables what has been called in literature as conflict directed backjumping, intelligent or non-chronological backtracking; as

Algorithm 2.5 Find new literal to watch

```
procedure FindNewWatch($x$)
    for each literal $y$ in the clause, $y \neq x$ do
        if $y$ is set to true then
            return                                   ▷ do nothing, satisfied clause
        else if $y$ is watched then
            $w \leftarrow y$                          ▷ $w$ is the other watched literal
        else if $y$ is unassigned then
            move watch from $x$ to $y$                ▷ found a new literal to watch
            return
        end if
    end for
    add $w$ to the queue for unit propagation        ▷ unit clause detected
                          ▷ if $\tilde{w}$ is already in the queue, this detects a conflict
end procedure
```

well as many other combinations of the same sort of expressions.

The basic idea is fairly simple. At each step of the DLL algorithm a literal is set to true, i.e. decided, and the consequences of this decision are propagated. The process iterates until a conflict is found. At this point one may collect the set of decision literals, say $\{x_1, \dots, x_d\}$, that were actually responsible for generating the empty clause (note that the last decision literal, say x_d, *must* be in that set). Now we know that one of those decisions was wrong or, in other words, that a solution should also satisfy the clause $(\tilde{x}_1 \vee \cdots \vee \tilde{x}_d)$. Such clause, called a *conflict clause*, can be added to the formula, i.e. 'learnt', in order to avoid performing a similar search again that will finish in the same conflict.

It might be the case, moreover, that recent decisions (except from x_d) did not participate in the generation of this conflict. Then, it is possible to directly jump to the last relevant decision level and assert $\tilde{x}_d$ (now as an implied fact, *not* a decision) while pruning a large part of the search space. In order to accommodate for this facilities the original DLL algorithm has to be modified, Algorithm 2.6 shows a common formulation which is the base of most modern solvers.

The algorithm is more easily described (and implemented!) in an iterative rather that recursive form. As in the original DLL, an unassigned literal is selected and the consequences of assigning it to true are computed using Unit-Propagate (note however that additional reasoning, and not exclusively unit propagation, may be included at this point). If no conflict occurs then the algorithm continues performing more decisions and adding their implications. When

Algorithm 2.6 Conflict directed backjumping

```
procedure CBJ
    $level \leftarrow 0$
    while there are unassigned literals do
        $level \leftarrow level + 1$
        pick an unassigned literal $x$                          ▷ decision
        add $x$ to the queue for unit propagation
        while UnitPropagate returns "Conflict" do               ▷ deduction
            if $level = 0$ return "Unsatisfiable"
            $C$      $\leftarrow$ derived conflict clause                ▷ conflict analysis
            $y$      $\leftarrow$ asserting literal in $C$
            $level$ $\leftarrow$ level where $y$ is asserted
            undo assignments and decisions until $level$             ▷ backjumping
            add the clause $C$ to the formula
            add $y$ to the queue for unit propagation
        end while
    end while
    return "Satisfiable"
end procedure
```

a conflict occurs a special procedure, called *conflict analysis*, is invoked to derive a conflict clause. Such clause must contain the negation of: exactly one literal from the current decision level, known as the *asserting literal*, and zero or more literals from previous levels. The the second highest decision level in the clause (or 0 if the clause contains only one literal) is the *asserting level* of the clause. One can backtrack to this level (where the conflict clause becomes unit), learn the conflict clause, assert the new literal and continue propagation.

Example 2.1. An example taken from Nieuwenhuis et al. (2004) and shown in Figure 2.3, serves to clarify these ideas and introduce the computation of conflict clauses. To improve readability atomic predicates are denoted by natural numbers and decision literals are written in bold. Literals that are removed as a consequence of the current assignment are cancelled out, and the *reason* of each implied literal is underlined.

The reason for an implied literal, also called its *antecedent*, is the clause that became unit and justified its addition to the current assignment. Decision literals have no reasons, since they are arbitrarily assigned by the algorithm. The execution proceeds by simple decision and propagation of literals until step 7 when, since both literals $\tilde{6}$ and 6 have been implied, a conflict is detected. In this simple

$$\begin{array}{lll}
1: & \tilde{1} \vee 2,\ \tilde{3} \vee 4,\ \tilde{5} \vee \tilde{6},\ 6 \vee \tilde{5} \vee \tilde{2} & \mathbf{1} \\
2: & \underline{\cancel{\tilde{1}} \vee 2},\ \tilde{3} \vee 4,\ \tilde{5} \vee \tilde{6},\ 6 \vee \tilde{5} \vee \tilde{2} & \mathbf{1}, 2 \\
3: & \cancel{\tilde{1}} \vee 2,\ \tilde{3} \vee 4,\ \tilde{5} \vee \tilde{6},\ 6 \vee \tilde{5} \vee \cancel{\tilde{2}} & \mathbf{1}, 2, \mathbf{3} \\
4: & \cancel{\tilde{1}} \vee 2,\ \underline{\cancel{\tilde{3}} \vee 4},\ \tilde{5} \vee \tilde{6},\ 6 \vee \tilde{5} \vee \cancel{\tilde{2}} & \mathbf{1}, 2, \mathbf{3}, 4 \\
5: & \cancel{\tilde{1}} \vee 2,\ \cancel{\tilde{3}} \vee 4,\ \tilde{5} \vee \tilde{6},\ 6 \vee \tilde{5} \vee \cancel{\tilde{2}} & \mathbf{1}, 2, \mathbf{3}, 4, \mathbf{5} \\
6: & \cancel{\tilde{1}} \vee 2,\ \cancel{\tilde{3}} \vee 4,\ \underline{\cancel{\tilde{5}} \vee \tilde{6}},\ 6 \vee \tilde{5} \vee \cancel{\tilde{2}} & \mathbf{1}, 2, \mathbf{3}, 4, \mathbf{5}, \tilde{6} \\
7: & \cancel{\tilde{1}} \vee 2,\ \cancel{\tilde{3}} \vee 4,\ \cancel{\tilde{5}} \vee \tilde{6},\ \underline{6 \vee \cancel{\tilde{5}} \vee \cancel{\tilde{2}}} & \mathbf{1}, 2, \mathbf{3}, 4, \mathbf{5}, \tilde{6}, 6\ (\lightning) \\
8: & \cancel{\tilde{1}} \vee 2,\ \tilde{3} \vee 4,\ \tilde{5} \vee \tilde{6},\ 6 \vee \tilde{5} \vee \cancel{\tilde{2}},\ \underline{\cancel{\tilde{1}} \vee \tilde{5}} & \mathbf{1}, 2, \tilde{5}
\end{array}$$

Figure 2.3: Example of conflict learning in DLL

example it is easy to observe that the assignment of the decision literal **3** has no relevance to the conflict, and thus a valid conflict clause is $(\tilde{1} \vee \tilde{5})$. A bit more formally this formula is derived by a sequence of resolution operations between the antecedents of the conflicting literals:

$$\begin{array}{c}
(\tilde{5} \vee \tilde{6}) \otimes (6 \vee \tilde{5} \vee \tilde{2}) \\
\downarrow \\
(\tilde{5} \vee \tilde{2}) \otimes (\tilde{1} \vee 2) \\
\downarrow \\
(\tilde{1} \vee \tilde{5})
\end{array}$$

This conflict clause has the asserting literal $\tilde{5}$, which is asserted at level 1 (i.e. it only depends on decisions made up to that level). The algorithm would continue by adding the conflict clause to the formula (shown in step 8) and propagate the literal just asserted. Also note that this literal does have an antecedent, the conflict clause, that can be used later in the search to compute more conflict clauses.

Observe that, in the previous example, the clause $(\tilde{5} \vee \tilde{2})$ is also a valid conflict clause. In fact, the only requirement for a conflict clause is that it contains exactly one literal from the current decision level. Algorithm 2.7 shows the description of a very general procedure to compute conflict clauses. The symbol A_x is used to denote the antecedent of a literal x.

A number of learning schemes have been proposed in the literature being supported by several different intuitions (Marques Silva and Sakallah, 1996b; Bayardo and Schrag, 1997; Zhang et al., 2001). The work of Zhang et al., in particular, provides an experimental comparison of various learning schemes. The

Algorithm 2.7 Conflict analysis algorithm

```
procedure ConflictAnalysis($x$)
    $C \leftarrow A_x \otimes A_{\bar{x}}$
    while stop criterion not met do
        pick an implied literal $w$ in $C$
        $C \leftarrow C \otimes A_{\bar{w}}$
        add the clause $C$ to the formula            ▷ this step is optional
    end while
    $y \quad \leftarrow$ literal from current level in $C$
    $level \leftarrow \max \{ \text{level}(x) \mid x \in C \setminus \{y\} \}$
    return $C$, $y$, $level$
end procedure
```

Scheme	*selects literals in*	*until*
First UIP	current level	first UIP found.
2nd UIP	two highest levels	first UIP of each level found.
3rd UIP	three highest levels	first UIP of each level found.
⋮	⋮	⋮
All UIP	all decision levels	first UIP of each level found.
Last UIP	current level	last UIP (decision literal) found.
Decision	all decision levels	only decision literals remain.

Table 2.1: Schemes for conflict analysis found in literature

proposed variations usually differ on the strategy to pick literals to resolve, the particular stop criterion enforced and the number of intermediate clauses that are added to the formula.

An important concept in the design of conflict analysis schemes is the *unique implication point* (UIP). During the process described in Algorithm 2.7, a clause is said to have a UIP at level l if there is exactly one literal in the clause from that decision level. Note that the sole requirement for a conflict clause to allow backjumping is to have an UIP from the current decision level. The simplest conflict analysis strategy, known as *First UIP*, is to resolve literals from the current level and stop as soon as the first UIP is found. Other possible strategies, which were considered by Zhang et al. (2001), are listed in Table 2.1; Last UIP is sometimes referred as the REL_SAT scheme since it was originally proposed and implemented by Bayardo and Schrag (1997) in the REL_SAT solver.

Another proposed idea was to actually find the shortest possible conflict clause from the chain of implications, this turns out to be equivalent to the vertex mincut problem in graphs. Finally GRASP employs a technique similar to First UIP, but where each intermediate clause in the resolution process is added to the formula (the optional step in Algorithm 2.7). The experimental study of Zhang et al. (2001) found First UIP to be the best learning strategy from those examined, although no convincing reason or explanation for that has been given.

In his master's thesis, Ryan (2004) observes that it is inappropriate to justify the superiority of First UIP just because it is easier to compute. Even the cost of All UIP is negligible compared to other operations such as unit propagation. Moreover, in his own experiments and implementation, First UIP consistently allows the solver to explore a much reduced search space compared to other approaches. He suggests a possible correlation between the average number of resolution steps required to derive conflict clauses (of which First UIP performs the fewer) and the overall solver performance. He also indicates that sometimes adding a few intermediate clauses is beneficial, but why and when this technique works is still largely unclear.

Another key observation from Ryan (2004) is that the implementation details of unit propagation and clause learning are very relevant to the actual performance of each other. In particular different implementations of unit propagation may select different antecedents for implied literals, resulting in different learnt clauses and, ultimately, in completely different search behaviours. Implementations engineered to increase the chances of finding of short antecedents, may also induce the generation of shorter and more useful conflict clauses.

Recently a further improvement for conflict analysis has been proposed by Eén and Sörensson (2005) applying the self-subsumption rule, already described in Section 2.3, to further simplify conflict clauses. This technique, called *conflict clause minimisation*, essentially checks if some of the literals in the conflict clause can be eliminated by applying self-subsumption with the implicants involved in the resolution process. The success of this idea has been empirically demonstrated in MINISAT at the SAT 2005 competition (Le Berre and Simon, 2005).

2.4.3 Clause database management

Clause learning is a crucial element in modern satisfiability solving that allows systems to prune vast regions of search space. However, by adding a clause to the formula each time a conflict is reached, and since the number of conflicts to solve an instance can be exponentially large in the worst case, the memory requirements grow exponentially while the search proceeds. Unless some clauses are eventually deleted, as the cost of unit propagation depends directly on the size of the formula, the raw performance of the algorithm will also quickly degrade.

We end up in a situation where the satisfiability problem also becomes a database management problem. Clauses have to be efficiently stored in memory allowing fast access to the literals in them. Moreover, since clauses are added and deleted on the fly, tasks such as garbage collection and explicit formula simplification need to be periodically performed.

A number of deletion policies have been proposed and implemented: in Grasp (Marques Silva and Sakallah, 1996a) large clauses are deleted as soon as they are not required to justify the current assignment; in zChaff (Moskewicz et al., 2001) conflict clauses that, upon backtracking, get about 100-200 unassigned literals are deleted; siege (Ryan, 2004) augments the same technique by periodically removing a random selection of clauses; BerkMin (Goldberg and Novikov, 2002) deletes a fraction of clauses that are either too old or too large, and have not participated actively in recent conflicts.

For the actual storage of the clause database, early implementations used pointer heavy structures such as linked lists which allowed to efficiently perform some database operations but were not very memory efficient. Most modern solvers store the clause database in a large array with clauses delimited by sentient values. This is not always a very flexible data structure but significantly reduces the memory requirements. The use of a contiguous area of memory also allows a more efficient utilisation of the processors' cache (Zhang and Malik, 2002; Mitchell, 2005).

Ryan (2004) goes even further by using only 21 bits to represent each literal so that three of them —rather than two using the usual 32 bits— fit within a 64-bit word. This, however, imposes a limitation on the number of atomic predicates that the solver is able to handle and becomes a significant limitation for some industrial applications.

Another important observation from Pilarski and Hu (2002) is that problem instances, particularly those arising from electronic design, often contain a large number of binary clauses. The use of a procedure such as the two watched literal scheme to perform unit propagation on those clauses is evidently an overkill. Following the same argument, Ryan (2004) proposes the use of specialised data structures to manage clauses of length two and three.

2.4.4 Decision heuristics

The Davis-Logemann-Loveland procedure, depicted in Algorithm 2.4, is nondeterministic in the sense that at each decision point it does not specify *which* literal should be chosen. Any choice strategy will produce the correct result but, as it was realised since the early days of satisfiability testing, different strategies do lead to very different sizes of spaces to search. Thus, the design of effective decision heuristics is crucial to achieve high levels of performance in a solver.

Most of the first heuristics (Jeroslow and Wang, 1990; Freeman, 1995) were focused on trying to select the literal that will produce greater simplification after unit propagation is applied. Several criterion were proposed to 'measure' the amount of simplification using literal counts and other statistics of the clauses in the formula. This kind of heuristics have, however, two main drawbacks: first they are expensive to compute and maintain (e.g. updating literal counts at each decision and backtracking point) and second it seems that, through these syntactic statistics, it is not always possible to capture the 'real structure' of the problem that is being processed. Moreover, in a comparative study from Marques Silva (1999) in the context of the first clause learning algorithms, it was found that simply selecting a literal at random is, in many cases, as good as the other heuristics that have been proposed so far.

Another strategy, also presented originally by Freeman (1995), is to perform the propagation of each literal and then choose the literal that effectively produced the greatest simplification of the formula. This technique, usually known as *lookahead*, has been implemented in SATZ (Li and Anbulagan, 1997) and more recently in MARCH_EQ (Heule et al., 2004) sometimes combined with other heuristics to avoid performing propagation on *all* literals. Although very effective on a few classes of structured and random problems, lookahead is often too expensive and the reductions in search space do not always compensate its cost.

After clause learning was introduced, Moskewicz et al. (2001) were the first authors to propose a heuristic that directly takes into account the information gained in the search space that has been already explored. They also argued that an effective decision strategy, since it is required at each node of the search tree, should be inexpensive to compute. The heuristic named *variable state independent decaying sum* (VSIDS) is easily incorporated into the clause learning framework, Algorithm 2.6, as follows:

- A counter, initialised to zero, is assigned to each literal.

- At each conflict, increment the counters of literals in the conflict clause.
- Periodically, divide all counters by a constant.

When a decision has to be made, the literal with the highest counter is selected. Since counters only have to be updated when a conflict occurs, the strategy has very little overhead.

Some variations of the strategy have been proposed, such as initialising counters with literal counts and using combined scores (e.g. of x and $\tilde{x}$) when making a decision. However a significant improvement was observed after a suggestion from Goldberg and Novikov (2002), implemented in BerkMin, that involves incrementing the counters not only of literals in the conflict clause, but also from all the clauses involved in the conflict analysis. The intuition is that this focuses even more the search in trying to solve recently generated conflicts. In particular, it was found useful to restrict the next decision to the literals in the most recent unsatisfied conflict clause.

Variable move to front (VMTF) is another strategy, proposed by Ryan (2004) in the siege solver, that maintains a list of predicates originally sorted according to their number of occurrences in the formula. Each time a conflict occurs, counters are incremented as in BerkMin and a random selection of predicates from the conflict clause are moved to the front of the list. For decisions, the first unassigned predicate from the list is selected, using the counters to decide which value to assign it first. One of the most important qualities of this approach is that it is extremely cheap to compute, since it avoids sorting scores to find the highest one, and has also been found to be very competitive in practise.

2.4.5 Restart strategies

A common phenomenon observed while performing experiments and benchmarking satisfiability solvers is that they suffer from a great variability on running times. Sometimes even a small change in the order of atomic predicates for decision making is the difference between solving a problem in hours instead of a few seconds. Gomes et al. (1997) proposed an explanation based on *heavy-tailed distributions*. In a few words many problems are usually solved within a reasonable amount of time, but the probability of sometimes requiring an extraordinarily large amount of time is not negligible.

In order to eliminate this undesired behaviour, the authors proposed to add some randomisation to the methods (e.g. in decision making) and restart the search frequently, with the hope to find one of the short solutions for the problem. This idea was effectively integrated later into zChaff (Moskewicz et al., 2001), BerkMin (Goldberg and Novikov, 2002) and most other modern solvers.

The actual details depend on the implementation, but it is important to note that clause learning solvers keep most (or at least some) of their learnt clauses between restarts. So that the previous search effort is not lost, while allowing the solver to escape from deep conflicts and start afresh searching into another space.

Another important detail to note is that when restarts and clause deletion are used together, the completeness of the algorithm is lost. There is nothing to guarantee that the solver will not keep learning and forgetting the same set of clauses forever. On some solvers completeness is retained by making restarts less frequently, or by keeping more and more clauses between restarts. The advantages of doing any of these, however, are not very clear since, in practise, solvers usually do not run long enough so that we can notice any difference.

As a concluding remark, restarts do not usually provide great speed gains or increased search efficiency. But they do provide robustness to the solver, making them capable of solving broader classes of instances compared to the same solver without restarts.

2.5 Stochastic local search

The use of local search and other approximate methods to solve computationally hard problems, can be traced back to algorithms for solving the travelling salesman problem (Lin, 1965). After having some success in the constraint satisfaction community (Minton et al., 1990) these ideas were later imported into satisfiability independently in the works of Selman et al. (1992) and Gu (1992).

The early ideas were fairly simple, and the authors of these first works were themselves astonished by their relative good performance. An initial truth assignment is built by assigning truth values to each atomic predicate at random. The procedure then counts the number of unsatisfied clauses and looks for a predicate which when *flipped*, i.e. changed from *true* to *false* or vice versa, provides the highest decrease of unsatisfied clauses. Such predicate is flipped and the process iterated. If at some point the number of unsatisfied clauses reduces to zero, then a solution for the problem has been found.

This simple procedure has many visible inconveniences. It is easy, for example, for the search to get stuck into some local minima and never find a solution. To overcome this, the procedure is periodically restarted with different random assignments with the hope of eventually finding a solution. More significantly, since the algorithm has no memory about the search space already explored, it cannot guarantee to always find a solution when one exists. Moreover, since it is also unable to prove the unsatisfiability of a formula, this kind of algorithms are often incomplete.

In certain applications, notably the case of planning, this might not be a strong issue; since problems are likely to have solutions and the main concern is just to find any of them. Moreover, since these simple ideas seemed to outperform complete methods in finding satisfying assignments, a great amount of research was spent on truly understanding them.

Algorithm 2.8 Stochastic local search algorithm

```
procedure SLS($F$)
    repeat $MaxTries$ times
        $A \leftarrow$ random truth assignment
        if $A$ satisfies $F$ return "Satisfiable"
        repeat $MaxFlips$ times
            pick an atomic predicate $p$ in $F$          ▷ choose candidate
            flip the value of $p$ in $A$
            if $A$ satisfies $F$ return "Satisfiable"
        end repeat
    end repeat
    return "Don't know"
end procedure
```

Algorithm 2.8 depicts the common structure of a stochastic local search satisfiability solver. Few research has been done into how to efficiently implement the mechanics involved in this procedure (but see e.g. Fukunaga, 2004). Most of the focus has been placed in designing and evaluating good heuristics to choose the atomic predicate to flip. Some of the most popular and successful approaches are presented in Algorithms 2.9 and 2.10. The following statistics are used in some of those algorithms: *Score*(p) is the number of remaining unsatisfied clauses if p is flipped; *BreakCount*(p) is the number of satisfied clauses that will become unsatisfied if p is flipped; and *TimeStamp*(p) is the iteration step at which the predicate p was last flipped.

Algorithm 2.9 Heuristics to choose candidate in local search (Part 1)

```
procedure GSAT                                              ▷ Selman et al. (1992)
    Best ← set of predicates that minimise Score
    pick an atomic predicate in Best at random
end procedure

procedure GWSAT                                             ▷ Selman et al. (1994)
    with probability $p$ do
        Best ← set of predicates occurring in unsatisfied clauses
    otherwise
        Best ← set of predicates that minimise Score
    end with
    pick a predicate in Best at random
end procedure

procedure WSAT                                              ▷ Selman et al. (1994)
    choose an unsatisfied clause $C$ at random
    pick a predicate in $C$ at random.
end procedure

procedure WalkSat/SKC                                       ▷ Selman et al. (1994)
    choose an unsatisfied clause $C$ at random
    Best ← set of predicates in $C$ that minimise BreakCount
    if BreakCount of predicates in Best is zero then
        pick a predicate in Best at random.
    else
        with probability $p$ do
            pick a predicate in Best at random
        otherwise
            pick a predicate in $C$ at random
        end with
    end if
end procedure
```

Selman et al. (1994) evaluated many of the early heuristics and proposed the original WalkSat approach, sometimes called the SKC variant after its authors, though it was not completely described in that early paper (presumably the first available description of the heuristic was the source code of the program itself). Significant improvements were obtained by McAllester et al. (1997) in Novelty. Strangely enough, Novelty was found to suffer from stagnation problems. Hoos

Algorithm 2.10 Heuristics to choose candidate in local search (Part 2)

```
procedure NOVELTY                                          ▷ McAllester et al. (1997)
    choose an unsatisfied clause $C$ at random
    sort the list of predicates in $C$ by $Score$, break ties with $TimeStamp$
    $Best1$ ← first predicate in the sorted list
    $Best2$ ← second predicate in the sorted list
    $Young$ ← predicate in $C$ with lower $TimeStamp$
    if $Best1 \neq Young$ then
        pick $Best1$
    else
        with probability $p$ do
            pick $Best1$
        otherwise
            pick $Best2$
        end with
    end if
end procedure

procedure NOVELTY$^+$                                              ▷ Hoos (1999)
    choose an unsatisfied clause $C$ at random
    with probability $wp$ do
        pick a predicate in $C$ at random
    otherwise
        continue using NOVELTY strategy ...
    end with
end procedure
```

(1999) found a theoretical explanation for such behaviour, by introducing the *probabilistically approximately completeness* (PAC), and showed that NOVELTY does not have such property. Through a very simple fix, just by adding random steps with some low probability, the PAC property is restored and the stagnation problem solved. This is the base of the NOVELTY$^+$ heuristic (Hoos, 1999).

Note that, common to most of these strategies, is the use of a parameter that allows to 'tune' the amount of noise introduced in the search. It was observed, however, that there is often no *optimal* tunning parameter; different families and classes of problems were better solved with different parameter settings. This lead Hoos (2002) into the idea of ADAPTIVE NOVELTY$^+$, an algorithm where those tunning parameters are set and modified on the fly as the search proceeds. When the search has been successfully reducing the number of unsatisfied clauses, the parameters are tunned for more *greediness*; in the other case, if no improvement has been found for a number of steps, more noise is introduced allowing the search to escape from local minima.

A final remark has to be made on the work of Tompkins and Hoos (2004), who have provided a clean and extensible implementation of a general stochastic local search solver UBCSAT. Together with particular instantiations of most of the local search procedures ever published in literature, this project has served as a solid framework for comparative experimentation between different approaches and the rapid evaluation of new ideas.

2.6 Chapter summary

In this chapter we have introduced most of the background information about propositional logic and the propositional satisfiability problem which is required to more clearly understand the rest of this thesis. In doing so, we have also aimed at providing a general picture of the state of the art in satisfiability solving technologies and implementations.

After giving first a formal definition of the syntax and semantics of propositional logic, we have detailed some of the normal forms which are commonly used to simplify the treatment of logical formulae both in theory and in practise. In particular we introduced Tseitin's translation which allows one to translate arbitrary propositional formulae into clausal normal form. In the following chapter we will propose a few improvements on this translation, as well as study the effect that varieties of the translation have on the performance of satisfiability solvers.

We then gave a survey of preprocessing methods which simplify the structure of propositional formulae. And this was followed by a detailed account on the theory and implementation techniques of the Davis-Logemann-Loveland algorithm currently used by most complete modern solvers. Finally a brief survey on stochastic local search methods, which are incomplete but also often more effective on satisfiable problems, was also given.

The following chapter will now introduce some of our contributions in the context of propositional logic. This includes a model to randomly generate propositional formulae which has a non-trivial structure and which are difficult for existing systems to solve. Then, we also have a look at clausal form translation methods, and propose a few ideas to improve on existing approaches.

Chapter 3

Encoding problems in propositional logic

This chapter describes the main thesis contributions in the context of propositional logic and propositional satisfiability. It consists of two main sections, the first one dealing with a model that we propose to randomly create non-clausal propositional formulae. Most of this section was submitted and accepted for publication at the Twentieth National Conference on Artificial Intelligence (Navarro Pérez and Voronkov, 2005), and a summary was also presented at the Workshop on Automated Reasoning 2005 (Navarro Pérez, 2005).

Our proposed model is useful to generate a large number of satisfiability problems which have a non-trivial structure and, at the same time, are particularly challenging for existing satisfiability solving technology. This makes them particularly suitable for benchmarking the latest and new emerging systems which try to exploit the structure information often present in problems derived from real-world applications. Moreover, we also use existing clausal solvers to measure the difficulty of the generated problems, while raising the question on the effect that clausal form translations have on these generated problems.

The second section of this chapter summarises several of our research results while studying encodings of problems in propositional logic. It includes first an improved version of Tseitin's translation which we devised while trying to reduce the negative effects that the addition of many new predicates tends to produce. This new translation, in particular, raises the idea that perhaps satisfiability solvers, able to deal with kinds of constraints more general than simple clauses, might perhaps be better suited to exploit the structure and symmetries of problems. Early ideas on this direction were also presented at the Workshop on Automated Reasoning 2006 (Navarro Pérez, 2006). We finally describe a brief study on how properties of different translations and simplifications affect the actual performance of solving procedures, in an attempt to figure out what constitutes a *good* encoding for a problem in propositional logic.

3.1 Generating hard non-clausal problems

Randomly generated formulae have often been used as benchmarks to evaluate the performance of satisfiability solving procedures. However it is important, as already pointed out by Mitchell et al. (1992) as well as Mitchell and Levesque (1996), to have a clear understanding of the distribution of such formulae in order to avoid reaching incorrect conclusions from deceiving experimental results. An algorithm may quickly solve several thousands of problems not because it is clever or effective but, unfortunately, because of a poor sampling mechanism that has a tendency to produce easy problems.

A model that has been recognised as being able to produce challenging benchmarks for the satisfiability problem is random k-SAT. Formulae are produced by randomly selecting clauses of length k built from a set with a given number of atomic predicates. For one parameter of the model, namely the ratio between the number of clauses and the number of predicates, an interesting pattern has been observed: the sets of generated formulae exhibit a sharp transition between almost all being satisfiable to almost none. Moreover, problems generated near this critical region are hard to solve for all existing systems, while problems far from it are either easy or only moderately hard. Researchers have shown a lot of interest in the study of random k-SAT and related problems (such as determining bounds for the critical region) and, at the same time, hard random 3-SAT formulae became a standard benchmark for testing satisfiability procedures.

The results of the SAT Competition (Le Berre and Simon, 2004), where new and state-of-the-art solvers are tested against several benchmarks, have shown that the best solvers on random 3-SAT are not necessarily the most effective on real-world applications and vice versa. One of the possible explanations is that such random formulae, which are just large sets of short and independent clauses, are unable to simulate problems with some kind of structure.

This section introduces one of our first contributions, a generalisation of the random k-SAT model, that is useful to produce test formulae with non-trivial structure. Our proposed *fixed shape model* is based on the idea of generating non-clausal formulae, i.e. arbitrary propositional formulae not necessarily in clausal normal form, and has several interesting features. First it produces a family of instances controlled by a number of parameters, allowing to evaluate solvers under different settings including critical conditions. Examples from real-world applications usually do not allow this amount of control. At the same time, our proposed model produces instances with some level of structure. We would like to point out that our motivation is not to produce 'harder' problems, but to provide non-clausal instances to serve as benchmarks for emerging solvers that try to exploit problem structure or directly work with the non-clausal representation.

After the definition of our proposed model, we experimentally study the probability distribution of the generated formulae and observe interesting features such as a sharp phase transition and the existence of hard problems in a critical region. We also report the results on some experiments that we performed to compare the performance of different state-of-the-art solvers in combination with two clausal form translations. We address the question of how the choice of a translation affects the properties of the generated problems and the performance of the solvers when trying to solve them. Our results point out that no translation is always better than the other, and that more research in this direction is needed.

3.1.1 The fixed clause-length model

In this section we present the fixed clause-length model, also known in literature as random k-SAT, that is used to generate random clausal formulae. This model has three parameters: the number of atomic predicates n, the number of clauses m and the length k of the clauses to be produced. The parameter $r = m/n$, the ratio of clauses to predicates, is often used instead of m to describe particular instances of the model.

A formula is generated by selecting clauses uniformly at random from the set of all clauses of length k. Slight variations of the model are found depending on whether trivial clauses (with complementary or repeated literals) are allowed or not. This, however, does not seem to affect the general behaviour of the distribution. In extensive research on random k-SAT (Mitchell et al., 1992; Mitchell and Levesque, 1996; Cook and Mitchell, 1997) two main features are frequently pointed out:

Sharp phase transition: For each k and n, the probability that a generated formula is satisfiable changes, as the value of r increases, from almost 1 to almost 0 in a very narrow region. Moreover, as the value of n increases, the transition seems to take place in a narrower area around some *crossover* point r^*. Friedgut (1999) was able to show that, indeed, the size of the critical region shrinks as n increases. His theoretical result, however, does not give any clues about the value of r^*, or even if such a value should actually exist. For the case of random 3-SAT, experimental evidence suggests a value near 4.25. Bounds for the crossover region are also known: $3.52 < r^* < 4.506$ (Kaporis et al., 2003; Dubois et al., 2000).

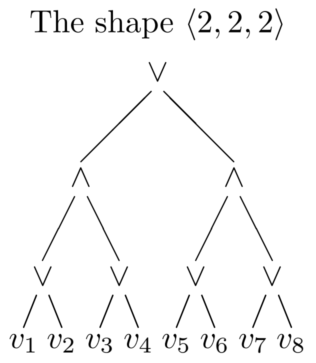
The easy-hard-easy pattern: The difficulty of the generated problems (usually measured as the number of branches explored by a DLL-based algorithm) exhibits a pattern that goes from very easy, for small values of r, to very hard, when r enters the phase transition, to easy (or moderately hard) when r becomes large. These phenomena is usually explained by the fact that, for low values of r, a formula with few clauses is under-constrained and very easy to satisfy. On the other hand, for large r, the formula is over-constrained and a complete satisfiability checking procedure can quickly find contradictions to finish the search. The hardest problems appear in the transition region where there are just enough clauses to make the problem potentially unsatisfiable, but not too many to make it easy for a solver to determine. The difficulty of a particular distribution of formulae clearly depends on the procedure used to solve it, but several authors have conjectured that this general pattern will hold for any reasonable complete method (Cook and Mitchell, 1997).

3.1.2 The fixed shape model

Our proposed model is closely related to the fixed clause-length model introduced in the previous section. We follow the same idea to go from under- to over-constrained areas but, instead of clauses of a fixed length, we use formulae generated according to a particular fixed shape.

In the following definition we use Σ to denote a set of atomic predicates, and $Lits_\Sigma$ to denote the set of literals that are built using such predicates.

Definition 3.1. A *shape* is a propositional formula S such that (i) S is built using the conjunction and disjunction connectives only; and (ii) every predicate appearing in S has exactly one occurrence in it. A Σ-*instance* of a shape is any


The shape $\langle 2,2,2\rangle$
$\vee$
$\wedge$ $\wedge$
$\vee$ $\vee$ $\vee$ $\vee$
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8

Sample instances:

$$((\neg x_3 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_3))$$
$$\vee((\ x_9 \vee \neg x_5) \wedge (\ x_3 \vee \ x_7))$$

$$((\neg x_6 \vee \ x_7) \wedge (\neg x_6 \vee \ x_2))$$
$$\vee((\neg x_9 \vee \neg x_7) \wedge (\neg x_1 \vee \ x_4))$$

$$((\ x_4 \vee \neg x_4) \wedge (\ x_8 \vee \ x_6))$$
$$\vee((\ x_7 \vee \ x_4) \wedge (\ x_4 \vee \ x_2))$$

Figure 3.1: Structure of the shape $\langle 2,2,2\rangle$ and sample instances

formula obtained by replacing every predicate in the shape by a literal from the set $Lits_\Sigma$. A *randomly generated* Σ*-instance* of a shape $\mathcal{S}$, is a formula obtained by independently and uniformly choosing literals from the set $Lits_\Sigma$ to replace each predicate occurring in $\mathcal{S}$. ■

In the sequel we assume that Σ is clear from the context and simply use the term instances instead of Σ-instances. The formula $(v_1 \wedge v_2) \vee v_3$ is an example of a shape. Two $\{x_1, x_2, x_3, x_4\}$-instances of this shape are $(\neg x_3 \wedge x_2) \vee \neg x_1$ and $(\neg x_4 \wedge x_3) \vee x_4$. Let us introduce a special kind of shape, called *balanced conjunctive-disjunctive shapes*; informally these are balanced trees of alternating conjunctions and disjunctions.

Definition 3.2. Given d integers $k_1, \ldots, k_d$ (with $d \geq 0$ and $k_i \geq 2$) we define two sets of formulae $[\![k_1, \ldots, k_d]\!]$ and $\langle k_1, \ldots, k_d\rangle$ recursively as follows.

1. If $d = 0$, then the formulae in both $[\![\,]\!]$ and $\langle\,\rangle$ are literals.

2. If $d \geq 1$ then every formula in $[\![k_1, k_2, \ldots, k_d]\!]$ is a conjunction of k_1 formulae in $\langle k_2, \ldots, k_d\rangle$. Likewise, every formula in $\langle k_1, k_2, \ldots, k_d\rangle$ is a disjunction of k_1 formulae in $[\![k_2, \ldots, k_d]\!]$. ■

If we have a large enough set of predicates Σ, then every set $\langle k_1, \ldots, k_d\rangle$ contains a shape S; moreover $\langle k_1, \ldots, k_d\rangle$ is the set of all instances of this shape (and similar for $[\![k_1, \ldots, k_d]\!]$). For this reason we will sometimes refer to $\langle k_1, \ldots, k_d\rangle$ as a *balanced disjunctive shape* and to $[\![k_1, \ldots, k_d]\!]$ as a *balanced conjunctive shape.* The value d is called the *depth* of the shape.

Note that these balanced shapes and their instances are formulae in negation normal form (NNF); Figure 3.1 presents one example. Moreover every formula in negation normal form is an instance of some shape. We define now the *random $\langle k_1, \ldots, k_d \rangle$-SAT* model as follows: The parameters are the number of predicates n and a real number r. A formula is produced as the conjunction of $[rn]$ randomly generated $\{x_1, \ldots, x_n\}$-instances of $\langle k_1, \ldots, k_d \rangle$, where $[rn]$ denotes the integer closest to rn. Note that the case $\langle k \rangle$ gives us exactly the random k-SAT model. Moreover, there is no need to consider the model of random $[\![k_1, \ldots, k_d]\!]$-SAT since this would be equivalent to $k_1[rn]$ random instances of $\langle k_2, \ldots, k_d \rangle$.

Several properties of balanced shapes are useful to characterise the hardness of the generated random formulae. The following theorem, for example, is useful to compute the probability that one of the generated formulae will be satisfied by an arbitrary truth assignment.

Theorem 3.1. *Let t be an arbitrary but fixed truth assignment. The probability $p_{\langle k_1, \ldots, k_d \rangle}$ that t satisfies a random instance of $\langle k_1, \ldots, k_d \rangle$ can be computed as follows:*

$$p_{\langle\rangle} = 1/2\,,$$
$$p_{\langle k_1, \ldots, k_d \rangle} = 1 - (p_{\langle k_2, \ldots, k_d \rangle})^{k_1}\,.$$

Proof. The probability is easily obtained, using very simple combinatorial arguments, as the number of instances of the shape that are satisfied by the fixed truth assignment divided by the total number of instances of the shape (with respect to a set Σ with a fixed number of predicates). □

Intuitively shapes with a value of p very close to 0 are very hard to satisfy, so a fewer number of them are sufficient to make a randomly generated problem unsatisfiable. Conversely a value of p very close to 1 would make a random instance quite easy to satisfy, so only very large formulae have a chance of being unsatisfiable. The latter effect has been experimentally observed on random k-SAT for large values of k (Mitchell and Levesque, 1996) and is confirmed by analytical lower bounds of the crossover region (Achlioptas and Peres, 2003).

Theorem 3.2. *The probability that a random instance of $\langle k_1, \ldots, k_d \rangle$-SAT is satisfiable, with n predicates and density r, tends to 0 as $n \to \infty$ for all values of $r > \log 2/\log(1/p)$. Where p is computed as in Theorem 3.1.*

Proof. A fixed truth assignment t satisfies a conjunction of $[rn]$ instances of $\langle k_1, \ldots, k_d \rangle$ with probability $p^{[rn]}$. The expected number of satisfying assignments is therefore $2^n p^{\lceil rn \rceil}$. This value (and thus the probability of the instance being satisfiable) tends to 0 as $n \to \infty$ when $r > \log 2 / \log(1/p)$. $\square$

This simple argument, useful to estimate the location of the critical region, has also been used to give an easy upper bound of the random k-SAT crossover point (Cook and Mitchell, 1997).

3.1.3 Translation to clausal normal form

While the formulae generated by our proposed model are non-clausal, modern satisfiability solving procedures are designed under the assumption that their input is a clausal formula. There is a recent interest on the design of non-clausal satisfiability testing algorithms (Thiffault et al., 2004; Giunchiglia and Sebastiani, 2000; Stachniak, 2002), but mature implementations are not readily available (see Section 3.1.5). So, in order to measure the *hardness* of our formulae we decided to translate them into clausal normal form first and then use a standard clausal solver. This raises the important question on how the choice of a particular translation affects the performance of existing solving procedures.

To test our formulae we used two kinds of translations. The *standard translation* (equivalence preserving) is simply based on distributive properties of disjunction and conjunction. It is well known, as we already saw on Section 2.2, that such translation causes an exponential increase in the size of the problem.

Theorem 3.3. *The standard translation of the balanced shape $\langle k_1, \ldots, k_d \rangle$ produces a CNF formula with clauses of the same length. Moreover, the length is the product of all the k_i with i odd.*

Table 3.1, in particular the left column under the 'length' header, illustrates this theorem showing the clause lengths of several shapes. The second translation we consider is an *optimised translation* (structure preserving). It uses the so called *naming technique* of Tseitin's translation, also discussed in Section 2.2, which avoids the exponential size increase by introducing new predicates.

Given a clausal formula $F = C_1 \wedge \cdots \wedge C_n$, i.e. each C_i is a clause, the *name of the formula F*, denoted p_F, is a fresh new predicate and its *clausal definition* is the clausal formula $\Delta_F = (\neg p_F \vee C_1) \wedge \cdots \wedge (\neg p_F \vee C_n)$. The formula Δ_F, which is equivalent to $p_F \to F$, is used to fix the meaning of the newly introduced predicate in the problem. Also if $n = 1$ the new name is not required, in that case we let p_F denote the formula F itself and let Δ_F be the 'empty' formula.

The translation works in a bottom up fashion, translating each child subformula into CNF before the parent. The naming technique is used to translate a disjunction $F_1 \vee \cdots \vee F_n$ of CNF formulae into $p_{F_1} \vee \cdots \vee p_{F_n}$ and adding the clauses in $\Delta_{F_1}, \ldots, \Delta_{F_n}$ to a stack of definitions. It is easy to show (see e.g. Tseitin, 1968; Plaisted and Greenbaum, 1986) that the final translated formula augmented with the stack of definitions is satisfiable if and only if the original propositional formula is.

The optimised translation has the main advantage of keeping the size of the translated formula small (linear with respect to the original), at the cost of introducing new predicates. It was interesting to see how modern solvers cope with this increase in the number of predicates and to determine whether the introduced optimisations are useful or not.

3.1.4 Experimental study

In order to experimentally observe the distribution of our randomly generated formulae we started by running small simulations with different shapes and parameter values on a variety of solvers. Table 3.1 shows properties of formulae tested at this stage. In this table, 'preds' is the number of predicates in the shape and p is the probability that a random instance is satisfied by a truth assignment, see Theorem 3.1. Then r_u is an upper bound of the crossover region, see Theorem 3.2. The 'weight' of each shape is the product of the number of predicates and r_u; this serves to estimate the magnitude (i.e. the number of literals) that the generated formulae would have in the hard region.

Consider for example the $\langle 2,4,2 \rangle$ shape. Although it is bigger and more complex than a simple clause of length 4, we only need a few instances of them ($\lceil 1.1n \rceil$ instead of $\lceil 10.8n \rceil$) to produce formulae which are difficult to solve. Low weight shapes are interesting because they seem appropriate to generate hard and short problems. The 'length' header has two columns; the left one shows the length of the resulting clauses for the standard translation, as in Theorem 3.3; while the right one shows the average clause length for the optimised translation. Finally 'fresh' is the number of fresh predicates introduced by the optimised translation for each generated instance of the shape.

Using this information we designed several experiments whose results we detail now. At this stage we considered four solvers: zChaff (2004.5.13), a carefully engineered implementation of the DLL procedure (Moskewicz et al., 2001);

shape	preds	p	r_u	weight	length		fresh
$\langle 3, 2\rangle$	6	0.578	1.26	7.59	3	2.14	3
$\langle 3\rangle$	3	0.875	5.19	15.57	3	3.00	0
$\langle 2, 4, 2\rangle$	16	0.533	1.10	17.61	4	2.89	2
$\langle 6, 3\rangle$	18	0.551	1.16	20.95	6	2.21	6
$\langle 2, 2, 3, 2\rangle$	24	0.557	1.18	28.41	6	2.27	14
$\langle 4\rangle$	4	0.938	10.83	43.32	4	4.00	0
$\langle 3, 3, 2\rangle$	18	0.807	3.23	58.11	6	3.00	3
$\langle 2, 2, 4, 2\rangle$	32	0.716	2.08	66.46	8	2.32	18
$\langle 2, 5, 3\rangle$	30	0.763	2.56	76.78	6	3.82	2
$\langle 5\rangle$	5	0.969	21.83	109.16	5	5.00	0
$\langle 2, 2, 2, 2, 2\rangle$	32	0.880	5.42	173.51	8	2.95	10

Table 3.1: List of properties of some balanced shapes

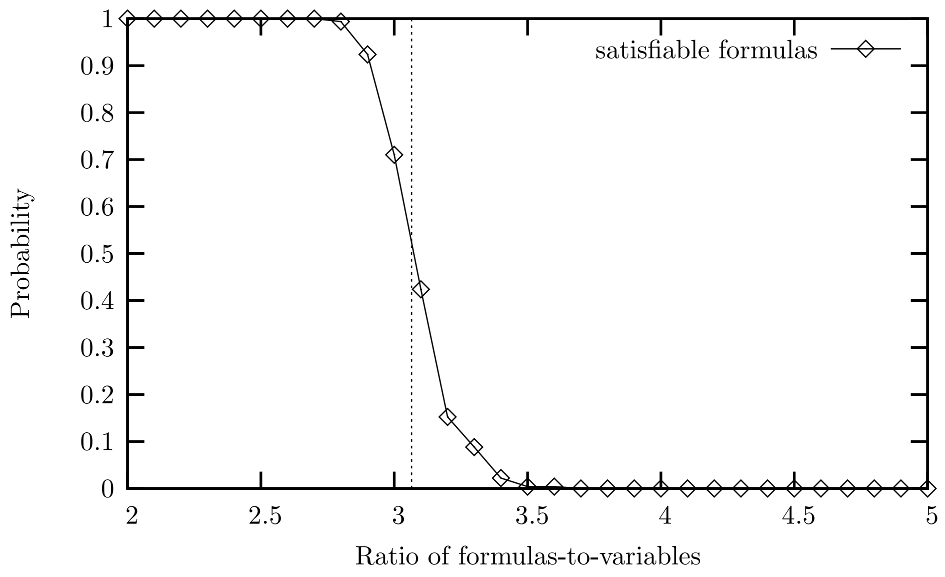


Figure 3.2: Probability of satisfiability of random $\langle 3, 3, 2\rangle$-SAT with 70 predicates

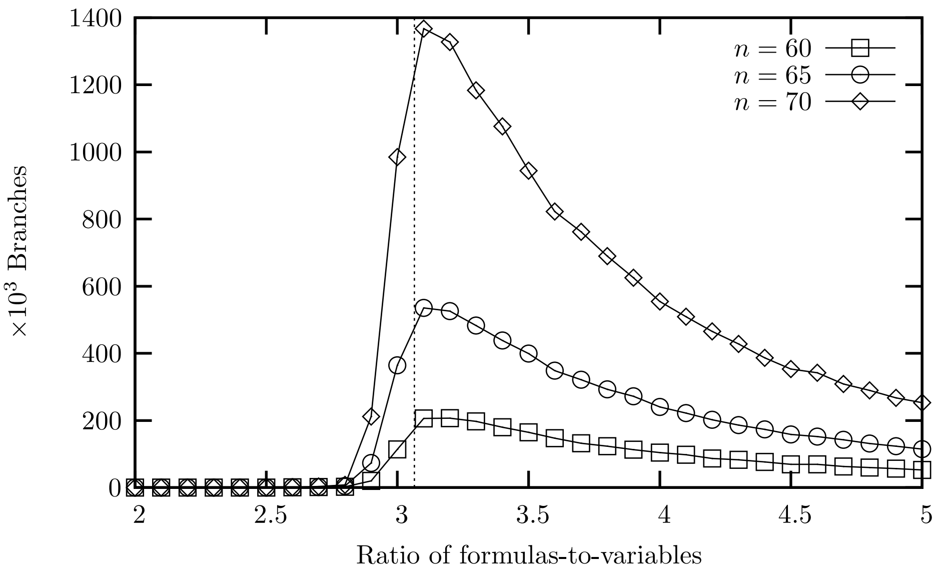


Figure 3.3: Branches explored by zCHAFF on random $\langle 3,3,2\rangle$-SAT with 70 predicates

MARCH_EQ (2004.3.20, 100% lookahead), which integrates equivalence reasoning techniques (Heule et al., 2004); KCNFS (2003.2.12), a solver with efficient heuristics to solve random k-SAT formulae (Dubois and Dequen, 2001); and the ADAPTIVE NOVELTY$^+$ stochastic local search algorithm (Hoos, 2002) implemented in the UBCSAT (2004.07.27) experimentation environment (Tompkins and Hoos, 2004). These solvers were selected using the results of the SAT Competition 2004 (Le Berre and Simon, 2004) as a reference. Moreover, we wanted to use very diverse solvers in order to observe how different strategies and clausal form translations perform in this setting with mixed randomness and structure. The experiments were run in parallel on 45 computers, each having an Intel III 1GHz CPU and 512Mb RAM.

In a first experiment we performed an analysis of random $\langle 3,3,2\rangle$-SAT formulae generating 500 samples for each parameter value. The purpose of this experiment was to obtain an accurate description of the probability distribution of this shape. Figure 3.2 shows an already familiar picture: the probability that a generated formula is satisfiable changes from almost 1 to almost 0 in a narrow region around the 0.5 probability point, in this case close to $r = 3.07$.

Figure 3.3 shows the median of the number of branches explored by zCHAFF when solving these formulae. The easy-hard-easy pattern is reproduced with the hardest problems near the crossover point. The same basic pattern was found

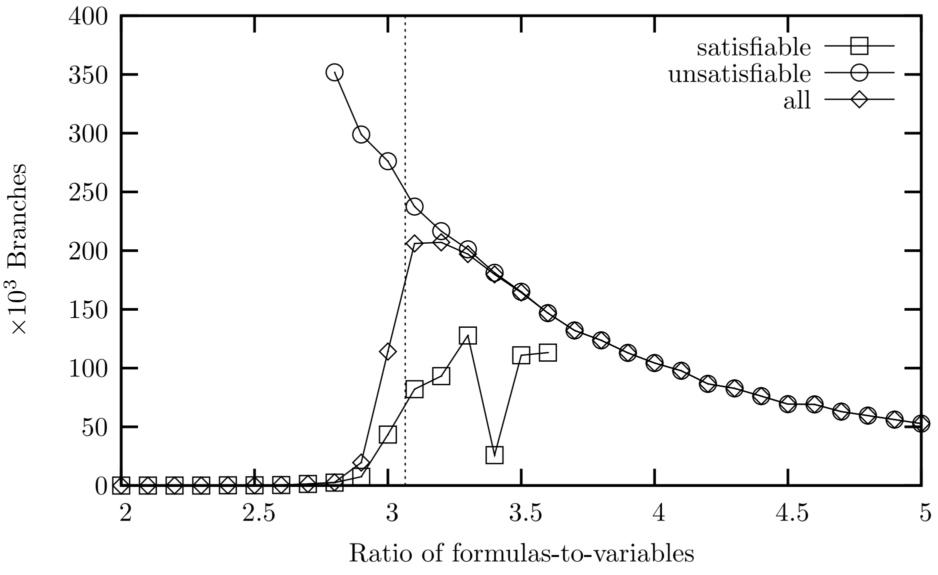


Figure 3.4: Branches explored by zChaff on random $\langle 3,3,2\rangle$-SAT with 60 predicates, factored by satisfiability

in all our experiments with different solvers and shapes. Compared to analogous results on random 3-SAT, the transition from easy to hard is much more sudden (increasing from a few hundred to more than 1.3 million branches in a region of length 0.3), while the decay after leaving the critical region is gradual and slow. Figure 3.4, which presents these results factored into satisfiable and unsatisfiable groups, suggests that most of the satisfiable formulae are rather easy to solve; while the unsatisfiable ones, several order of magnitudes harder, dominate the behaviour of the curve as soon as they appear. This figure also shows, however, that the few satisfiable formulae to the right of the crossover point also sometimes have a significant difficulty.

Using a more intense sampling near the critical region (1000 test cases per data point) we observed the so called scaling window effect. Let ϵ be a real number ($0 < \epsilon < 0.5$), the ϵ-*window* is the interval of values of r where the probability of satisfiability lies within ϵ and $1 - \epsilon$. Figure 3.5 shows how the length of the 0.1 and 0.01-windows (the former with a thicker plot line) decreases as the value of n increases; the crossover point is also marked with a small circle. This serves to provide observable evidence that sharp phase transition is likely to occur.

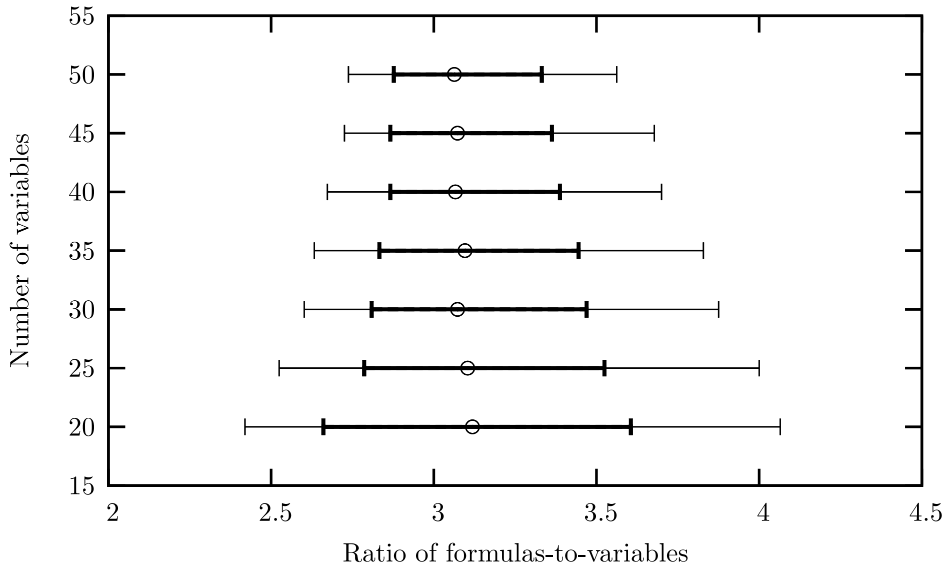


Figure 3.5: Scaling 0.1 and 0.01-windows on random $\langle 3, 3, 2\rangle$-SAT formulae

Local search methods

The ADAPTIVE NOVELTY$^+$ algorithm was considered to evaluate the effectiveness of local search methods for solving this class of formulae. Recall that this is an incomplete procedure and thus it is only able to find solutions of satisfiable instances. This imposes some limitations on the kind of experiments that we can perform since, for example, it solves *all* the satisfiable instances of the previous experiment in just a few minutes. The number of predicates had to be increased in order to obtain more significant data to be analysed. But this, in turn, makes it impossible the use of complete solvers to filter out unsatisfiable instances in a reasonable amount of time. We decided to generate 500 formulae with 140 predicates for each parameter value where, according to Figure 3.2, some satisfiable instances were expected to be found.

One of the first observations that we made is that the choice of a clausal form translation has a direct impact on the raw efficiency of the solver. It performs about 1 million flips per second on formulae obtained with the standard translation. While the shorter formulae produced by the optimised translation allow up to 11.9 million flips per second. Taking this into account, the cutoff parameter was set giving each of the two translations roughly the same amount of CPU time to solve each problem.

Table 3.2 shows the percentage of satisfiable formulae found with each trans-

r	standard	optimised
2.0	100.0%	100.0%
⋮	⋮	⋮
2.8	100.0%	99.6%
2.9	99.6%	91.6%
3.0	80.4%	41.0%
3.1	32.4%	12.2%
3.2	8.6%	3.2%
3.3	0.2%	0.2%

Table 3.2: Success rate of ADAPTIVE NOVELTY$^+$ on random $\langle 3, 3, 2\rangle$-SAT with 140 predicates

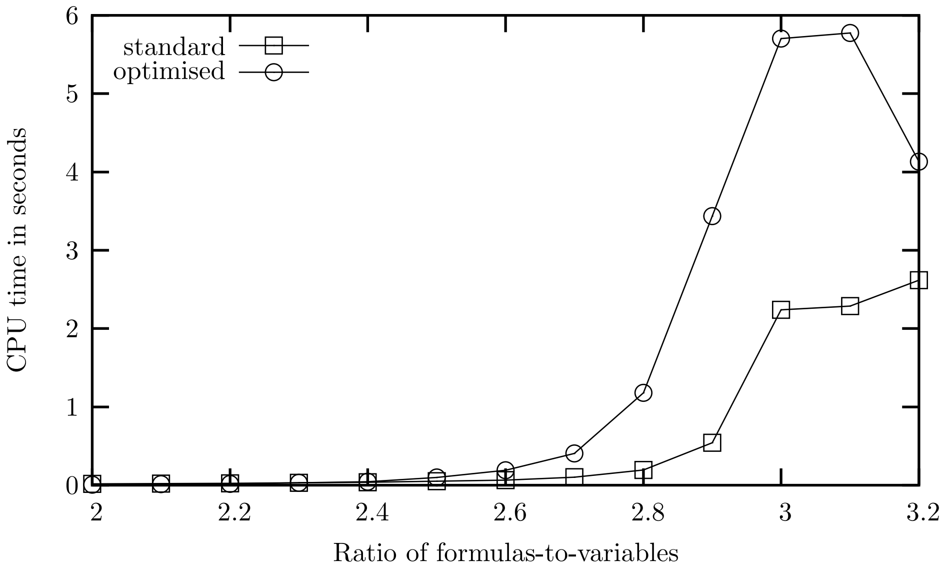


Figure 3.6: Average CPU time of ADAPTIVE NOVELTY$^+$ on random $\langle 3, 3, 2\rangle$-SAT with two different translations

Translation	clauses	length	predicates
standard	2240	4	140
optimised	1260	2.89	420

Table 3.3: Statistics on the translations of random $\langle 2,4,2\rangle$-SAT with 140 predicates and a value of $r = 1.0$

Translation	zChaff	march_eq	kcnfs
standard	431.5 min	58.3 min	14.8 min
optimised	722.8 min	31.9 min	19.1 min

Table 3.4: CPU time for different solvers and translations on random $\langle 2,4,2\rangle$-SAT with 140 predicates

lation at several settings for the parameter r. It shows that the standard translation, despite of its inferior raw performance, is far more effective in finding solutions than the optimised one. Moreover, as Figure 3.6 shows, the CPU time required to find these solutions is also considerably smaller. This results show how the reductions achieved by the optimised translation, at the cost of the introduction of many new predicates, adversely affects the overall effectiveness of the solver.

Complete methods

We also wanted to compare the performance of the complete solvers with respect to the clausal form translation applied. For this test we generated a smaller set of problems (50 samples per point) with instances of random $\langle 2,4,2\rangle$-SAT. In this case the crossover point was found near the $r = 1.0$ sample. Table 3.3 shows some statistics that compare the clausal representations provided by the two translations. In Table 3.4 the total CPU time usage of the solvers on each translation is shown.

Figure 3.7 shows the relation between the two translations for several values for r. The symbol $\mathsf{branch}(x, y)$ denotes the total number of branches explored while solving the 50 problems, with a fixed value of r, for each combination of a solver x and a translation y. The proportion $\mathsf{branch}(x, \mathsf{opt})/\mathsf{branch}(x, \mathsf{std})$ helps to provide a fair comparison indicating how the use of the optimised instead of the standard translation improved (< 1) or deteriorated (> 1) the performance of the solver. It is quite surprising that no translation was found decisively better than the other.

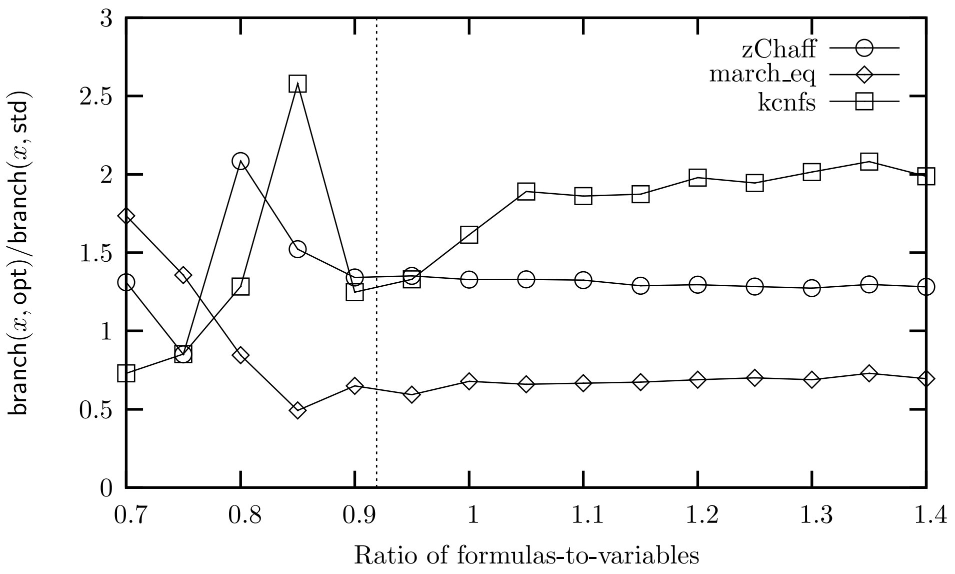


Figure 3.7: Effectiveness of the optimised translation on random $\langle 2, 4, 2\rangle$-SAT

The solvers zChaff and kcnfs showed a better performance with the standard translation and, conversely, march_eq found more useful the optimised one. We suspect that, since march_eq incorporates equivalence reasoning, the use of the optimised translation helps the solver to figure out the structure of the problem. While, for the other two solvers, the introduction of new predicates by the optimised translation has the undesirable effect of increasing the total space that needs to be searched. It is worth mentioning that we also performed some experiments with other shapes and parameter values, where the general observations already discussed in this section were also found.

3.1.5 Related work

The study and development of non-clausal procedures for the satisfiability problem is quite recent. Some authors have initiated a search for tractable classes of non-clausal problems (Altamirano and Escalada-Imaz, 2000), while others look for possible ways to generalise the DLL method (Thiffault et al., 2004; Giunchiglia and Sebastiani, 2000; Stachniak, 2002). It would have been interesting to test our formulae with the system NoClause developed by Thiffault et al. (2004). We encountered, however, some portability issues with the current version of the software that prevented us from doing some experimentation. In the work of Stachniak (2002) a first attempt to build hard non-clausal formulae is made, they are instances of $[\![2, 2, [rn], 3]\!]$, however no evidence of sharp phase or difficulty patterns were reported.

Another interesting related work is an analytical study on the satisfiability of randomly generated and-or trees, similar to the formulae presented here but with randomness applied to the structure, developed by Luby et al. (1998). Also, after the publication of our work (Navarro Pérez and Voronkov, 2005), new tighter upper bounds for the critical region of random $\langle k_1, \ldots, k_d \rangle$-SAT were computed by Santillán Rodríguez (2007).

Although most of the research on randomly generated SAT problems is focused on the random k-SAT model, other variants are also found in literature. Monasson and Zecchina (1997) proposed a random $(2+p)$-SAT model that, based on insights from statistical mechanics, mixes 2- and 3-SAT clauses. Other variable length models have also been considered (constant probability, or expected length) but they were found unsuitable for the production of hard problems (Mitchell et al., 1992; Mitchell and Levesque, 1996).

Generation of structured hard instances for the SAT problem has usually been done by translating problems from other domains (graphs, combinatorics, optimisation, etc.) into propositional formulae. Other generators, such as XOR-SAT (Barthel et al., 2002), are particularly designed to produce only satisfiable instances. There has also been a lot of interest in the more general setting of random constraint satisfaction problems (Gent et al., 2001), and generalised satisfiability problems (Creignou and Daude, 2002).

3.1.6 Results and conclusions

Extensive research on the satisfiability problem has lead to a deeper understanding of this and many other important problems in AI and related fields. Very efficient implementations of solving procedures are now easily available, and the performance of state-of-the-art solvers keeps improving each year. We believe that, in the near future, a great deal of research effort will be spent on the development of new theories and procedures that, extending current known approaches, will be able to handle general classes of formulae that encode information in a more succinct and efficient way.

In this section we have presented a model that generates hard non-clausal random formulae. We expect this procedure to provide diverse and challenging material to evaluate the performance of current and next generation solvers that have started to introduce non-clausal features. An example is the system developed by Muhammad and Stuckey (2006), who have already used our model as a source of benchmarks. We have carried out a careful experimental observation of the properties exhibited by these formula distributions where the sharp phase phenomenon and easy-hard-easy patterns were found.

Another contribution of this work, not done before to the best of our knowledge, is a first study on how the use of a particular clausal translation affects the performance of existing clausal solving procedures. Moreover, although these results apply only to our randomly generated formulae, they raise interesting questions on how to efficiently deal with real-world problems which are often described in a non-clausal setting.

3.2 Encoding real-world problems

In order to simplify its design and implementation, modern satisfiability solvers typically work on formulae in *clausal normal form* so that arbitrary propositional formulae first need to be translated into this format. This has long been the standard approach since the translation proposed by Tseitin (1968), and later improver by Plaisted and Greenbaum (1986), allows to efficiently translate arbitrary propositional formula into an equisatisfiable set of clauses.

It has been noted, however, that this translation tends to introduce many unnecessary atoms. One of the simplifications of the SatELite preprocessor (Eén and Biere, 2005), for example, is specifically targeted to identify atoms introduced by Tseitin's translations, as introduced in Section 2.2, and 'undo' their introduced definitions. This motivated our interest to design a translation which, while still producing a result of polynomial size with respect to the input, tries to introduce as few new atoms as possible.

This section explores first some ideas in this direction, on the definition of a more economical clausal form translation, and later discusses a possibility to extend satisfiability solvers to deal with more kinds of constraints other than just clauses. Valuable lessons are learnt from this research work, which suggests that a more clear understanding on the properties of real-world problems, and their clausal encodings, is needed. This section finishes with reflections on how to characterise and evaluate the properties of propositional problem encodings.

3.2.1 An improved clausal form translation

In the following we assume that logical connectives are not only binary, but that they have multiple arity. In particular we will allow n-ary (with $n \geq 2$) conjunctions, disjunctions and parity formulae. A parity formula is an expression of the form $A_1 \oplus \cdots \oplus A_n$ which evaluates to true under an interpretation if an odd number of the A_i subformulae evaluate to true. Note that $A \oplus B$, a binary parity formula, corresponds to the connective of exclusive disjunction, and that an equivalence $A \leftrightarrow B$ can be rewritten as $\neg A \oplus B$.

Moreover, we will also assume that these connectives are already *flattened*; this implies that a formula such as $((a \wedge b) \wedge c)$ is identified with $(a \wedge b \wedge c)$. We also assume that the *true* and *false* constants do not appear as a subformula. It is easy to see that these constants, as in Section 2.2, can be eliminated by evaluation of the formulae they appear in. The only case where the $\top$ or $\bot$ connectives are allowed is when the entire formula is simplified to one of them.

We do not explicit assume or disallow the commutativity of our connectives (i.e. whether two formulae that only differ by a permutation of their operands are identified with each other or not), this might be an implementation dependant decision of the translations presented here.

Extended normal forms

We introduce now a particular negation normal form that, unlike the standard definition given in Section 2.2, still allows the use of parity formulae in the final reduced expression.

Definition 3.3. A formula A is in *negation normal form*, or simply *NNF*, if the formula is either $\top$, or $\bot$, or is built from literals using the connectives of conjunction, disjunction and parity only. The negation connective, in particular, is only allowed to appear in front of an atom to form a literal.

As usual, a formula B is called a *negation normal form* of a formula A if B is equivalent to A and B is in negation normal form. ■

Algorithm 3.1 (Negation normal form translation)**.** The rewrite rule system on formulae of Figure 3.8 gives an algorithm to translate arbitrary propositional formulae into a negation normal form representation. Given an input formula, the algorithm simply applies the rewrite rules in a nondeterministic manner until a negation normal form is obtained.

$$
\begin{aligned}
\neg(A_1 \wedge \cdots \wedge A_n) &\quad\Rightarrow\quad (\neg A_1 \vee \cdots \vee \neg A_n) && (3.1)\\
\neg(A_1 \vee \cdots \vee A_n) &\quad\Rightarrow\quad (\neg A_1 \wedge \cdots \wedge \neg A_n) && (3.2)\\
\neg(A_1 \oplus A_2 \oplus \cdots \oplus A_n) &\quad\Rightarrow\quad (\neg A_1 \oplus A_2 \oplus \cdots \oplus A_n) && (3.3)\\
(A_1 \leftrightarrow A_2) &\quad\Rightarrow\quad (\neg A_1 \oplus A_2) && (3.4)\\
\neg\neg A &\quad\Rightarrow\quad A && (3.5)
\end{aligned}
$$

Figure 3.8: Rewrite rules for negation normal form translation

The reader can easily verify that the rewriting rule system is terminating and that the resulting expression is, indeed, a negation normal form. Similar to standard normal form translations, this algorithm works by pushing negation inwards into other connectives. In particular note that rule (3.4) allows to completely remove equivalence and, when applying rule (3.3), only one of the operands of the parity constraint needs to be modified.

Remark. This negation normal form has some useful properties. In particular the effect of the negation connective is now much simpler, and we do not need to explicitly take into account the *polarity* of subformulae as the approach of Plaisted and Greenbaum (1986) does. In their terms, all subformulae have a positive polarity, except for those below the scope of parity constraints which have a polarity of zero.

We now introduce another normal form that, as we will see in later sections, is quite useful to achieve an efficient clausal form translation. We will see, in particular, that to make a "half-definition" of such formulae (the term is properly introduced later) only one new predicate is required.

Definition 3.4. An *extended clause* is either a regular clause or a parity clause, i.e. a parity formula built using only literals. A formula A is in *extended clausal normal form*, or simply *XCNF*, if it is a conjunction of extended clauses.

Likewise, a formula B is called an extended clausal normal form of a formula A if B is equivalent to A and B is in extended clausal normal form. ■

Renaming translation

We now proceed to introduce a variation of the structure preserving clausal form translation due to Tseitin (1968). To clarify the presentation it is presented as two separate stages: First some subformulae are replaced by new predicates and definitions for these predicates are introduced. In a second stage, the definitions themselves are translated into clausal normal form.

Definition 3.5. A *full-definition* is a formula $x \leftrightarrow A$, where x is an atom and A a formula. Similarly, a *half-definition* is a formula of the form $x \rightarrow A$. ■

Algorithm 3.2 (Economic subformula renaming)**.** This algorithm takes as input a formula A in negation normal form. It produces as output a formula A^s and a set Δ_A of half-definitions, full-definitions, and parity clauses.

Figure 3.9 depicts three formula mappings: B^f, B^h and B^s; that are applied to subformulae of A. These are called the *full*, *half* and *skip* renaming mappings respectively. Each of these rules is evaluated as follows:

- First, if any, the formula to the far right (next to the "$\mid$" symbol) is constructed. This might involve recursion and further evaluation of other formula mappings. After the formula has been constructed it is added to the global set Δ_A.
- Then the formula to the right of the "$\Rightarrow$" symbol is similarly constructed and returned as the result of each particular mapping application.

The algorithm works by computing A^s, then it returns this formula together with the final set Δ_A obtained.

We provide now several remarks and claims about the previous algorithm.

Theorem 3.4. *Algorithm 3.2 and the mappings defined in Figure 3.9 satisfy the following properties:*

1. *The renaming mappings B^f and B^h always return a single literal.*
2. *The renaming mapping B^s returns a formula in XCNF.*
3. *The algorithm terminates and produces a formula A^s in XCNF.*
4. *Moreover, the set of definitions Δ_A only contains:*
   (a) *Full-definitions of the form $x \leftrightarrow (x_1 \wedge \cdots \wedge x_n)$.*
   (b) *Parity clauses of the form $(\tilde{x} \oplus x_1 \oplus \cdots \oplus x_n)$.*
   (c) *Half-definitions of the form $x \rightarrow B$, where B is in XCNF.*

$$
\begin{array}{rcll}
x^f & \Rightarrow & x & \qquad (3.6)\\
(B_1 \wedge \cdots \wedge B_n)^f & \Rightarrow & x_B & \mid\ x_B \leftrightarrow (B_1^f \wedge \cdots \wedge B_n^f) \qquad (3.7)\\
(B_1 \vee \cdots \vee B_n)^f & \Rightarrow & x_B & \mid\ \tilde{x}_B \leftrightarrow (\tilde{B}_1^f \wedge \cdots \wedge \tilde{B}_n^f) \qquad (3.8)\\
(B_1 \oplus \cdots \oplus B_n)^f & \Rightarrow & x_B & \mid\ (\tilde{x}_B \oplus B_1^f \oplus \cdots \oplus B_n^f) \qquad (3.9)\\
\\
x^h & \Rightarrow & x & \qquad (3.10)\\
(B_1 \wedge \cdots \wedge B_n)^h & \Rightarrow & x_B & \mid\ x_B \rightarrow (B_1^s \wedge \cdots \wedge B_n^s) \qquad (3.11)\\
(B_1 \vee \cdots \vee B_n)^h & \Rightarrow & x_B & \mid\ x_B \rightarrow (B_1^h \vee \cdots \vee B_n^h) \qquad (3.12)\\
(B_1 \oplus \cdots \oplus B_n)^h & \Rightarrow & x_B & \mid\ x_B \rightarrow (B_1^f \oplus \cdots \oplus B_n^f) \qquad (3.13)\\
\\
x^s & \Rightarrow & x & \qquad (3.14)\\
(B_1 \wedge \cdots \wedge B_n)^s & \Rightarrow & B_1^s \wedge \cdots \wedge B_n^s & \qquad (3.15)\\
(B_1 \vee \cdots \vee B_n)^s & \Rightarrow & B_1^h \vee \cdots \vee B_n^h & \qquad (3.16)\\
(B_1 \oplus \cdots \oplus B_n)^s & \Rightarrow & B_1^f \oplus \cdots \oplus B_n^f & \qquad (3.17)
\end{array}
$$

Figure 3.9: Mapping rules for the economic subformula renaming

Proof. Each item follows by a careful examination of Figure 3.9 and the validity of previous items. Item 2 is, perhaps, the less clear one; but it actually follows from Item 1 and a simple examination of rules (3.14–3.17). Item 4(d) follows by a similar argument on definitions introduced by rules (3.10–3.13). □

Remark. The distinction of full- and half-renaming mappings take into account the usual optimisations in terms of subformula polarity (see e.g. Plaisted and Greenbaum, 1986).

Compared to the standard formalisations and implementations of the renaming translation, the most significant improvement is condensed in rule (3.11) which avoids the renaming of operands of conjunctions with positive polarity at any formula depth (not only at the root, which is a usual optimisation).

Moreover, after a close examination the reader might notice that rule (3.12) is never invoked in the computation of A^s (under the assumption of a flattened formula). The rule is anyway included for completeness of the definition.

Theorem 3.5. *A formula A is satisfiable if and only if so is $\Delta_A \cup \{A^s\}$ is.*

Proof. The proof follows by structural induction on the definitions in Figure 3.9 and the standard textbook arguments for the correctness of Plaisted and Greenbaum (1986) translation. In particular note that $(\tilde{x}_B \oplus x_{B_1} \oplus \cdots \oplus x_{B_n})$ is just an alternative way of writing $x_B \leftrightarrow (x_{B_1} \oplus \cdots \oplus x_{B_n})$. □

We begin now with the second phase of the translation which takes the output of Algorithm 3.2 and produces a formula in CNF. This, in turn, is done in two steps: first parity clauses are replaced by sets of regular clauses, then half- and full-definitions are are also replaced by their clausal form equivalents.

Algorithm 3.3 (Clausal form translation of parity clauses)**.** Parity clauses of length $n > 3$ are translated first to a set of $n-2$ ternary parity clauses by introducing $n-3$ new predicates, denoted here as $y_1, \ldots, y_{n-3}$, as follows:

$$(x_1 \oplus x_2 \oplus \cdots \oplus x_{n-1} \oplus x_n) \quad \Rightarrow \quad \begin{array}{lll} x_1 & \oplus\, x_2 & \oplus\, y_1 \\ \tilde{y}_1 & \oplus\, x_3 & \oplus\, y_2 \\ \vdots & \vdots & \vdots \\ \tilde{y}_{n-4} & \oplus\, x_{n-2} & \oplus\, y_{n-3} \\ \tilde{y}_{n-3} & \oplus\, x_{n-1} & \oplus\, x_n \ . \end{array}$$

Then, for binary and ternary parity clauses, the standard clausal form translation is finally applied:

$$x_1 \oplus x_2 \quad \Rightarrow \quad \begin{array}{l} x_1 \vee x_2 \\ \tilde{x}_1 \vee \tilde{x}_2 \ , \\ \\ \\ \end{array} \qquad x_1 \oplus x_2 \oplus x_3 \quad \Rightarrow \quad \begin{array}{l} x_1 \vee x_2 \vee x_3 \\ x_1 \vee \tilde{x}_2 \vee \tilde{x}_3 \\ \tilde{x}_1 \vee x_2 \vee \tilde{x}_3 \\ \tilde{x}_1 \vee \tilde{x}_2 \vee x_3 \ . \end{array}$$

Remark. After applying Algorithm 3.3 to the output of Algorithm 3.2 we obtain a formula with similar properties to the ones described in Theorem 3.4 except for the fact that the returned formula, as well as the right hand side of half-definitions, are now in clausal form. Moreover, the set of definitions does not contain parity clauses any more, but just plain clauses.

Algorithm 3.4 (Clausal form translation of definitions)**.** This algorithm takes as input a set of half- and full-definitions as already simplified by previous algorithm. It then applies the rules described in Figure 3.10 to replace each definition by a number of clauses. Note that half-definitions have the form $x \to (C_1 \wedge \cdots \wedge C_n)$ where each C_i is a clause. The output is the set of clauses computed.

$$
\begin{array}{rcl}
x \to (C_1 \wedge \cdots \wedge C_n) & \Rightarrow & \tilde{x} \vee C_1 \\
 & & \vdots \\
 & & \tilde{x} \vee C_n \\
 & & \\
x \leftrightarrow (x_1 \wedge \cdots \wedge x_n) & \Rightarrow & x \vee \tilde{x}_1 \vee \cdots \vee \tilde{x}_n \\
 & & \tilde{x} \vee x_1 \\
 & & \vdots \\
 & & \tilde{x} \vee x_n
\end{array}
$$

Figure 3.10: Rewrite rules for half- and full-definitions

Theorem 3.6. *Given formula A, let A' be the result of applying Algorithms 3.1 through 3.4 including the conjunction of the set of definitions also computed by them. Then A' is in clausal normal form and is equisatisfiable with A.*

Proof. It follows by the correctness of each individual algorithm. □

Experimental evaluation

In order to evaluate the effectiveness of our proposed improvements on the clausal form translation, we carried out some experiments to compare this approach with existing available systems. For this we took an example from Biere et al. (1999), the proof of a reachability property on a barrel shifter with k registers, and compared the performance of the translation done by the BMC tool, also described by these authors, and our own implementation. We then ran MiniSat (Eén and Sörensson, 2005) to evaluate the performance that a modern satisfiability solver has while solving the generated formulae.

Table 3.5 summarises the results of these experiments, showing the number of predicates and clauses required to build the formula by each translation, as well as the running time of MiniSat in seconds. The time required to generate the formulae was negligible, and roughly equivalent, for both translations. As can be seen, our proposed translation not only produces fewer predicates and clauses, but also significantly improves the performance of the solver when processing these examples.

	BMC tool			new translation		
k	preds	clauses	time	preds	clauses	time
2	50	159	0.0	33	103	0.0
3	275	942	0.0	184	546	0.0
4	578	2035	0.0	369	1107	0.0
5	1407	5383	0.9	881	2808	0.4
6	2306	8931	4.7	1405	4503	2.3
7	3523	13765	25.1	2101	6758	5.1
8	5106	20083	90.2	2993	9651	38.9
9	8903	36606	180.8	5176	17328	37.1
10	11982	49483	570.2	6881	23083	92.4

Table 3.5: Experimental evaluation of improved clausal form translation

3.2.2 Using specialised constraints

During the development of the translation described in the previous section, we observed that, potentially, most atoms are introduced while clausifying parity clauses. This was our motivation to implement a solver with direct support of parity clauses, with the hope that it would improve the performance on the solver on instances with many parity clauses.

Moreover, most implementation techniques discussed in Chapter 2 can easily be modified in order to handle parity clauses. The most significant changes were needed to adapt the *watched literals scheme* for parity clauses. In this case, it is enough to watch two literals on each parity clause, but literals have to be watched on both of their positive and negative phases. In total we need to set 4 watches for every parity clause (independent of its length), compare this with the 8 watches needed if a ternary parity clause is clausified, or the $8(n-2)$ for a parity clause of length n clausified using Algorithm 3.3.

Algorithm 3.5 shows the modified version of the algorithm to update the watched literals in a parity clause when performing unit propagation. This idea was implemented in a version of MiniSat (Eén and Sörensson, 2005) with support of custom constraints but, unfortunately, it was not as successful as expected. First we observed that problems from typical applications do contain large amounts of parity clauses, but most of them are of size 2 or 3, which do not introduce new atoms when clausified and for which the savings in literals to watch are not really dramatic. Moreover the possible advantages of the approach, such as having a more compact representation and a reduced number of literals to watch, turned out to be negligible for these short parity clauses.

Algorithm 3.5 Find new literal to watch on a parity clause

```
procedure FindNewParityWatch($x$)
    $s \leftarrow$ false
    for each literal $y$ in the clause, $y \neq x$ do
        if $y$ is set to true then
            $s \leftarrow \neg s$                              ▷ parity so far
        else if $y$ is watched then
            $w \leftarrow y$                  ▷ $w$ is the other watched literal
        else if $y$ is unassigned then
            move watch from $x$ to $y$          ▷ found a new literal to watch
            move watch from $\tilde{x}$ to $\tilde{y}$
            return
        end if
    end for
    if $s$ is true then $w \leftarrow \tilde{w}$
    add $w$ with to the queue for unit propagation       ▷ unit clause detected
                          ▷ if $\tilde{w}$ is already in the queue, this detects a conflict
end procedure
```

Another problem we faced is that there was no obvious way to translate formulae such as $x \to (a \oplus b)$, which are often produced in half-definitions, in terms of clauses and parity clauses only. One approach was to clausify the right hand side and loose any advantage of having support of such parity constraints. The other approach was to ignore the polarity optimisation and write a single parity clause $(\neg x \oplus a \oplus b)$. This, however, dramatically decreased the effectiveness of the solver, making it explore a considerably larger search space in all our experiments.

3.2.3 Properties of problem encodings

The experience and lessons learnt from our experience with parity clauses pointed out that, in order to develop better propositional encodings, one has to take into account the properties and features that are typically found on problems from the applications that we are more interested in solving. Moreover, it is also important to understand the effect that properties of the generated encodings will have on the performance and running time of a satisfiability solver when processing them.

Having this in mind, we decided to run a series of experiments to evaluate the effect that a number of clausal form simplification techniques, in particular those proposed by Eén and Biere (2005) and implemented in SatELite (see Section 2.3), have in the behaviour of a DLL-style satisfiability solver.

For this we first gathered a number of SAT benchmarks which have been commonly used in literature. This includes problems from bounded model checking (Biere et al., 1999), microprocessor verification (Velev and Bryant, 2001) and a number of assorted instances from the SATLIB library of problems (Hoos and Stützle, 2000). After filtering out problems that were either too easy or too difficult to solve, 104 satisfiable and 28 unsatisfiable problems (solvable between a few seconds and 20 minutes by standard MiniSat) remained.

We then ran two versions of MiniSat, with and without simplifications, and collected a number of statistics such as the total running time, number of decisions, unit propagations, restarts, conflicts and the amount of literals in conflict clauses generated by these two solvers on each problem instance. We also analysed the behaviour of two other defined predicates

$$\begin{aligned} \text{compactness} &= \text{propagations} \;/\; \text{decisions}, \\ \text{constrainedness} &= \text{decisions} \;/\; \text{conflicts}. \end{aligned}$$

The first one, *compactness*, is the average number of unit propagations that a solver has to do after deciding the value of a predicate. Intuitively a good encoding for a problem should be compact, i.e. the solver should be able to quickly determine all the consequences of a given truth assignment.

The second, *constrainedness*, estimates the average number of guesses, or decisions, that the solver had to make before reaching a conflict point. A priori, it is unclear whether one would prefer to to increase or reduce this indicator. On unsatisfiable problems, the constrainedness of a problem estimates the depth of the decision tree that describes the total search space to explore. A small constrainedness value might suggest a smaller search space, however it might also be the case, in particular for satisfiable problems, that solutions are more difficult to guess and that it might be harder to find one.

One of the first observations we made on the results, is that the behaviour of the solvers is quite different between satisfiable and unsatisfiable problems. In general, changes between solving an unsatisfiable problem with and without simplifications tend to be more consistent. In anyway, the solver still has to exhaust the search space before determining that the given problem has no solutions. On the other hand, satisfiable problems are much more unstable since 'luck', not only simplifications and optimisations, is often an important factor on whether a solution is found early or late in the search. In view of this, we analysed satisfiable and unsatisfiable problems separately.

	unsat	sat
time	−46.9%	−25.8%
decisions	− 5.1%	−22.2%
propagations	−52.1%	−43.5%
restarts	+ 4.3%	− 9.3%
conflicts	+33.2%	−31.3%
conf. lits.	+ 2.3%	−40.6%
compactness	−61.0%	−28.3%
constrainedness	+ 0.5%	+13.2%

Table 3.6: Summary the of effects of simplification techniques

Table 3.6 shows a summary of how the statistics collected were affected by the application of preprocessing simplifications. On average, unsatisfiable problems were solved almost 50% faster when applying the simplifications, while satisfiable ones had an average speedup of about 25%. Interestingly, when simplifications are applied, unstatisfiable problems tend to generate more conflicts (+33.2%) which only causes a small increase on memory consumption (reflected on the +2.3% increase on conflict literals). This also suggests that the simplifications performed are helping the solver to learn 'better' (i.e. shorter) conflict clauses. On the other hand, the number of conflicts is considerably reduced (−31.3%), as well as the number of conflict literals (−40.6%), for instances which are satisfiable. Overall it is interesting to observe that the number of unit propagations is considerably reduced on both unsatisfiable (−52.1%) and satisfiable cases (−43.5%).

Now, to get a deeper understanding of how these variables relate with the actual time required to solve a problem, Table 3.7 shows some contingency tables to illustrate these relations. Each one of the inner cells shows the percentage of problems on which the simplifying solver reported more (resp. less) occurrences of some event while also the time required to solve it was more (resp. less). For example, in 28.57% of the unsatisfiable problems both the number of decisions and the total running time was incremented; and in 29.81% of the satisfiable problems, although the number of unit propagations was reduced, the running time did increase nevertheless. The third and fifth column, as well as the bottom row, show marnginal totals where it is shown, for example, that about 70% of the unsatisfiable problems were solved faster using the simplifications while, for satisfiable ones, it is less clear whether the simplifications were useful or not.

Some interesting points to observe are that the simplifications performed do effectively tend to reduce the number of unit propagations and, as one might

		unsat			sat		
		time △	time ▽		time △	time ▽	
decisions	△	28.57%	39.29%	67.86%	25.00%	1.92%	26.92%
	▽	0.00%	32.14%	32.14%	27.88%	45.19%	73.07%
propagations	△	25.00%	3.57%	28.57%	23.08%	1.92%	25.00%
	▽	3.57%	67.86%	71.43%	29.81%	45.19%	75.00%
restarts	△	28.57%	53.57%	82.14%	37.50%	3.85%	41.35%
	▽	0.00%	17.86%	17.86%	15.38%	43.27%	58.65%
conflicts	△	28.57%	42.86%	71.43%	32.69%	1.92%	34.61%
	▽	0.00%	28.57%	28.57%	20.19%	45.19%	65.38%
conf. lits.	△	28.57%	10.71%	39.28%	29.81%	3.85%	33.66%
	▽	0.00%	60.71%	60.71%	23.08%	43.27%	66.35%
compactness	△	3.57%	0.00%	3.57%	23.08%	2.88%	25.96%
	▽	25.00%	71.43%	96.43%	29.81%	44.23%	74.04%
constrainedness	△	7.14%	21.43%	28.57%	13.46%	38.46%	51.92%
	▽	21.43%	50.00%	71.43%	39.42%	8.65%	48.07%
		28.57%	71.43%	100.00%	52.88%	47.11%	100.00%

Table 3.7: Contingency tables with effects of simplification techniques

expect, a reduction in the number of propagations does increase the chances of speeding up the solving process. Moreover, a Fisher's exact test (Devore, 1999) performed with the JMP Statistical Discovery software (SAS, 2007) finds very convincing statistical significance for this hypothesis with p-values of less than 0.0001 for both unsatisfiable and satisfiable instances. In statistic analysis, a smaller p-value is an indicator of stronger confidence in the hypothesis being tested. Interestingly, no such strong significance is found, in the unsatisfiable case, for other variables such as the number of decisions or the number of conflicts.

On the other hand, for the case of satisfiable instances, Fisher's exact test does find very convincing statistical significance for the hypothesis that if the compactness of a formula is increased (i.e. there are less unit propagations per decision) then it is more likely that the solution time will be reduced. And, confirming our early speculation, if the constrainedness is reduced (i.e. there are more decisions between conflicts) then it is more likely that the solution will be found earlier.

3.2.4 Results and conclusions

In this section we have explored many ideas while trying to understand the effect that different propositional encodings have when they are applied to logical formulae. It is indeed very puzzling to figure out what constitutes the basis for a *good* encoding and, moreover, satisfiability solvers tend to be very sensitive to even the slightest changes in the representation of an input formula.

We have, nevertheless, being able to propose an improved clausal form translation which, while trying to reduce the number of newly introduced predicates, provides a significant reduction not only in the size of the generated formulae, but also in the time that is required to solve them. Furthermore, as a valuable lesson learnt from our experiments and implementations, we found that one should always take into account, while designing encodings for satisfiability, the general characteristics of the actual problems that we are willing to solve.

Finally, as a result of some statistical analysis, we do were able to obtain some general advise which can be helpful for designers of propositional encodings. First, better encodings seem to be those which are *compact*, i.e. that the effects of assigning a truth value to a predicate are quickly propagated and computed using unit propagation. Second, good encodings —particularly for unsatisfiable formulae— are also *constrained*. This means that one should be able to quickly detect conflicts when assigning inappropriate values to variables.

More comprehensive statistical studies are definitively needed in order to gain more insight on the characteristics of propositional satisfiability encodings. Nonetheless, we believe that our work sets an important precedent on the way that research in this direction can be carried out, while already obtaining some simple conclusions on this issue.

3.3 Concluding remarks

We have explored and documented in this Chapter most of the thesis contributions in the context of the propositional satisfiability problem. One of our earliest contributions, which has been already considered as useful and interesting by the research community (Muhammad and Stuckey, 2006; Gomes and Walsh, 2006; Santillán Rodríguez, 2007), is a model to randomly generate logical formulae not necessarily in clausal normal form. These formulae, moreover, are useful to create benchmarks to test satisfiability solvers which either directly work with non-clausal representations, or alternatively try to exploit the structural information implicitly available on clausal encodings.

A couple of other contributions, found in the second section of this chapter, include the proposal of an improved clausal form translation, as well as some empirical and statistical study on properties of encodings generated from real-world applications. From these we are able to draw some interesting conclusions which, moreover, translate into some useful advise for designers of propositional encodings. In particular we observe that two indicators, the compactness and constrainedness of an encoding, are directly related with the relative ease with which problems can be solved.

This chapter, as well as already noted in the previous Chapter 2, starts to highlight some of the inconveniences that one commonly faces when dealing with propositional encodings. First, the formulae generated from applications tend to quickly grow beyond the scope of existing technology, while their original structure is clouded by normal form translations. The remaining chapters will therefore introduce another alternative which, by the use of a logic with a higher level of abstraction, attempts to provide a solution for this kind of issues.

Chapter 4

Effectively propositional logic

It has become customary in automated reasoning and artificial intelligence to tackle problems by reducing them to checking the satisfiability of a formula in some logical framework. Then one has to either find a *model* of the given logical formula, or produce a *refutation* which proves that no such model exists. And different logic systems, with distinct trade-offs between complexity and expressiveness, have been developed to use in this context.

Propositional logic, introduced in Chapter 2 and the main theme of previous chapters, is an alternative which, while being one of the simplest options, offers enough expressive power to encode a significant number of problems and applications in computer science. As we have seen, however, this logical language has some drawbacks: the structure of problems generated from applications often tends to be lost or hidden behind the propositional encoding, and the size of the encodings themselves tend to quickly grow beyond the capabilities of existing solvers.

On the other hand of the spectrum, we have first-order logic which offers a significantly increased expressibility but at the cost of a much higher complexity. The problem of deciding whether a formula is satisfiable or not becomes *undecidable*, i.e. there is no algorithm which, given as input a first-order formula, will always terminate in finite time after deciding the satisfiability status of the formula. The problem is, nevertheless, *semi-decidable*: there are procedures which, if the input is unsatisfiable, will eventually terminate with a refutation of the formula; but, if the problem is satisfiable, then the procedure might run forever without never finding a solution.

As a compromise between these two possibilities, we consider the use of effectively propositional logic (EPR). Unlike propositional logic, formulae in this logic can use variables, although the use of quantification is restricted and function symbols are not allowed. This fragment of first-order logic, which also corresponds to the Bernays-Schönfinkel class of formulae, has an $\exists^*\forall^*$ quantifier prefix when written in prenex normal form and is as expressive as propositional logic.

Alternatively, one can describe effectively propositional formulae as those having a finite Herbrand universe. Thus, as a consequence of Herbrand's theorem formally stated later in Section 4.2, when testing for satisfiability it is possible to replace an effectively propositional formula by all its ground instances which, in turn, can be interpreted as simple propositional formulae and sent to a propositional satisfiability solver. Through this process, known in literature as instantiation or grounding, it is possible to reduce any arbitrary effectively propositional formula into an equisatisfiable propositional representation, and hence the name 'effectively propositional' used to describe them.

Moreover, the introduction of variables and a limited amount of quantification are enough to allow the design of problem encodings which are often exponentially shorter than their propositional counterparts. On the one hand, this allows one to write problem descriptions which are usually much more succinct and even natural for humans to read and understand. While, on the other, it also enables the use of reasoning techniques which work directly with formulae written at a higher level of abstraction.

Although quite recent, there has been a significant interest of the automated reasoning community on effectively propositional logic. The CADE ATP System Competition has, since its JC instalment in 2001, a division for EPR problems (Pelletier et al., 2002). Moreover, a number of theorem provers particularly geared towards this class of formulae have also been developed. These include, for example, EGROUND by Schulz (2002), PARADOX by Claessen and Sörensson (2003), DARWIN by Fuchs (2004), and iPROVER by Korovin (2006).

In this chapter we will formally introduce the syntax and semantics of effectively propositional logic; and then briefly summarise some of the existing techniques that have been currently designed to deal with this kind of formulae. We will then also introduce another language, which we call *finite domain predicate logic*, which puts some syntactic sugar on top of the effectively propositional language in order to even more naturally describe problems from applications.

4.1 Syntax and semantics

As in the propositional case, effectively propositional formula are built from atoms combined with logical connectives. In this case, however, atoms are built from predicates which may also have arguments. The following definition formally introduces the syntax of effectively propositional logic.

Definition 4.1. The syntax of an *effectively propositional logic* is composed of a non-empty but finite set of *constant symbols* $\mathcal{D}$, also sometimes referred to as the *domain* of the logic; and an infinite supply of *variables* which will be often denoted by: x, y, ...[1] There is also a set of *predicate symbols* $\mathcal{P}$ and, moreover, each predicate symbol is associated with a positive integer which we call its *arity*.

A *term* is either a variable or a constant symbol. An *atom* is an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of arity n and each t_i is a term. *Effectively propositional formulae* are then built from atoms using the standard propositional connectives of truth (***true***, $\top$), falsity (***false***, $\bot$), conjunction (***and***, $\wedge$), disjunction (***or***, $\vee$), and negation (***not***, $\neg$). Other connectives such as implication and equivalence are defined in terms of those already introduced, e.g. $F \rightarrow G \equiv \neg F \vee G$ and $F \leftrightarrow G \equiv (F \rightarrow G) \wedge (G \rightarrow F)$. A set of formulae will be also often referred to as a *set of constraints*. The *size* of a formula, or set of formulae, is counted as the number of symbols in it.

A *literal* is either an atom of the form $p(t_1, \ldots, t_n)$, a literal with a *positive phase*, or its negation $\neg p(t_1, \ldots, t_n)$, a literal with *negative phase*. We will also consider the notion of a *signed predicate symbol* which is either a predicate symbol p or its negation $\neg p$. Then, a literal $s(t_1, \ldots, t_n)$ can be alternatively seen as a pair of a signed predicate symbol s and a n-tuple of terms $(t_1, \ldots, t_n)$; where n matches the arity of the predicate symbol in s. We will also use $\tilde{s}$ to denote the *logical complement* of a signed predicate symbol s, i.e. $\tilde{s} = \neg p$ when $s = p$ and $\tilde{s} = p$ when $s = \neg p$. Similarly, if a literal is of the form $l = s(t_1, \ldots, t_n)$, then we also define $\tilde{l} = \tilde{s}(t_1, \ldots, t_n)$.

A *clause* is a disjunction of literals and a *clausal formula* is a set of clauses. Although a clausal formula can be equivalently defined as a conjunction of clauses, having them in the form of sets of constraints will later simplify the treatment of its semantics. As in the propositional case, a *unit clause* is one that contains a single literal, and the *empty clause* is the one that contains no literals at all.

[1]Not to be confused with the notation of literals in previous chapters.

As usual, a formula is said to be in *clausal normal form* (CNF) if it is a clausal formula. ■

The following definition introduces the concept of a substitution, which is used in expressions to replace variables by terms. Moreover, we also to define other important related notions such as instances of formulae and unification.

Definition 4.2. A *substitution* is a function σ that maps variables to terms, and behaves like the identity function almost everywhere. We denote by $F\sigma$ the result of applying a substitution σ to a formula F, i.e. the formula obtained after uniformly replacing every variable x in F with the corresponding term $\sigma(x)$. A substitution is often denoted by explicitly enumerating the non-trivial mappings that it makes, e.g.: $\{x_1 \to t_1, \ldots, x_k \to t_k\}$.

We also say that a formula F' is an *instance of* F if there is a substitution σ such that $F' = F\sigma$; also σ is a *unifier* of the formulae F and F' if $F\sigma = F'\sigma$, and a *most general unifier* if, for any other unifier σ', it is always possible to find a substitution ρ such that $\sigma' = \sigma\rho$. In a similar way, substitutions can be applied to sets of formulae, clauses, atoms, terms or even tuples of terms.

As a convenience, if F is a formula, we will also often write $F(x_1, \ldots, x_k)$, where the sequence $x_1, \ldots, x_k$ includes all the variables appearing in F, to explicitly denote the dependency of F on such variables. Moreover, after enumerating the variables of F in such a way, we will also use the expression $F(t_1, \ldots, t_k)$ to denote the formula $F\{x_1 \to t_1, \ldots, x_k \to t_k\}$.

A set of formulae, a formula, an atom, a term or a tuple of terms is said to be *ground* if it contains no variables. The *Herbrand domain* is the set of all ground atoms which, for the case of effectively propositional logic, is always finite. Finally, a *ground instance* of a formula F is any instance of F which is also ground. ■

We now proceed to define the semantics of effectively propositional formulae in terms of Herbrand interpretations. Intuitively, these interpretations determine which atoms are to be taken as true or false, while the truth value of a more complex expression is evaluated from these values by using the standard meaning of propositional connectives.

Definition 4.3. A *Herbrand interpretation* is a set of ground atoms, i.e. a subset of the Herbrand domain. The notion of whether a Herbrand interpretation $\mathcal{I}$ is a *model* of a ground formula F, denoted by $\mathcal{I} \models F$, is defined as follows:

$$\begin{array}{llllll} \mathcal{I} \models A & \text{iff} & A \in \mathcal{I}, & \mathcal{I} \models F \wedge G & \text{iff} & \mathcal{I} \models F \text{ and } \mathcal{I} \models G, \\ \mathcal{I} \models \neg F & \text{iff} & \mathcal{I} \not\models F, & \mathcal{I} \models F \vee G & \text{iff} & \mathcal{I} \models F \;\text{ or }\; \mathcal{I} \models G. \end{array}$$

Also $\mathcal{I}$ is always a model of $\top$ and never of $\bot$. Now a Herbrand interpretation $\mathcal{I}$ is said to be a *model of a non-ground formula* F if it is a model of every ground instance of F, and a model of a set of constraints if it is a model of every formula in the set. A set constraints is said to be *satisfiable* if it has at least one model, and *unsatisfiable* if it has no models. ■

Since in the following we will mostly be dealing with effectively propositional formulae and Herbrand interpretations, we will often simply say *formula* or *predicate formula* when we refer to an effectively propositional formula and *interpretation* when we refer to a Herbrand interpretation.

Although we will not always write sets of constraints in clausal normal form, we do make sure that they can be easily rewritten in such form by using simple logical identities (e.g. distributing connectives and changing implication for disjunction). This is more of a convenience and is used as an aid to more clearly convey the intuitive meaning of a clause. For example

$$\mathsf{big}(x) \wedge \mathsf{round}(x) \wedge \mathsf{orbits\text{-}star}(x) \rightarrow \mathsf{planet}(x)$$

is equivalent to

$$\neg\mathsf{big}(x) \vee \neg\mathsf{round}(x) \vee \neg\mathsf{orbits\text{-}star}(x) \vee \mathsf{planet}(x)\ ,$$

but the meaning of the former is perhaps clearer to grasp.

Moreover, it is also possible to apply the translation proposed by Tseitin (1968) to reduce any effectively propositional formula into an equisatisfiable clausal formula. The translation is linear, both in time and in the size of its output, and similar to the propositional case works by replacing complex subformulae by atoms built with fresh new predicates. In fact, the same Algorithm 2.1 can be used first to translate an effectively propositional formula into negation normal form. Then the following version of Tseitin's translation for effectively propositional formulae is applied.

Algorithm 4.1 (Tseitin's clausal form translation)**.** Let G be a formula in negation normal form. For each subformula of the form $F(x_1, \ldots, x_k)$ introduce a new predicate symbol p_F of arity k and also let $\hat{F}$ be the atom $p_F(x_1, \ldots, x_k)$. Then let Δ_G be a set containing, for each subformula $F = A \wedge B$, the pair of constraints

$$\hat{F} \to \hat{A}$$
$$\hat{F} \to \hat{B}$$

and, for each subformula $F = A \vee B$, the constraint

$$\hat{F} \to \hat{A} \vee \hat{B} \ .$$

The algorithm produces as output the set of constraints $\Delta_G \cup \{\hat{G}\}$.

The following theorem, whose proof is completely analogous to the propositional case, establishes the correctness of Tseitin's translation.

Theorem 4.1. *Let F be an effectively propositional formula. The formula F and the set of constraints $\Delta_F \cup \{\hat{F}\}$ are equisatisfiable.*

Although there are methods such as analytic tableaux (Hähnle, 2001) which work on formulae with arbitrary structure, a great deal of theory and implementation techniques have been developed to work with formula written in a clausal normal form. In the following sections we will briefly survey some methods which have been proposed to check the satisfiability of effectively propositional formulae.

4.2 Grounding-based methods

As a consequence of Herbrand's theorem (see e.g. Fitting, 1996), checking the satisfiability of effectively propositional formula can be reduced to the problem of propositional satisfiability. Formally, we have the following statement.

Theorem 4.2 (Herbrand's theorem)**.** *Let F be a formula in clausal normal form. The formula F is unsatisfiable if and only if there is an unsatisfiable set G containing only ground instances of clauses in F.*

Since the number of constant symbols is finite, the number of ground terms as well as the number of ground instances of an effectively propositional clause is finite. One can actually, in the most naive approach, replace F by the set containing *all* the ground instances of clauses in F. Such set can then be interpreted as a set of propositional clauses and fed to a propositional satisfiability solver in order to determine the satisfiability of F.

This naive approach, however, often does not tend to scale. The size of the generated propositional problem is in general exponential in the size of the original EPR representation since, given a clause with k variables and a domain $\mathcal{D}$ with n constant symbols, the clause has a total of n^k different ground instances. Researchers have therefore explored many ways to reduce the number of generated clauses while still preserving the satisfiability of the original formula. While many of these ideas have been developed to deal even with more general classes of formulae such as first-order logic, our discussion will be restricted to the effectively propositional case.

4.2.1 Splitting

One technique to reduce the number of generated instances is to split clauses into components which are then grounded into fewer instances (Schulz, 2002; Claessen and Sörensson, 2003). Suppose that a clause is written in the form $C(\bar{x}) \vee D(\bar{y})$ where both C and D are subclauses with respective variables $\bar{x} = x_1, \ldots, x_k$ and $\bar{y} = y_1, \ldots, y_{k'}$. Also let $\bar{x} \cap \bar{y}$ denote a sequence containing the variables that occur both in $\bar{x}$ and $\bar{y}$. This clause is then equivalent to the pair of constraints

$$\begin{aligned}&C(\bar{x}) \vee p(\bar{x} \cap \bar{y})\\&\neg p(\bar{x} \cap \bar{y}) \vee D(\bar{y})\end{aligned}$$

where p is a fresh new predicate symbol of the appropriate arity.

If there is a variable which appears in $\bar{x}$ and not in $\bar{y}$, and vice versa, then the generated constraints have less literals per clause and, therefore, less ground instances. Moreover, after a clause has been split, it is sometimes possible to further split the newly generated clauses. However, since different sequences of splits can yield a number of different outcomes, this opens up the problem of finding a strategy to apply splits in a way that reduces the final number of variables per clause as much as possible.

Finding an optimal sequence of splits has turned out to be a rather difficult problem, and often one prefers to implement a cheap although imperfect solution. A first heuristic, proposed by Schulz (2002) and implemented in EGROUND, enumerates all possible subsets (small ones first) that contain only variables from the original clause. For each such subset V, it is checked in a linear time operation whether the clause can be split so that variables in the intersection are exactly those in V. If this is possible, then the split is performed and the whole process iterated. Otherwise, since the number of subsets is exponential, the algorithm gives up after trying some fixed amount of checks.

An alternative approach, proposed by Claessen and Sörensson (2003) and implemented in Paradox, checks first which variables are connected in the clause. It is said that a pair of variables is *connected* if there is a literal in the clause in which both of them appear. Then the variable which is connected to the *least* number of other variables, say x, is selected. All literals containing x are then moved to left hand side of the split, and the remaining to the right. The procedure is then iterated but on the right hand side only; since all literals in the left contain x, it cannot be further split. This splitting heuristic runs in polynomial time and, moreover, is guaranteed to always find a split if one is possible.

4.2.2 Pure predicates

If all the occurrences of a predicate symbol are always in literals of the same phase, i.e. either always positive or always negative, then it is said that the predicate is *pure*, and all clauses where this pure predicate appears can be removed without altering the satisfiability of the formula. This is justified since any model of the original set of clauses is also a model of the reduced one. And it is easy to extend any model of the reduced set in a sensible way so that it becomes a model of the original formula; e.g. if the pure predicate always appears in positive literals, then extend the interpretation so that it evaluates all ground atoms containing this predicate symbol to true.

This reduction is also well know in the propositional case, where it was originally suggested by Davis et al. (1962) to be applied in the main loop of the DLL algorithm. The idea was, however, later abandoned since it becomes too expensive to keep track of literal counts on each phase in order to quickly detect the ones that become pure. On the other hand, in the case of effectively propositional logic, it is enough to perform it once as a preprocessing step, and can be of a great help to reduce the number of instances to be generated later. Recent versions of Paradox implement exactly this strategy.

4.2.3 Linking restrictions

Notice that, in a sense, the simplification described in the previous section is just a very rough lifting of the pure literals reduction from the propositional to the effectively propositional level. Suppose that, for some binary predicate symbol p, all its positive occurrences are instances of the literal $p(a, x)$, while all its negative occurrences are instances of $\neg p(x, b)$, where a and b are constant symbols. Since the predicate occurs in literals in both phases, p is not pure. However, while generating ground clauses during the instantiation process, all ground literals of the form $p(a, c)$, where c is a constant symbol different from b, will be pure in the propositional sense, because all negative instances of this predicate have a b in the second argument. In fact, only literals containing the atom $p(a, b)$ will not be pure after grounding.

Linking restrictions are another approach to reduce the number of ground clauses created at instantiation by trying to avoid, as much as possible, the generation of pure literals in the resulting set of propositional clauses. The following definition introduces some useful notation that will be helpful to later define some examples of linking restrictions.

Definition 4.4. Let F be a clausal formula, and let s be a signed predicate symbol of arity n. The set T_s of n-tuples of terms is defined as

$$T_s = \{(t_1, \ldots, t_n) \mid \text{the literal } s(t_1, \ldots, t_n) \text{ occurs in } F\} \ .$$

Also, given an integer $1 \leq i \leq k$, we define the set of terms $T_{s.i}$ as the i-th projection of tuples in T_s, i.e.

$$T_{s.i} = \{t_i \mid (t_1, \ldots, t_n) \in T_s\} \ . \qquad \blacksquare$$

The following restriction, which we call *complete linking* and is based on the *hyper-linking* approach from Lee and Plaisted (1992), gives a very general linking restriction to limit the generation of clauses during instantiation.

Algorithm 4.2 (Complete linking)**.** During the instantiation process, discard all clauses that contain a literal of the form $s(c_1, \ldots, c_n)$ unless the tuple $(c_1, \ldots, c_n)$ is an instance of some tuple in $T_{\bar{s}}$.

Although we are not aware of any modern prover implementing the complete linking restriction as described here, we do discuss later a possible efficient implementation in Section 7.1.1. Another possible approach is a linking restriction proposed by Schulz (2002), under the name of *structural constraints*, and implemented in EGROUND as a cheaper alternative to complete linking.

Algorithm 4.3 (Positional linking)**.** Let s be a signed predicate symbol with an arity of n, and let $1 \leq i \leq n$. We define the set of constant symbols $C_{s.i} = T_{s.i}$ if there are no variables in $T_{s.i}$, or $C_{s.i} = \mathcal{D}$ otherwise.

During the instantiation process, discard all clauses that contain a literal of the form $s(c_1, \ldots, c_n)$ unless, for every $1 \leq i \leq n$, the constant symbol $c_i \in C_{\tilde{s}.i}$.

We call this *positional linking* since it restricts grounding by information collected independently for each position in the list of arguments of literals.

4.2.4 Sort inference

The use of *sorts* was proposed by Claessen and Sörensson (2003) as an alternative to the linking restrictions of EGROUND. Nevertheless, as we will see later in Section 7.1.1, both approaches are orthogonal and can even be used simultaneously.

The idea behind this approach is that often, when encoding problems from real-world applications, constant symbols are used to represent different 'concepts' or 'sorts' of the original application. For example, in a problem domain involving students taking courses, an atom $\mathsf{takesCourse}(x, y)$ can be used where the variable x stands for a student name, while y stands for a subject. Nonetheless, when the problem is encoded as an effectively propositional formula, both variables are implicitly quantified over a single unified pool $\mathcal{D}$ of constant symbols. By reducing the instantiation of variables only to their appropriate sorts, one hopes to reduce the number of generated clauses, as well as simplify both the search of models and the interpretation of the models found (i.e. avoid models asserting facts such as $\mathsf{takesCourse}(\mathsf{logic}, \mathsf{french})$ or other nonsense). The following definition formally introduces the notion of sorts that we will use in the context of effectively propositional formulae.

Definition 4.5. Given a predicate symbol p of arity n and positive integer i, with $1 \leq i \leq n$, a *predicate position* is an expression of the form $p.i$. Then a *sort assignment* A is a function that maps each predicate position $p.i$ to a set of constant symbols $A_{p.i} \subseteq \mathcal{D}$. Each set $A_{p.i}$ is also known as a *sort*.

Moreover, a clause C is said to be *compatible* with a sort assignment A if for every term t appearing at some predicate position $p.i$, the term t is either a constant symbol in $A_{p.i}$ or, if it is a variable x, then all other occurrences of x in the clause also have the sort $A_{p.i}$. A set of clauses is *compatible* with a sort A, if all its clauses are compatible with A.

Given a clausal formula F and a sort assignment A, we will use $F|_A$ to denote the set of all ground instances of clauses in F which are compatible with A. ∎

An interesting property of sort assignments is that, when they are compatible with a formula, then the instantiation process can be reduced to those ground instances which are compatible with the sort assignment. More formally, we have the following theorem.

Theorem 4.3. *Let F be a clausal formula, and let A be a sort assignment compatible with F. The formulae F and $F|_A$ are equisatisfiable.*

Proof. Recall that, by definition, an interpretation $\mathcal{I}$ is a model of an effectively propositional clause C iff $\mathcal{I}$ is a model of every ground instance of C. From this it directly follows that any model of F is also a model of $F|_A$.

For the converse suppose that $\mathcal{I}$ is a model of $F|_A$. Also, for each sort of the form $A_{p.i}$, let $[A_{p.i}]$ be some distinguished fixed symbol in the set $A_{p.i}$. Then define the interpretation $\mathcal{I}'$ such that

$$\mathcal{I}' \models p(c_1, \ldots, c_n) \quad \text{iff} \quad \mathcal{I} \models p(\hat{c}_1, \ldots, \hat{c}_n) \ ,$$

where each

$$\hat{c}_i = \begin{cases} c_i & \text{if } c_i \in A_{p.i} \ , \\ [A_{p.i}] & \text{otherwise} \ . \end{cases}$$

We will show that $\mathcal{I}'$ is a model of F. For this take any clause C in F, and let $C\sigma$ be one of its ground instances. Now, define a new substitution σ' such that, for every variable x in the domain of σ,

$$x\sigma' = \begin{cases} x\sigma & \text{if } x\sigma \in A_x \ , \\ [A_x] & \text{otherwise} \ , \end{cases}$$

where A_x is the sort of the position where the variable x occurs in the clause. This is well defined since, because of the sort assignment being compatible with the formula F, all occurrences of x should have the same sort. Also note that, by construction, the ground clause $C\sigma'$ is compatible with A and, by hypothesis,

$\mathcal{I} \models C\sigma'$. But, from the construction of the interpretation $\mathcal{I}'$, we know that this is the case only if $\mathcal{I}' \models C\sigma$. Since $C\sigma$ was an arbitrary ground instance of an arbitrary clause in F, it follows that $\mathcal{I}' \models F$. □

As a consequence of this theorem, sort assignments are very helpful to reduce the number of generated instances in a grounding approach. It is however often the case that this sort of information is not explicitly given in most existing benchmarks such as, for example, those available at the TPTP problem library (Sutcliffe and Suttner, 1998). A simple method, however, can be used to extract sort information from a previously unsorted formula. The following procedure, which is based on the *sort inference* as proposed by Claessen and Sörensson (2003) and implemented in Paradox, builds a sort assignment which is compatible with the given input formula.

Algorithm 4.4 (Sort inference)**.** Given an effectively propositional formula as input, initially create a sort assignment giving unrelated empty sorts to each predicate position.

Processing one clause at a time, and as a union-find algorithm: for each variable in the clause, merge the sorts assigned to all predicate positions where that variable occurs; and, for each constant symbol, add the constant symbol to the sort assigned to the predicate position where it appears.

Finally, add a dummy constant symbol to any sort that still remained empty at the end of this procedure.

From this description it easily follows, as Claessen and Sörensson (2003) point out, that the sort assignment produced by this algorithm is compatible with the given input formula and, from Theorem 4.3, that it is also *valid* in the sense of the following formal statement.

Theorem 4.4. *Let F be a clausal formula, and let A be the result of applying the sort inference algorithm to F. The formulae F and $F|_A$ are equisatisfiable.*

Although in this section we only explored the usefulness of sort inference in the context of an instantiation-based approach, we will also see later in Section 7.2.2 how it is also possible to couple sort assignments with reasoning mechanisms that work at a higher levels of abstraction.

4.2.5 Incremental search

A problem with the grounding approach as has been presented so far is that, when the domain of the problem is large, it is easy to run out of time or space resources even before attempting to do any search of models. A very simple approach, implemented for example in EGROUND, is to try and build at least one instance of every clause before running out of resources. The resulting set of clauses is not equisatisfiable with the original formula anymore, but if one is lucky enough to prove that the set is unsatisfiable then, by Theorem 4.2, the original formula also is.

A more elaborated approach can, for example, successively create instances of clauses of the input formula and run satisfiability checks from time to time. If, at some point, the unsatisfiability of the generated set of ground clauses is established, then the procedure stops declaring the input formula unsatisfiable. Otherwise, if a model is found, more ground instances are created, perhaps using the information provided by model as a guide, and the procedure is iterated.

Variants of this kind of procedures have been proposed by Lee and Plaisted (1992), Plaisted and Zhu (2000), and Hooker et al. (2002). Moreover, the approach also allows a tighter integration and interleaving of propositional methods with other more general forms of reasoning. Combinations of tableaux related techniques with propositional satisfiability checking have been proposed by researchers such as Billon (1996), Letz and Stenz (2001). Fuchs (2004), as well as Korovin (2006) have also implemented combinations of first-order reasoning with instantiation in, respectively, the DARWIN and IPROVER systems; currently two of the leading effectively propositional provers at the CADE ATP System Competition (Sutcliffe, 2007). The later Section 4.3 describes in a bit more detail these approaches.

Another kind of incremental approach proposed by McCune (1994a), and also implemented by Claessen and Sörensson (2003) in the PARADOX system, is known as *MACE-style* model finding. In the context of an effectively propositional formulae, this technique works by noting that it is often enough to search for models in domains with a reduced number of constant symbols. Formally, we have the following statements.

Definition 4.6. Let $\mathcal{D}'$ be a set of constant symbols and let $\pi\colon \mathcal{D} \to \mathcal{D}'$ be a function that maps symbols from one domain to the other. Given an effectively propositional clausal formula F, let $\pi(F)$ be the formula obtained after replacing every constant symbol c in F with the corresponding $\pi(c)$.

If the clausal formula $\pi(F)$ is satisfiable, we say that F has a *model of size* n, where n is the number of elements in $\mathcal{D}'$. ■

Theorem 4.5. *Let F be an effectively propositional clausal formula and n a positive integer. If F has a model of size n, then F is satisfiable.*

Moreover, Claessen and Sörensson (2003) show how to build, given an input formula F, a set of propositional clauses G_i which are satisfiable if and only if the formula F has a model of size i. One can therefore incrementally test the satisfiability of each set $G_1, \ldots, G_n$, where n is the number of constant symbols originally in F. If at some point one finds a satisfiable set G_i then the procedure stops and reports the satisfiability of F. If it turns out that even G_n is unsatisfiable, then F is also unsatisfiable (just let π be the identity function), and the search is also stopped.

Note however that, when the satisfiability solver if searching for a solution of the set G_i, it has to guess not only truth values for each ground atom in the reduced domain, but also the mapping π used to actually reduce it. This introduces many symmetries in the search space, some of which can be eliminated by throwing in even more clauses into the encoding of G_i. Nevertheless, since there are a lot of similarities between each set G_i and G_{i+1} —many clauses in the later were already present in the former— a satisfiability solver with an interface for incremental search can be used to reuse information and increase the speed of individual satisfiability checks.

The approach is often useful for satisfiable problems when n, the original number of constant symbols, is very large and a model with a shorter domain size is easy to find. On the other hand, for unsatisfiable problems, this has the effect of introducing a heavy and unnecessary overhead: it is easier and more efficient just to directly check the satisfiability of F instead of all the sets $G_1, \ldots, G_n$. Later in Section 7.1.2 we will revisit this issue and compare the incremental against a one-shot approach which directly tries to check the satisfiability of the input formula without reducing the domain.

Finally, another advantage of incremental search as implemented in Paradox is that it allows one to use the instantiation approach not only on effectively propositional formulae, but on arbitrary first-order logic as well. This is not possible in general if one uses a one-shot approach.

4.3 Non-ground reasoning methods

In the previous section we have briefly described many techniques to reduce the satisfiability problem from effectively to plain propositional logic. In doing so it becomes easy to import years of research and existing software implementations, such as all those described in Chapter 2, very easily into the field of effectively propositional logic. Nevertheless, one is still constrained by the inefficiency of the propositional language to succinctly represent problems and, sooner or later, one runs out of resources after having generated so many ground clauses.

We will now look at alternative solving approaches that do not necessarily resort to a grounding process. Instead, reasoning methods are developed which perform inferences at the more general effectively propositional level. In particular, we will briefly introduce three different logical calculi. The first of them, resolution, is one of the major cornerstones in automated reasoning that was developed for reasoning with first-order logic. The later two are the model evolution and instantiation calculi, which have been especially successful in the effectively propositional arena.

4.3.1 Resolution calculus

Before dwelling into notion of resolution and its applicability to reasoning with effectively propositional formulae, we will first introduce some concepts and notations about formal proof systems in general. This will be useful not only for explaining resolution, but also the calculi in following sections and material in the later Chapter 7.

Definition 4.7. In general, an *inference rule* is a relation among clauses. Given a tuple $(C_1, \ldots, C_m, C)$ in such a relation, more often written as

$$\frac{C_1 \quad \cdots \quad C_n}{C} \ ,$$

we say that $C_1, \ldots, C_n$ are the *premises* of the inference, and that C *follows from*, or is a *direct consequence* of, its premises.

Given a set of clauses S, a *proof of a clause C derived from S by a set of inference rules $\mathcal{R}$*, denoted by $S \vdash_{\mathcal{R}} C$, is a sequence of clauses $C_1, \ldots, C_m$ where the last $C_m = C$ and each C_i is either a clause in S or is a direct consequence of some previous clauses in the sequence by an inference rule in $\mathcal{R}$. If clear by context, we will often omit the subscript $\mathcal{R}$ in the proves relation. Moreover, we say that m is the *length* of the proof while its *size* is the sum of the sizes of all clauses in the sequence.

A *refutation* of a set of clauses S is a proof of the empty clause derived from S. A set of inference rules is *refutationally sound* if, whenever S accepts a refutation, the set S is unsatisfiable; and is *refutationally complete* if, whenever the set S is unsatisfiable, there is a refutation of S. ■

We will now formally introduce the inference rule of resolution which was originally proposed by Robinson (1965) and, as we have observed before, has been one of the main foundations in the research of automated reasoning and theorem proving.

Definition 4.8. The inference rule of *binary resolution with factoring*, later simply referred to as *resolution*, is defined as

$$\frac{C \vee A_1 \vee \cdots \vee A_k \quad \neg A \vee D}{C\sigma \vee D\sigma}\ \mathsf{Res}$$

where σ is the most general unifier such that $A_1\sigma = \cdots = A_k\sigma = A_i\sigma$. ■

The inference system of resolution is both refutationally sound and complete and, when equipped with orders and selection functions, can be turned into a semi-decision procedure for first-order logic by a process of saturation (see e.g. Bachmair and Ganzinger, 2001). Systems such as SPASS by Weidenbach et al. (1996), OTTER by McCune and Wos (1997), and VAMPIRE by Riazanov and Voronkov (2002), implement many variations and optimisations of this resolution calculus. Resolution-based provers were quickly found as a very successful approach for first-order theorem proving, in much the same way that for effectively propositional logic they weren't.

Surprisingly, as noted first by Joiner (1976), it seemed notoriously difficult to design a decision procedure based on resolution for effectively propositional logic, i.e. the Bernays-Schönfinkel class of formulae. And, although such procedure was eventually found based on *semantic clash resolution* (Fermüller et al., 1993), resolution-based methods have been found relatively ineffective even when compared with the most simple propositional approaches (Sutcliffe et al., 2002).

As a consequence, other approaches have been devised trying to provide new reasoning techniques which are suitable for effectively propositional logic without having to resort to a full instantiation process.

4.3.2 Model evolution calculus

The model evolution calculus is an attempt to lift the main components of the DLL algorithm, which as we saw in Chapter 2 has been very successful in the propositional setting, to the effectively propositional and first-order level. The first step towards this direction was given by Baumgartner (2000) who proposed, in his *FDPLL calculus*, an extension of the splitting rule in order to be directly applied to first-order formulae without grounding.

Based on those early ideas, Baumgartner and Tinelli (2003) developed later the *model evolution calculus* incorporating first-order versions of the splitting and unit propagation inference rules, being the later a key component on the success of DLL. Moreover, in the same way that DLL is able to provide a model of the input formula whenever the formula is satisfiable, the model evolution calculus is also able to import this model generation process to the first-order case.

In order to do this, the calculus maintains a structure Λ, a finite set of possibly non-ground literals, known as a *context*. The context implicitly defines an interpretation $\mathcal{I}_\Lambda$ as a candidate model for the input clausal formula F. Initially, the search starts with a default context Λ_0 whose corresponding interpretation assigns all ground atoms to false. While $\mathcal{I}_\Lambda$ is not a model of F, the several inference rules of the calculus are then used to *evolve* the context Λ so that it becomes one. The search continues until either a model is found, or $\mathcal{I}_\Lambda$ becomes irreparable and therefore F is unsatisfiable.

For first-order formulae the calculus is refutationally sound and complete, as well as a semi-decision procedure for testing unsatisfiability. Moreover, the calculus is terminating for effectively propositional logic and, therefore, it also is a decision procedure for the Bernays-Schönfinkel class of formulae.

This calculus has been implemented by Fuchs (2004) in the Darwin system, with more implementation details also given by Baumgartner et al. (2005), and has been found particularly successful in solving effectively propositional problems since its first participation at the 20th instalment of the the CADE ATP System Competition (Sutcliffe, 2006).

4.3.3 Instantiation calculus

The instantiation calculus, originally proposed by Ganzinger and Korovin (2003), is a competing approach that interleaves propositional satisfiability checks with first-order inferences based on unification. The main inference rule of the calculus, similar to resolution, unifies complementary literals in a pair of clauses but, instead of the resolvent of the two clauses, instances of each clause using the appropriate unifier are generated.

Definition 4.9. The inference rule of *binary instance generation* is defined as

$$\frac{C \vee A \qquad \neg A' \vee D}{C\sigma \vee A\sigma \quad \neg A'\sigma \vee D\sigma}\,\mathsf{InstGen}$$

where σ is a proper most general unifier such that $A\sigma = A'\sigma$. ■

The procedure works by maintaining a set of clauses S, originally the set of input clauses, and interleaving propositional reasoning with applications of instance generation. First a propositional set of clauses $S{\downarrow}$ is built, where $\downarrow$ is the constant substitution that maps all variables in S to some distinguished constant symbol. The set $S{\downarrow}$ is sent to a propositional satisfiability solver and, if it is proved unsatisfiable, then the input formula is also unsatisfiable. Otherwise, if the satisfiability solver finds a propositional model $\mathcal{I}$; this interpretation is used to select some literals on clauses on which to apply instance generation inferences which, at the same time, helps to (implicitly) build an interpretation $\mathcal{I}_S$ as a candidate model for S. If no new clauses can be generated then satisfiability of S is detected being $\mathcal{I}_S$ one of its models. Otherwise the new clauses are added to the set S and the process is iterated.

This process is a semi-decision procedure for first-order logic, and a decision procedure for effectively propositional formulae. This follows from the fact that, as Ganzinger and Korovin (2003) proved, if the set S is saturated using instance generation, then it is possible to determine the satisfiability of S from the propositional satisfiability of $S{\downarrow}$. Formally, we have the following statement.

Theorem 4.6. *Let S be a set of clauses closed under instance generation. The sets of clauses S and $S{\downarrow}$ are equisatisfiable.*

Their proof, moreover, shows that given a model $\mathcal{I}$ of the propositional set of clauses $S{\downarrow}$, the induced interpretation $\mathcal{I}_S$ is a model of the saturated set S. Korovin (2006) also implemented this calculus in the iProver system, using a saturation strategy based on the given-clause algorithm (McCune, 1994b), a procedure commonly found in resolution theorem provers. Additional simplifications, which detect and eliminate redundancies based on resolution and other approaches, made iProver a successful contestant in the EPR category of the latest CADE ATP System Competitions (Sutcliffe, 2007).

4.4 Finite domain predicate logic

In the previous section we introduced both the syntax and semantics of effectively propositional logic, which is the main focus of this thesis. As we have already argued, this language is much more concise than propositional encodings and also clearer for humans to read and understand. Nevertheless, the language often lacks of a few constructs which are commonly used in applications such as functions and quantifiers.

Function symbols and quantifiers are, of course, what distinguishes effectively propositional formulae from first-order logic. But it turns out, however, that if one limits the logic to a fixed finite domain, then it is still possible to reduce the resulting logic to the effectively propositional case.

In this section we introduce the finite domain predicate logic. This is a sorted logic which allows the use of function symbols, quantification and even equality. Then, we will also show how to reduce formulae in this logic to the effectively propositional case as introduced first in Section 4.1. The added syntactic sugar will be useful in later chapters to describe the encodings of applications in a more natural fashion.

Definition 4.10. The language of *finite domain predicate logic* consists of a number of non-empty disjoint sets of *constant symbols*, which we denote by sans-serif uppercase letters: A, B, $\ldots$. Each of these sets is also known as a *sort*. It also also has a set of *predicate symbols* $\mathcal{P}$ and a set of function symbols $\mathcal{F}$. Moreover, each predicate and function symbol is associated with a *signature*, which for a predicate symbol is an expression of the form $\mathsf{A}_1 \times \cdots \times \mathsf{A}_n$ and for a function symbol of the form $\mathsf{A}_1 \times \cdots \times \mathsf{A}_n \to \mathsf{A}$. In either case n is the arity of the symbol.

For each sort we also have an infinite supply of variables which are denoted, as usual, by the letters x, y, $\ldots$. Terms are defined inductively as follows: an expression t is a *term* of sort A if

- t is a variable of sort A.
- t is a constant symbol in A.

- t is an expression of the form $f(t_1, \ldots, t_n)$, where f is a function symbol with signature $\mathsf{A}_1 \times \cdots \times \mathsf{A}_n \to \mathsf{A}$ and each t_i is a term of sort A_i.

A *predicate atom* is an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol with signature $\mathsf{A}_1 \times \cdots \times \mathsf{A}_n$ and each t_i is a term of sort A_i. And an *equality atom* is an expression of the form $t_1 = t_2$ where both t_1 and t_2 are terms of the same sort. An *atom* is either a predicate atom or an equality atom.

Using atoms as basic building blocks, we construct formulae by using the the primitive connectives of *falsity* ($\bot$), *negation* ($\neg F$), *conjunction* ($F \wedge G$) and *quantification* ($\forall x.\, F$); where F and G are formulae, and x is a variable. Duals of these operators and additional connectives are introduced as abbreviations:

$$\begin{aligned} \top &\equiv \neg\bot & \exists x.\, F &\equiv \neg(\forall x.\, \neg F) \\ F \vee G &\equiv \neg(\neg F \wedge \neg G) & F \to G &\equiv \neg F \vee G \end{aligned}$$

The standard notion of *free* and *bound* variables with respect to the scope of quantifiers also applies here. A *closed formula* is a formula with no free variables. We will often use $\bar{x}$ to denote a sequence of variables $x_1, \ldots, x_n$ whose length is specified in the context where it is used. This allows, for example, to write $\exists \bar{x}.\, F$ instead of the longer expression $\exists x_1. \ldots \exists x_n.\, F$. Similarly we will write $\bar{x} = \bar{y}$ as a shorthand for $\bigwedge_{i=1}^{n} x_i = y_i$. ■

A *constraint* is just an alternative name for a formula. A *clause* is special kind of constraint which is just a disjunction of literals, and a formula is said to be in *clausal normal form* if it is a set of clauses.

Definition 4.11. An *interpretation* for a finite domain predicate logic is a structure which assigns:

- for each predicate symbol p a relation $p^{\mathcal{I}} \subseteq \mathsf{A}_1 \times \cdots \times \mathsf{A}_n$, and
- for each function symbol f a function $f^{\mathcal{I}} \colon \mathsf{A}_1 \times \cdots \times \mathsf{A}_n \to \mathsf{A}$,

according to the signature associated with the corresponding symbol. Also, a *variable mapping* σ is a function that maps variables to constant symbols of the appropriate sort. We also say that σ' is an x-variant of σ if they differ, at most, in their valuation for the variable x.

The interpretation of a term t with respect to an interpretation $\mathcal{I}$ and a variable mapping σ, denoted by $t^{\mathcal{I},\sigma}$, is a constant symbol computed as follows:

- if t is a variable, then $t^{\mathcal{I},\sigma} = \sigma(t)$,
- if t is a constant symbol, then $t^{\mathcal{I},\sigma} = t$,
- if t is an expression $f(t_1, \ldots, t_n)$, then $t^{\mathcal{I},\sigma} = f^{\mathcal{I}}(t_1^{\mathcal{I},\sigma}, \ldots, t_n^{\mathcal{I},\sigma})$.

We will now define when an interpretation $\mathcal{I}$ and a variable mapping σ *satisfies* a formula F, denoted $\mathcal{I} \models_\sigma F$, recursively as follows:

$$\begin{array}{lll}
\mathcal{I} \not\models_\sigma \bot & & \\
\mathcal{I} \models_\sigma p(t_1, \ldots, t_n) & \text{iff} & (t_1^{\mathcal{I},\sigma}, \ldots, t_n^{\mathcal{I},\sigma}) \in p^{\mathcal{I}}, \\
\mathcal{I} \models_\sigma t_1 = t_2 & \text{iff} & t_1^{\mathcal{I},\sigma} \text{ is equal to } t_2^{\mathcal{I},\sigma}, \\
\mathcal{I} \models_\sigma \neg F & \text{iff} & \mathcal{I} \not\models_\sigma F, \\
\mathcal{I} \models_\sigma F \wedge G & \text{iff} & \mathcal{I} \models_\sigma F \text{ and } \mathcal{I} \models_\sigma G, \\
\mathcal{I} \models_\sigma \forall x.\, F & \text{iff} & \mathcal{I} \models_{\sigma'} F \text{ for every } \sigma' \text{ which is an } x\text{-variant of } \sigma,
\end{array}$$

An interpretation $\mathcal{I}$ is a *model* of a finite domain predicate formula F and is denoted by $\mathcal{I} \models F$ if, for every variable mapping σ, we have that $\mathcal{I} \models_\sigma F$. We also say that $\mathcal{I}$ is a *model* of a set of formulae, also known as a *set of constraints*, if $\mathcal{I}$ is a model of every constraint in the set. A set of constraints is said to be *satisfiable* if it has at least one model. ■

Observe that the semantics of equality, as introduced in the previous definition, is evaluated syntactically, i.e. using the unique name assumption, with respect to the names of constant symbols after the terms on each side of the equal sign have been evaluated under the current interpretation $\mathcal{I}$ and variable mapping σ. In particular note that constant symbols evaluate to *themselves* and, therefore, the equality or inequality of constant symbols does not actually depend on the interpretation $\mathcal{I}$.

Negation normal form As with other logics, in order to simplify the exposition and avoid dealing with the polarity of subformulae, we assume that formulae are put first in *negation normal form.* It is easy to achieve this by pushing negation inwards and replacing implications with disjunctions. The resulting logical formulae will therefore use any of the connectives: $\bot, \top, \wedge, \vee, \forall, \exists$; and negation is restricted to occur only in literals.

Algorithm 4.5 (Negation normal form translation)**.** The rewrite rule system on formulae of Figure 4.1 gives an algorithm to translate arbitrary finite domain

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
\neg\bot & \Rightarrow & \top & \neg\top & \Rightarrow & \bot \\
\neg(F \wedge G) & \Rightarrow & (\neg F \vee \neg G) & \neg(F \vee G) & \Rightarrow & (\neg F \wedge \neg G) \\
\neg(\forall x.\, F) & \Rightarrow & (\exists x.\, \neg F) & \neg(\exists x.\, F) & \Rightarrow & (\forall x.\, \neg F) \\
\neg\neg A & \Rightarrow & A & & &
\end{array}
$$

Figure 4.1: Rewrite rules for negation normal form translation

predicate formulae into a negation normal form representation. Given an input formula, the algorithm simply applies the rewrite rules in a nondeterministic manner until a negation normal form is obtained.

The reader can easily verify that this algorithm produces a formula in negation normal form which is equivalent to its input. In the following we will assume that all formulae are in negation normal form.

Quantification Note that quantification of variables is done with respect to finite sorts, so that a formula $\forall x.\, F(x)$ can actually be unfolded into the conjunction $F(c_1) \wedge \cdots \wedge F(c_k)$, where $\mathsf{A} = \{c_1, \ldots, c_k\}$ is the corresponding sort of the variable x. Nevertheless, naively unfolding quantifiers in this way will potentially produce an exponential blowup in the size of the formula. We therefore now introduce an alternative approach which, following the style of Tseitin's translations discussed in previous chapters, only incurs in a linear size increase.

Definition 4.12. We define the set of constraints Sorts as the set that contains, for every sort $\mathsf{A} = \{c_1, \ldots, c_k\}$ in a finite domain predicate logic, the constraints:

$$
\begin{array}{l}
\mathsf{first}_\mathsf{A}(c_1) \wedge \mathsf{succ}_\mathsf{A}(c_1, c_2) \wedge \cdots \wedge \mathsf{succ}_\mathsf{A}(c_{k-1}, c_k) \wedge \mathsf{last}_\mathsf{A}(c_k) \\
\mathsf{succ}_\mathsf{A}(x, y) \rightarrow \mathsf{in}_\mathsf{A}(x) \wedge \mathsf{in}_\mathsf{A}(y)
\end{array}
$$

where succ_A, $\mathsf{first}_\mathsf{A}$, last_A, and in_A are fresh new predicate symbols, the first of them with signature $\mathsf{A} \times \mathsf{A}$ and the rest with signature A. ■

The intuition behind the constraints in Sorts is to enumerate all constant symbols in a sort A by providing a predicate succ_A that allows one to iterate over them, while the predicates $\mathsf{first}_\mathsf{A}$ and last_A can be used to determine the bounds of this sequence. The intended meaning of $\mathsf{in}_\mathsf{A}(x)$ is to represent that $x \in \mathsf{A}$. Note that Sorts defines only the positive polarity of these predicate symbols. In other words, if $c_i \in \mathsf{A}$ then the predicate $\mathsf{in}_\mathsf{A}(c_i)$ should be true in models of Sorts, but the converse does not necessarily have to hold.

Algorithm 4.6 (Quantifier removal translation)**.** Let Γ be a set of constraints. The set Γ^{ea} is defined as the result of iterating the following procedure until all quantifiers in Γ have been removed.

- If there is a subformula $F = \forall x.\, G(x, \bar{y})$, where $x, \bar{y}$ are all the free variables in G, then replace the subformula F with the atom $\mathsf{forall}_F(\bar{y})$ and add the constraint:

$$\mathsf{in}_\mathsf{A}(x) \land \mathsf{forall}_F(\bar{y}) \to G(x, \bar{y})$$

  where A is the sort of x and forall_F is a fresh new predicate symbol whose signature matches the sorts of the $\bar{y}$ variables.

- Similarly, replace a subformula of the form $F = \exists x.\, G(x, \bar{y})$ with the atom $\mathsf{exists}_F(\bar{y})$ and add the constraints:

$$\begin{aligned}
&\mathsf{first}_\mathsf{A}(x) \land \mathsf{exists}_F(\bar{y}) \to \mathsf{find}_F(x, \bar{y})\\
&\mathsf{find}_F(x, \bar{y}) \to G(x, \bar{y}) \lor \mathsf{xfind}_F(x, \bar{y})\\
&\mathsf{succ}_\mathsf{A}(x, z) \land \mathsf{xfind}_F(x, \bar{y}) \to \mathsf{find}_F(z, \bar{y})\\
&\mathsf{last}_\mathsf{A}(x) \land \mathsf{xfind}_F(x, \bar{y}) \to \bot
\end{aligned}$$

  where A is the sort of x and exists_F, find_F, xfind_F are fresh new predicate symbols with appropriate signatures.

Theorem 4.7. *The sets of constraints* Γ *and* $\mathsf{Sorts} \cup \Gamma^{\mathsf{ea}}$ *are equisatisfiable.*

Proof. The argument is similar to the one used in structural clausal form translations where subformulae are replaced by fresh new atoms and constraints are added to give a meaning to those atoms.

In particular it can be shown that, if $F = \forall x.\, G(x, \bar{y})$ and a given interpretation $\mathcal{I}$ is a model of $\mathsf{Sorts} \cup \{F\}^{\mathsf{ea}} \cup \{\mathsf{forall}_F(\bar{y})\}$, then $\mathcal{I}$ is also a model of F. And, for the converse, that if $\mathcal{I} \models F$ then it is always possible to find an interpretation $\mathcal{I}'$ such that $\mathcal{I}' \models \mathsf{Sorts} \cup \{F\}^{\mathsf{ea}} \cup \{\mathsf{forall}_F(\bar{y})\}$. The case for the existential quantifier is also analogous. $\square$

After Algorithm 4.6 has been applied to to soak existentials and remove all quantifiers, a standard translation such as the one by Plaisted and Greenbaum (1986) can be applied to obtain a set of constraints in clausal normal form. In the following we will assume that formulae are always put in clausal normal form, although function symbols and equality are still allowed to appear anywhere in the literals of clauses.

Flattening A first step in order to remove function symbols is to employ the process known as *flattening*. This allows one to unfold the nesting of terms in order to more easily remove them in the later steps.

Definition 4.13. A term is *shallow* if it is either a variable or a constant symbol, and an atom is *shallow* if it is either of the form:

- $p(t_1, \ldots, t_n)$,
- $f(t_1, \ldots, t_n) = t$, or
- $t_1 = t_2$,

where all terms t_i and t are shallow. ■

The following algorithm is used to flatten a clause by making all terms shallow.

Algorithm 4.7 (Clause flattening algorithm)**.** Given an input clause C iterate the following procedure until all atoms are shallow: If there is a subterm t in C which is not shallow then:

- Replace all occurrences of t in C with a new variable x of the same sort.
- Replace C with $t \neq x \vee C$.

It is easy to see that the output clause of the previous algorithm is logically equivalent to its input. Moreover, if it is iteratively applied to all the clauses in a set Γ to obtain another set Γ' all whose atoms are shallow, then Γ and Γ' are also equivalent.

Function symbols On a flattened formula it is now easy to remove function symbols by replacing them with new predicate symbols.

Definition 4.14. Let Functs be a set that contains, for every function symbol f of signature $\mathsf{A}_1 \times \cdots \times \mathsf{A}_n \to \mathsf{A}$ in a finite domain predicate logic, the constraints:

$$\begin{aligned}&\exists y.\, \hat{f}(x_1, \ldots, x_n, y)\\ &\hat{f}(x_1, \ldots, x_n, y) \wedge \hat{f}(x_1, \ldots, x_n, z) \to y = z\end{aligned}$$

where $\hat{f}$ is a fresh new predicate symbol with a sort of $\mathsf{A}_1 \times \cdots \times \mathsf{A}_n \times \mathsf{A}$. ■

Theorem 4.8. *Let Γ be a set of flattened clauses, and let Γ^{fn} be the set obtained after replacing all shallow atoms of the form $f(t_1, \ldots, t_n) = t$ with $\hat{f}(t_1, \ldots, t_n, t)$. The sets Γ and $\Gamma' = \mathsf{Sorts} \cup \mathsf{Functs}^{\mathsf{ea}} \cup \Gamma^{\mathsf{fn}}$ are equisatisfiable. Moreover, Γ' is a set of flattened clauses that does not contain neither function symbols nor quantifiers.*

Proof. The result easily follows by the observation that, if $\mathcal{I}$ is a model of Functs, then the relation assigned to a predicate symbol $\hat{f}^{\mathcal{I}}$ must actually be a function. Using this observation, it is easy to build an interpretation $\mathcal{I}'$ which behaves on the function symbol f as $\mathcal{I}$ behaves on the predicate symbol $\hat{f}$ and, therefore, transfer models between Γ and Γ'. □

Equality After removing function symbols and quantifiers, we only have left to remove equality.

Definition 4.15. The set Equals is defined as the set containing, for every sort A in a finite domain predicate logic, the constraints:

$$\begin{aligned}&\mathsf{succ}_{\mathsf{A}}(x, y) \to \mathsf{less}_{\mathsf{A}}(x, y)\\ &\mathsf{less}_{\mathsf{A}}(x, y) \wedge \mathsf{less}_{\mathsf{A}}(y, z) \to \mathsf{less}_{\mathsf{A}}(x, z)\\ &\mathsf{less}_{\mathsf{A}}(x, y) \to \neg\mathsf{eq}_{\mathsf{A}}(x, y) \wedge \neg\mathsf{eq}_{\mathsf{A}}(y, x)\\ &\mathsf{eq}_{\mathsf{A}}(x, x)\end{aligned}$$

where $\mathsf{less}_{\mathsf{A}}$ and eq_{A} are fresh new predicate symbols with a signature of $\mathsf{A} \times \mathsf{A}$. ■

Theorem 4.9. *Let Γ be a set of flattened clauses, and let Γ^{eq} be the set obtained after replacing all shallow atoms of the form $t_1 = t_2$ with $\mathsf{eq}_{\mathsf{A}}(t_1, t_2)$ for the appropriate sort A. The sets Γ and $\Gamma' = \mathsf{Sorts} \cup \mathsf{Equals} \cup \Gamma^{\mathsf{eq}}$ are equisatisfiable. Moreover, Γ' is a set of flattened clauses that does not contain equality.*

Proof. The result easily follows by the observation that, if an interpretation $\mathcal{I}$ is a model of $\mathsf{Equals} \cup \{\mathsf{eq}_\mathsf{A}(t_1, t_2)\}$ then it is also the case that $\mathcal{I} \models t_1 = t_2$. From this observation it is then possible to transfer models between Γ and Γ'. □

As a result of Algorithms 4.5, 4.6, and 4.7 together with the translation of Plaisted and Greenbaum (1986), as well as Theorems 4.8 and 4.9, one can translate any arbitrary finite domain predicate formulae into a set of clauses without any quantification, function symbols or equality. Syntactically, this corresponds exactly with the effectively propositional formulae as defined in Section 4.1. However, since the generated formulae still have to be evaluated taking into account the sort information, semantically the two classes of formulae are different. The following theorem will, therefore, fill in the remaining gap.

Theorem 4.10. *Let F be a set of plain clauses written in the language of a finite domain predicate logic. The set F has a finite domain predicate model if and only if it has an effectively propositional model.*

Proof. It actually follows from Theorem 4.3, since interpreting F with respect to the finite domain predicate logic, is equivalent to checking the satisfiability of the ground instances $F|_A$ in effectively propositional logic, where A encodes the original sort information of the logic. □

Moreover, the resulting formula is of linear size with respect to the original input. In the following we will then freely use equality and finite quantification knowing that they do not add any complexity to the logic.

4.5 Chapter summary

In this chapter we have introduced the syntax and semantics of effectively propositional logic. This is a fragment of first-order logic which, we argue, is a better alternative, compared to propositional logic, both to encode and solve problems derived from applications.

Although the use of effectively propositional language, also known in literature as the Bernays-Schönfinkel class of formulae, is rather new in the automated reasoning community; there already are some theoretical foundations and implementations of systems that more efficiently deal with this particular logic. The central sections of this chapter give a quick overview on the state of the art of such technologies and existing reasoning methods.

Finally, we also introduced the finite domain predicate logic, which corresponds to a syntactic sugar placed on top of the effectively propositional language. Therefore, it still remains as a decidable fragment of first-order logic with features such as equality, function symbols and finite quantification. The motivation for developing such a logic is that it enables us to succinctly and naturally encode problems from applications, as we will do in the following Chapters 5 and 6.

Chapter 5

Encoding LTL bounded model checking

Model checking is a technique suitable for verifying that a hardware or software component works according to some formally specified expected behaviour. This is usually done by building a description of the system, often modelled as a finite state machine in a formal language suitable for further deployment, and using a temporal logic to specify the properties that the system is expected to satisfy.

One of the first accomplishments in model checking consists in the use of symbolic model checkers (McMillan, 1993), where the transition system of the finite state machine is symbolically, rather than explicitly, represented. These symbolic representations, which usually take the form of a binary decision diagram (BDD), provide significant improvements over previous techniques; but some formulae are still hard to encode succinctly using BDDs and, moreover, the encoding itself is often highly sensitive to the variable order used to create the representation.

Another significant achievement in the state of the art of model checking came when Biere, Cimatti, Clarke, and Zhu (1999) proposed the now widely known technique of bounded model checking (BMC). In bounded model checking instead of trying to prove the correctness of the given property, one searches for counterexamples within executions of the system of a bounded length. A propositional formula is created and a decision procedure for propositional logic, such as DPLL (Davis et al., 1962), is used to find models which, in turn, represent bugs in the system. When no models are found the bound is increased trying to search for longer counterexamples.

Although this basic method is not complete, i.e. it is not able to prove properties but rather just *disprove* them, it has been found as a useful tool to quickly find simple bugs in systems (Biere et al., 1999; Copty et al., 2001; Strichman, 2004) and as a good complement to other BDD-based techniques. A significant amount of research has been spent recently on extending this technique to more expressive temporal logics (Jehle et al., 2005), obtaining better propositional encodings (Latvala et al., 2004), and proposing termination checks to regain completeness (Prasad et al., 2005). A recent survey on the state of the art is found in the work of Biere et al. (2006).

Bounded model checking has been largely focused on generating and solving problems encoded in propositional logic. We observe, however, that bounded model checking problems can also be easily and naturally encoded within the Bernays-Schönfinkel class of formulae. One of our motivations is to obtain a new source of problems for first-order reasoners, particularly those geared towards the effectively propositional division (EPR) of the CASC system competitions (Sutcliffe and Suttner, 2006).

Moreover, we believe that the EPR encoding has several advantages over the propositional approach. First, it gives a more succinct and natural description of both the system and the property to verify. It is not needed, for example, to replicate copies of the temporal formula for every step of the execution trace where it has to be checked. Furthermore, it is possible to directly translate systems descriptions which are written in a modular way, without requiring to flatten or expand module definitions beforehand. A prover can potentially use this information to better organise the search for a proof or counterexamples.

On the other hand, our encoding may also turn out to be useful for propositional, satisfiability-based, approaches to bounded model checking. Indeed, it preserves the structure of the original bounded model checking problem in the obtained effectively propositional formula and reduces the problem of finding an optimised propositional encoding of the model checking problem to the problem of finding an optimised propositional instantiation of the effectively propositional description.

After introducing a number of formal definitions in Section 5.1, we present in Section 5.2 two different encodings of linear temporal logic (LTL) into effectively propositional formulae. The first encoding takes as input an LTL formula and a bound k, in order to produce a set of constraints which capture the execution paths satisfying the temporal property. The second encoding is an improvement on the first that produces two sets of constraints: one that depends on the LTL formula only (i.e. not the bound) and its output is linear with respect to its input; and another, with a size of $O(k)$, that depends on the bound k only. Compare with propositional encodings where, if n is the size of the LTL formula, the output is typically of size $O(nk)$ instead of $O(n + k)$ with our approach. Furthermore, with a binary encoding of states, the size of the later component is reduced to $O(\log^2 k)$.

We also present, in Section 5.3, an approach to encode of modular descriptions of model checking problems while preserving their modularity and hierarchical representation. We show, in particular, how several features of a software and hardware description language such as SMV (Cimatti et al., 2002) can be easily represented within the effectively propositional fragment. Using the ideas depicted here, it is also possible to develop a tool to translate system descriptions in industry standard formats (e.g. Verilog HDL or SMV) automatically into a format such as TPTP (Sutcliffe and Suttner, 1998) suitable for consumption by first-order theorem provers.

5.1 Introduction to model checking

In this section we introduce the main formal definitions that we will use in the context of model checking. We first define the linear temporal logic (LTL) in a way that closely follows the standard definitions found in literature but with a few modifications to better represent the notion of bounded executions.

5.1.1 Linear temporal logic

This is a temporal logic which deals with individual executions of a system. In can represent properties such that “there is some execution path in which the system eventually reaches some particular state” and that “in all execution paths, two atomic predicates will never be set simultaneously to true”. Formally, we start by defining the notion of an execution path.

Definition 5.1. Let $\mathcal{V} = \{p_1, \dots, p_n\}$ be a set of elements called *state variables.* A subset $s \subseteq \mathcal{V}$ is known as a *state.*

A *path* $\pi = s_0 s_1 \dots$ is a, finite or infinite, sequence of states. The *length* of a finite path $\pi = s_0 \dots s_k$, denoted by $|\pi|$, is $k + 1$; while, for an infinite path, we define $|\pi| = \omega$, where $\omega > k$ for every number k.

A *k-path* is either a finite path of the form $\pi = s_0 \dots s_k$, or an infinite path with a loop of the form $\pi = s_0 \dots s_{l-1} s_l \dots s_k s_l \dots s_k \dots$, in the sequel also written as $\pi = s_0 \dots s_{l-1}(s_l \dots s_k)^\omega$. ■

We will assume that system executions are always infinite paths, i.e. there are no deadlock states. Finite paths, however, are also needed to represent the prefix of an execution of the system up to a bounded length. With this intuition in mind we now define the semantics of LTL formulae in negation normal form; these are formulae built using propositional and temporal connectives, but negation is only allowed in front of atomic propositions.

This is a technical restriction that simplifies the presentation of our results and the definition of the logic itself, but it is not a limitation in practise since it is possible to put arbitrary LTL formulae into negation normal form by applying a simple translation.

Definition 5.2. A path $\pi = s_0, s_1, \dots$ is a *model* of an LTL formula ϕ at a state s_i, where $i < |\pi|$, denoted by $\pi \models_i \phi$, if

$$\begin{array}{lll}
\pi \models_i p & \text{iff} & p \in s_i, \\
\pi \models_i \neg p & \text{iff} & p \notin s_i, \\
\pi \models_i \psi \wedge \phi & \text{iff} & \pi \models_i \psi \text{ and } \pi \models_i \phi, \\
\pi \models_i \psi \vee \phi & \text{iff} & \pi \models_i \psi \;\text{ or }\; \pi \models_i \phi, \\
\pi \models_i \mathbf{X}\phi & \text{iff} & i+1 < |\pi| \text{ and } \pi \models_{i+1} \phi, \\
\pi \models_i \mathbf{F}\phi & \text{iff} & \exists j,\, i \le j < |\pi|,\, \pi \models_j \phi, \\
\pi \models_i \psi\mathbf{W}\phi & \text{iff} & \text{either: } \pi \text{ is infinite and } \forall j,\, i \le j,\, \pi \models_j \psi, \\
 & & \text{or: } \exists j',\, i \le j' < |\pi|,\, \pi \models_{j'} \phi \text{ and } \forall j,\, i \le j < j',\, \pi \models_j \psi.
\end{array}$$

Also π is a model of $\top$ for every state s_i with $i < |\pi|$, and of $\bot$ for no state. We write $\pi \models \phi$ to denote $\pi \models_0 \phi$. ■

Note that we introduced the *weak until*, $\mathbf{W}$, as a primary connective of our temporal logic. Other standard temporal connectives —such as *until*, *release* and *globally*— are introduced as abbreviations of the other existing connectives: $\psi\mathbf{U}\phi = \mathbf{F}\phi \wedge (\psi\mathbf{W}\phi)$, $\psi\mathbf{R}\phi = \phi\mathbf{W}(\psi \wedge \phi)$, and $\mathbf{G}\phi = \phi\mathbf{W}\bot$.

If we consider infinite paths only, then the definition given matches the standard notion of LTL that is often found in literature; in particular dualities such

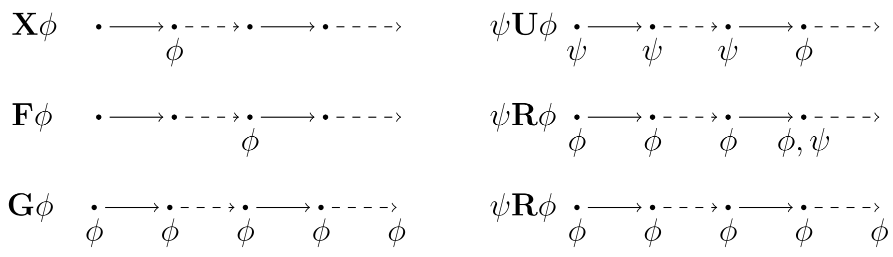

$\mathbf{X}\phi$
ϕ
$\mathbf{F}\phi$
ϕ
$\mathbf{G}\phi$
ϕ ϕ ϕ ϕ ϕ
$\psi\mathbf{U}\phi$
ψ ψ ψ ϕ
$\psi\mathbf{R}\phi$
ϕ ϕ ϕ ϕ, ψ
$\psi\mathbf{R}\phi$
ϕ ϕ ϕ ϕ ϕ

Figure 5.1: Example execution paths of LTL semantics

as $\neg\mathbf{F}\phi \equiv \mathbf{G}\neg\phi$ do hold. Since we assume that system executions are always infinite, one can make use of these identities and put formulae into negation normal form without any loss of generality.

Now the finite case is defined so that if $\pi \models_i \phi$ then, for all possible infinite paths π' extending π, it is also the case that $\pi' \models_i \phi$. Here we deviate a little from usual definitions of LTL and dualities such as the above-mentioned do not hold anymore. For example, neither $\mathbf{F}\phi$ nor $\mathbf{G}\neg\phi$ hold in a finite path where $\neg\phi$ holds at all states. In particular, since finite paths in the temporal logic defined are interpreted as prefixes of longer paths, it is not possible to write a formula to test for the end of a path.

5.1.2 Kripke structures

In the context of model checking, the systems that we want to verify also need to be abstracted using some particular formalism. As commonly done in literature, we will assume that systems are described by Kripke structures, which simply are transition systems representing how the execution of the system in question can change from one state to another. The following is the formal definition.

Definition 5.3. A *Kripke structure* over a set of state variables $\mathcal{V}$ is a tuple $M = (S, I, T)$ where $S = 2^{\mathcal{V}}$ is the set of all states, $I \subseteq S$ is a set whose elements are called *initial states*, and T is a binary relation on states, $T \subseteq S \times S$, called the *transition relation* of the system. We also make the assumption that the transition relation is total, i.e. for every state $s \in S$ there is a state $s' \in S$ such that $(s, s') \in T$.

A *path* $\pi = s_0 s_1 \ldots$ *is in the structure* M if $s_0 \in I$ and for every $0 < i < |\pi|$ we have $(s_{i-1}, s_i) \in T$. We say that a path π in M is a *prefix path* if it is finite, and a *proper path* otherwise.

An LTL formula ϕ is *satisfiable* in a Kripke structure M if there is a proper path π in M such that $\pi \models \phi$. Similarly, a formula ϕ is *valid* in M if, for every proper path π in M, $\pi \models \phi$. ■

Note that, if π is a prefix path in M and $\pi \models \phi$, then for every extension π' of π we also have $\pi' \models \phi$, thus prefix paths are enough for testing satisfiability. Observe, however, that formulae such as $\mathbf{G}\psi$ or $\psi \mathbf{W} \phi$ (where ϕ never holds) are never satisfied by (finite) prefix paths.

Now, while the symbol s_i represents a state in a path, we will use s_i to represent a constant symbol in a predicate formulae. The similar notation was intentionally chosen since a constant s_i will be used as a symbolic representation of a state s_i. The intended meaning should always be clear by context, but a different typeface is also used as a hint to distinguish the two possibilities.

Similarly, it is assumed throughout this section that the set $\mathcal{P}$ of predicate symbols contains a unary predicate symbol p for every state variable $p \in \mathcal{V}$. The atom $\mathsf{p}(\mathsf{s}_i)$ symbolically represents the fact that a variable p is true at the state s_i of a path (i.e. $p \in s_i$), and the symbol $\mathcal{P}_\mathcal{V}$ denotes the set of predicates representing state variables. Our next aim is to define the notion of a symbolic representation of Kripke structures along the lines of representations commonly used in the propositional case.

Let us define the *canonical first-order structure* for $\mathcal{P}_\mathcal{V}$, denoted by $C_\mathcal{V}$. This structure is an interpretation which, instead of the symbolic representations s_i used elsewhere, draws constant symbols from the domain $2^\mathcal{V}$, its signature is the set of predicate symbols $\mathcal{P}_\mathcal{V}$, and the interpretation of every predicate $\mathsf{p} \in \mathcal{P}_\mathcal{V}$ is defined as $C_\mathcal{V} \models \mathsf{p}(s)$ iff $p \in s$.

Definition 5.4. Let $I(x)$ and $T(x, y)$ be two predicate formulae of variables x and x, y, respectively, using predicate symbols $\mathcal{P}_\mathcal{V}$ and no constants. We say that this pair of formulae *symbolically represents* a Kripke structure M if

1. a state s is an initial state of M iff $C_\mathcal{V} \models I(s)$.
2. a pair (s, s') belongs to the transition relation of M iff $C_\mathcal{V} \models T(s, s')$. ■

The idea described in this definition is extended to represent paths in a Kripke structure M by a Herbrand interpretation as follows.

Definition 5.5. Given an interpretation $\mathcal{I}$ over the domain $\mathcal{D} = \{\mathsf{s}_0, \ldots, \mathsf{s}_k\}$, we define *the k-path induced by $\mathcal{I}$*, denoted by $\pi^\mathcal{I}$, by $\pi^\mathcal{I} = s_0^\mathcal{I} \ldots s_k^\mathcal{I}$, where

$s_i^{\mathcal{I}} = \{p \in \mathcal{V} \mid \mathcal{I} \models \mathsf{p}(\mathsf{s}_i)\}$, for all $0 \leq i \leq k$. We will rather informally refer to the states $s_i^{\mathcal{I}}$ as *induced states*. For the induced k-path $\pi^{\mathcal{I}}$ we will often omit the superscripts on the induced states and simply write $\pi^{\mathcal{I}} = s_0 \dots s_k$.

Given a value l with $0 \leq l \leq k$, we also introduce the notation $\pi^{l,\mathcal{I}}$ to denote the infinite k-path $s_0 \dots s_{l-1}(s_l \dots s_k)^{\omega}$ with a loop starting at s_l. ■

In the sequel we will assume that the set of initial states I and the transition relation T of our Kripke structures are always symbolically described in this way. Definition 5.5 immediately implies the following fact.

Lemma 5.1. *Let $M = (S, I, T)$ be a Kripke structure and $\mathcal{I}$ an interpretation.*

1. *$\mathcal{I} \models I(\mathsf{s}_i)$ iff $s_i^{\mathcal{I}}$ is an initial state of M.*

2. *$\mathcal{I} \models T(\mathsf{s}_i, \mathsf{s}_j)$ iff $(s_i^{\mathcal{I}}, s_j^{\mathcal{I}})$ belongs to the transition relation of M.*

5.2 Encoding of temporal properties

In this section we present a translation that allows one to encode an LTL formula as an effectively propositional formula. Following the results from Biere et al. (1999), it has been shown that, if one wants to check the satisfiability of an LTL formula, it is enough to search for k-paths that satisfy this formula.

Theorem 5.1 (Biere et al., 1999)**.** *An LTL formula ϕ is satisfiable in a Kripke structure M iff, for some k, there is a k-path π in M with $\pi \models \phi$.*

Our translation makes use of this result by creating, for a given value k and a Kripke structure M, a predicate formula whose models correspond to k-paths of the system satisfying the original LTL formula (the details of such correspondence are given later in Proposition 5.2). We begin giving a set of constraints that characterise the k-paths of Kripke structures and define some auxiliary symbols, which are used later in the translation.

Definition 5.6. Let $M = (S, I, T)$ be a Kripke structure, and also let $k \geq 0$. The *predicate encoding of k-paths*, denoted by $[k]$, is defined as the set of constraints:

$$\begin{aligned}
&\mathsf{succ}(\mathsf{s}_0, \mathsf{s}_1)\\
&\mathsf{succ}(\mathsf{s}_1, \mathsf{s}_2)\\
&\dots\\
&\mathsf{succ}(\mathsf{s}_{k-1}, \mathsf{s}_k)\\
&\mathsf{succ}(x, y) \to \mathsf{less}(x, y)\\
&\mathsf{succ}(x, y) \land \mathsf{less}(y, z) \to \mathsf{less}(x, z)\\
&\mathsf{succ}(x, y) \to \mathsf{trans}(x, y)\\
&\mathsf{hasloop} \to \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_0) \lor \cdots \lor \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_k)
\end{aligned}$$

And the *predicate encoding* of the structure M, denoted by $[M]$, is defined as:

$$\begin{aligned}
&\mathsf{trans}(x, y) \to T(x, y)\\
&I(\mathsf{s}_0)
\end{aligned}$$

We also define $[M, k] = [M] \cup [k]$. Note that of the predicates $\mathsf{succ}(x, y)$, $\mathsf{less}(x, y)$, $\mathsf{trans}(x, y)$ and $\mathsf{hasloop}$ are fresh new predicates not in $\mathcal{P}_\mathcal{V}$. ■

The intuition behind the predicates introduced in the previous definition is to model paths in the Kripke structure. It easily follows, for example, that if an interpretation $\mathcal{I} \models \mathsf{trans}(\mathsf{s}_i, \mathsf{s}_j)$ then the pair $(s_i^\mathcal{I}, s_j^\mathcal{I})$ is in the transition relation of the structure. The encoding of temporal formulae will then use the $\mathsf{hasloop}$ predicate as a trigger that enforces, since it would make $\mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$ true for some l, the paths accepted as models to be infinite. The following proposition summarises important properties of the models of $[M, k]$.

Proposition 5.1. *Let M be a Kripke structure, and let $\mathcal{I}$ be a model of the set of constraints $[M, k]$. Then for every $0 \leq i, j, l \leq k$:*

1. *If $i < j$ then $\mathcal{I} \models \mathsf{less}(\mathsf{s}_i, \mathsf{s}_j)$.*
2. *The induced k-path $\pi^\mathcal{I} = s_0 \dots s_k$ is a finite path in M.*
3. *If $\mathcal{I} \models \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$ then the induced k-path $\pi^{l,\mathcal{I}} = s_0 \dots s_{l-1}(s_l \dots s_k)^\omega$ is an infinite path in M.*

Proof.

1. By a straightforward induction, from the first $k + 2$ constraints in $[k]$.
2. Since $\mathcal{I} \models I(\mathsf{s}_0)$ it immediately follows by Lemma 5.1 that the induced state s_0 is an initial state of the system. It is easy to see that the constraints imply $T(\mathsf{s}_i, \mathsf{s}_{i+1})$ for all $i < k$ so, by the same lemma, the pair (s_i, s_{i+1}) belongs to the transition relation of M, hence $s_0, \dots, s_k$ is a path in M.

3. If $\mathcal{I} \models \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$ it follows again that (s_k, s_l) belongs to the transition relation and, together with the previous item, it proves that $\pi^{l,\mathcal{I}}$ is an infinite k-path in M. □

The following two definitions give the translation of an LTL formula ϕ into a predicate encoding following an approach similar to structural clausal form translations: a new predicate symbol is first introduced to represent each subformula, here denoted by $\Theta_\phi(x)$ in Definition 5.7, and then a set of constraints, given in Definition 5.8, are added to give $\Theta_\phi(x)$ its intended meaning.

Definition 5.7. We define the *symbolic representation* of an LTL formula γ, a predicate formula $\Theta_\gamma(x)$, as follows:

$$\begin{aligned}
\Theta_\top(x) &= \top & \Theta_\bot(x) &= \bot \\
\Theta_p(x) &= \mathsf{p}(x) & \Theta_{\neg p}(x) &= \neg\mathsf{p}(x) \\
\Theta_{\psi\wedge\phi}(x) &= \Theta_\psi(x) \wedge \Theta_\phi(x) & \Theta_{\psi\vee\phi}(x) &= \Theta_\psi(x) \vee \Theta_\phi(x) \\
\Theta_{\mathbf{X}\phi}(x) &= \mathsf{next}_\phi(x) & \Theta_{\mathbf{F}\phi}(x) &= \mathsf{evently}_\phi(x) \\
\Theta_{\psi\mathbf{W}\phi}(x) &= \mathsf{weak}_{\psi,\phi}(x)
\end{aligned}$$

where $\mathsf{next}_\phi(x)$, $\mathsf{evently}_\phi(x)$ and $\mathsf{weak}_{\psi,\phi}(x)$ are fresh new predicates, not already in $\mathcal{P}_\mathcal{V}$, introduced as needed for subformulae of γ. ■

Definition 5.8. For every pair of LTL formulae ψ, ϕ and a value $k \geq 0$, we define the following sets of constraints:

$\Phi^k_{\mathbf{X}\phi}$: x1: $\mathsf{next}_\phi(x) \wedge \mathsf{trans}(x,y) \rightarrow \Theta_\phi(y)$
x2: $\mathsf{next}_\phi(\mathsf{s}_k) \rightarrow \mathsf{hasloop}$

$\Phi^k_{\mathbf{F}\phi}$: f1: $\mathsf{evently}_\phi(x) \rightarrow \mathsf{event}_\phi(x,\mathsf{s}_0) \vee \cdots \vee \mathsf{event}_\phi(x,\mathsf{s}_k)$
f2: $\mathsf{event}_\phi(x,y) \rightarrow \Theta_\phi(y)$
f3: $\mathsf{event}_\phi(x,y) \wedge \mathsf{less}(y,x) \rightarrow \mathsf{hasloop}$
f4: $\mathsf{event}_\phi(x,y) \wedge \mathsf{less}(y,x) \wedge \mathsf{trans}(\mathsf{s}_k,z) \wedge \mathsf{less}(y,z) \rightarrow \bot$

$\Phi^k_{\psi\mathbf{W}\phi}$: w1: $\mathsf{weak}_{\psi,\phi}(x) \rightarrow \Theta_\phi(x) \vee \mathsf{xweak}_{\psi,\phi}(x)$
w2: $\mathsf{xweak}_{\psi,\phi}(x) \wedge \mathsf{trans}(x,y) \rightarrow \mathsf{weak}_{\psi,\phi}(y)$
w3: $\mathsf{xweak}_{\psi,\phi}(x) \rightarrow \Theta_\psi(x)$
w4: $\mathsf{xweak}_{\psi,\phi}(\mathsf{s}_k) \rightarrow \mathsf{hasloop}$

Again $\mathsf{event}_\phi(x, y)$ and $\mathsf{xweak}_{\psi,\phi}(x)$ are fresh new predicates not in $\mathcal{P}_\mathcal{V}$.

We finally introduce the set of *structural definitions* of an LTL formula γ (with depth k), denoted by $[\gamma, k]$, as the union of the sets Φ^k_ϕ for every temporal subformula ϕ of the original γ. ■

Later in Proposition 5.2 we show how the models of such formulae relate to the k-paths satisfying an LTL formula. We need first to introduce the concept of a *rolling function* which will be used as a tool in the proof of such proposition.

Definition 5.9. Given a k-path π we define its *rolling function* δ, a function defined for every $0 \leq i < |\pi|$ and with range $\{0, \dots k\}$, as follows:

- If π is of the form $s_0 \dots s_{l-1}(s_l \dots s_k)^\omega$, then

$$\delta(i) = \begin{cases} i & i \leq k \\ l + [(i - l) \bmod (k + 1 - l)] & \text{otherwise}. \end{cases}$$

- Otherwise, if $\pi = s_0 \dots s_k$, then $\delta(i) = i$ for every $0 \leq i < |\pi|$. ■

The rolling function is a notational convenience used to unfold an infinite k-path $\pi = s_0 \dots s_{l-1}(s_l \dots s_k)^\omega$ as the sequence $\pi = s_{\delta(0)} s_{\delta(1)} \dots$, without explicitly showing the loop. We emphasise the fact that the rolling function is defined only when $0 \leq i < |\pi|$; in particular, if π is finite, the function is not defined for indices outside of the path. Also notice that, for both finite and infinite paths, the rolling function acts as the identity for all i with $0 \leq i \leq k$. Moreover, for $0 \leq i < |\pi|$, it is always the case that $s_i = s_{\delta(i)}$; in fact, the following stronger result holds.

Lemma 5.2. *Let π be a k-path, ϕ an LTL formula, $i < |\pi|$ and δ the rolling function of π. Then it follows that $\pi \models_i \phi$ if and only if $\pi \models_{\delta(i)} \phi$.*

We will now prove one of the main propositions which shows how to obtain, from models of the encoded formula, a k-path in the given Kripke structure that, moreover, satisfies the original LTL formula at a particular state.

Proposition 5.2. *Let M be a Kripke structure, γ an LTL formula, and $\mathcal{I}$ a model of the formula $[M, k] \cup [\gamma, k]$ with domain $D = \{\mathsf{s}_0, \dots, \mathsf{s}_k\}$. We define a path π according to the following two cases:*

1. *If $\mathcal{I} \models \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$, for some $0 \leq l \leq k$, then let $\pi = \pi^{l,\mathcal{I}}$ for any such l.*

2. If $\mathcal{I} \not\models \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$*, for every* $0 \leq l \leq k$*, then let* $\pi = \pi^{\mathcal{I}}$*.*

Let $i < |\pi|$*, and let* δ *be the rolling function of* π*. If* $\mathcal{I} \models \Theta_\gamma(\mathsf{s}_{\delta(i)})$ *then* $\pi \models_i \gamma$*.*

Proof. Although the proofs for items 1 and 2 are independent, we will prove them together since most of their arguments can be shared. Before starting we define, for a formula ϕ, the set

$$J_\phi = \left\{ j \mid i \leq j < |\pi| \text{ and } \mathcal{I} \models \Theta_\phi(\mathsf{s}_{\delta(j)}) \right\} ,$$

which is the set of points after i where the symbolic representation of ϕ holds under the current interpretation. Also observe that, if $0 < j < |\pi|$, it is always the case that $\mathcal{I} \models \mathsf{trans}(\mathsf{s}_{\delta(j-1)}, \mathsf{s}_{\delta(j)})$.

Note as well that, if the interpretation $\mathcal{I} \models \mathsf{hasloop}$, then it would follow from the rule $\mathsf{hasloop} \rightarrow \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_0) \vee \cdots \vee \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_k)$ in $[M, k]$ that $\mathcal{I} \models \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$ for some l and, therefore, π must be an infinite path according to item 1 of the proposition's statement.

Equipped with this information we will now start the proof by induction over the structure of the formula γ.

- If $\gamma = \top$ (or $\bot$), there is nothing to prove since also $\Theta_\gamma(x) = \top$ (or $\bot$).

- If $\gamma = \mathsf{p}$ (or $\neg\mathsf{p}$), then $\mathcal{I} \models \mathsf{p}(\mathsf{s}_{\delta(i)})$ (or $\neg\mathsf{p}(\mathsf{s}_{\delta(i)})$). By definition we know that $p \in s^{\mathcal{I}}_{\delta(i)}$ (or $p \notin s^{\mathcal{I}}_{\delta(i)}$) and, since the states $s_i = s_{\delta(i)}$, we have $\pi \models_i \gamma$.

- If $\gamma = \psi \odot \phi$, where $\odot \in \{\wedge, \vee\}$, then $\mathcal{I} \models \Theta_\psi(\mathsf{s}_{\delta(i)}) \odot \Theta_\phi(\mathsf{s}_{\delta(i)})$ and, by induction, we get that $\pi \models_i \psi \odot \phi$.

- If $\gamma = \mathbf{X}\phi$, then $\mathcal{I} \models \mathsf{next}_\phi(\mathsf{s}_{\delta(i)})$. Now observe that, if $i = k$ then, by rule $\mathsf{x2}$, it would be the case that $\mathcal{I} \models \mathsf{hasloop}$ and we are in the case of an infinite path. In any case we would always have that $i + 1 \leq |\pi|$.

  So now we have that $\mathcal{I} \models \mathsf{trans}(\mathsf{s}_{\delta(i)}, \mathsf{s}_{\delta(i+1)})$, by rule $\mathsf{x1}$ it is also the case that $\mathcal{I} \models \Theta_\phi(\mathsf{s}_{\delta(i+1)})$, by induction $\pi \models_{i+1} \phi$ and, therefore, we are able to conclude $\pi \models_i \mathbf{X}\phi$.

- If $\gamma = \mathbf{F}\phi$, then $\mathcal{I} \models \mathsf{evently}_\phi(\mathsf{s}_{\delta(i)})$. We also know, from rule $\mathsf{f1}$, that $\mathcal{I} \models \mathsf{event}_\phi(\mathsf{s}_{\delta(i)}, \mathsf{s}_j)$ for some $0 \leq j \leq k$ and, from $\mathsf{f2}$, that $\mathcal{I} \models \Theta_\phi(\mathsf{s}_j)$. Using this information we will first prove that J_ϕ is not empty.

If $j \geq \delta(i)$, then it is always possible to find a $j' > i$ such that $j = \delta(j')$ and, for any such j', since $\mathcal{I} \models \Theta_\phi(\mathsf{s}_{\delta(j')})$, we have that $j' \in J_\phi$.

Suppose that otherwise $j < \delta(i)$, then $\mathcal{I} \models \mathsf{less}(\mathsf{s}_j, \mathsf{s}_{\delta(i)})$ and, by rule f3, it follows $\mathcal{I} \models \mathsf{hasloop}$. So we must have an infinite path as defined in item 1, in particular $\mathcal{I} \models \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$ for some l. To avoid a contradiction from rule f4, it must be the case that $\mathcal{I} \not\models \mathsf{less}(\mathsf{s}_j, \mathsf{s}_l)$ and, therefore, $l \geq j$. Again, since the point j is after the loop, it is always possible to find a $j' > i$ such that $j = \delta(j')$ and, similarly, such $j' \in J_\phi$.

Now, since J_ϕ is not empty, let j^* be any element in it. Then we know that $\mathcal{I} \models \Theta_\phi(\mathsf{s}_{\delta(j^*)})$ and, by induction, $\pi \models_{j^*} \phi$. Moreover, $i \leq j^*$ and, therefore, $\pi \models_i \mathbf{F}\phi$.

- If $\gamma = \psi \mathbf{W} \phi$, then $\mathcal{I} \models \mathsf{weak}_{\psi,\phi}(\mathsf{s}_{\delta(i)})$. We consider now the set J_ϕ. If it is not empty then let $j^* = \min J_\phi$, otherwise let $j^* = \omega$. Note that, by definition of J_ϕ, for every j in the interval

$$i \leq j < \min(j^*, |\pi|) , \tag{5.1}$$

we have $\mathcal{I} \not\models \Theta_\phi(\mathsf{s}_{\delta(j)})$ and, from rule r1, we obtain that

$$\mathcal{I} \models \mathsf{weak}_{\psi,\phi}(\mathsf{s}_{\delta(j)}) \rightarrow \mathsf{xweak}_{\psi,\phi}(\mathsf{s}_{\delta(j)}) . \tag{5.2}$$

Now it is easy to prove, for every j in the interval (5.1) and by a simple induction over $m = j - i$, that

$$\mathcal{I} \models \mathsf{weak}_{\psi,\phi}(\mathsf{s}_{\delta(j)}) \wedge \mathsf{xweak}_{\psi,\phi}(\mathsf{s}_{\delta(j)}) . \tag{5.3}$$

The base case when $i = j$ follows, since $\delta(j) = \delta(i)$, by the original hypothesis and from (5.2). Now for $i < j$ we know, by use of the inductive hypothesis, that $\mathcal{I} \models \mathsf{xweak}_{\psi,\phi}(\mathsf{s}_{\delta(j-1)})$. From rule r2 and also because $\mathcal{I} \models \mathsf{trans}(\mathsf{s}_{\delta(j-1)}, \mathsf{s}_{\delta(j)})$ it follows that $\mathcal{I} \models \mathsf{weak}_{\psi,\phi}(\mathsf{s}_{\delta(j)})$ and, using equation (5.2) again, $\mathcal{I} \models \mathsf{xweak}_{\psi,\phi}(\mathsf{s}_{\delta(j)})$.

Now, if $\mathcal{I} \models \mathsf{weak}_{\psi,\phi}(\mathsf{s}_{\delta(j)})$, from rule r3, $\mathcal{I} \models \Theta_\psi(\mathsf{s}_{\delta(j)})$ and, by structural induction, $\pi \models_j \psi$. In particular we obtain,

$$\pi \models_j \psi \qquad \text{for every } i \leq j < \min(j^*, |\pi|) . \tag{5.4}$$

So, if J_ϕ is empty we conclude from (5.3) that $\mathcal{I} \models \mathsf{xrelease}_{\psi,\phi}(\mathsf{s}_k)$ and, from rule $\mathsf{r4}$, we obtain that the path must be infinite. It then follows that $\min(j^*, |\pi|) = \omega$ and, from (5.4), $\pi \models_j \psi$ for every $i \leq j$. In particular $\pi \models_i \psi \mathbf{W} \phi$.

If, otherwise, the set J_ϕ is not empty, then it must be that $j^* \in J_\phi$. By construction $\mathcal{I} \models \Theta_\phi(\mathsf{s}_{\delta(j^*)})$ and, by structural induction, $\pi \models_{j^*} \phi$. Moreover, $j^* < |\pi|$ and $\min(j^*, |\pi|) = j^*$. Again from equation (5.4) we have $\pi \models_j \psi$ for every $i \leq j < j^*$ so we finally conclude that $\pi \models_i \psi \mathbf{W} \phi$. □

The previous proposition shows that, under the given assumptions, if an interpretation $\mathcal{I} \models \Theta_\phi(\mathsf{s}_{\delta(i)})$ then there is a path π, determined by $\mathcal{I}$, such that $\pi \models_i \phi$. Note, however, that the converse is not always true, e.g. $\mathcal{I} \not\models \Theta_\phi(\mathsf{s}_{\delta(i)})$ does not necessarily imply $\pi \not\models_i \phi$ for the possible induced paths.

Additional constraints can be added to the set $[\phi, k]$ in order to make the converse hold but, since we are mostly interested in satisfiability of the LTL formulae, this is not required for the correctness of our main result. Whether the addition of such constraints would be helpful for the solvers to find solutions more quickly, is an interesting question left open for further research.

What we do need to show is that, if there is a path that satisfies an LTL formula, then it is also possible to find an interpretation that satisfies its symbolic representation. The following definition shows how to build such interpretation and later, in Proposition 5.3, we prove that it serves the required purpose.

Definition 5.10. Let π be a k-path and δ its rolling function. We define an interpretation $\mathcal{I}^\pi$ with domain $D = \{\mathsf{s}_0, \ldots, \mathsf{s}_k\}$, for every $\mathsf{s}_i, \mathsf{s}_j \in D$ and pair of LTL formulae ψ, ϕ, as follows:

$\mathcal{I}^\pi \models \mathsf{p}(\mathsf{s}_i)$	iff	$p \in s_i$, for $p \in \mathcal{V}$.
$\mathcal{I}^\pi \models \mathsf{less}(\mathsf{s}_i, \mathsf{s}_j)$	iff	$i < j$.
$\mathcal{I}^\pi \models \mathsf{succ}(\mathsf{s}_i, \mathsf{s}_j)$	iff	$i + 1 = j$.
$\mathcal{I}^\pi \models \mathsf{trans}(\mathsf{s}_i, \mathsf{s}_j)$	iff	$\delta(i + 1) = j$.
$\mathcal{I}^\pi \models \mathsf{hasloop}$	iff	π is an infinite path.
$\mathcal{I}^\pi \models \mathsf{next}_\phi(\mathsf{s}_i)$	iff	$\pi \models_i \mathbf{X}\phi$.
$\mathcal{I}^\pi \models \mathsf{evently}_\phi(\mathsf{s}_i)$	iff	$\pi \models_i \mathbf{F}\phi$.
$\mathcal{I}^\pi \models \mathsf{event}_\phi(\mathsf{s}_i, \mathsf{s}_j)$	iff	$\pi \models_j \phi$ and there is a $j' \geq i$ with $\delta(j') = j$.
$\mathcal{I}^\pi \models \mathsf{weak}_{\psi,\phi}(\mathsf{s}_i)$	iff	$\pi \models_i \psi \mathbf{W} \phi$.
$\mathcal{I}^\pi \models \mathsf{xweak}_{\psi,\phi}(\mathsf{s}_i)$	iff	$\pi \models_i \psi \mathbf{W} \phi \wedge \neg\phi$.

■

Proposition 5.3. *Let π be a k-path in a Kripke structure M, and δ its rolling function. Also let γ be an arbitrary LTL formula, and let $i < |\pi|$.*

1. $\mathcal{I}^\pi \models \Theta_\gamma(\mathsf{s}_{\delta(i)})$ *iff* $\pi \models_i \gamma$,
2. $\mathcal{I}^\pi \models [M, k] \cup [\gamma, k]$.

Proof.

1. For most cases of the formula γ (propositional and temporal formulae) it immediately follows by definition that $\mathcal{I}^\pi \models \Theta_\gamma(\mathsf{s}_{\delta(i)})$ iff $\pi \models_{\delta(i)} \gamma$. Also recall that, by Lemma 5.2, $\pi \models_{\delta(i)} \gamma$ iff $\pi \models_i \gamma$. If γ is a conjunction or disjunction the argument follows by a straightforward induction.

2. Most constraints in $[M, k]$ can trivially be shown to be satisfied simply by definition of $\mathcal{I}^\pi$. The only interesting cases relate to the transition relation and initial states.

   Suppose that $\mathcal{I}^\pi \models \mathsf{trans}(\mathsf{s}_i, \mathsf{s}_j)$ and, by definition, $\delta(i+1) = j$. In particular the pair (s_i, s_j) must belong to the transition relation and, by Lemma 5.1, it follows that $\mathcal{I}^\pi \models T(\mathsf{s}_i, \mathsf{s}_j)$. In general $\mathcal{I}^\pi \models \mathsf{trans}(x, y) \rightarrow T(x, y)$. A similar argument, since the state s_0 is an initial state, shows that $\mathcal{I}^\pi \models I(\mathsf{s}_0)$.

   Now for the set $[\gamma, k]$ it is enough to check that $\mathcal{I}^\pi$ satisfies arbitrary ground instances of the constraints in $\Phi^k_{\mathbf{X}\phi}$, $\Phi^k_{\mathbf{F}\phi}$, and $\Phi^k_{\psi\mathbf{R}\phi}$. We will consider instances obtained mapping $x \rightarrow \mathsf{s}_i$, $y \rightarrow \mathsf{s}_j$ and $z \rightarrow \mathsf{s}_l$.

   x1 If $\mathcal{I}^\pi \models \mathsf{next}_\phi(\mathsf{s}_i)$ then $\pi \models_i \mathbf{X}\phi$, by definition, $\pi \models_{i+1} \phi$ and, by Lemma 5.2, $\pi \models_{\delta(i+1)} \phi$. Now, from $\mathcal{I}^\pi \models \mathsf{trans}(\mathsf{s}_i, \mathsf{s}_j)$, we obtain that $\delta(i+1) = j$, so that $\pi \models_j \phi$ and, by previous item 1, $\mathcal{I}^\pi \models \Theta_\phi(\mathsf{s}_j)$.

   x2 Similarly, $\mathcal{I}^\pi \models \mathsf{next}_\phi(\mathsf{s}_k)$ implies $\pi \models_k \mathbf{X}\phi$ and, by definition of the LTL semantics, $k+1 < |\pi|$. But this is only possible when the path is infinite and, therefore, $\mathcal{I}^\pi \models \mathsf{hasloop}$.

   f1 If $\mathcal{I}^\pi \models \mathsf{evently}_\phi(\mathsf{s}_i)$ then $\pi \models_i \mathbf{F}\phi$ and, by definition, there must be a value $j \geq i$ on which $\pi \models_j \phi$. From Lemma 5.2 also $\pi \models_{\delta(j)} \phi$ and, by definition, it also follows that $\mathcal{I}^\pi \models \mathsf{event}_\phi(\mathsf{s}_i, \mathsf{s}_{\delta(j)})$. In particular $\mathcal{I}^\pi \models \mathsf{event}_\phi(\mathsf{s}_i, \mathsf{s}_0) \vee \cdots \vee \mathsf{event}_\phi(\mathsf{s}_i, \mathsf{s}_k)$.

   f2 If $\mathcal{I}^\pi \models \mathsf{event}_\phi(\mathsf{s}_i, \mathsf{s}_j)$ then, by definition, $\pi \models_j \phi$ and, by the previous item 1, $\mathcal{I}^\pi \models \Theta_\phi(\mathsf{s}_j)$. (Recall $j \leq k$, so $\delta(j) = j$.)

f3 If $\mathcal{I}^\pi \models \mathsf{less}(\mathsf{s}_j, \mathsf{s}_i)$ then $j < i$. If additionally $\mathcal{I}^\pi \models \mathsf{event}_\phi(\mathsf{s}_i, \mathsf{s}_j)$ then, by definition, there must be a $j' \geq i$ with $\delta(j') = j$, so that $j < i \leq j'$. In particular $j = \delta(j') \neq j'$, which is only the case when the path is infinite, $j \geq l$ and $\mathcal{I}^\pi \models \mathsf{hasloop}$.

f4 Again if $\mathcal{I}^\pi \models \mathsf{less}(\mathsf{s}_j, \mathsf{s}_i) \wedge \mathsf{event}_\phi(\mathsf{s}_i, \mathsf{s}_j) \wedge \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$ the path has a loop that must start exactly at point l. As in the previous argument, it also must be the case that $j \geq l$ and, in particular, $\mathcal{I}^\pi \not\models \mathsf{less}(\mathsf{s}_j, \mathsf{s}_l)$. Therefore $\mathcal{I}^\pi$ will never satisfy the body of rule $\mathsf{f4}$.

w1 Follows by the logical tautology $\psi \mathbf{W} \phi \rightarrow \phi \vee (\psi \mathbf{W} \phi \wedge \neg\phi)$.

w2 If $\mathcal{I}^\pi \models \mathsf{xweak}_{\psi,\phi}(\mathsf{s}_i)$ then $\pi \models_i \psi \mathbf{W} \phi \wedge \neg\phi$. From the LTL theorem $\psi \mathbf{W} \phi \wedge \neg\phi \rightarrow \mathbf{X}(\psi \mathbf{W} \phi)$ and Lemma 5.2, it follows $\pi \models_{\delta(i+1)} \psi \mathbf{W} \phi$. Again from $\mathcal{I}^\pi \models \mathsf{trans}(\mathsf{s}_i, \mathsf{s}_j)$ we get $\delta(i+1) = j$ and, finally, we obtain $\mathcal{I}^\pi \models \mathsf{weak}_{\psi,\phi}(\mathsf{s}_j)$.

w3 Follows by the LTL theorem $\psi \mathbf{W} \phi \wedge \neg\phi \rightarrow \psi$.

w4 From the same argument as in **w2** we get $\pi \models_k \mathbf{X}(\psi \mathbf{W} \phi)$, which is only possible when the path is infinite. □

With these results being put in place we are able now to show, in Theorem 5.2, how the problem of testing the satisfiability of an LTL formula in a Kripke structure can be translated into the problem of checking satisfiability of predicate formulae.

Definition 5.11. Let M be a Kripke structure, ϕ an LTL formula and $k \geq 0$. The *predicate encoding* of M and ϕ (with depth k), denoted by $[M, \phi, k]$, is defined as the set of constraints $[M, k] \cup [\phi, k] \cup \{\Theta_\phi(\mathsf{s}_0)\}$. ■

Theorem 5.2. *Let ϕ be an LTL formula, and M a Kripke structure.*

1. *ϕ is satisfiable in M iff $[M, \phi, k]$ is satisfiable for some $k \geq 0$.*

2. *ϕ is valid in M iff $[M, \mathsf{NNF}(\neg\phi), k]$ is unsatisfiable for every $k \geq 0$.*

Proof. We only need to prove item 1, since 2 is just its dual.

Suppose first that ϕ is satisfiable in M. Then, by Theorem 5.1, there is a k-path π in M such that $\pi \models \phi$. Then, by following Proposition 5.3, we also know that $\mathcal{I}^\pi \models [M, k] \cup [\phi, k] \cup \{\Theta_\phi(\mathsf{s}_0)\}$ and, therefore, $[M, \phi, k]$ is satisfiable.

On the other hand, suppose that $[M, \phi, k]$ is satisfiable and let $\mathcal{I}$ be one of its models. If $\mathcal{I} \models \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$ for some $0 \leq l \leq k$ then, by Proposition 5.2, we know $\pi^{l,\mathcal{I}} \models \phi$ and, by item 3 in Proposition 5.1, $\pi^{l,\mathcal{I}}$ is a (proper) path in M.

Similarly, if $\mathcal{I} \not\models \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$ for every $0 \leq l \leq k$ then, by Proposition 5.2, we know $\pi^{\mathcal{I}} \models \phi$ and, by item 2 in Proposition 5.1, $\pi^{\mathcal{I}}$ is a path in M. Now $\pi^{\mathcal{I}}$ is a prefix path in M, but it can be arbitrarily extended to a proper path π' in M. For such proper path π' we would also have $\pi' \models \phi$. In either case the temporal formula ϕ is satisfiable in the system M. □

Implicit bound encoding As seen in Definition 5.8, the encoding just presented makes explicit use of the bound k in order to build the symbolic representation of an LTL formula. Notice that, in particular, a constraint of size $O(k)$ is created for every subformula of the form $\mathbf{F}\phi$ of the property to be checked. In this section we present an alternate encoding, which only uses the bound in an implicit way.

Definition 5.12. Given pair of LTL formulae ψ, ϕ, we define the following sets of constraints:

$\Phi'_{\mathbf{F}\phi}$:

- f1': $\mathsf{evently}_\phi(x) \to \Theta_\phi(x) \vee \mathsf{xevently}_\phi(x)$
- f2': $\mathsf{xevently}_\phi(x) \wedge \mathsf{succ}(x, y) \to \mathsf{evently}_\phi(y)$
- f3': $\mathsf{xevently}_\phi(x) \wedge \mathsf{last}(x) \to \mathsf{hasloop}$
- f4': $\mathsf{xevently}_\phi(x) \wedge \mathsf{last}(x) \wedge \mathsf{trans}(x, y) \to \mathsf{evently2}_\phi(y)$
- f5': $\mathsf{evently2}_\phi(x) \to \Theta_\phi(x) \vee \mathsf{xevently2}_\phi(x)$
- f6': $\mathsf{xevently2}_\phi(x) \wedge \mathsf{succ}(x, y) \to \mathsf{evently2}_\phi(y)$
- f7': $\mathsf{xevently2}_\phi(x) \wedge \mathsf{last}(x) \to \bot$

The sets $\Phi'_{\mathbf{X}\phi}$ and $\Phi'_{\psi\mathbf{W}\phi}$ are identical to $\Phi^k_{\mathbf{X}\phi}$ and $\Phi^k_{\psi\mathbf{W}\phi}$, except for the following constraints which replace x2 and w4 respectively.

$\Phi'_{\mathbf{X}\phi}$: x2': $\mathsf{next}_\phi(x) \wedge \mathsf{last}(x) \to \mathsf{hasloop}$

$\Phi'_{\psi\mathbf{W}\phi}$: w4': $\mathsf{xweak}_{\psi,\phi}(x) \wedge \mathsf{last}(x) \to \mathsf{hasloop}$

We finally introduce the set of *implicit structural definitions* of an LTL formula γ, denoted simply by $[\gamma]$, as the union of the sets Φ'_ϕ for every temporal subformula ϕ of the original γ. ■

Note that the newly defined sets Φ'_ϕ, do not explicitly use the value of the bound k anymore. We replaced the explicit references to s_k with a predicate $\mathsf{last}(x)$ which should be made true for the constant symbol representing the last state. Moreover, since the size of $\Phi'_{\mathbf{F}\phi}$ is constant, the size of the encoding $[\gamma]$ is now linear with respect to the size of γ.

The k-paths that satisfy an LTL formula ϕ can therefore now be captured with the set of constraints $[k] \cup \{\mathsf{last}(\mathsf{s}_k)\} \cup [\phi] \cup \{\Theta_\phi(\mathsf{s}_0)\}$. This representation is convenient since it breaks the encoding in two independent parts, one depending on the bound only and the other on the LTL formula only. Moreover it has a size of $O(n + k)$ where n is the size of the original temporal formula.

A complete instance of the bounded model checking problem would be then represented, analogous to Definition 5.11, as

$$[M, \phi, k]^* = [M] \cup [k] \cup \{\mathsf{last}(\mathsf{s}_k)\} \cup [\phi] \cup \{\Theta_\phi(\mathsf{s}_0)\}$$

and, for such set of constraints, the statement of Theorem 5.2 also holds.

This encoding is particularly useful when searching for counterexamples in an incremental setting, since both the system description and the temporal formula have to be encoded only once. Just the small set $[k] \cup \{\mathsf{last}(\mathsf{s}_k)\}$ needs to be updated while testing for increasing bounds. If using a model finder that supports incremental solving features, then one only needs to add $\mathsf{succ}(\mathsf{s}_k, \mathsf{s}_{k+1})$ and replace $\mathsf{last}(\mathsf{s}_k)$ with $\mathsf{last}(\mathsf{s}_{k+1})$.

Logarithmic encoding of states As seen in the previous section, the only part of the translation where there is an increase of size with respect to the input is in $[k]$, because of the series of facts of the form $\mathsf{succ}(\mathsf{s}_i, \mathsf{s}_{i+1})$ and the constraint

$$\mathsf{hasloop} \rightarrow \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_0) \vee \cdots \vee \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_k)\ . \tag{5.5}$$

This group of constraints, which is of size $O(k)$, can be more compactly encoded by representing the names of states in binary notation. For this we introduce a pair of constant symbols $\{\mathsf{b0}, \mathsf{b1}\}$ in order to write, for example when $k = 2^4$, the following definition for the succ predicate:

$$\begin{aligned}
&\mathsf{succ}(\,x_3, x_2, x_1, \mathsf{b0},\ x_3, x_2, x_1, \mathsf{b1}\,)\\
&\mathsf{succ}(\,x_3, x_2, \mathsf{b0}, \mathsf{b1},\ x_3, x_2, \mathsf{b1}, \mathsf{b0}\,)\\
&\mathsf{succ}(\,x_3, \mathsf{b0}, \mathsf{b1}, \mathsf{b1},\ x_3, \mathsf{b1}, \mathsf{b0}, \mathsf{b0}\,)\\
&\mathsf{succ}(\,\mathsf{b0}, \mathsf{b1}, \mathsf{b1}, \mathsf{b1},\ \mathsf{b1}, \mathsf{b0}, \mathsf{b0}, \mathsf{b0}\,)
\end{aligned}$$

In general we only need $w = \lceil \log k \rceil$ constraints, with a total size of $O(\log^2 k)$. On the other hand, the constraint (5.5) is rewritten as:

$$\begin{aligned}
&\mathsf{hasloop} \to \mathsf{loopafter}(\overline{\mathsf{b0}})\\
&\mathsf{loopafter}(\bar{x}) \wedge \mathsf{last}(\bar{y}) \to \mathsf{trans}(\bar{y}, \bar{x}) \vee \mathsf{xloopafter}(\bar{x})\\
&\mathsf{xloopafter}(\bar{x}) \wedge \mathsf{succ}(\bar{x}, \bar{y}) \to \mathsf{loopafter}(\bar{y})\\
&\mathsf{xloopafter}(\bar{x}) \wedge \mathsf{last}(\bar{x}) \to \bot
\end{aligned}$$

where $\overline{\mathsf{b0}}$ is a string of w symbols $\mathsf{b0}$, $\bar{x} = x_{w-1}, \ldots, x_0$ and similarly for $\bar{y}$.

One also has to replace everywhere else occurrences of s_0 with $\overline{\mathsf{b0}}$, the constant symbol s_k with its binary representation (e.g. for $k = 13$ use $\mathsf{b1}, \mathsf{b1}, \mathsf{b0}, \mathsf{b1}$), and variables such as x and y with the corresponding $\bar{x}$ or $\bar{y}$. The resulting set of constraints, which we denote by $[M, k, \phi]^b$, is of size $O(n \log k + \log^2 k)$, where n is the compound size of M and ϕ, and satisfies the statement of Theorem 5.2.

5.3 Encoding of the system description

Generating an instance of the bounded model checking problem requires three parameters as input: a system description M, a temporal formula ϕ and a bound k. In the previous section we showed how to encode an LTL formula as a predicate formula (w.r.t. the bound), but we generally assumed that the system (a Kripke structure M) was already symbolically described.

In this section we deal with how to encode a system, which is originally given in some industry standard format suitable to describe software and hardware components, in the form of a predicate formula. An advantage of using an effectively propositional, rather than just propositional, encoding is that important features for component development, such as the ability to describe systems in a modular and hierarchical way, can be directly represented in the target language. There is no need, for example, to perform a flattening phase to create and instantiate all modules of a system description before doing the actual encoding.

We will show now, by means of an example, how a system described in the SMV language can be succinctly and naturally encoded within the effectively propositional fragment. Although we would prefer to formally define the fragment of SMV considered here, the number of different SMV variants and the lack of documentation on the formal semantics in existing implementations made this task particularly difficult. Anyway, the explanation of the ideas presented in this section is always general enough so that it is possible to apply them to other arbitrary systems, not only the one in the example, and even implemented to be performed in an automated way.

For our running example we consider a distributed mutual exclusion (DME) circuit first described by Martin (1985) and then made available in the SMV format with the distribution of the NuSMV model checker (Cimatti et al., 2002). The system description is fragmented in a number of modules, each being a separate unit specifying how a section of the system works. The DME, for example, organises modules in a hierarchical way: the most basic modules are *gates* which perform simple logical operations, then a number of gate modules are replicated and assembled together to form the module of a *cell*, finally a number of cells are also replicated and linked together in the *main* module which represents the entire system.

Module variables A module usually defines a number of variables and describe how their values change in time. In the DME example, a typical gate module looks like:

```
module and−gate(in1,in2)
var
  out: boolean;
assign
  init(out) := 0;
  next(out) := (in1 & in2) union out;
```

This is a module named 'and−gate' which defines two boolean variables as input ('in1' and 'in2') and an output boolean variable ('out'). The initialisation part causes the output of all 'and−gate' instances to hold the value zero (i.e. false) when the system starts to execute. At each step the module nondeterministically chooses to compute the logical and of its inputs and update the output, or keep the output from the last clock cycle. The '**union**' operator in SMV effectively creates a set out of its two operands and nondeterministically chooses an element of the set as the result of the expression. So, this is the model of an asynchronous logical gate; fairness constraints (which are also encoded as LTL formulae) can be added to ensure, for example, that the gate eventually computes the required value.

In the symbolic description we represent each of these variables with a predicate symbol such as, in this particular example, $\mathsf{and\_gate\_in1}(v_1, v_2, x)$. The variable name is prefixed with the module name so that variables of different modules do not interfere with each other. Since, moreover, several instances of the 'and−gate' can be created, the first arguments v_1, v_2 serve to distinguish among such instances, the following section explains this in more detail. The last argument x represents a time step within the execution trace. Using this naming convention, it is possible to describe the module as follows:

$$\begin{array}{l}
\neg\mathsf{and\_gate\_out}(v_1, v_2, \mathsf{s}_0) \\
\mathsf{trans}(x, y) \rightarrow \\
\qquad (\mathsf{and\_gate\_out}(v_1, v_2, y) \leftrightarrow \mathsf{and\_gate\_in1}(v_1, v_2, x) \wedge \mathsf{and\_gate\_in2}(v_1, v_2, x)) \\
\quad \vee (\mathsf{and\_gate\_out}(v_1, v_2, y) \leftrightarrow \mathsf{and\_gate\_out}(v_1, v_2, x))
\end{array}$$

Note that, although the original SMV description distinguishes between inputs and outputs of the module, our proposed encoding does not need to.

Submodel instances Modules can also create named instances of other modules and specify how its own variables and the variables of the its submodule instances relate to each other. There is also one designated 'main' module, an instance of which represents the entire system to verify. One has to distinguish between the notions of a module (the abstract description of a component) and its possibly many module instances, which actually conform the complete system. In our running example, the DME circuit, part of the definition of a cell module looks like:

```
module cell(left, right, token)
var
  ack: boolean;
  c: and−gate(a.out, !left.ack);
  d: and−gate(b.out, !u.ack)
  ⋮
```

Here two submodule instances ‘c’ and ‘d’ are created, both instances of the ‘and−gate’ module. The elements ‘a, b: mutex_half’ and ‘u: user’ are instances of other modules also created within the cell, with definitions of other internal variables such as ‘out’ and ‘ack’. The elements ‘ left ’ and ‘right’ are references to other ‘ cell ’ instances, these are explained later in the following section.

Symbolically, the relations between the inputs and outputs of these modules are described using the constraints:

$$\mathsf{cell\_left}(v, w) \rightarrow \begin{array}{l} \mathsf{and\_gate\_in1}(v, \mathsf{c}, x) \leftrightarrow \mathsf{mutex\_half\_out}(v, \mathsf{a}, x) \\ \mathsf{and\_gate\_in2}(v, \mathsf{c}, x) \leftrightarrow \neg\mathsf{cell\_ack}(w, x) \\ \mathsf{and\_gate\_in1}(v, \mathsf{d}, x) \leftrightarrow \mathsf{mutex\_half\_out}(v, \mathsf{b}, x) \\ \mathsf{and\_gate\_in2}(v, \mathsf{d}, x) \leftrightarrow \neg\mathsf{user\_ack}(v, \mathsf{u}, x) \end{array} \tag{5.6}$$

Here the variable v stands for a particular cell instance, the second argument of the predicates is now filled in with the instance names of the different modules.

In general, if a module M_1 creates instances of a module M_2, we say that M_2 is a *submodule* of M_1. The submodule relation must then create a directed acyclic graph among the modules of a system; and the *submodule depth* of a module is the length of the longest path that reaches it from the designated ‘main’ module. The depth of the ‘main’ module, for example, is always 0; and the depth of a module is strictly less than the depth of its submodules.

In a module of depth d we will therefore use $d+1$ arguments in the predicates that represent the module’s boolean variables. The last argument always denotes time, and the interpretation of the other d arguments is the string of names that represent each created instance in a chain of submodules. Consider for example the ‘out’ variable of a module ‘some−gate’ which corresponds to an instance with the fully qualified name of ‘main.sub1.sub2.sub3.sub4.out’; symbolically we would represent such variable with the predicate

$$\mathsf{some\_gate\_out}(\mathsf{sub1}, \mathsf{sub2}, \mathsf{sub3}, \mathsf{sub4}, x)\,.$$

Finally note that instances of the same module can in principle be reached from the main module by paths of different lengths. Consider for example a module ‘m1’ that creates instances of ‘m2’ and ‘m3’, but ‘m2’ also creates instances of ‘m3’. In this example the module ‘m3’ is of depth 2 and not 1. So in general, if a module of depth d creates an instance named ‘sub’ of another module of depth d'; and the sequence of constant symbols $\mathsf{m}_1, \ldots, \mathsf{m}_d$ is used to identify an instance of the first module, then the sequence of d' constant symbols $\mathsf{m}_1, \ldots, \mathsf{m}_d, \ldots, \mathsf{o}, \ldots, \mathsf{sub}$ —where a number of dummy constant symbols 'o' (unused anywhere else) serve as padding to get the required length— is the identifier of the second.

Module references Another feature of the SMV language is that modules can get references to other modules as parameters (e.g. 'left' and 'right' in the cell example). This feature is encoded introducing a new predicate, c.f. $\mathsf{cell\_left}(v, w)$ in (5.6), that establishes this relation between the two modules. References are used in our running example to communicate three different cells 'e−1', 'e−2' and 'e−3':

```
module main
var
  e-3: process cell(e-1,e-2,1);
  ⋮
```

which is encoded as: $\{\mathsf{cell\_left}(\mathsf{e\_3}, \mathsf{e\_1}), \mathsf{cell\_right}(\mathsf{e\_3}, \mathsf{e\_2}), \mathsf{cell\_token}(\mathsf{e\_3}, x)\}$. In general, the reference from a 'module1' to another 'module2' is encoded as:

$$\mathsf{module1\_link}(\bar{v}, \bar{w}) \rightarrow \mathsf{module1\_var1}(\bar{v}, x) \leftrightarrow \mathsf{module2\_var2}(\bar{w}, x)$$

where $\bar{v}$ and $\bar{w}$ are sequences of variables of appropriate lengths according to the depths of each module, and 'link' is the local name which the first module uses to reference the second. Compare this with the relevant constraint in (5.6).

Enumerated types Finally, another common feature of component description languages is the use of enumerated types. Using standard encodings, such variables are represented with an additional argument to denote the value currently hold. Also, a number of constraints have to be added in order to ensure that one (and only one) value of an enumerated variable holds at a time.

For example, an enumerated variable 'colour: {red, green, blue}' would be encoded as:

$$\begin{array}{l}
\mathsf{m\_colour}(\bar{v}, x, \mathsf{red}) \vee \mathsf{m\_colour}(\bar{v}, x, \mathsf{green}) \vee \mathsf{m\_colour}(\bar{v}, x, \mathsf{blue}) \\
\neg\mathsf{m\_colour}(\bar{v}, x, \mathsf{red}) \vee \neg\mathsf{m\_colour}(\bar{v}, x, \mathsf{green}) \\
\neg\mathsf{m\_colour}(\bar{v}, x, \mathsf{red}) \vee \neg\mathsf{m\_colour}(\bar{v}, x, \mathsf{blue}) \\
\neg\mathsf{m\_colour}(\bar{v}, x, \mathsf{green}) \vee \neg\mathsf{m\_colour}(\bar{v}, x, \mathsf{blue})
\end{array}$$

Where $\bar{v}$ represents a number of variables according to the depth of the module, and the variable x stands for the time step during execution.

5.4 Evaluation of the approach

In order to evaluate the ideas presented here, we developed a tool SMV2TPTP that —taking as input a SMV description, an LTL formula, and a bound k— produces an effectively propositional formula in the TPTP format suitable for use with existing effectively propositional and first-order reasoners. The tool, as well as some of the generated benchmarks, have been made publicly available online.[1]

This translator is able to read a subset of the SMV specification language with support of features such as: boolean and enumerated types, modules and references to modules, limited support of process modules, and LTL specifications. Since this is just an early proof of concept, we did not implemented more complex features such as vectors or arithmetic which, moreover, would perhaps be better implemented using technology from satisfiability modulo theories (e.g. Nieuwenhuis and Oliveras, 2005; Barrett and Berezin, 2004).

We then proceeded to select a number of problems which our tool was able to read. Although the lack of support for arithmetic prevented us from using many of the benchmarks made available by the community, we still were able to find some interesting test cases among the examples distributed together with the NuSMV model checker (Cimatti et al., 2002). Table 5.1 lists the selected instances together with a brief description of the model they represent as well as the property to be checked. The first three systems correspond to very simple designs, while the three later ones are closer to real-world applications. Although NuSMV, in the standard BDD mode, is able to prove the validity of these properties within a few seconds; the benchmarks are interesting because they help to evaluate the scalability of the bounded model checking approach.

[1] `http://www.cs.man.ac.uk/~navarroj/tools/`

After translating the problems into the TPTP format, we ran the resulting effectively propositional formulae trough a number of different provers. We considered the Darwin (Fuchs, 2004) and iProver (Korovin, 2006) systems which implement two different calculi that lift existing propositional techniques to the first-order case and are particularly geared towards the effectively propositional fragment. To compare with alternative approaches we also included Paradox (Claessen and Sörensson, 2003) which implements an instantiation method that reduces the problem to propositional satisfiability, and Vampire (Riazanov and Voronkov, 2002) the leading resolution-based first-order theorem prover. Also, as a reference to the existing SAT-based approach, we ran NuSMV in BMC mode on the original SMV input files.

Recall that, since the properties to check are valid, all the formulae created are unsatisfiable. In this context, when a prover finishes claiming that a formula generated with a bound k is unsatisfiable, what we obtain is a proof that there are no counterexamples of length $\leq k$. We therefore ran, for each combination of a benchmark and a prover, a number of individual tests for increasing values of k. We stopped until the prover consistently took more than one hour to solve a particular instance, or when all instances up to $k = 100$ were solved.

Table 5.2 shows the results of a first analysis on the outcome of this experiment. The table shows either a number indicating the last bound that the prover was able to refute within 1 hour or, in parenthesis, the time it took for the prover to refute all bounds of a length of k less than or equal to 100. Notice that the *mutex* benchmark is reported twice since two symmetric properties, from the point of view of each of the two process being modelled, were tested for this system.

From these results some interesting conclusions can already be drawn. First, the performance of NuSMV is comparable to that of Darwin and iProver. Moreover, for the easy problems Darwin is even able to outperform NuSMV, while on the hard problems iProver is still competitive. Another observation is that, although both NuSMV and Paradox are based on instantiation methods, NuSMV performs much better. The better performance of NuSMV is most likely explained by the fact that their translation to SAT is able to make use of domain specific information which is not directly available to Paradox. It is promising and interesting to note however that, while also both Darwin and iProver do not have access to any domain specific information, they still compete with the BMC mode of NuSMV. Finally, the performance of Vampire is

counter

Model:	3 bit counter
Property:	Always eventually the counter will overflow.
File path:	examples/ctl-ltl/counter.smv

short

Model:	Simple access control
Property:	If one requests access to the resource, it will eventually enter the busy state.
File path:	examples/example_cmu/short.smv

mutex

Model:	Mutual exclusion algorithm
Property:	If a process asks for access to the critical region, it will eventually be granted. Two symmetric properties are tested, one for each of the two processes.
File path:	examples/example_cmu/mutex.smv

prodcell

Model:	Production cell control model
Property:	A liveness property that the system will not enter a deadlock state.
File path:	examples/production-cell/production-cell.smv

dme1

Model:	Distributed mutual exclusion algorithm
Property:	Two users will never have simultaneous access to the shared resource.
File path:	examples/example_cmu/dme1.smv

gigamax

Model:	Gigamax cache coherence protocol
Property:	Two processors will not write simultaneously to the cache.
File path:	examples/example_cmu/gigamax.smv

Table 5.1: NuSMV benchmarks selected for experimentation

	NuSMV	Darwin	iProver	Paradox	Vampire
counter	(3m 45s)	(12s)	(2m 05s)	17	10
short	(1m 22s)	(44s)	(1m 17s)	12	25
mutex1	(2m 38s)	(1m 02s)	100	16	17
mutex2	(2m 39s)	(1m 06s)	100	16	17
prodcell	60	80	43	22	3
dme1	71	9	37	9	5
gigamax	77	6	27	8	3

Table 5.2: Evaluation of provers on bounded model checking problems

very limited in this setup, adding to the experimental evidence that resolution is not always the best approach for solving effectively propositional formulae.

Another interesting thing to observe, is how the running time of the provers scales while the bound for the model checking problem is increased. For Darwin, for example, the running times in Figure 5.2 show a pattern with the difficulty of solving the 'mutex' problem increasing along two different and clearly defined trend lines. There seems to be a group of 'easy' problems when $k = 0, 3 \pmod 4$ and a group of 'difficult' problems when $k = 1, 3 \pmod 4$. This behaviour is explained by the structure of the underlying system whose paths are periodic with a period of length 4. Apparently, the prover finds the problem easier at some points of this cycle.

iProver, however, shows a completely different behaviour. Its running times shown in Figure 5.3 for the same problem, do not seem to be particularly affected by the underlying system being checked and, furthermore, they show a lot more of variance. Although most problems for different bounds are solved within 15 minutes, some few exceptional cases —placed at seemingly random points— take up to one hour of running time. This effect was also observed when solving other problems, Darwin often shows very well defined difficulty patterns, while iProver running times tend to look more erratic.

Another interesting case is that of Darwin when solving the 'prodcell' problem. The prover manages to solve the $k = 80$ problem within 4:18 minutes, but times out after one hour for $k = 81$. Interestingly, if one extrapolates the running times of the prover on previous bounds, one finds that Darwin should be able to solve even the $k = 100$ problem in just below 8 minutes. A closer look at Darwin's performance reveals that the solver timed out at $k = 81$ since, because of its memory requirements, the operating system was kept busy most of the time

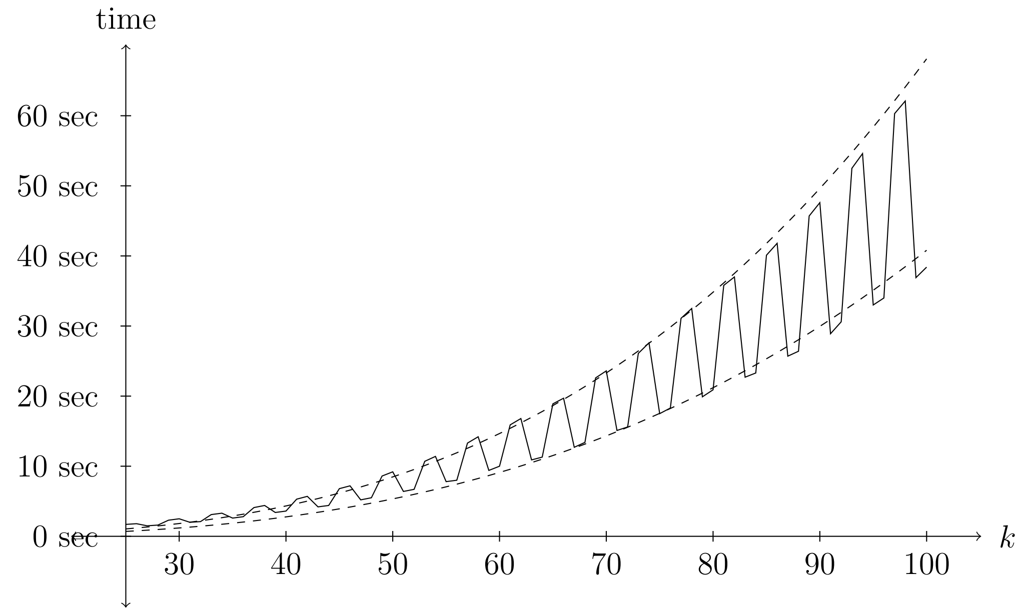


Figure 5.2: DARWIN running time solving first property of ‘mutex’

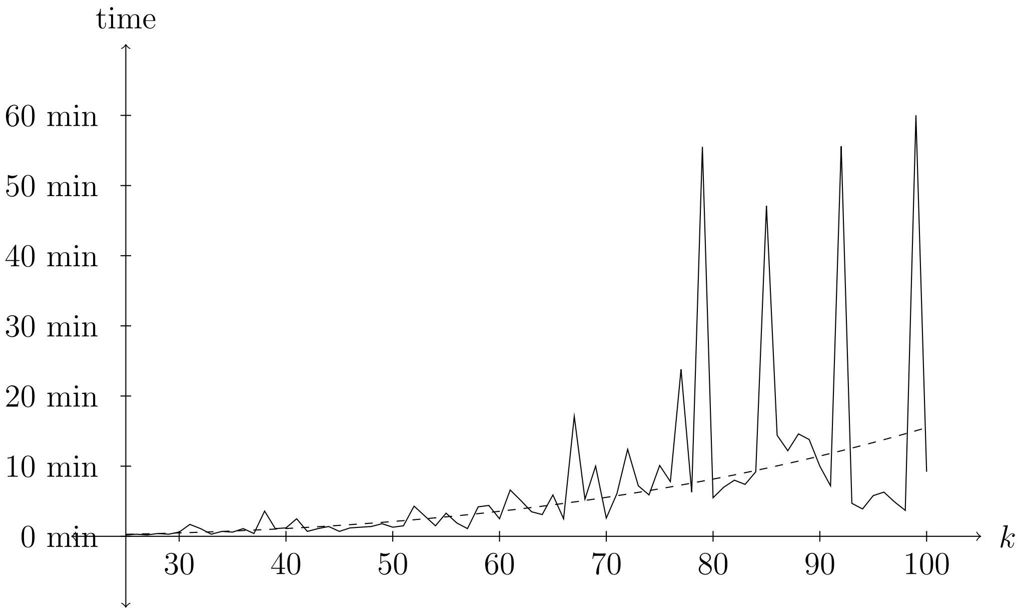


Figure 5.3: IPROVER running time solving first property of ‘mutex’

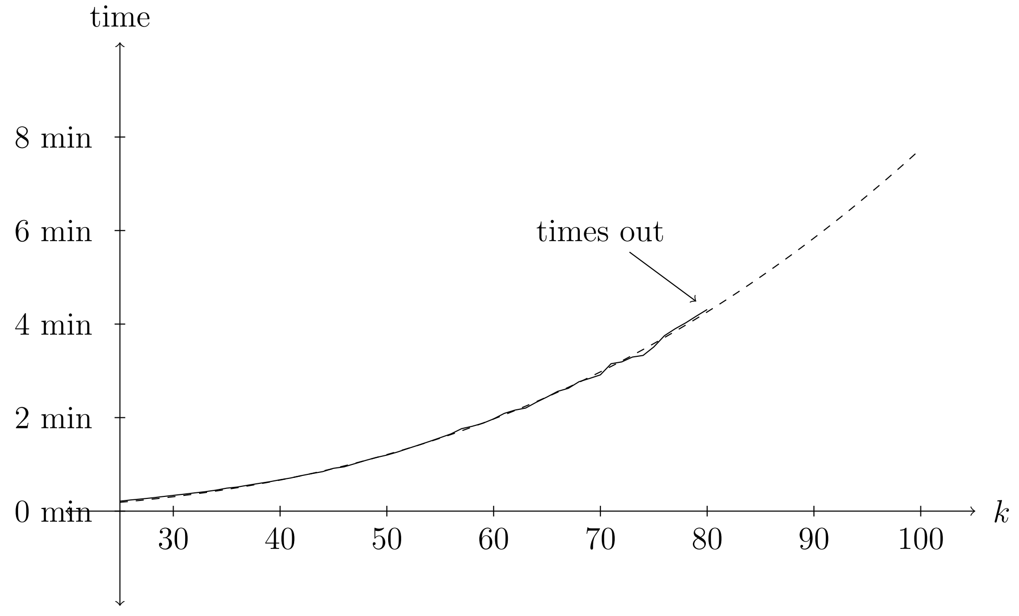


Figure 5.4: Darwin running time solving ‘prodcell’

doing memory swaps. A similar issue prevented Darwin of having a reasonable performance on the other two industrial size problems ‘dme1’ and ‘gigamax’, but the trend is much more evident in ‘prodcell’.

Finally, many trend lines were also computed for the best effectively propositional prover on each of these problems, giving us an indication on the scalability of the approach. All problems had a best fit when approximated by a polynomial equation of the form $t = e^b\, k^a$ where t represents the expected running time in minutes, and the coefficients a and b are given in Table 5.3 for different problems. The running time of most problems seems to be of order x^3, except for the last two which are closer to x^4. Also note that, in the case of the ‘mutex’ problems two trend lines were found for each of the two properties. The ‘counter’ problem also has a clearly distinguished *easier* trend line (not shown) when solved with Darwin for problems with $k = 0 \pmod 8$; recall that the system being modelled is a 3-bit counter, so it also loops at every 8 states.

5.5 Concluding remarks

In this chapter we discussed several strategies to encode instances of the bounded model checking problem as a predicate formula in the Bernays-Schönfinkel class. We showed a translation which, given a linear temporal logical formula and a

	prover	a	b
counter	Darwin	2.761954	−14.2671
short	Darwin	2.963619	−13.9819
mutex1a	Darwin	2.935007	−13.9017
mutex1b	Darwin	3.006557	−13.7189
mutex2a	Darwin	2.969525	−14.0579
mutex2b	Darwin	2.991756	−13.6480
prodcell	Darwin	2.679634	−10.2932
dme1	iProver	3.981209	−10.3161
gigamax	iProver	3.750581	−7.97828

Table 5.3: Trend line coefficients for scalability patterns

bound k, produces a set of constraints whose models represent all the possible paths (of bounded length k) which satisfy the given property. We also discussed how to further improve this translation and generate an output of size $O(n + k)$ where n is the size of the input LTL formula. The translation is also further improved by using a binary representation to denote the states.

We then proceeded to show how to efficiently describe transition systems as effectively propositional formulae, and demonstrated how many features commonly found in software and hardware description languages are succinctly and naturally encoded within our target language. Most significantly, modular and hierarchical system descriptions are directly encoded without a significant increase in the size; unlike propositional encodings where a preliminary, and potentially exponential, flattening phase needs to be applied to the system description.

Finally, a tool to translate a subset of the SMV language automatically into effectively propositional formulae was developed and made available online for the research community to use. Some experimental results were also reported, indicating that although the capabilities of current technology are not mature enough for industrial applications, the approach seems to be a promising solution for the scalability problems of plain propositional translations. Directions for future work include the extension to more general forms of temporal logics (such as μTL), the inclusion of more features to describe systems (such as arrays and arithmetic) and the application of similar encoding techniques to other suitable application domains.

Chapter 6

Encoding planning problems

Planning has been the focus of attention of many researchers in the field of artificial intelligence, where it was originally conceived as a formalisation of deduction processes (Green, 1969). Alternatively, the problem of finding a sequence of actions to reach, from an initial state, a set of desired goals, has also been reduced to the problem of finding a satisfying truth assignment for a propositional logical formula (Kautz and Selman, 1992; Kautz et al., 1996).

In this chapter we follow a similar approach but, instead of a propositional encoding, we make use of the finite domain predicate logic introduced near the end of Chapter 4, allowing a much more succinct and natural representation of problems. The size of the resulting formula is linear in the size of, for example, a STRIPS description (Fikes and Nilsson, 1971) of the original planning problem. And, moreover, the formula can also be linearly translated to plain effectively propositional logic without any further adornments.

This enables the use of reasoning mechanisms that work at a level of abstraction higher than propositional logic. On the other hand, our encoding may also turn out to be useful for propositional, SAT-based, approaches to planning. Indeed, it preserves the structure of the original planning problem in the obtained effectively propositional formula and reduces the problem of finding an optimised propositional encoding to the problem of finding an optimised propositional instantiation of the EPR description. Thinking in this more general fragment of first-order logic, often allows one to find simplifications or alternative encodings that one might miss if only focused in the propositional case.

Reasoning with effectively propositional theories is a relatively new area of research, which seems to offer a language with a good compromise between expressibility and complexity. There are many computer scientists currently developing ideas and procedures in order to more efficiently deal with this kind of formulae. Unfortunately, there is also a lack of benchmarks for these researchers to experiment and test their systems. An important contribution of this chapter is to aid filling in this gap with two alternative planning encodings which are a new and rich source of problems with close links to real-world applications. Finally, we also include some empirical evaluation on the performance of existing systems while solving some of these generated problems.

6.1 Introduction to planning

In this section we formally introduce several notions and concepts related to planning. We first introduce the notion of a planning domain where applicable actions, their preconditions and consequences, are described. We then proceed to define what a planning problem and its solutions are. This formalism corresponds to STRIPS-style planning as introduced by Fikes and Nilsson (1971).

Note that, unlike elsewhere in this thesis, in this chapter we will represent variables with uppercase letters. This is because now many variables will be needed, and using uppercase it will become easier to distinguish variables from constant symbols which remain lowercase.

Definition 6.1. The language of a planning domain consists of a triple of finite sets of symbols $(\mathcal{O}, \mathcal{F}, \mathcal{A})$ which are respectively called *object*, *fluent* and *action symbols*. Fluent and action symbols have, moreover, an associated natural number which we call the *arity* of the symbol. If f is a fluent symbol of arity m, then an expression of the form $f(t_1, \ldots, t_m)$, where each t_i is either a variable or an object symbol, is called a *fluent*.

An *action* is a triple $(\alpha_{\text{req}}, \alpha_{\text{add}}, \alpha_{\text{del}})$ where $\alpha = a(X_1, \ldots, X_n)$, for an action symbol $a \in \mathcal{A}$ of arity n, is the *signature* of the action. Each element in the triple is a set of fluents of the form $f(t_1, \ldots, t_m)$ where each t_i is either an object symbol or a variable X_j with $1 \leq j \leq n$. We say that these are the fluents that, respectively, the action *requires*, *adds* and *deletes* when it is executed. A *planning domain* $\mathcal{Dom}$ is simply a set of actions and its size, denoted $|\mathcal{Dom}|$, is defined as the number of symbols occurring in the description of all its actions.

An *action instance* $\alpha' = \alpha\sigma$, where σ is any substitution, corresponds to the triple of sets of fluent instances $({\alpha'}_{\text{req}}, {\alpha'}_{\text{add}}, {\alpha'}_{\text{del}})$, where ${\alpha'}_{\text{req}} = \alpha_{\text{req}}\sigma$, etc. ■

Example 6.1. We will consider as a running example for this section, a planning domain in the context of logistics. This domain has, among others, an action $\mathsf{load\text{-}truck}$ that takes three parameters: a package X_1 to load, a truck X_2 where to load the package, and a location X_3 where the loading takes place. The definition of such an action would probably look like:

$$\begin{array}{l}\mathsf{load\text{-}truck}(X_1, X_2, X_3)\\ \quad \text{Req: } \mathsf{at}(X_1, X_3), \mathsf{at}(X_2, X_3)\\ \quad \text{Del: } \mathsf{at}(X_1, X_3)\\ \quad \text{Add: } \mathsf{in}(X_1, X_2)\end{array}$$

where at and in are binary fluent symbols. In words, the $\mathsf{load\text{-}truck}$ action requires both the package, represented by the variable X_1, and the truck, represented by X_2, to be at the same location, represented by X_3. The action removes the package from the location and places it, instead, in the truck. A ground instance of this action, say $\mathsf{load\text{-}truck}(\mathsf{pk3}, \mathsf{w238}, \mathsf{man})$, would load the particular package $\mathsf{pk3}$ into the truck with license plate $\mathsf{w238}$ when both items are in Manchester (man).

The size of such definition is 16 (1 action symbol + 4 fluent symbols + 11 variable occurrences). We can imagine that the planning domain also contains other actions to unload the truck and drive it from one location to another; as well as more object symbols to identify different packages, trucks and locations.

Definition 6.2. Let α and β be two distinct ground actions. We say that α *interferes with* β, if the action α deletes fluents that are either required or added by β (i.e. $\alpha_{\text{del}} \cap (\beta_{\text{req}} \cup \beta_{\text{add}}) \neq \emptyset$). We say that a pair of ground actions is *interfering* if one of them interferes with the other. ■

Example 6.2. The ground action $\mathsf{load\text{-}truck}(\mathsf{pk3}, \mathsf{w238}, \mathsf{man})$ interferes with the other ground action $\mathsf{load\text{-}truck}(\mathsf{pk3}, \mathsf{y659}, \mathsf{man})$ since the former deletes the fluent $\mathsf{at}(\mathsf{pk3}, \mathsf{man})$ while the later requires it. Note that this is how, implicitly, the functionality of the fluent at is preserved, i.e. no object is allowed to end up at two different places simultaneously.

Definition 6.3. Given a set of ground fluents $\mathbf{S}$ and a set of ground actions $\mathbf{A}$, we say that $\mathbf{A}$ *is executable in* $\mathbf{S}$ *and produces* $\mathbf{S}'$, denoted by $\mathbf{S} \xrightarrow{\mathbf{A}} \mathbf{S}'$, if:

- $\mathbf{A}$ does not contain interfering actions,

- $\mathbf{A}_{\text{req}} \subseteq \mathbf{S}$,
- $\mathbf{S}' = \mathbf{S} \setminus \mathbf{A}_{\text{del}} \cup \mathbf{A}_{\text{add}}$.

where $\mathbf{A}_{\text{req}} = \bigcup_{\alpha \in \mathbf{A}} \alpha_{\text{req}}$, etc. ■

Definition 6.4. A *planning problem* is given as a pair $\mathbf{I}, \mathbf{G}$ of sets of ground fluents, respectively known as the *initial* and *goal states* of the problem.

A *solution plan* for the problem is a sequence $\mathbf{A}_1, \dots, \mathbf{A}_k$ of sets of ground actions such that the sequence $\mathbf{S}_1 \xrightarrow{\mathbf{A}_1} \mathbf{S}_2 \xrightarrow{\mathbf{A}_2} \cdots \xrightarrow{\mathbf{A}_k} \mathbf{S}_{k+1}$ holds, the set $\mathbf{I} = \mathbf{S}_1$ and $\mathbf{G} \subseteq \mathbf{S}_{k+1}$. ■

The kind of plans just defined are often known as *plans with parallel actions*. The semantics of such plans is that, at each step, it is possible to execute the actions in a set $\mathbf{A}_i$ in any order (even simultaneously) while still reaching the same outcome. In our example both load-truck(pk3, w238, man) and load-truck(pk4, w238, man) can be simultaneously executed in order to load both packages pk3 and pk4 into the truck w238. Alternatively, a *linear plan* is a plan where each $\mathbf{A}_i$ is a singleton. Trivially, any plan with parallel actions can be translated into a linear plan just by sequencing parallel actions into an arbitrary order, e.g. first load package pk3, then load pk4.

6.2 Encoding of planning problems

In this section we will consider an encoding of planning problems into finite domain predicate logic as introduced in Section 4.4. Given a planning domain and a bound k, we construct a set of constraints Γ_k whose models correspond to plans of length k. Linear plans of shorter lengths ($< k$) can also be encoded by allowing the use of a nop action that does nothing or, in plans with parallel actions, having steps where no action is executed (i.e. an empty $\mathbf{A}_i$).

Although fluents and actions were already defined as atoms in predicate logic, these predicate symbols will now play the role of constant symbols so that it becomes possible to quantify over them in our encoding. For example, if $f(\overline{Y})$ is a fluent in a planning domain then the predicate holds$(f, \overline{Y}, T)$ will be used to denote the fact that an instance of the fluent $f(\overline{Y})$ holds at a step T of the plan.

Note that this sort of encoding requires, however, all fluents (resp. actions) to have the same arity. It is easy to achieve this by padding actions with additional variables (which will be unused in its fluents), and padding fluents in actions with some dummy constant symbol $o \in \mathcal{D}$.

We will split the encoding of a planning domain into four groups of clauses. The first group Bound_k specifies the length of the plans to be considered, the second group $\mathsf{Act}_{\mathcal{D}om}$ encodes the definitions of actions, the third $\mathsf{Prob}_{\mathbf{I},\mathbf{G}}$ encodes the initial and goal states of a particular problem instance, and the fourth and last one $\mathsf{Frame}_{\mathcal{D}om}$ encodes the frame conditions. Frame conditions are the ones responsible to state that all fluents, whose status is not changed by any of the executed actions, must remain unmodified. We will, actually, show two different encodings for frame conditions that produce plans which are either linear or with parallel actions. The following definition formally introduces the particular case of finite domain predicate logic used to encode this formulae.

Definition 6.5. We consider a finite domain predicate logic having four different sorts of constant symbols: $\mathcal{O}$, $\mathcal{F}$, $\mathcal{A}$ which are directly inherited from the planning domain, and the sort $\mathcal{T} = \{s_1, \ldots, s_k\}$ which contains several constant symbols to denote time steps in a plan.

We will also use the variables X, Y —possibly subscripted— that range over constant symbols of the sort of objects $\mathcal{O}$; as well as F as a variable of sort $\mathcal{F}$; the pair A, B of variables of sort $\mathcal{A}$; and T, U as variables of sort $\mathcal{T}$ that usually represent, respectively, the *current* and *next step* in a plan. We will also use, for example, $\overline{X}$ and $\bar{c}$ to respectively denote sequences, whose length will be clear from context, of variables or constant symbols. We moreover assume that there are two fixed positive numbers n and m which, respectively, denote the arity of actions and fluents.

The logic also has six predicate symbols. A binary predicate next of sort $\mathcal{T} \times \mathcal{T}$ to represent the sequence of steps in a plan; three predicates reqs, adds and dels all of the sort $\mathcal{A} \times \mathcal{O}^n \times \mathcal{F} \times \mathcal{O}^m$ to represent fluents that are required, added or deleted by each action; a predicate holds of sort $\mathcal{F} \times \mathcal{O}^m \times \mathcal{T}$ and a predicate $\mathsf{executes}$ of sort $\mathcal{A} \times \mathcal{O}^n \times \mathcal{T}$ that respectively represent the fluents that hold and actions that execute at each step in the plan. ■

Using this logic we are now able to define the several components which conform our planning encoding.

Definition 6.6. Given a positive number k, the set Bound_k is simply defined as the set containing $\mathsf{next}(s_i, s_{i+1})$ for every $i \leq 0 < k$. ■

This simple set with a size of $O(k)$ is used to define an order among steps in the plan and to determine, from each step, which is the next one. The following set encodes the actions available in the domain.

Definition 6.7. Given a planning domain $\mathcal{Dom}$, the *domain definition* $\mathsf{Act}_{\mathcal{Dom}}$ is the set that contains, for each action in the domain, the constraints:

$$\begin{array}{ll} \mathsf{reqs}(a, \overline{X}, f, \overline{Y}\sigma) & \text{for each } f(\overline{Y})\sigma \text{ required by } a(\overline{X}) \\ \mathsf{dels}(a, \overline{X}, f, \overline{Y}\sigma) & \text{for each } f(\overline{Y})\sigma \text{ deleted by } a(\overline{X}) \\ \mathsf{adds}(a, \overline{X}, f, \overline{Y}\sigma) & \text{for each } f(\overline{Y})\sigma \text{ added by } a(\overline{X}) \end{array}$$

together with the following three constraints:

$$\begin{array}{l} \mathsf{reqs}(A, \overline{X}, F, \overline{Y}) \land \neg\mathsf{holds}(F, \overline{Y}, T) \rightarrow \neg\mathsf{executes}(A, \overline{X}, T) \\ \mathsf{next}(T, U) \land \mathsf{adds}(A, \overline{X}, F, \overline{Y}) \land \mathsf{executes}(A, \overline{X}, T) \rightarrow \mathsf{holds}(F, \overline{Y}, U) \\ \mathsf{next}(T, U) \land \mathsf{dels}(A, \overline{X}, F, \overline{Y}) \land \mathsf{executes}(A, \overline{X}, T) \rightarrow \neg\mathsf{holds}(F, \overline{Y}, U) \end{array}$$

that make actions have their corresponding preconditions and effects. ■

Example 6.3. In our running example, the action load-truck which was defined in Example 6.1 would be encoded as:

$$\begin{array}{l} \mathsf{reqs}(\mathsf{load\text{-}truck}, X_1, X_2, X_3, \mathsf{at}, X_1, X_3) \\ \mathsf{reqs}(\mathsf{load\text{-}truck}, X_1, X_2, X_3, \mathsf{at}, X_2, X_3) \\ \mathsf{dels}(\mathsf{load\text{-}truck}, X_1, X_2, X_3, \mathsf{at}, X_1, X_3) \\ \mathsf{adds}(\mathsf{load\text{-}truck}, X_1, X_2, X_3, \mathsf{in}, X_1, X_2) \end{array}$$

Similar constraints are added for other actions in the domain. The last few constraints of $\mathsf{Act}_{\mathcal{Dom}}$ would ensure that an action is not executed when one of its requirements does not hold or, if the action is executed, that fluents are added or deleted accordingly. It is also easy to see that $\mathsf{Act}_{\mathcal{Dom}}$ has a size of $O(|\mathcal{Dom}|)$.

We now move to the encoding of a problem instance using the set of constraints $\mathsf{Prob}_{\mathbf{I},\mathbf{G}}$. Typically, in propositional encodings of a planning problem, one has to completely specify the initial state $\mathbf{I}$ stating, for every ground fluent, whether $f(\bar{c})$ or $\neg f(\bar{c})$ should hold. To avoid this, we define a special action setup that adds all the ground fluents to be true at the initial state and does not require or delete anything. Quantifying over all fluents it is easy to express that “initially nothing holds” and then make the setup action execute at the step zero of the plan, the frame conditions will then ensure that everything not added by setup remains false.

Definition 6.8. Given a planning problem defined by an initial state $\mathbf{I}$ and goals $\mathbf{G}$, the encoding of the problem instance $\mathsf{Prob}_{\mathbf{I},\mathbf{G}}$ is defined as the set of constraints:

$$\begin{array}{ll}
\neg\mathsf{holds}(F, \overline{Y}, s_0) & \\
\mathsf{adds}(\mathsf{setup}, \overline{X}, f, \bar{c}) & \text{for every } f(\bar{c}) \text{ in } \mathbf{I} \\
\mathsf{executes}(\mathsf{setup}, \bar{o}, s_0) & \\
\mathsf{next}(T, U) \rightarrow \neg\mathsf{executes}(\mathsf{setup}, \overline{X}, U) & \\
\mathsf{holds}(f, \bar{c}, s_k) & \text{for every } f(\bar{c}) \text{ in } \mathbf{G}
\end{array}$$

where $\bar{o}$ simply represents the sequence $o, \ldots, o$ of dummy constant symbols of the required length. ■

Example 6.4. Suppose that initially we have two packages in Manchester, a truck in London, and our goal is to get the packages to Edinburgh. This corresponds to an initial state $\mathbf{I} = \{\mathsf{at}(\mathsf{pk3}, \mathsf{man}), \mathsf{at}(\mathsf{pk4}, \mathsf{man}), \mathsf{at}(\mathsf{w238}, \mathsf{lon})\}$, a set of goals $\mathbf{G} = \{\mathsf{at}(\mathsf{pk3}, \mathsf{edn}), \mathsf{at}(\mathsf{pk4}, \mathsf{edn})\}$ and would be encoded as $\mathsf{Prob}_{\mathbf{I},\mathbf{G}}$:

$$\begin{array}{l}
\neg\mathsf{holds}(F, Y_1, Y_2, s_0) \\
\mathsf{adds}(\mathsf{setup}, X_1, X_2, X_3, \mathsf{at}, \mathsf{pk3}, \mathsf{man}) \\
\mathsf{adds}(\mathsf{setup}, X_1, X_2, X_3, \mathsf{at}, \mathsf{pk4}, \mathsf{man}) \\
\mathsf{adds}(\mathsf{setup}, X_1, X_2, X_3, \mathsf{at}, \mathsf{w238}, \mathsf{lon}) \\
\mathsf{executes}(\mathsf{setup}, o, o, o, s_0) \\
\mathsf{next}(T, U) \rightarrow \neg\mathsf{executes}(\mathsf{setup}, X_1, X_2, X_3, U) \\
\mathsf{holds}(\mathsf{at}, \mathsf{pk3}, \mathsf{edn}, s_k) \\
\mathsf{holds}(\mathsf{at}, \mathsf{pk4}, \mathsf{edn}, s_k)
\end{array}$$

The first constraint makes all fluents false at time s_0, then we have the definition of the setup action. A pair of constraints follow that make `setup` execute in the first step, and only then. Finally we specify that the goals should hold at the final state s_k. Note again that we do not have to specify where packages *are not*, such as $\neg\mathsf{at}(\mathsf{pk3}, \mathsf{lon})$, or that the truck is empty (because there is nothing in it).

We finally proceed to describe the rules that actually encode the frame conditions and, at the same time, to disallow the execution of interfering actions. The following sections consider two alternatives that correspond to plans that are either linear or with parallel actions.

6.2.1 Linear plans

One possibility is to allow only one action to execute at a time, this way the frame conditions can be directly expressed stating that the truth value of fluents not added or deleted by an action do not change. Moreover, in order to allow plans whose length is shorter than the bound k, a nop action that does nothing should be added to the definition of the planning domain.

Definition 6.9. Given a planning domain $\mathcal{Dom}$, the *linear frame encoding* of the domain, denoted by $\mathsf{LFrame}_{\mathcal{Dom}}$, is the set containing, for each action symbol $a \in \mathcal{A}$ and fluent $f \in \mathcal{F}$, the constraint

$$\begin{aligned}&\mathsf{next}(T,U) \wedge \mathsf{executes}(a,\overline{X},T) \wedge \textstyle\bigwedge_{\sigma\in\Xi_{a,f}} \overline{Y} \neq \overline{Y}\sigma \rightarrow \\ &\qquad\qquad\qquad\qquad \mathsf{holds}(f,\overline{Y},T) \leftrightarrow \mathsf{holds}(f,\overline{Y},U)\end{aligned}$$

and the pair of constraints

$$\begin{aligned}&\exists A,\overline{X}.\, \mathsf{executes}(A,\overline{X},T)\\ &\mathsf{executes}(A,\overline{X},T) \wedge \mathsf{executes}(B,\overline{Z},T) \rightarrow A = B \wedge \overline{X} = \overline{Z}\end{aligned}$$

where the set $\Xi_{a,f}$ contains all substitutions σ for which the fluent $f(\overline{Y})\sigma$ is either added or deleted by $a(\overline{X})$. ■

Example 6.5. In our example the linear frame conditions for the $\mathsf{load\text{-}truck}$ action would be expressed as follows:

$$\begin{aligned}&\mathsf{next}(T,U) \wedge \mathsf{executes}(\mathsf{load\text{-}truck}, X_1, X_2, X_3, T) \wedge\\ &\quad \neg(Y_1 = X_1 \wedge Y_2 = X_3) \rightarrow \mathsf{holds}(\mathsf{at}, Y_1, Y_2, T) \leftrightarrow \mathsf{holds}(\mathsf{at}, Y_1, Y_2, U)\end{aligned}$$

$$\begin{aligned}&\mathsf{next}(T,U) \wedge \mathsf{executes}(\mathsf{load\text{-}truck}, X_1, X_2, X_3, T) \wedge\\ &\quad \neg(Y_1 = X_2 \wedge Y_2 = X_3) \rightarrow \mathsf{holds}(\mathsf{in}, Y_1, Y_2, T) \leftrightarrow \mathsf{holds}(\mathsf{in}, Y_1, Y_2, U)\end{aligned}$$

In words these constraints state that, except for the package X_1 moved by the action, all other objects remain at their same locations and in their same containers. The last few constraints of $\mathsf{LFrame}_{\mathcal{Dom}}$ encode the fact that one, and only one, ground action executes at any given time.

Note that this encoding requires $|\mathcal{A}|\,|\mathcal{F}|$ constraints to represent the frame conditions, where $|\mathcal{A}|$ (resp. $|\mathcal{F}|$) denotes the number of action (resp. fluent) symbols. Additionally, each fluent added or deleted by actions must appear represented as a substitution in the set $\Xi_{a,f}$ for one of such constraints. Therefore the set of constraints $\mathsf{LFrame}_{\mathcal{Dom}}$ has a size of $O(|\mathcal{A}|\,|\mathcal{F}| + |\mathcal{Dom}|)$.

6.2.2 Plans with parallel actions

Alternatively, several actions can be executed at once as long as they do not interfere with each other. We consider an *explanatory* encoding following ideas proposed by Haas (1987), Schubert (1990) and later applied in the propositional case by Kautz et al. (1996); where it is expressed that, if a fluent changes its value from one step to another, then one of the actions that modify it must have been executed.

Definition 6.10. Given a planning domain $\mathcal{Dom}$, the *parallel frame encoding* of the domain, denoted by $\mathsf{PFrame}_{\mathcal{Dom}}$, is the set containing, for each fluent $f \in \mathcal{F}$, the constraints:

$$\begin{aligned}&\mathsf{added}(f, \overline{Y}, T) \to \textstyle\bigvee_{(a,\sigma)\in\Delta_f} \exists\overline{X}.\, (\mathsf{executes}(a, \overline{X}, T) \wedge \overline{Y} = \overline{Y}\sigma)\\ &\mathsf{deleted}(f, \overline{Y}, T) \to \textstyle\bigvee_{(a,\sigma)\in\nabla_f} \exists\overline{X}.\, (\mathsf{executes}(a, \overline{X}, T) \wedge \overline{Y} = \overline{Y}\sigma)\end{aligned}$$

together with the three constraints

$$\begin{aligned}&\mathsf{next}(T, U) \wedge \neg\mathsf{holds}(F, \overline{Y}, T) \wedge \quad \mathsf{holds}(F, \overline{Y}, U) \to \mathsf{added}(F, \overline{Y}, T)\\ &\mathsf{next}(T, U) \wedge \quad \mathsf{holds}(F, \overline{Y}, T) \wedge \neg\mathsf{holds}(F, \overline{Y}, U) \to \mathsf{deleted}(F, \overline{Y}, T)\\ &\mathsf{dels}(A, \overline{X}, F, \overline{Y}) \wedge \mathsf{reqs}(B, \overline{Z}, F, \overline{Y}) \wedge\\ &\qquad\qquad \mathsf{executes}(A, \overline{X}, T) \wedge \mathsf{executes}(B, \overline{Z}, T) \to A = B \wedge \overline{X} = \overline{Z}\end{aligned}$$

where the set Δ_f (resp. ∇_f) contains the pair (a, σ) whenever the fluent $f(\overline{Y})\sigma$ is added (resp. deleted) by the action $a(\overline{X})$. ■

Example 6.6. In this case, the predicates added and $\mathsf{deleted}$ are defined for each fluent. Consider for instance the following constraint that encodes the frame conditions for the fluent $\mathsf{at}(Y_1, Y_2)$:

$$\begin{aligned}&\mathsf{added}(\mathsf{at}, Y_1, Y_2, T) \to\\ &\qquad\quad \exists\overline{X}.\, (\mathsf{executes}(\mathsf{unload\text{-}truck}, \overline{X}) \wedge Y_1 = X_1 \wedge Y_2 = X_3)\\ &\qquad \vee \exists\overline{X}.\, (\mathsf{executes}(\mathsf{drive\text{-}truck}, \overline{X}) \wedge Y_1 = X_1 \wedge Y_2 = X_3)\end{aligned}$$

If a fluent $\mathsf{at}(Y_1, Y_2)$ is added at some state, then it must be the case that either a package $Y_1 = X_1$ was unloaded at a location $Y_2 = X_3$ (from some truck X_2) or, similarly, a truck was driven to that location from another.

The last few constraints trigger the predicates added and $\mathsf{deleted}$, whenever a change in the truth value of a fluent occurs, in order to search for an explanation of such change. The final constraint disables the execution of two actions when one deletes a requirement of the other and, therefore, they are interfering. It is also not possible to execute two actions such that one deletes the fluent added by the other, a contradiction will occur in $\mathsf{Act}_{\mathcal{D}om}$ when both actions try to assign contradictory values to the fluent.

Note that, in this case, the number of clauses in $\mathsf{PFrame}_{\mathcal{D}om}$ is linear with respect to the number of fluent symbols in $\mathcal{F}$. Moreover, the size of the clauses only depends on the number of actions that add or delete each given fluent. Overall, $\mathsf{PFrame}_{\mathcal{D}om}$ has only a size of $O(|\mathcal{D}om|)$ and does not directly depend on the number of actions or fluents as in the previous case.

Theorem 6.1. *Given a planning domain $\mathcal{D}om$, a problem $\mathbf{I}, \mathbf{G}$ and a positive integer k, the finite domain predicate formula $\mathsf{Bound}_k \wedge \mathsf{Act}_{\mathcal{D}om} \wedge \mathsf{Prob}_{\mathbf{I},\mathbf{G}} \wedge \mathsf{Frame}_{\mathcal{D}om}$, where $\mathsf{Frame}_{\mathcal{D}om}$ is either $\mathsf{LFrame}_{\mathcal{D}om}$ or $\mathsf{PFrame}_{\mathcal{D}om}$, is satisfiable if and only if the planning problem has a solution plan, respectively linear or with parallel actions, of length less than or equal to k.*

Proof. If an interpretation $\mathcal{I}$ is a model of the encoding, then the plan where $\mathbf{A}_i = \{a(\bar{c}) \mid \mathcal{I} \models \mathsf{executes}(a, \bar{c}, s_i)\}$, for $1 \leq i \leq k$, is a valid solution to the planning problem.

Conversely, if $\mathbf{A}_1, \ldots, \mathbf{A}_{k'}$ is a solution plan (linear or with parallel actions) with $k' < k$, then it is possible to build an interpretation $\mathcal{I}$, giving appropriate values to predicates, such that $\mathcal{I}$ is a model of the encoding. □

This finishes the presentation of our first encoding which, although becoming a bit involved in some points, easily and directly encodes the general framework of STRIPS-style planning. Also note that the encoding, in particular the one with parallel actions, has a size of $O(|\mathcal{D}om| + k)$; depending only linearly on the size of the original input description and the bound k.

6.3 Alternative state-based encoding

The previous section followed an approach to directly encode STRIPS-style planning descriptions into the language of finite domain predicate logic, essentially lifting ideas such as those by Kautz and Selman (1992) into a higher level language. In this section we present an alternative encoding which, instead of the STRIPS formalism, reformulates planning as a *reachability* problem.

We present first as motivation an encoding for the game of Tower of Hanoi, and then introduce a second example that more closely resembles planning problems from real-world applications. We finally end this section with a discussion of several issues raised by this kind of encodings, in particular some caveats of STRIPS-style problem descriptions.

6.3.1 Tower of Hanoi

Tower of Hanoi is a popular mathematical puzzle which involves three towers and a series of disks of decreasing lengths. Initially all the disks are stacked in order on the first of the towers, the smallest at the top and the largest at the bottom. One disk at a time is then moved from the top of a tower to the top another one. The move is valid as long as the disk is smaller than the smallest disk on the destination tower. The goal is to eventually move all the disks to the last of the three towers; where they should appear, again, stacked in the correct order.

Consider then the following encoding of the Tower of Hanoi problem with n disks. We will use a predicate $p(T_1, \dots, T_n)$ to encode a state, i.e. the configuration of the disks, during the game. Each variable T_i will range over $\{0, 1, 2\}$ denoting the tower on which the disk i is currently located. A smaller index corresponds to a smaller disk, for example T_1 is the location of the smallest disk.

Now, following the rules of Tower of Hanoi, one is allowed to move a disk k from a tower i to a tower j if

- the disk k is the smallest one in the tower i.
- the disk k would be the smallest one in the tower j.

Alternatively, these constraints can be reformulated as follows:

- the disks $1, \dots, k-1$ are not in the tower i.
- the disks $1, \dots, k-1$ are not in the tower j.

So, to encode the legal moves of Tower of Hanoi we use n clauses of the form

$$\begin{aligned} &p(T_1, \dots, I, \dots, T_j) \wedge T_1 \neq I \wedge \cdots \wedge T_{k-1} \neq I \\ &\qquad \wedge T_1 \neq J \wedge \cdots \wedge T_{k-1} \neq J \rightarrow p(T_1, \dots, J, \dots, T_j)\ , \end{aligned}$$

as well as initial and goal conditions

$$p(0,\dots,0) \qquad \neg p(2,\dots,2)\ .$$

Equality can be removed using the general method of Section 4.4 or, for this simpler case, by adding a number of unit clauses

$$\begin{array}{lll} \mathsf{neq}(0,1) & \mathsf{neq}(0,2) & \mathsf{neq}(1,2) \\ \mathsf{neq}(1,0) & \mathsf{neq}(2,0) & \mathsf{neq}(2,1) \end{array}$$

after replacing every occurrence of $X \neq Y$ with $\mathsf{neq}(X,Y)$. Although not necessary, a unit clause of the form $\neg\mathsf{neq}(X,X)$ can also be added to help provers in pruning clauses faster.

Note that this encoding works by propagating and asserting $p(T_1,\dots,T_n)$ for every configuration that is reachable from $p(0,\dots,0)$. Since the goal configuration $p(2,\dots,2)$ is also reachable, the set becomes unsatisfiable and a solution (i.e. a series of steps to get from one configuration to the other) can be extracted, for example, from a refutation object found by a prover.

It is interesting to observe that, if we decide to ground this set of clauses in order to feed it to a propositional satisfiability solver, we will obtain a rather large set of $O(n3^n)$ clauses which can be, however, shown to be inconsistent by the use of *unit propagation only*. This is because the encoding is *Horn*, i.e. each clause has at most one positive literal.

Although the size of the produced ground encoding is rather large, because of its structure it becomes very simple and easy to process. Moreover, higher level reasoning approaches might not need to generate all those ground clauses at all, and be better able to exploit the simple structure of this Horn encoding.

6.3.2 A logistics example

In order to further explore the possibilities of similar encodings, where each possible state of the domain is encoded as a single predicate $p(\dots)$, we now discuss the application of these ideas in a slightly more realistic planning problem.

We decided to encode one of the simple versions of the DriverLog domain presented at the Third International Planning Competition in 2002 (Fox and Long, 2002). In this domain, trucks are used to move packages among different locations, with the added complexity that drivers are also needed and they have to walk between trucks in order to drive them.

There are three kinds of objects in this domain: *packages*, *drivers*, and *trucks*. Each of these objects can be situated *at* a location in the map. Additionally a package can be *in* a truck, and a driver can be *driving* a truck. We note, however, that these fluents have a functional behaviour, e.g. a package can only be either at one location or in a single truck at any moment in time.

Consider then the following encoding. We will use two sets of constant symbols to identify locations and trucks. Assuming that there are n packages, m drivers and l trucks, we will use a predicate

$$p(P_1, \ldots, P_n, D_1, \ldots, D_m, T_1, \ldots, T_l) \tag{6.1}$$

to encode a possible state of the world in this problem. Variables P_i, D_j and T_k specify, respectively, the current location of packages, drivers and trucks. In particular both P_i and D_j range over locations and trucks, while T_k ranges over locations only. In order to save space and also improve readability, we will write $p[X \to t]$ to denote the state predicate (6.1) after a variable X has been replaced with the term t.

It is now possible to encode actions by specifying how to get from one state to another. For the *load* action we use nl clauses, for each $1 \leq i \leq n$ and $1 \leq k \leq l$, of the form

$$p[P_i \to T_k] \to p[P_i \to t_k] \ ,$$

where t_k is the constant symbol representing the k-th truck. This will allow to put a package in a truck if the package was originally situated at the same location as the truck. The *unload* action is quite simply the converse of *load*, so we add another nl clauses

$$p[P_i \to t_k] \to p[P_i \to T_k] \ .$$

We also have a *board* action which allows a driver to get into a truck. It is very similar to *load*, but has the added restriction that the truck must be originally empty. This is encoded with ml clauses, for each $1 \leq j \leq m$ and $1 \leq k \leq k$, of the form

$$p[D_j \to T_k] \wedge D_1 \neq t_k \wedge \cdots \wedge D_m \neq t_k \to p[D_j \to t_k] \ .$$

To *disembark* a truck we simply need ml clauses

$$p[D_j \to t_k] \to p[D_j \to T_k] \ .$$

If a driver is on board of a truck, then it can *drive* the truck from one to another connected location. This will require ml clauses

$$p[D_j \to t_k; T_k \to L_1] \wedge \mathsf{link}(L_1, L_2) \to p[D_j \to t_k; T_k \to L_2] \ .$$

Finally drivers can also *walk* along paths, this action only requires n clauses

$$p[D_j \to L_1] \wedge \mathsf{path}(L_1, L_2) \to p[D_j \to L_2] \ .$$

At last we only have to add unary clauses, instances of the atoms $\mathsf{link}(X, Y)$ and $\mathsf{path}(X, Y)$, to describe the map that the trucks and drivers use to travel. It is also possible to handle equality by replacing $X \neq Y$ with $\mathsf{neq}(X, Y)$, and asserting this predicate for every pair of distinct trucks and locations. Initial and goal states are given, respectively, as positive and negative instances of the state predicate (6.1). Interestingly, initial and goal states do not need to be ground. If we are just interested, for example, in the actual locations of packages at the end of a plan, variables for drivers and trucks can be left uninstantiated.

Although the planning domain encoded is rather simple, it already serves to highlight some interesting features of our proposed alternative encoding approach

- The resulting is a set of Horn clauses. In particular, propositional instantiations can be solved by unit propagation alone.
- The size of the encoding is only moderately large, with $O(k^2)$ clauses and $O(k^3)$ literals where $k = \max(n, m, l)$.
- Inertial rules, i.e. frame conditions, are easily and transparently encoded simply by using the same variable (e.g. P_i or T_k) in the original and resulting state of objects which are not modified by an action.
- The encoding is in principle unbounded, i.e. it does not restrict plans to a fixed upper bound on the length of the plan. In practise, the only restriction would be the memory available to the prover.

- There are no added complications, such as detecting conflicting or mutually exclusive actions, in order to model parallelism in plans. In principle, a prover can even reuse and compose fragments of plans since it has still access to a high level description of actions.

- The encoding is *natural* and *clear* to understand.

Some other limitations and areas for improvement, however, are also noted:

- This approach requires all the information describing a state to be encoded within a single predicate. If states are complex and involve many variables, they will end up having many arguments and thus rendering a naive instantiation approach completely impractical. This, however, motivates to further the research in non-instantiation methods.

- There is no guarantee that the obtained plans (which can be extracted from refutations of the clause set) will be optimal in any sense. Heuristics in provers, nevertheless, can be used to tune the quality of the plans. Provers are, in any case, already tuned trying to find short refutations, which would in turn correspond to shorter plans.

- Unfortunately there seems to be no easy way to directly apply these ideas to planning domains already defined in standard planning languages, such as STRIPS (Fikes and Nilsson, 1971) and PDDL extensions (Ghallab et al., 1998), that are currently being used by the planning community. The problem is that such descriptions often do not make explicit some information which was essential in the construction of our encoding.

  A common description would use, for example, a series of predicates such as $\mathsf{at}(X, Y)$ and $\mathsf{in}(X, Y)$ to determine the location of objects in the world. But the information that these predicates actually have a function-like behaviour is only implicitly contained in the definitions of actions.

  Moreover, there is usually no distinction between *fluents* that change its value from one state to another, and predicates whose value remains fixed throughout the whole plan. In the DriverLog domain, for example, the fact that the path and link predicates do not change over time, and therefore do not need to be encoded within the states, is again only implicitly available.

disks	Ground	MiniSat	Paradox	Darwin	iProver	Vampire
6	0s	0s	0s	0s	11s	1s
7	1s	0s	0s	1s	1m 03s	3s
8	2s	0s	0s	3s	4m 18s	16s
9	8s	2s	1s	11s	14m 16s	8m 04s
10	28s	5s	2s	41s	– [1]	– [2]
11	1m 38s	19s	7s	2m 34s	–	–
12	5m 35s	1m 05s	19s	– [1]	–	–
13	18m 47s	3m 44s	– [2]	–	–	–
14	48m 39s	11m 21s	–	–	–	–
15	– [1]	–	–	–	–	–

[1]Timed out after 1 hour. [2]Gave up.

Table 6.1: Running times solving Tower of Hanoi

We feel that this provides a good justification to revise the languages that are currently being used by the planning community, since relevant information which is potentially useful to improve the performance of planners is missing from existing domain descriptions.

6.4 Evaluation of the approach

We also ran a batch of experiments in order to evaluate the performance of existing systems trying to solve the problem of Tower of Hanoi with our proposed encoding. The first system considered is a combination of an ad hoc implementation to generate the ground version of this encoding and MiniSat (Eén and Sörensson, 2005), a propositional satisfiability solver. Other systems considered were the three first-order reasoners Paradox (Claessen and Sörensson, 2003), Darwin (Fuchs, 2004), iProver (Korovin, 2006), and Vampire (Riazanov and Voronkov, 2002).

The results, shown in Table 6.1, draw a clear picture on how state-of-the-art systems cope with our proposed encoding. The first two columns represent the total time taken to solve an instance of the problem, and the time spent by MiniSat parsing and performing unit propagation. As can be seen, most of the time is actually spent on the generation of the grounded formulae.

Paradox follows on the list. This is a prover based on a grounding technique and also calls MiniSat as a back end. Interestingly, since Paradox performs some simplifications while grounding, the time it takes to both ground and solve a problem is comparable to the time spent by MiniSat solving an already ground version of the formula. The prover, however, gives up even before trying to ground the 13 disks problem, since it estimates that memory requirements would be rather too large.

Darwin is another prover geared towards effectively propositional formulae, which implements a non-ground version of the DPLL algorithm (Baumgartner and Tinelli, 2003). It also has a competitive performance, but eventually suffers from memory limitations. Although Darwin reports to be able to solve the 12 disks problem in under 20 minutes CPU time, it actually took almost 10 hours on wall clock to complete. The huge difference in time is explained by the operating system being busy swapping memory for the prover to complete its operation. The difference between CPU time and wall clock was not significant when solving problems with fewer disks.

Finally we also included iProver, another prover specialised on effectively propositional formulae, and Vampire, the leading resolution first-order prover, to see how they work on this state-based encoding. We see that the performance of both provers is limited with respect to the others, but they still manage to solve problems with a fewer number of disks. From personal communication with Korovin, this behaviour is explained because of the time spend by these resolution provers on expensive subsumption checks which turned out to be ineffective for this kind of problems. When subsumption is partially disabled on iProver, then the results become comparable to those of Darwin.

Then we also ran a number of experiments with problems semi-automatically translated from the basic STRIPS instances used at the Third International Planning Competition in 2002 (Fox and Long, 2002). A small script in Perl was used to translate problem instances in this planning domain to effectively propositional formulae in the TPTP language. This script, as pointed out on the discussion at the end of Section 6.3.2, uses domain specific information and therefore it is not able to translate arbitrary STRIPS files.

From this we built 20 instances, as shown in Table 6.2, with different numbers of *locations* (l) and objects (o), i.e. packages, drivers and locations. It turned out that only 4 of the easier instances, involving up to 9 objects, were solved in less than half an hour by any of the first-order reasoners. In order to compare with standard approaches, we also ran the SatPlan 2006 release (Kautz et al., 2006)

	l	o	SatPlan	Paradox	Darwin	iProver	Vampire
DriverLog01	3	6	1.0s	8s	8s	2s	1m 03s
DriverLog02	3	7	0.4s	16s	10s	1m 00s	32s
DriverLog03	3	8	0.7s	–	49s	4m 03s	5m 33s
DriverLog04	3	9	0.9s	–	5m 18s	–	58s
DriverLog05	3	10	1.0s	–	–	–	–
DriverLog06	3	11	0.4s	–	–	–	–
DriverLog07	3	12	0.9s	–	–	–	–
DriverLog08	3	13	1.7s	–	–	–	–
DriverLog09	5	11	5.8s	–	–	–	–
DriverLog10	6	11	4.1s	–	–	–	–
DriverLog11	7	11	2m 01s	–	–	–	–
DriverLog12	10	11	1m 13s	–	–	–	–
DriverLog13	12	11	–	–	–	–	–
DriverLog14	10	12	27s	–	–	–	–
DriverLog15	12	16	2m 42s	–	–	–	–
⋮	⋮	⋮	–	–	–	–	–

Table 6.2: Running times solving the 'DriverLog' planning domain

using MiniSat as the back-end satisfiability solver. We can see that SatPlan scales much better, being able to solve the first 10 instances (with up to 11 objects and more locations) in just a few secconds. Problems until the 15th instance (with up to 16 objects) can still be solved in minutes, but then the planner also starts to time-out after half an hour of search.

6.5 Concluding remarks

This chapter has explored many possible alternatives to encode applications from the planning domain using effectively propositional formulae or, equivalently, the finite domain predicate logic.

In particular we have shown first how planning problems, including their frame conditions, can be easily encoded within the proposed fragment of logic. Moreover, the size of the generated formulae is linear with respect to size of a standard description, e.g. in the STRIPS language, of the original planning problem. This is in contrast with propositional encodings where the size of the resulting formulae is often exponential in the size of the input.

We also proposed a second alternative approach which uses a single predicate to encode the complete state of the world at each step in the execution of a plan.

We argue that this encoding has several interesting features. For one it results in a simple set of Horn clauses, despite the fact that a direct translation from existing STRIPS descriptions is less suitable.

We believe that the ideas presented here are of great value to both the planning and automated reasoning communities. First, by describing problems at a higher level of abstraction, it becomes easier to apply more general reasoning techniques that do not necessarily deal with the handling of individual objects one at a time. On the other hand, the ideas presented here might also turn out to be useful for propositional SAT-based approaches. Since the problem of finding optimised propositional encodings, including but not limited to planning, is reduced to finding an appropriate instantiation of the obtained finite domain formula.

Moreover, in particular for the automated reasoning community, we are also providing a source of new and relevant benchmarks which developers of first-order reasoners, particularly those geared towards the effectively propositional fragment, can use to test the capabilities of their systems. As a contribution of this thesis, in fact, we have made available a set of effectively propositional formulae, obtained from the experiments performed in Section 6.4, in the TPTP format for the research community to use. Another interesting issue which has been left open until now, and which is the matter of the following chapters, is how existing reasoners can be improved, and new ones designed, in order to more efficiently solve this kind of problems.

Chapter 7

Solving problems in effectively propositional logic

The previous two chapters were mainly concerned with the translation of problems from different applications into effectively propositional formulae. And, although some experiments were performed where the generated problems were fed into existing theorem provers, the issue of how to design algorithms and reasoning mechanisms to deal with such problems has not been examined so far.

Earlier in Chapter 4, a brief description of the state of the art on techniques to check the satisfiability of effectively propositional formulae was given. The aim of this chapter will therefore be to study and evaluate some possible improvements on these reasoning techniques.

This chapter is divided in two main sections. The first one mostly deals with ideas and approaches to checking the satisfiability of effectively propositional logical formulae by reducing them to propositional logic. After such reduction, standard propositional techniques, such as those described in Chapter 2, are applied in order to actually solve the problem. In particular we propose a linking restriction which complements earlier ideas of Sections 4.2.3 and 4.2.4; and present a comparison between incremental and one-shot methods to solve EPR formulae.

Then, the second section discusses some reasoning techniques which can be directly applied to effectively propositional clauses, without having perform a preliminary ground instantiation. We show how inference rules, such as resolution, are potentially more efficient when directly operating with clauses in this form; and propose a new generalisation inference rule which, furthermore, allows to more efficiently generate proofs and refutations.

7.1 By reduction to propositional logic

As anticipated, in this section we will first consider techniques within an instantiation framework where, in order to solve an EPR problem, one tries to generate an equisatisfiable ground version which is finally solved by a propositional satisfiability solver. The advantage of this approach is that it becomes easy to incorporate all the technology and optimisations that have already been developed by the propositional satisfiability community in recent years.

On the other hand, by designing efficient procedures to translate effectively propositional formulae into plain propositional, one provides an alternative approach to produce propositional encodings of real-world problems. Instead of carefully crafting encodings of different application domains into propositional logic, one can instead take advantage of the more friendly and natural language of effectively propositional logic, and then use a single unified translator which provides a number of generic simplifications and optimisations simultaneously to all application domains.

In particular, we discuss two main ideas. The first one is an alternative to the linking restriction proposed by Schulz (2002) and implemented in EGROUND. Our proposed ground term linking is inspired by issues raised when solving real-world problems and, as we show, is rather easy to implement on top of existing instantiation-based systems such as PARADOX (Claessen and Sörensson, 2003).

The second idea revisits the issue on whether, for effectively propositional formulae, an incremental or a one-shot approach is more convenient. PARADOX, which is one of the leading instantiation-based provers for effectively propositional problems, uses an incremental approach. We challenge this design decision and show that, in many cases, employing a simple one-shot approach the efficiency of the prover is increased by several orders of magnitude.

7.1.1 Improving linking restrictions

Consider an effectively propositional formula which includes a binary predicate symbol p all whose positive occurrences are the ground instances: $p(0,1)$, $p(1,2)$, $\dots$, $p(k-1,k)$. Then, negative instances of $\neg p(x,y)$ only need to be instantiated with exactly those combinations of terms, since any other would result in the creation of a pure literal. Interestingly, neither EGROUND's linking restrictions nor PARADOX's sort inference are able to capture this information and both generate an amount of $O(k^2)$ of instances instead.

On the other hand, it is fairly common for this kind of situation to arise on problems generated from real-world applications. In the bounded model checking problems of Chapter 5, for example, all the positive occurrences of the atom succ are ground and, if the bound of the problem is k, there are only k different instances of them. However, again, both EGROUND and PARADOX would create an amount of $O(k^2)$ of ground instances for each negative $\neg\mathsf{succ}(x, y)$. From this we came up with the following algorithm as an alternative linking restriction, where the set T_s is defined as in Definition 4.4 in the previous Section 4.2.3.

Algorithm 7.1 (Ground term linking)**.** For each signed predicate symbol s: if all tuples in the set T_s are ground then, during instantiation, a clause containing the literal $\tilde{s}(t_1, \ldots, t_n)$ is generated only if the tuple of terms $(t_1, \ldots, t_n) \in T_s$.

Although this linking restriction might seem to be very weak, it is also very useful in practise since, first, it is applicable to many instances generated from real-world problems and, second, it is easy to implemented on top of existing systems such as PARADOX with very little overhead. Moreover, this linking restriction is orthogonal to other optimisations such as sort inference.

One first has to analyse the input formula F to find predicate symbols p such that either T_p or $T_{\neg p}$ contain only ground terms. This process is simple, is done only once and in linear time at the beginning of the solving process. Then, since PARADOX already maintains a hash to map every ground term to a propositional atom for use by MINISAT (Eén and Sörensson, 2005), the back end SAT solver, one can use this hash table to record the admissible ground instances of the relevant predicate symbols.

After recording such ground terms, and before starting the search for models, one locks the hash so that no more instances are generated. Later, during the instantiation process, any attempt to create an instance which is not already present in a locked hash will be denied. Thus allowing the prover to discard clauses containing those instances even before they are generated and sent to the satisfiability solver.

It is interesting to note that the stronger notion of complete linking from Algorithm 4.2, very similar to the *hyper-linking* restriction of Lee and Plaisted (1992), is also easy to implement on top of PARADOX. One would have to collect, for each signed predicate symbol s, the set of terms T_s. Then, before generating a clause containing the ground literal $\tilde{s}(t_1, \ldots, t_n)$ one would have to perform an 'instance of' check of the tuple $(t_1, \ldots, t_n)$ against all tuples in T_s.

Moreover, this basic idea can be improved in at least two ways. First, the set of tuples T_s can be pruned by removing any tuple which is an instance of another tuple already in T_s (i.e. subsumption checking). And second, the 'instance of' checks are only needed when a new ground term is seen by the prover, since the result of this check can also be stored in the hash table of each predicate symbol.

Nevertheless, there is still a small overhead if one were to implement this strategy. The 'instance of' checks, even if done only once for each ground term, might turn out to be expensive and outweigh the possible benefits obtained by limiting the instance generation with this complete linking approach.

In either case, we found that these techniques are easier to implement when using a one-shot rather than an incremental generation approach. Thus raising the question of —particularly for the case of effectively propositional formulae— whether the incremental strategy, as implemented e.g. in Paradox, is the most appropriate when instantiation is combined with linking restrictions.

7.1.2 Incremental and one-shot methods

The idea behind the implementation of Paradox works by, instead of trying to find a refutation of the set of input clauses, searching for a finite model which would establish the satisfiability of such clauses. This technique, also known in the literature as MACE-style finite model finding (McCune, 1994a) as we saw earlier on Section 4.2.5, generates a set of propositional clauses which is satisfiable if and only if the original input set has a model with a domain of size k.

For arbitrary first-order formulae, there is no a priori bound until when the search can be stopped. Hence, the typical implementation incrementally tries all possible domain sizes until either a model is found or resources are exhausted. It is however known that, as a consequence of Herbrand's theorem, if a formula is satisfiable then it must have a model whose domain is its Herbrand domain. For effectively propositional formulae, the Herbrand domain is finite and, therefore, it is enough to search for a model of size $\hat{k}$, where $\hat{k}$ is the number of constant symbols in the original input description.

Systems such as eground (Schulz, 2002) implement a one-shot approach for effectively propositional problems where only a search for a model of size $\hat{k}$ is attempted. Paradox's implementation, however, uses the incremental approach

Problem domain	sat	unsat	–	
Syntactic	90	362	98	452
Natural language processing	177	7	38	184
Algebra - Groups	56	51	2	107
Other	37	46	32	115
	360	466	170	996

Table 7.1: Benchmarks used to test instantiation methods

trying all domain sizes from 1 until $\hat{k}$ for effectively propositional formulae under the rationale that, if the problem is satisfiable, "it is very likely that it is not needed to go all the way up to the biggest case, and that a smaller model can be found much quicker" (Claessen and Sörensson, 2003). This design decision seems to be appropriate since, even in unsatisfiable problems, Paradox often outperforms eground on a number of effectively propositional benchmarks from the TPTP problem library.

We observed that, however, in order to implement the ideas discussed in the previous section a one-shot approach is often more convenient. Moreover, many simplifications in the encoding generated by Paradox become possible, for example symmetry reduction clauses can be simplified or even avoided altogether in problems with no equality or function symbols. Experiments in the following section explore and compare the effectiveness of the incremental and the one-shot approach when combined with ground term linking.

7.1.3 Evaluation of the approach

In order to experimentally evaluate the ideas that have just been briefly discussed in this section, we modified a version of Paradox which implements the ground term linking restriction combined with a one-shot approach to solve effectively propositional formulae.

We then selected a number of effectively propositional benchmarks from the TPTP library. Table 7.1 shows the number of problems used by their domain and satisfiability status. A number of problems remained unsolved by both the original and the modified versions of Paradox after 15 minutes of search, those are listed in the third column.

All problems solved by the original version of Paradox were also solved by the modified version, plus two unsatisfiable problems which originally remained unsolved and now were completed almost instantly. Table 7.2 shows a summary

	sat	unsat
time	+ 97.43%	− 88.83%
domain size	+291.83%	− 8.21%
atoms	+ 31.51%	− 2.43%
clauses	+292.38%	− 25.97%
conflicts	− 2.89%	− 99.97%

Table 7.2: Summary of the effects of new instantiation methods

of how some of the statistics changed from the original to the modified version for satisfiable and unsatisfiable problems.

As we expected, because of the use of a one-shot instead of an incremental approach, the time required to solve satisfiable problems increased and, for these particular benchmarks, it almost doubled on average. The largest domain size for which a model was sought, as well as the number of ground clauses generated, almost quadrupled. While the number of distinct ground atoms also had an increase of about 30%. Interestingly, the number of conflicts generated by the SAT solver, an indication of the search space explored, did not change significatively. One might speculate that, although it is possible to find shorter models with the incremental approach, the actual search space explored by the SAT solver while looking for a larger model does not necessarily change too much.

It is also worth mentioning that most satisfiable problems are, nevertheless, rather easy and are solved within a few seconds by both the original and the modified versions of Paradox. About 90% of the problems are still solved in less than one second even by the modified version which does perform full instantiation. Also, when looking at individual problems, in 59% of the cases the encodings of the generated problem were still smaller or equal in the modified version, while 71% of the problems also required less or the same ammount of search (number of conflicts).

On the other hand, the results for unsatisfiable problems are rather impressive. Running time was reduced about 89%, that is almost 9 times faster than the original implementation of Paradox. The largest domain size for which a model was sought is also reduced. This is because, if a contradiction arises during the instantiation phase, the procedure is stopped and no search is performed at all. This was the case in a few problems, where the contradiction was not detected for smaller domain sizes but it was easily spotted when trying to do the full instantiation. Some significant reductions are also seen in the number of distinct ground atoms, and ground clauses generated. These are result of the use of our proposed linking restriction. Finally the number of conflicts made by the SAT solver were also dramatically reduced by several orders of magnitude from about a hundred thousand conflicts to just 30 (on average).

Again, looking at individual problems, in 13% of the problems the running times were reduced from minutes to a few seconds while the rest (87%) were solved in roughly the same time. In 49% of the problems the size of the generated ground instance was significatively reduced (more than 40%), while in 48% the search space (number of conflicts) was at least halved. Interestingly, 59% of the problems were solved by unit propagation only, as opposed to 32% for the original version of Paradox.

7.1.4 Results and conclusions

In this section we investigated a couple of techniques which, particularly for unsatisfiable problems, are useful to significantly reduce both the search space explored and the total running time of instantiation-based provers. As expected, in the case of satisfiable problems, the use of a one-shot instead of an incremental approach often has an adverse effect. We observed, however, that the magnitude of these negative effects are often rather small since satisfiable problems tend to be easier anyway.

While these techniques were described in the context of instantiation-based methods, it might also be possible to apply them in different or broader circumstances. The proposed linking restriction, for example, can also be used in a saturation-based framework in order to discard many clauses which might not be relevant for obtaining a proof. The following section discusses these ideas, where reasoning is done directly at the effectively propositional level, without resorting to a propositional-based approach.

7.2 Reasoning in effectively propositional logic

In the previous section we mainly focused on approaches that, although designed to more efficiently deal with effectively propositional formulae, at the end they still resort to a reduction of problems to propositional satisfiability checking where all the actual *reasoning* is performed. We will now show how methods to directly reason at the effectively propositional level open up the possibility of developing even more efficient reasoning techniques.

In order to do so, we will first show how resolution can be exponentially more efficient when used directly at the effectively propositional rather than through a reduction to the propositional level. Although this result is not surprising, and even perhaps expected, we are not aware of any formal proof of this in the literature. We will, moreover, give an explicit example of a family of formulae which have short quadratic first-order resolution refutations, while even their smallest propositional refutations are exponential in size.

Then, as one of the contributions of this thesis, we propose a new inference rule which we call *generalisation*. This makes use of sort assignments in order to infer general information out of facts known about individual elements. We first show that resolution can be augmented with generalisation giving rise to a system which is still refutationally sound and complete. We will also show how some standard sort inference algorithms, such as the one presented in Section 4.2.4, are compatible with generalisation.

Finally we will give a formal proof that refutations with generalisation can become exponentially smaller than those using resolution only. This is done, similar to the comparison of effectively propositional and propositional resolution, by giving an explicit example of a family of formulae which exhibit this exponential gap between the two reasoning methods.

7.2.1 Propositional vs first-order resolution

In order to further motivate the use of Bernays-Schönfinkel formulae as a formal language to represent problems, we prove in this section that resolution reasoning within this fragment can be exponentially more efficient than in propositional logic. This shows that the language not only provides means for creating more compact encodings, but that the actual solving time is potentially reduced by the use of this approach.

We consider proofs using the resolution inference system, as introduced earlier in Section 4.3.1, which operates on sets of effectively propositional clauses; for additional details we also refer the reader to the work of Bachmair and Ganzinger (2001). In particular, we add now the notion of a *propositional refutation*, which simply is a refutation all whose formulae are ground.

We will now proceed to show that there is a family of sets of clauses S_i with respective unsatisfiability proofs Γ_i, where the shortest propositional refutation of S_i is exponentially larger than Γ_i. Recall that, although clauses formally are disjunctions of literals, we will often write them as implications in order to improve readability. The following theorem states the main result of this section in a formal way.

Theorem 7.1. *There is a sequence of sets of clauses $S_1, S_2, \ldots$ of increasing sizes such that each S_i has a refutation of a size quadratic in i, while the shortest propositional refutation of S_i has a size exponential in i.*

Proof. Take a logic whose language has the set of constant symbols $\mathbb{B} = \{0, 1\}$, and a single predicate symbol s of arity i. We will denote by $\bar{0}$, $\bar{1}$, $\bar{x}$ etc. sequences of constants 0, 1 and variables, respectively, whose length will be clear from the context. The set S_i consists of the following clauses:

$$s(\bar{0}) \ . \tag{7.1}$$

i clauses of the form:

$$s(\bar{x}, 0, \bar{1}) \to s(\bar{x}, 1, \bar{0}) \ . \tag{7.2}$$

The clause

$$s(\bar{1}) \to \bot \ . \tag{7.3}$$

This set of clauses is unsatisfiable and its size is quadratic in i.

Note that every ground atom is of the form $s(\bar{b})$, where $\bar{b}$ is a sequence of bits representing a number between 0 and $2^i - 1$ written in binary notation. For a number n such that $0 \leq n < 2^i$ let us denote by $\underline{n}$ the sequence of i bits denoting this number. Then (7.1) asserts $s(\underline{0})$ and (7.3) asserts $\neg s(\underline{2^i - 1})$, while the ground instances of clauses in (7.2) assert $s(\underline{n}) \to s(\underline{n+1})$. Using this observation it is not hard to argue that every unsatisfiable set of ground instances of clauses in S_i contains all ground instances of (7.2), and so all propositional refutations of this set have a size exponential in i.

Let us show that S_i has a non-ground refutation of a quadratic size. To this end, we will show, by induction on the length of a non-empty sequence of constants $\bar{1}$, resolution proofs of the clauses

$$s(\bar{x}, \bar{0}) \to s(\bar{x}, \bar{1}), \tag{7.4}$$

having a number of steps linear in the length of $\bar{1}$.

When the length is 1, then (7.4) is an instance of (7.2). When the length is greater than 1, by induction, we know that there is such a refutation of a clause

$$s(\bar{x}, y, \bar{0}) \rightarrow s(\bar{x}, y, \bar{1}) \ . \tag{7.5}$$

From this and (7.2) we derive by a resolution inference the clause

$$s(\bar{x}, 0, \bar{0}) \rightarrow s(\bar{x}, 1, \bar{0}) \ .$$

From this and (7.5) we derive by a resolution inference the clause

$$s(\bar{x}, 0, \bar{0}) \rightarrow s(\bar{x}, 1, \bar{1}) \ .$$

and we are done.

This implies that there is a resolution proof of the clause

$$s(\bar{0}) \rightarrow s(\bar{1})$$

having a number of steps linear in i, and hence a refutation having a number of steps linear in i. Moreover, the size of each clause in the refutation is linear in i, so the size of the refutation is quadratic in i. □

This example shows how, for some families of formulae, the difference between reasoning at the effectively propositional rather that just propositional level can be a very significant one. Interestingly, as we have already pointed out in Section 4.3.1, resolution was found not to be very competitive on effectively propositional theorem proving. In the following section we introduce another inference rule, particularly designed for effectively propositional formulae, as a proposal to complement the resolution approach.

7.2.2 The generalisation inference rule

Suppose that, using some inference system such as resolution, a prover has been able to derive, from a set of constraints F, that the fact $p(c_i)$ is true for *all* elements in the domain of the logic. Imagine, moreover, that F includes a rule of the form $p(x) \rightarrow G(x)$, where G represents some large fragment of F. Logically, one should now be able to infer that $G(x)$ is true *for all* x; using only resolution, however, the best that one can obtain is, after performing one inference for each constant symbol, that each individual $G(c_i)$ is true. Inspired by this example is that we propose the generalisation inference rule which, unlike resolution, is capable of performing this kind of inferences.

Anticipating material in later sections, we will introduce the definition of the generalisation inference rule with respect to a sort assignment, those previously introduced in Section 4.2.4. In particular recall that a sort assignment A is a function that maps each predicate position to a *sort*, i.e. a set of constant symbols; and that, for a formula F, the notation $F|_A$ denotes the set of all ground instances of F which are compatible with A. Now, formally, we have the following definition.

Definition 7.1. Given a sort assignment A, the inference rule of *generalisation with respect to A* is

$$\frac{C_1 \vee s(\bar{x}, c_1, \bar{z})\sigma_1 \quad \cdots \quad C_n \vee s(\bar{x}, c_n, \bar{z})\sigma_n}{C_1\sigma \vee \cdots \vee C_n\sigma \vee s(\bar{x}, y, \bar{z})\sigma}\ \mathsf{Gen}_A$$

where $\{c_1, \ldots, c_n\}$ is the sort assigned to the relevant predicate position with respect to A, and σ is the most general unifier such that $s(\bar{x}, c_i, \bar{y})\sigma_i\sigma = s(\bar{x}, c_i, \bar{y})\sigma$, for every $1 \leq i \leq n$. ■

One of the first results that we want to show, is that when resolution is combined with generalisation, then the obtained inference system is still, as resolution alone, refutationally sound and complete. For this we begin by introducing the following set of clauses which will be useful to simulate generalisation using resolution alone.

Definition 7.2. Given a domain restriction A, we define the set of *generalisation clauses* $[\![A]\!]$ as the set containing all the possible clauses of the form

$$s(\bar{x}, c_1, \bar{z}) \wedge \cdots \wedge s(\bar{x}, c_n, \bar{z}) \rightarrow s(\bar{x}, y, \bar{z})\ , \tag{7.6}$$

where $\{c_1, \ldots, c_n\}$ is the domain restriction of the relevant predicate position with respect to A. ■

Notice that all clauses in $[\![A]\!]|_A$ are tautologies, since the variable y would have to be mapped to a constant c_i of its appropriate sort.

Lemma 7.1. *If there is a refutation of a set of clauses S using resolution and generalisation with respect to a domain restriction A, then there is a refutation of $S \cup [\![A]\!]$ using only resolution.*

Proof. The result easily follows by noticing that generalisation inference steps can be simulated by resolving the n clauses of the form $C_i \vee s(\bar{x}, c_i, \bar{z})\sigma_i$ with the corresponding clause in the set $[\![A]\!]$ from Definition 7.2. □

Now it is only left to provide some sufficient conditions to obtain suitable sort assignments and show that, for such assignments, the set $[\![A]\!]$ is redundant and can therefore be eliminated.

Definition 7.3. A *sort inference function* Ξ is a function that yields a sort assignment given a set of clauses as input. Moreover, we say that Ξ is

- *valid* if, for any set of clauses S, the sets S and $S|_A$ are equisatisfiable; and
- *stable* if, for any set of clauses S, $\Xi(S \cup [\![A]\!]) = A$.

where, in any case, $A = \Xi(S)$. ■

The first condition, validity, states that when checking the satisfiability of S by using instantiation-based methods, the generation of ground instances can be restricted to those which are compatible with A. The condition of stability asserts that the sort inference procedure is not affected when the set of generalisation clauses $[\![A]\!]$ is added to a formula.

Getting closer to the proof of the main claim in this section, the following theorem proves that resolution can be extended with generalisation preserving soundness.

Theorem 7.2. *Let S be a set of clauses, and let Ξ a valid and stable sort inference function. Also let $A = \Xi(S)$. If there is a refutation of S using resolution and generalisation with respect to A then there is a refutation of S using resolution only.*

Proof. If there is a refutation of S using resolution and generalisation with respect to A then, by Lemma 7.1 and since resolution is sound, $S \cup [\![A]\!]$ is unsatisfiable.

Now, since the sort inference function is stable, $\Xi(S \cup [\![A]\!]) = A$, and, because of its validity, the sets $S \cup [\![A]\!]$ and $(S \cup [\![A]\!])|_A = S|_A \cup [\![A]\!]|_A$ are equisatisfiable. But recall that $[\![A]\!]|_A$ contains only tautologies and, therefore, $S|_A \cup [\![A]\!]|_A$ and $S|_A$ are equisatisfiable. Finally, again by validity of Ξ, $S|_A$ and S are equisatisfiable.

So, since $S \cup [\![A]\!]$ is unsatisfiable, then S also is, and since resolution is refutation complete, there is a refutation of S using resolution only. □

From this, our main result now follows as a simple corollary.

Corollary 7.1. *An inference rule system based on resolution and generalisation, with respect to a valid and stable sort inference function, is both refutationally sound and complete.*

Proof. Soundness follows by Theorem 7.2, while completeness is directly inherited from the completeness of resolution. □

From this result it follows that, any valid and stable sort inference function is suitable to be combined with generalisation in order to produce a sound and complete inference system. It still remains open, however, the question on how to obtain such kind of sort inference functions. This is the matter of the following section.

Sort inference for generalisation

In this section we will explore some possibilities in order to generate sort information that can be combined with the generalisation inference rule. A first option, though not very interesting, is to assign the trivial sort assignment to all sets of clauses.

Definition 7.4. Given an effectively propositional language with a domain $\mathcal{D}$ of constant symbols, the *trivial sort assignment* is the function that maps every predicate position to $\mathcal{D}$. ■

That is, it uses the domain of the logic itself as the sort for all variables and positions in predicates. This procedure is clearly stable since, irrelevant to the particular set of input clauses, the trivial sort assignment is always used. Moreover, from Herbrand's theorem, Theorem 4.2, this procedure is also valid and therefore a suitable candidate to be used together with generalisation.

In the following Section 7.2.2, we will see how even this simple approach can already represent a significant advantage over using the resolution inference rule alone. However, particularly on problems from applications, it is very likely that more specialised sort inference functions are able to give even better results in practise.

In Section 4.2.4 we already saw an example of a sort inference function, i.e. Algorithm 4.4, as a method proposed by Claessen and Sörensson (2003) and implemented in Paradox in the context of grounding-based model finding. Moreover, it is not hard to argue, as it is done by Claessen and Sörensson (2003), that this domain inference function is valid —in the sense of our Definition 7.3— and that, moreover, is not affected when adding the set of clauses $[\![A]\!]$ to S. So, from Theorem 7.2, it follows that we already have a sort inference method that, without any further modifications, can be directly used to empower generalisation inferences.

It is to be expected, that if one obtains a sort inference method by extending some available technique to restrict the number of generated instances in a grounding approach, then the obtained method will most likely be valid. This follows since such kind of methods actually work by replacing the satisfiability testing for an effectively propositional formula, to checking instead an equisatisfiable set of propositional instances. This equisatisfiability is, precisely, what the property of validity asks for.

Unfortunately, however, not any ground restriction method can be so easily integrated with generalisation. From linking restrictions of Section 4.2.3, in particular the positional linking restriction, i.e. Algorithm 4.3, from Schulz (2002) and implemented in EGROUND, one can easily define the following sort inference function.

Algorithm 7.2 (Positional sort inference)**.** Given a set of clauses S, and for every signed predicate position of the form $s.i$, build as in Algorithm 4.3 the sets $C_{s.i}$ of all the constants that appear in a literal with signed predicate s at position i, or all constant symbols in $\mathcal{D}$ if a variable appears in some literal at that position.

For each predicate position $p.i$, the *positional sort inference* is defined as the function that maps each such set S, to the sort assignment $A_{p.i} = C_{p.i} \cap C_{\neg p.i}$.

This sort inference function is clearly also valid. It works by the observation that literals which are not compatible with the generated sort assignment would be pure, i.e. they only appear in one of the two possible phases, and so they can be discarded. However, this sort inference function is not stable, as the following example shows.

Example 7.1. Consider the following satisfiable set of clauses S:

$$p(a)$$
$$\neg p(x) \vee q(x)$$
$$\neg q(b)$$

The positional sort would have $A_{p.1} = \{a\}$, $A_{q.1} = \{b\}$, and the restricted set $S|_A$ is simply $\{p(a), \neg q(b)\}$. Note that, however, adding the clause

$$\neg p(a) \vee p(x)$$

which is part of $[\![A]\!]$, would cause $A_{p.1} = \mathcal{D} = \{a, b\}$ —because now a variable appears in $p(x)$ on both positive and negative phases— making the sort inference function unstable and rendering the set of clauses $S \cup [\![A]\!]$ unsatisfiable.

In this section we have shown how a sort inference method, as proposed by Claessen and Sörensson (2003), can be used together with the generalisation inference rule in order to make it more easily applicable in practise. In the following we give, in the form a theoretical result, some evidence on why combining generalisation with resolution is likely to produce a powerful reasoning system.

Generalisation vs resolution

In this section, and in order to further motivate the use of the generalisation inference rule in combination with resolution, we show a family of formulae which, similar to the one given in Section 7.2.1, shows that refutations can become exponentially shorter when combining resolution with the generalisation inference rule. For doing so we will show an example of a series of unsatisfiable sets of clauses $S_1, \ldots, S_n$ such that the length of shortest resolution refutation of S_n is exponential in n, while using both generalisation and resolution it is possible to find a refutation of size quadratic in n. In the following we will use x_i to represent variables, b_i and c_i for constant symbols, as well as s_i and t_i for arbitrary terms.

Definition 7.5. Take a logic whose language has a set of constant symbols $\mathbb{B} = \{0, 1\}$ and let n be a non-negative number. For every i, with $0 \leq i \leq n$, there is a pair of predicate symbols p_i and q_i both of arity i.

Now let S_n be the set of clauses that contains: the clause

$$p_0 \ , \tag{7.7}$$

$2n$ clauses, two for every $0 \leq i < n$, of the form

$$\begin{aligned} p_i(x_1, \ldots, x_i) &\rightarrow p_{i+1}(x_1, \ldots, x_i, 0) \ , \\ p_i(x_1, \ldots, x_i) &\rightarrow p_{i+1}(x_1, \ldots, x_i, 1) \ , \end{aligned} \tag{7.8}$$

the clause

$$p_n(x_1, \ldots, x_n) \rightarrow q_n(x_1, \ldots, x_n) \ , \tag{7.9}$$

n claques, one for every $0 \leq i < n$, of the form

$$q_{i+1}(0, x_{\underline{i}}, \ldots, x_n) \land q_{i+1}(1, x_{\underline{i}}, \ldots, x_n) \rightarrow q_i(x_{\underline{i}}, \ldots, x_n) \ , \tag{7.10}$$

where $\underline{i} = n - i + 1$, and the clause

$$q_0 \rightarrow \bot \ . \tag{7.11}$$

Moreover, we will assume that generalisation inferences are applied with respect to the trivial sort assignment that simply maps every predicate position to the domain set $\mathbb{B} = \{0, 1\}$. ■

Intuitively, clauses of the form (7.8) encode the fact that if $p_i(b_1, \ldots, b_i)$ is true, then p_n should be true for all n-bit strings with a prefix of $b_1, \ldots, b_i$. A dual of this is encoded by clauses of the form (7.10): if $q_i(b_{\underline{i}}, \ldots, b_n)$ is false, then q_n should be false for some n-bit string with a suffix of $b_{\underline{i}}, \ldots, b_n$.

From clauses (7.7) and (7.8) we get that $p_n(b_1, \ldots, b_n)$ is true for all n-bit strings. Then from (7.9) that $q_n(b_1, \ldots, b_n)$ is also true for all n-bit strings and, therefore from (7.10), the atom q_0 should be true. But this causes a contradiction with (7.11), so the set S_n is unsatisfiable.

Indices in variables and terms have been chosen to enforce the *prefix* and *suffix* intuition of these predicate symbols. Formally, in the atoms $p_i(t_1, \ldots, t_i)$ and $q_i(t_{\underline{i}}, \ldots, t_n)$, we say that the position of the term t_i is the i-th *bit position*. Note that in all clauses of S_n, the variable x_i only appears at the i-th bit position of an atom.

Theorem 7.3. *There is a refutation of S_n, using both generalisation and resolution inference rules, which is of size quadratic in n.*

Proof. We start our refutation with the clause (7.7) which is the fact p_0. Observe now that it is possible to extend a proof of

$$p_i(x_1, \ldots, x_i) \tag{7.12}$$

to a proof of

$$p_{i+1}(x_1, \ldots, x_i, x_{i+1}) \tag{7.13}$$

by adding a constant number of steps.

To do this, first apply a generalisation inference on the pair of clauses (7.8) to obtain

$$p_i(x_1, \dots, x_i) \rightarrow p_{i+1}(x_1, \dots, x_i, x_{i+1}) \ ,$$

and then resolve this with (7.12) to obtain (7.13).

After n iterations of this procedure we get a proof of $p_n(x_1, \dots, x_n)$ whose length is linear in n. Now, resolve this with (7.9) to obtain $q_n(x_1, \dots, x_n)$. Observe now that we can extend a proof of

$$q_{i+1}(x_{\underline{i}-1}, x_{\underline{i}}, \dots, x_n) \tag{7.14}$$

to a proof of

$$q_i(x_{\underline{i}}, \dots, x_n) \tag{7.15}$$

by adding a constant number of steps.

To do this, simply resolve (7.14) with (7.10) to obtain

$$q_{i+1}(1, x_{\underline{i}}, \dots, x_n) \rightarrow q_i(x_{\underline{i}}, \dots, x_n) \ ,$$

and again with (7.14) to finally obtain (7.15).

After n of such iterations we end with a proof of q_0 which is also of length linear in n. Finally resolving q_0 with (7.11) we obtain a refutation of S_n. Since the size of each clause in the refutation is also linear in n, the size of the refutation is quadratic in n. □

The following is the main theorem of this section. It shows that, using resolution alone, even the shortest refutation is of length at least exponential in n. We will now give the formal statement of this theorem, and a short sketch of its proof. The rest of this section will fill in all the details left out in this sketch.

Theorem 7.4. *A resolution refutation of S_n has a length of, at least, 2^n.*

Sketch of proof. To prove that any refutation of S_n has at least an exponential length, we will introduce a function on sets of clauses that, in a way, measures the accumulated progress achieved step by step on a refutation.

This *work* function, denoted by w, will map the set of clauses occurring in a partial proof Γ to the set of n-bit strings which, intuitively, have already been *consumed* while trying to build a refutation. This function should moreover satisfy $w(S_n) = \emptyset$, while the work of any refutation Γ is $w(\Gamma) = \mathbb{B}^n$.

It is then only left to show that, if a partial proof Γ' is extended to another Γ by an application of a resolution inference step, then $w(\Gamma)$ will include, at most, one element of $\mathbb{B}^n$ which was not already in $w(\Gamma')$. From this it follows that any refutation, which would have to include all elements in $\mathbb{B}^n$ one by one, has at least an exponential length. □

In order to fill in the details of this proof, we first have to identify the kind of clauses that appear in a refutation of S_n. We claim that, indeed, only two different kinds of clauses are possible. In the following description recall that x_i is used to represent variables, b_i and c_i are constant symbols, while s_i and t_i are arbitrary terms. Then, the first group I contains Horn clauses of the form

$$[p_i(x_1, \dots, x_i)] \to p_j\,(x_1, \dots, x_i, b_{i+1}, \dots, b_j\,) \tag{7.16}$$

$$[p_i(x_1, \dots, x_i)] \to q_n(x_1, \dots, x_i, b_{i+1}, \dots, b_n) \tag{7.17}$$

where the first literal is optional and, if present in (7.16), then $i < j$.

The second group II are Horn clauses of the form

$$l_1 \wedge \cdots \wedge l_m \to h \tag{7.18}$$

where

- the head h is either $\bot$, or a literal $q_i(t_{\underline{i}}, \dots, t_n)$ with $i < n$,
- the body of the clause can be empty and, if it is not, each literal should be either of the form
  - $q_j(c_{\underline{j}}, \dots, c_{\underline{i}-1}, t_{\underline{i}}, \dots, t_n)$ where $j > i$, or
  - $p_j(s_1, \dots, s_j)$ where, for every $\underline{i} \le k \le j$, $s_k = t_k$. That is, if the sequence of indices $1, \dots, j$ and $\underline{i}, \dots, n$ overlap, then the terms in this atom must agree with the corresponding terms in the head.

If a literal in the body of a clause has a predicate symbol p_j for some $j < \underline{i}$, i.e. its sequence of terms does not overlap with those in the head, then we say that the literal is *inactive*. All other literals are *active*.

In the following we use the notation $\langle S \rangle$ to denote the closure of a set of clauses S with respect to the resolution inference rule.

Proposition 7.1. *All clauses in $\langle S_n \rangle$ are either of type I or type II.*

Proof. Clauses (7.7), (7.8) and (7.9) are of type I, while (7.10) and (7.11) are of type II. It is also not hard to verify that resolving two clauses of type I yields another clause of type I. While resolving a clause of type II with any of type I or II, the result is still of type II.

Note, in particular, that the empty clause $\perp$ is of type II. □

We now introduce the notion of another property, which all clauses in the set $\langle S_n \rangle$ must satisfy, and that will be very helpful in the proof of our claim.

Definition 7.6. A clause in this logic is *safe* if: when a variable x_i appears somewhere in the clause, then it appears in all the literals of the clause and always at the i-th bit position. ■

Note that all clauses in the set S_n are safe. Moreover, when resolving two safe clauses with an unifier σ, and since each variable x_i is only allowed to appear at the i-th bit position, there is no need for the substitution σ to rename variables in order to perform the resolution inference; the substitution needs to map, at most, variables to constant symbols. We say that a substitution is a *ground substitution* if it maps each variable either to itself or to a constant symbol. In the following, unless stated otherwise, we assume that all substitutions are ground substitution. In particular, we also have the following remark.

Remark. If a clause C is safe, then all its instances $C\sigma$, where σ is a ground substitution, are also safe.

Lemma 7.2. *If two safe clauses C_1 and C_2 are resolved together, the result is a safe clause.*

Proof. Suppose that C_1 and C_2 were resolved with a unifier σ on a pair of literals $l_1\sigma = \neg l_2\sigma = l$ to obtain a clause C. Recall that, since the clauses are safe, we can moreover assume that σ is an ground substitution.

So, if a variable x_i appears in C, it must also appear either in $C_1\sigma$ or $C_2\sigma$ and, therefore, also in the literal l. But again, it must therefore appear in all literals of both $C_1\sigma$ and $C_2\sigma$. And, since literals in C are a subset of those, it implies that C is also safe. □

Corollary 7.2. *All clauses in $\langle S_n \rangle$ are safe.*

Proof. The result easily follows from the previous lemma and the fact that all clauses in S_n are safe. □

Corollary 7.3. *Let C be a clause of type II in $\langle S_n \rangle$. If C has an inactive atom, then the clause is ground.*

Proof. Let l be the inactive atom in C and h its head. Suppose that there is a variable x_i in h. Since the clause is safe, this variable should also appear in l at the i-th bit position. But this is not possible since, because the terms in l and h do not overlap, there is no i-th bit position in l! It must therefore be the case that there are no variables in h, nor anywhere else in the clause. □

We will now proceed to define the work function as anticipated earlier in the sketch of the proof.

Definition 7.7. Given a term t we define the set $\hat{t} \subseteq \mathbb{B}$ as the set of ground instances of t, i.e.

$$\hat{t} = \begin{cases} \{0\} & t = 0, \\ \{1\} & t = 1, \\ \{0,1\} & t \text{ is a variable.} \end{cases}$$

Then, given a clause C of type II with head $q_i(t_{\underline{i}}, \ldots, t_n)$, we first define the work of every literal l, a subset of $\mathbb{B}^n$, as follows:

$$w(l) = \begin{cases} \mathbb{B}^n & l = \bot, \\ \mathbb{B}^{n-j} \times \hat{t}_{\underline{j}} \times \cdots \times \hat{t}_n & l = q_j(t_{\underline{j}}, \ldots, t_n), \\ \hat{t}_1 \times \cdots \times \hat{t}_n & l = p_j(t_1, \ldots, t_j) \text{ for an active } l, \\ \emptyset & \text{if } l \text{ is inactive.} \end{cases}$$

We also let $w^c(l) = \mathbb{B}^n \setminus w(l)$. So, the work of C is defined as

$$w(C) = w^c(l_1) \cap \cdots \cap w^c(l_m) \cap w(h) \tag{7.19}$$

For clauses of type I we let $w(C) = \emptyset$. Finally, the work of a set of clauses is the union of the work of each clause in the set. ■

Remark. If l is a literal in a clause of type II and σ is a substitution, then $w(l\sigma) \subseteq w(l)$.

The following definition and lemmas will be helpful to prove some important property of the work function, namely Corollary 7.4, which states that, for any substitution σ and a clause C in a refutation of S_n, $w(C\sigma) = w(C)$.

Definition 7.8. We say that a clause C of type II is complete if

$$w(l_1) \cup \cdots \cup w(l_m) = w(h) ,$$

and all the sets $w(l_1), \ldots, w(l_m)$ are pairwise disjoint.

In the following we will use

$$w(l_1) \uplus \cdots \uplus w(l_m) = w(h) ,$$

to denote this union in order to emphasise the fact that its operands are disjoint. ■

Remark. If a clause C is complete then its work is empty. From this, and the fact that all clauses in S_n are either of type I or complete clauses of type II, it follows that $w(S_n) = \emptyset$.

On the other hand, the work of the empty clause, and therefore by any refutation of S_n, is $w(\bot) = \mathbb{B}^n$.

Lemma 7.3. *Let C be a clause of type II in $\langle S_n \rangle$, and let σ be a substitution. If the clause C is complete then $C\sigma$ also is.*

Proof. Suppose that C is of the form (7.18). The fact that the collection of sets $w(l_1\sigma), \ldots, w(l_m\sigma)$ is disjoint follows easily since, by hypothesis, the collection of sets $w(l_1), \ldots, w(l_m)$ was disjoint and we are now replacing each $w(l_k)$ with a smaller set $w(l_k\sigma)$.

We now proceed to show that

$$w(l_1\sigma) \uplus \cdots \uplus w(l_m\sigma) \subseteq w(h\sigma) . \tag{7.20}$$

Let $\bar{b} = (b_1, \ldots, b_n)$ be an element in the left hand side of (7.20), it must then be the case that $\bar{b} \in w(l_k\sigma) \subseteq w(l_k)$ for some k. Since C is complete, then we must also have $\bar{b} \in w(h)$. Now suppose that $\bar{b} \in w(h)$ but $\bar{b} \notin w(h\sigma)$, it must be the case that σ maps some variable x_i to the wrong bit, i.e. $\sigma(x_i) = 1 - b_i$. There is however a contradiction since, from Corollary 7.2, the clause C is safe and the variable x_i should also appear in l_k where it would also be mapped to the wrong bit. Therefore $\bar{b} \in w(h\sigma)$, the right hand side of (7.20).

The proof for the converse, that an item in the right hand side must also be in the left hand, is analogous. □

Lemma 7.4. *Let C be a clause of type II in $\langle S_n \rangle$. If C is non-ground, then it is also complete.*

Proof. The proof is by induction on the length of the derivation of C. For the base case it is easy, and enough, to verify that all clauses of form (7.10) are complete.

Suppose that C is obtained from a pair of clauses of types I and II:

$$\begin{aligned} C_1\sigma &: l'_1 \to l_1 \\ C_2\sigma &: l_1 \wedge l_2 \wedge \cdots \wedge l_m \to h \\ C &: l'_1 \wedge l_2 \wedge \cdots \wedge l_m \to h \end{aligned}$$

Since both C and $C_2\sigma$ are safe, and because C is non-ground, there must be a variable in h which is also in l_1. From this both $C_1\sigma$ and $C_2\sigma$, as well as C_1 and C_2, are all non-ground. Moreover, since C_1 is not ground and of type I, its body should not be empty. Now, by inductive hypothesis, C_2 is complete and, by Lemma 7.3, $C_2\sigma$ also is. Therefore

$$w(l_1) \uplus \cdots \uplus w(l_m) = w(h) \ .$$

Also, since both C and $C_2\sigma$ are non-ground and from Corollary 7.3, they do not contain inactive atoms. In particular both l_1 and l'_1 should be active literals and, since the terms in l'_1 are just a prefix of those in l_1, $w(l_1) = w(l'_1)$.

Suppose that, otherwise, C is obtained by resolving two clauses of type II:

$$\begin{aligned} C_1\sigma &: l'_1 \wedge \cdots \wedge l'_{m'} \to l_1 \\ C_2\sigma &: l_1 \wedge l_2 \wedge \cdots \wedge l_m \to h \\ C &: l'_1 \wedge \cdots \wedge l'_{m'} \wedge l_2 \wedge \cdots \wedge l_m \to h \end{aligned}$$

As in the previous case, both C_1 and C_2 most be non-ground, and all its literals are active. Therefore, by induction,

$$w(l'_1) \uplus \cdots \uplus w(l'_{m'}) = w(l_1)$$
$$w(l_1) \uplus w(l_2) \uplus \cdots \uplus w(l_m) = w(h)$$

which are easily combined to obtain the desired result. □

Corollary 7.4. *Let C be any clause in $\langle S_n \rangle$ and let σ be a substitution. It then follows that $w(C\sigma) = w(C)$.*

Proof. The result is trivial for clauses of type I which have an empty work. Also, if C is ground, then $C\sigma = C$ and the result is again trivial.

For a non-ground clause C of type II we have, from Lemma 7.4, that C is complete and, from Lemma 7.3, that $C\sigma$ is also complete. From previous remarks it follows that both $w(C) = w(C\sigma) = \emptyset$. □

We are now ready to prove the core result needed in the proof of our claim, namely that the work function increments only one element at a time during each step in a refutation of S_n. The following definition serves to formalise this statement.

Definition 7.9. Let $\Gamma = C_1, \ldots, C_m$ be a sequence of clauses. The *work increment* of a clause C_k in the sequence is denoted by $\delta(C_k)$ and defined as the set $\delta(C_k) = w(\{C_1, \ldots, C_k\}) \setminus w(\{C_1, \ldots, C_{k-1}\})$. ■

Lemma 7.5. *Let Γ be a resolution derivation of a clause C from S_n. Then, for every n-bit string $\bar{b} \in w(\Gamma)$ there is a clause C_k in Γ with $\delta(C_k) = \{\bar{b}\}$.*

Proof. The proof is by induction on the length of Γ. When the length is 1, then the clause C is in S_n, $w(\Gamma) = \emptyset$ and the result is trivial.

Suppose that otherwise $\Gamma = \Gamma', C$, for a non-empty sequence Γ'. Take an element $\bar{b} \in w(\Gamma)$. If $\bar{b} \in w(\Gamma')$ then, by induction, there is a clause C_k in the sequence Γ' with $\delta(C_k) = \{\bar{b}\}$. In the following suppose that, otherwise, the element $\bar{b} \notin w(\Gamma')$ and, as a consequence $\bar{b} \in \delta(C) \subseteq w(C)$.

Now, C is not of type I or a hypothesis in S_n since, for those clauses, we know that $w(C) = \emptyset$ while $\bar{b} \in w(C)$. The only remaining cases are when C is a clause of type II obtained by resolving two other clauses C_1 and C_2 in Γ' with a unifier σ.

Suppose that C is the result of resolving two clauses of types I and II.

$$\begin{aligned} C_1\sigma &: [l'_1] \to l_1 \\ C_2\sigma &: l_1 \land l_2 \land \cdots \land l_m \to h \\ C &: [l'_1] \land l_2 \land \cdots \land l_m \to h \end{aligned}$$

Since $\bar{b} \in w(C)$ we get that, in particular, $\bar{b} \in w^c(l_2) \cap \cdots \cap w^c(l_m) \cap w(h)$. However, since $\bar{b} \notin \Gamma'$ also $\bar{b} \notin w(C_2)$ and, by Corollary 7.4, $\bar{b} \notin w(C_2\sigma)$. So it must therefore be the case that $\bar{b} \notin w^c(l_1)$ or, equivalently, $\bar{b} \in w(l_1)$. In particular we have shown that $\delta(C) \subseteq w(l_1)$. Since by hypothesis $\delta(C)$ is not empty, and since $w(l_1)$ has at most one element (the predicate symbol in l_1 is either p_j or q_n), it must be the case that $\delta(C) = w(l_1) = \{\bar{b}\}$. So C is the clause for which the increment was exactly $\{\bar{b}\}$.

Suppose that otherwise C is the result of resolving two clauses of types II.

$$\begin{aligned} C_1\sigma &: l'_1 \land \cdots \land l'_{m'} \to l_1 \\ C_2\sigma &: l_1 \land l_2 \land \cdots \land l_m \to h \\ C &: l'_1 \land \cdots \land l'_{m'} \land l_2 \land \cdots \land l_m \to h \end{aligned}$$

Exactly as in the previous case we get, since $\bar{b} \in w(C)$, that $\bar{b} \in w(l_1)$. Now, however, from $\bar{b} \in w(C)$ we also get $\bar{b} \in w^c(l'_1) \cap \cdots \cap w^c(l'_{m'})$ and, from this, that $\bar{b} \in w(C_1\sigma) = w(C_1) \subseteq w(\Gamma')$, contradicting one of our hypothesis. □

The proof of our main Theorem 7.4 now follows as a simple corollary of the previous lemma.

Corollary 7.5. *A resolution refutation of S_n has a length of, at least, 2^n.*

Proof. Let Γ be a refutation of S_n. It contains the empty clause and, therefore, its work is $w(\Gamma) = \mathbb{B}^n$. Now, from Lemma 7.5, for each element $\bar{b} \in \mathbb{B}^n$ there is a clause C in Γ for which $\delta(C) = \{\bar{b}\}$. But this implies that there must be at least 2^n clauses in Γ, one for each such $\bar{b}$. □

7.2.3 Results and conclusions

This section includes several interesting theoretical results, in the context of reasoning within effectively propositional logic. First, we proved that resolution can be, in principle, exponentially more efficient when applied directly at this level rather than the lower propositional setting. In particular we gave a family of unsatisfiable formulae whose refutation proofs using first-order resolution are exponentially shorter than any propositional resolution proof.

Another pair of results, in the context of our proposed generalisation inference rule, state sufficient conditions in order to combine sort inference mechanisms with the inference rule itself. Finally, we also proved that, when resolution is combined with generalisation, the resulting inference system can produce, as well, exponentially shorter refutations for unsatisfiable formulae. This set of results provide, we believe, compelling evidence of the advantages that directly reasoning at the effectively propositional can provide.

7.3 Concluding remarks

In this chapter we have explored several reasoning techniques that are applicable to effectively propositional formulae. First, we briefly described some original ideas which provide some improvements in the context of grounding-based methods. These are presented in support of our claim that, in order to develop better propositional encodings, a generic translation from effectively propositional to propositional logic is also a suitable approach.

Then we also proposed, as one of the main contributions of this thesis, a new inference rule which we call *generalisation* and that is able to exploit sort information while reasoning directly within effectively propositional logic. We show how some existing sort inference techniques, such as those developed for grounding-based approaches, can be directly applied in this context; while some others, particularly based on linking restrictions, cannot be imported as easily.

We also show that resolution is exponentially more efficient at the effectively propositional, rather than just propositional, level; and, moreover, the combination of resolution with generalisation yields a further exponential improvement. This serves to suggest that the use of an effectively propositional encoding is useful not only to obtain a more compact representation of problems, but also to solve them more efficiently.

Incidentally, the proofs of these two exponential gaps provide us with a couple of benchmark families that might be interesting to test with existing systems. We have shown that, when reasoning with a particular inference system, some unsatisfiable problems have rather short refutations, but are existing implementations of theorem provers able to find such short proofs? Or, which heuristics can we use in order to find these shorter proofs with more probability?

Further directions for future work include the research on techniques to efficiently implement and integrate the proposed generalisation inference rule with other systems which already make use of resolution, such as iProver or perhaps even Vampire. Alternatively, it might also prove fruitful to investigate on possible extensions of this inference rules in order to make use of complete linking information which can perhaps better describe the underlying structure of the problem being solved.

Chapter 8

Discussion

We finalise this thesis with a summary of the research work carried out, and an appraisal on the evidence that has been gathered to support our main hypothesis. First an overview of the thesis is given, evaluating the degree to which the stated research aims and objectives were achieved. Then, the major thesis contributions are brought out, highlighting the extent of the research community for which this work will be most interesting. Finally, some directions for future work are also pointed out.

8.1 Thesis overview

The main objective of this thesis was to explore whether the language of effectively propositional logic is a suitable alternative to succinctly and naturally encode problems from different application domains. Moreover, we argue that reasoning and solving problems in this language offers the possibility to more easily exploit the structure that usually arises in problems derived from applications.

Substantial evidence is presented in order to support this claim. In particular, two case studies are developed where applications from model checking and planning are encoded using the proposed language. The resulting logical encodings are often exponentially shorter than standard encodings in propositional logic that have been commonly used and accepted by the research community. Finally, we also analyse and give some evidence on how reasoning at the effectively propositional, rather than just propositional, level is also potentially more efficient.

8.2 Impact of major contributions

In this section we highlight four of the major contributions of this thesis. The first of such contributions is the development of a method to randomly generate non-clausal propositional formulae which are difficult for propositional reasoners to solve. The second comprises a couple of case studies on how to encode problems from different application domains by using the language of effectively propositional logic. The third is, as a by product of the previous contribution and other ideas explored in this thesis, a set of effectively propositional benchmarks which have been made available to the research community. Finally, the last contribution deals with the formal study and comparison of inference rules and reasoning techniques in the context of effectively propositional logic.

8.2.1 Hard non-clausal propositional problems

In Chapter 3, particularly Section 3.1, we discussed a method to randomly generate propositional problems. These problems have, as opposed to other popular methods for generating random formulae, a non-trivial structure which is built on top of an alternating tree of conjunctions and disjunctions. Therefore, by nature, these problems are originally non-clausal and suitable for testing systems able to handle problems which have been described using this more general syntax. A tool to generate these problems, written in C++, has also been made available online for the research community to use.[1]

The proposed method should be of great interest to researchers —such as Thiffault et al. (2004), Giunchiglia and Sebastiani (2000) and Stachniak (2002)— implementing solvers and developing ideas in order to directly read and process non-clausal formulae. Muhammad and Stuckey (2006) have, in fact, already used our proposed method to evaluate the performance of their stochastic non-clausal solver. Our work was also found relevant in the constraint programming community, who have listed our method as an approach to randomly generate formulae in a handbook of their research area (Gomes and Walsh, 2006). There is also interest from people in theoretical computer science; such as Santillán Rodríguez (2007) who computed tighter bounds for the phase transition region of our random model, where the most difficult problems are expected to be found.

[1] http://www.cs.man.ac.uk/~navarroj/tools/

8.2.2 Effectively propositional encodings

Another of our main contributions, developed in Chapters 5 and 6, are two case studies of how to encode applications, respectively from the model checking and planning domains, as effectively propositional formulae. An empirical evaluation of how this technique compares to other alternative approaches is also carried out. The results obtained are encouraging and show that, although there is still a lot of room for improvement, the proposed approach is competitive with other existing procedures.

As a valuable product of our research we also devised the *finite domain predicate logic* which, adding syntactic sugar on top of the effectively propositional language, allows to even more easily and naturally describe constructs often required when modelling application domains. This goes in line with the idea of encoding individual applications into a feature rich high level language, and then design very efficient and optimised translations into lower level languages such as effectively propositional or even propositional logic.

Researchers from many diverse areas, within or even outside computer science, might find this as a useful technique to quickly prototype and experiment with logic-based reductions of the problems that they typically encounter within their respective application domains. On the other hand, researchers in automated reasoning can concentrate their efforts on designing generic translation tools and developing highly optimised provers without worrying too much about the particular details of each individual application.

8.2.3 Benchmarks for effectively propositional provers

As a side effect of our work showcasing encodings of application problems into the language of effectively propositional logic, we had to actually generate many of those problems. And the research community, particularly people such as Baumgartner et al. (2005) and Ganzinger and Korovin (2003) which are actively working on the development of decision procedures for this logic, were very fond of having access to these benchmarks.

We have made available a tool, SMV2TPTP written in C++, to translate LTL properties and a subset of the SMV model checking language into effectively propositional formulae using the procedure described in Chapter 5. A number of simple Perl scripts that generate particular classes of formulae, such as those in Sections 6.3, 7.2.1 and 7.2.2, were also made available online for the research community to use.[2]

A number of premade benchmarks, created by selecting adequate parameters on these tools to obtain problems of increasing difficulty suitable for testing this and next generation provers, were also packed for more accessible consumption by system developers. Moreover, in order to make them even more widespread available for the automated reasoning community, these generated benchmarks will be also distributed as part of the TPTP v3.4.0 in the HWV domain of the problem library for theorem provers (Sutcliffe and Suttner, 2007).

8.2.4 Effectively propositional reasoning techniques

Finally, another contribution of this thesis is a formal analysis and comparison among different reasoning techniques, presented in Section 7.2, that are directly applied among effectively propositional clauses. We show, for example, how resolution can be exponentially more efficient on finding refutations in effectively propositional logic compared to plain propositional logic.

We also propose a new inference rule, which we call *generalisation*, that is a suitable complement to resolution and other reasoning strategies. Essentially, the inference of generalisation allows one to derive a general property after it has been individually proved for all the (finite amount of) constant symbols in an appropriate sort. In particular, we show that standard sort inference techniques can be used to compute these appropriate sorts and, moreover, that the resulting inference rule is compatible with resolution. In fact, we also show that generalisation and resolution combined can also be exponentially more efficient than resolution alone.

Although we do not pursue any research on an efficient implementation of the generalisation inference rule —since this lies beyond the intended scope of the thesis— we do think that the theoretical results obtained are encouraging and of great value for the automated reasoning community, who can now incorporate these ideas in the design of new systems. It is also worth noting that, although we only explored the use of these inference rules in the context of effectively propositional logic, they can be certainly applied and extended to more general settings such as full first-order logic.

[2]http://www.cs.man.ac.uk/~navarroj/tools/

8.3 Future work

Research on effectively propositional logic is, indeed, a rather new venture. This thesis makes, by providing the required benchmarks and some early theoretical results, an effort to encourage the community and accelerate the development of new techniques in this area. There are, therefore, many research directions open to explore and ideas to investigate.

To begin with, in the propositional case, more studies are needed on the impact that clausal form translations have on the difficulty of solving a problem. Surprisingly, very little seems to have been done in this respect. While it is widely known that satisfiability solvers are very sensitive to little variations in the problem encoding, it is not yet clear what should one aim for in order to design a *good* encoding. Section 3.2 tries to shed some light on these issues but, definitively, much more work is needed.

More research on how to extend satisfiability solvers to deal with arbitrary formulae or, at least, with extended kinds of constraints also would be very valuable. In particular designing methods to efficiently deal with predicates that have a *function-like* behaviour, which are very common in translations from applications in effectively propositional formulae, should be of great use to improve on current technology. In general, finding better reductions of effectively propositional into plain propositional logic looks like a fruitful area for future work.

A second direction for research in the future involves the reduction of even more applications into the language of effectively propositional logic. To start, in the field of model checking one can consider reductions of temporal logics more general than LTL, and larger fragments of SMV that include many more of the available features. On planning problems there is also a great room for improvement, translations are needed which are able to handle more than the most basic STRIPS syntax.

Applications where the use of satisfiability solvers has already been found useful are other natural candidates to benefit from these ideas, as well as any kind of problem which can be reduced to some form of reasoning or combinatorial search. Interaction with the also emerging community of *satisfiability modulo theories* (see e.g. Tinelli, 2002) is also perhaps necessary in order to support more complex features —such as theories of arrays, bit vectors, integers and reals— which are often needed while modelling many application domains.

Finally, another venue for future research work is to further the development of reasoning tools for effectively propositional formulae. First, more theory is needed about deduction and properties of inference rules. An important question, left open in this thesis, is how to combine the generalisation and resolution inference rules in order to devise a decision procedure for effectively propositional logic, a semi-decision procedure for first-order logic, or perhaps even both.

Then, research on how to efficiently implement these decision procedures, or to optimise existing ones such as the model evolution or instantiation calculi, would also help to improve the state of the art. Moreover, designing translators from feature rich languages into lower level formalisms will facilitate to people from other disciplines to more easily and quickly incorporate reasoning services into their applications, thus also widening the reach of our technology.

Bibliography

Dimitris Achlioptas and Yuval Peres. The threshold for random k-SAT is $2^k(\ln 2 - O(k))$. In *STOC'03: Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, pages 223–231, San Diego, CA, USA, 2003. ACM Press.

Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult instances of Boolean satisfiability in the presence of symmetry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22:1117–1137, 2003.

Edgar Altamirano and Gonzalo Escalada-Imaz. An efficient proof method for non-clausal reasoning. In *ISMIS'00: Proceedings of the 12th International Symposium on Foundations of Intelligent Systems*, pages 534–542. Springer, 2000. ISBN 3-540-41094-5.

Alessandro Armando and Luca Compagna. Automatic SAT-compilation of protocol insecurity problems via reduction to planning. In *FORTE'02: Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, pages 210–225. Springer, 2002. ISBN 3-540-00141-7.

Fahiem Bacchus. Exploring the computational tradeoff of more reasoning and less searching. In *SAT'02: Proceedings of the 5th International Symposium on Theory and Applications of Satisfiability Testing*, pages 7–16, Cincinnati, OH, USA, 2002.

Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT'03: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, number

2919 in Lecture Notes in Computer Science, pages 341–355. Springer, January 2003.

Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 26, pages 19–99. Elsevier, 2001.

Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *CAV'04: Proceedings of the 16th International Conference on Computer Aided Verification*, number 3114 in Lecture Notes in Computer Science, pages 515–518, Boston, MA, USA, July 2004. Springer.

W. Barthel, A. K. Hartmann, Michele Leone, F. Ricci-Tersenghi, M. Weigt, and Riccardo Zecchina. Hiding solutions in random satisfiability problems: A statistical mechanics approach. *Physical Review Letters*, 88, 2002.

Peter Baumgartner. FDPLL: A first-order Davis-Putnam-Logeman-Loveland procedure. In David McAllester, editor, *CADE-17: Proceedings of the 17th International Conference on Automated Deduction*, number 1831 in Lecture Notes in Artificial Intelligence, pages 200–219. Springer, 2000.

Peter Baumgartner and Cesare Tinelli. The model evolution calculus. In Franz Baader, editor, *CADE-19: Proceedings of the 19th International Conference on Automated Deduction*, number 2741 in Lecture Notes in Artificial Intelligence, pages 350–364. Springer, 2003.

Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Implementing the model evolution calculus. *Special Issue of the International Journal of Artificial Intelligence Tools (IJAIT)*, 2005.

Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real world SAT instances. In *AAAI'97: Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 203–208. AAAI Press, 1997.

Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research (JAIR)*, 22:319–351, December 2004.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS'99: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, number 1579 in Lecture Notes in Computer Science. Springer, 1999.

Armin Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan. Linear encodings of bounded LTL model checking. *Journal of Logical Methods in Computer Science*, 2(5:5), 2006.

Jean-Paul Billon. The disconnection method. A confluent integration of unification in the analytic framework. In *TABLEAUX'06: Proceedings of the 5th International TABLEAUX Workshop*, number 1071 in Lecture Notes in Artificial Intelligence, pages 110–126, Terrasini, Italy, May 1996. Springer.

Ronen I. Brafman. A simplifier for propositional formulas with many binary clauses. *IEEE Transactions on Systems, Man, and Cybernetics. Part B, Cybernetics*, 34:52–59, 2004.

Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components. In *ICSE'03: Proceedings of the 2003 ACM/IEEE International Conference on Software Engineering*, pages 385–395, Portland, OR, USA, May 2003.

Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *CAV'02: Proceedings of the 14th International Conference on Computer Aided Verification*, number 2404 in Lecture Notes in Computer Science, pages 359–364, London, U.K., 2002. Springer.

Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style model finding. In *MODEL'03: Proceedings of the Workshop on Model Computation*, 2003.

Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001. ISSN 0925-9856.

Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC'71: Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971.

Stephen A. Cook and David G. Mitchell. Finding hard instances of the satisfiability problem: A survey. In *Discrete Mathematics and Theoretical Computer Science*, volume 5 of *DIMACS*. American Mathematical Society, 1997.

F. Copty, L. Fix, R. Fraer, Enrico Giunchiglia, G. Kamhi, Armando Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *CAV'01: Proceedings of the 13th International Conference on Computer Aided Verification*, number 2102 in Lecture Notes in Computer Science, pages 436–453. Springer, 2001.

James M. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic (extended abstract). In *Proceedings of the AAAI Workshop on Tractable Reasoning, held at the Tenth National Conference on Artificial Intelligence (AAAI'92)*, San Jose, CA, USA, 1992.

James M. Crawford and Larry D. Auton. Experimental results on the crossover point in satisfiability problems. In *AAAI'93: Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 21–27, 1993.

James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI'94: Proceedings of the Twelfth National Conference on Artificial Intelligence (vol. 2)*, pages 1092–1097, Seattle, WA, USA, July 1994. AAAI Press. ISBN 0-262-61102-3.

James M. Crawford, Matthew L. Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *KR'96: Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 149–159, Boston, MA, USA, 1996.

N. Creignou and H. Daude. Random generalized satisfiability problems. In *SAT'02: Proceedings of the 5th International Symposium on Theory and Applications of Satisfiability Testing*, Cincinnati, OH, USA, 2002.

Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960. ISSN 0004-5411.

Martin Davis, George Logemann, and Dondald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

Jay L. Devore. *Probability and Statistics for Engineering and the Sciences.* Brooks/Cole Publishing Company, fifth edition, 1999.

Lyndon Drake, Alan Frisch, Inês Lynce, João P. Marques Silva, and Toby Walsh. Comparing SAT preprocessing techniques. In Frank van Harmelen, editor, *ECAI'02: Proceedings of the 15th European Conference on Artificial Intelligence*, pages 166–170, 2002.

Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *IJCAI'01: Proceedings of the 17th International Joint Conference on Artificial Intelligence*, 2001.

Olivier Dubois, Yacine Boufkhad, and Jacques Mandler. Typical random 3-SAT formulae and the satisfiability threshold. In *SODA'00: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 126–127, San Francisco, CA, USA, 2000. Society for Industrial and Applied Mathematics.

Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT'05: Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, number 3569 in Lecture Notes in Computer Science. Springer, 2005.

Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In *BMC'03: Proceedings of the First International Workshop on Bounded Model Checking*, volume 89 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2003.

Niklas Eén and Niklas Sörensson. MiniSat: A SAT solver with conflict-clause minimization. Poster presented at the SAT 2005 Competition, 2005.

Christian Fermüller, Alexander Leitch, Tanel Tammet, and Nikolai Zamov. *Resolution Methods for the Decision Problem.* Number 679 in Lecture Notes in Artificial Intelligence. Springer, Berlin, Germany, 1993.

Paolo Ferraris and Enrico Giunchiglia. Planning as satisfiability in nondeterministic domains. In *AAAI'00/IAAI'00: Proceedings of the Seventeenth National*

Conference on Artificial Intelligence and the Twelfth Conference on Innovative Applications of Artificial Intelligence, pages 748–753. AAAI Press / MIT Press, 2000. ISBN 0-262-51112-6.

R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, 1996. ISBN 0-387-94593-8.

Maria Fox and Derek Long. The third international planning competition: Temporal and metric planning. In *AIPS'02: Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems*, Toulouse, France, April 2002.

Jon William Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1995.

E. Friedgut. Sharp thresholds of graph properties and the k-SAT problem. *Journal of the American Mathematical Society*, 12(4):1017–1054, 1999.

Alexander Fuchs. Darwin: A theorem prover for the model evolution calculus. Master's thesis, University of Koblenz-Landau, 2004.

Alex Fukunaga. Efficient implementations of SAT local search. In *SAT'04: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, number 3542 in Lecture Notes in Computer Science, Vancouver, Canada, May 2004. Springer.

Harald Ganzinger and Konstantin Korovin. New directions in instantiation-based theorem proving. In *LICS'03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, page 55, Ottawa, Canada, June 2003. IEEE Computer Society. ISBN 0-7695-1884-2.

Ian P. Gent, Ewan Macintyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. Random constraint satisfaction: Flaws and structure. *Constraints*, 6 (4):345–372, 2001.

M. Ghallab, A. Howe, C. Knoblock, Drew V. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL: The planning domain definition language,

1998. Planning Committee of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS'98).

Enrico Giunchiglia and Roberto Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In *AI*IA'99: Proceedings of the 6th Congress of the Italian Association for Artificial Intelligence on Advances in Artificial Intelligence*, pages 84–94. Springer, 2000. ISBN 3-540-67350-4.

Enrico Giunchiglia, Alessandro Massarotto, and Roberto Sebastiani. Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In *AAAI'98/IAAI'98: Proceedings of the Fifteenth National Conference on Artificial Intelligence and the Tenth Conference on Innovative Applications of Artificial Intelligence*, pages 948–953, Madison, WI, USA, 1998. AAAI Press. ISBN 0-262-51098-7.

Evgueni Goldberg and Yakov Novikov. BerkMin: A fast and robust SAT-solver. In *DATE'02: Proceedings of the Conference on Design, Automation and Test in Europe*, page 142, Washington, DC, USA, 2002. IEEE Computer Society.

Evgueni Goldberg, Mukul R. Prasad, and Robert K. Brayton. Using SAT for combinational equivalence checking. In *DATE'01: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 114–121, Munich, Germany, 2001.

Carla P. Gomes and Toby Walsh. Randomness and structure. In F. Rossi, P. van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, chapter 18. Elsevier, 2006.

Carla P. Gomes, Bart Selman, and Nuno Crato. Heavy-tailed distributions in combinatorial search. In Gert Smolka, editor, *CP'97: Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, number 1330 in Lecture Notes in Computer Science, pages 121–135. Springer, 1997.

C. Green. Application of theorem proving to problem solving. In *IJCAI'69: Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 219–239, Washington, DC, USA, 1969.

Jun Gu. Efficient local search for very large-scale satisfiability problems. *SIGART Bulletin*, 3(1):8–12, 1992. ISSN 0163-5719.

A. R. Haas. The case for domain specific frame axioms. In F. M. Brown, editor, *Proceedings of the 1987 Workshop on The Frame Problem in Artificial Intelligence*, pages 343–348, Lawrence, KS, USA, 1987. Morgan Kaufmann.

R. Hähnle. Tableaux and related methods. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 3, pages 100–178. Elsevier, 2001.

Marijn Heule and Hans van Maaren. Aligning CNF- and equivalence-reasoning. In *SAT'04: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, number 3542 in Lecture Notes in Computer Science, pages 145–156, Vancouver, Canada, May 2004. Springer.

Marijn Heule, Joris van Zwieten, Mark Dufour, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient lookahead SAT solver. In *SAT'04: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, number 3542 in Lecture Notes in Computer Science, pages 345–359, Vancouver, Canada, May 2004. Springer.

J. Hooker, G. Rago, V. Chandru, and A. Shrivastava. Partial instantiation methods for inference in first order logic. *Journal of Automated Reasoning*, 28(3): 371–396, 2002.

Holger H. Hoos. An adaptive noise mechanism for WalkSAT. In *AAAI'02: Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 655–660. AAAI Press, 2002.

Holger H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *AAAI'99/IAAI'99: Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Conference on Innovative Applications of Artificial Intelligence*, pages 661–666, Orlando, FL, USA, 1999. AAAI Press. ISBN 0-262-51106-1.

Holger H. Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. In Ian P. Gent, Hans van Maaren, and Toby Walsh, editors, *SAT'00: Proceedings of the 3rd International Symposium on Theory and Applications of Satisfiability Testing*. IOS Press, 2000.

Markus Jehle, Jan Johannsen, Martin Lange, and Nicolas Rachinsky. Bounded model checking for all regular properties. In *BMC'05: Proceedings of the Third International Workshop on Bounded Model Checking*, volume 144 of *Electronic Notes in Theoretical Computer Science*, pages 3–18, Edinburgh, Scotland, 2005. Elsevier.

R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1(1–4):167–187, 1990.

William H. Joiner, Jr. Resolution strategies as decision procedures. *Journal of the ACM*, 23(3):398–417, 1976. ISSN 0004-5411.

Alexis C. Kaporis, Lefteris M. Kirousis, and Efthimios Lalas. Selecting complementary pairs of literals. In *Proceedings of the Workshop on Typical Case Complexity and Phase Transitions, held at the 18th Annual Symposium on Logic in Computer Science (LICS'03)*, Ottawa, Canada, June 2003.

Henry Kautz. Deconstructing planning as satisfiability. In *AAAI'06: Proceedings of the Twenty-first National Conference on Artificial Intelligence*, Boston, MA, USA, July 2006. AAAI Press.

Henry Kautz and Bart Selman. Planning as satisfiability. In *ECAI'92: Proceedings of the 10th European Conference on Artificial intelligence*, pages 359–363, Vienna, Austria, 1992. John Wiley & Sons, Inc.

Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *AAAI'96: Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996.

Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In *KR'96: Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, Boston, MA, USA, 1996.

Henry Kautz, Bart Selman, and Joerg Hoffmann. SatPlan: Planning as satisfiability. In *IPC'06: Abstracts of the 5th International Planning Competition*, 2006.

Vlado Keselj and Nick Cercone. A formal approach to subgrammar extraction for NLP. *Journal of Mathematical and Computer Modelling*, 45:394–403, February 2007.

Konstantin Korovin. Implementing an instantiation-based theorem prover for first-order logic. In C. Benzmueller, B. Fischer, and Geoff Sutcliffe, editors, *IWIL'06: Proceedings of the 6th International Workshop on the Implementation of Logics, held at the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'06)*, volume 212 of *CEUR Workshop Proceedings*, Phnom Penh, Cambodia, 2006.
See also: http://www.cs.man.ac.uk/~korovink/iprover/.

Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992. ISSN 0738-4602.

T. Latvala, Armin Biere, K. Heljanko, and T. Junttila. Simple bounded LTL model checking. In *FMCAD'04: Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design*, number 3312 in Lecture Notes in Computer Science, pages 186–200, Austin, TX, USA, 2004. Springer.

Daniel Le Berre and Laurent Simon. Fifty-five solvers in Vancouver: The SAT 2004 competition. In *SAT'04: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, number 3542 in Lecture Notes in Computer Science, Vancouver, Canada, May 2004. Springer.

Daniel Le Berre and Laurent Simon. The SAT 2005 competition website, June 2005. URL http://www.satcompetition.org/2005/.

S.-J. Lee and David A. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9(1):25–42, 1992.

Reinhold Letz and Gernot Stenz. DCTP: A disconnection calculus theorem prover. In *IJCAR'01: Proceedings of the First International Joint Conference on Automated Reasoning*, number 2083 in Lecture Notes in Computer Science, pages 381–385, Siena, Italy, June 2001. Springer.

Chu Min Li. Equivalent literal propagation in the DLL procedure. *Discrete Applied Mathematics*, 130(2):251–276, 2003. ISSN 0166-218X.

Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI'97: Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 366–371, Japan, 1997. Morgan Kaufmann. ISBN 1-55860-480-4.

Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157:115–137, August 2004.

S. Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269, December 1965.

Michael G. Luby, Michael Mitzenmacher, and M. Amin Shokrollahi. Analysis of random processes via and-or tree evaluation. In *SODA'98: Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 364–373, San Francisco, CA, USA, 1998. Society for Industrial and Applied Mathematics. ISBN 0-89871-410-9.

Inês Lynce and João P. Marques Silva. The puzzling role of simplification in propositional satisfiability. In *Proceedings of the EPIA Workshop on Constraint Satisfaction and Operational Research Techniques for Problem Solving, held at the Portuguese Conference on Artificial Intelligence (EPIA'01)*, pages 73–86, Porto, Portugal, 2001.

Inês Lynce and João P. Marques Silva. Probing-based preprocessing techniques for propositional satisfiability. In *ICTAI'03: Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, page 105, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2038-3.

Inês Lynce and João P. Marques Silva. SAT in bioinformatics: Making the case with halotype inference. In *SAT'06: Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, Seattle, WA, USA, 2006.

Inês Lynce and João P. Marques Silva. Breaking symmetries in SAT matrix models. In *SAT'07: Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, Lisbon, Portugal, 2007.

João P. Marques Silva. Improvements to the implementation of interpolant-based model checking. In Dominique Borrione and Wolfgang J. Paul, editors, *CHARME'05: Proceedings of the 13th Conference on Correct Hardware Design and Verification Methods*, number 3725 in Lecture Notes in Computer Science, pages 367–370, Saarbrücken, Germany, 2005. Springer.

João P. Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *EPIA'99: Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, pages 62–74, London, U.K., 1999. Springer. ISBN 3-540-66548-X.

João P. Marques Silva and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In *DAC'00: Proceedings of the 37th Conference on Design Automation*, pages 675–680, Los Angeles, CA, USA, June 2000. ACM Press. ISBN 1-58113-187-9.

João P. Marques Silva and Karem A. Sakallah. Grasp: A new search algorithm for satisfiability. In *ICCAD'96: Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, San Jose, CA, USA, 1996a. IEEE Computer Society. ISBN 0-8186-7597-7.

João P. Marques Silva and Karem A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *ICTAI'96: Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*, November 1996b.

Alain J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In H. Fuchs, editor, *VLSI'85: Proceedings of the 1985 Chapel Hill Conference on Very Large Scale Integration*, 1985.

Fabio Massacci and Laura Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24(1):165–203, 2000.

David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *AAAI'97: Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 321–326. AAAI Press, 1997.

William McCune. A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical Report ANL/MCS-TM-194, Argonne National Laboratory, Argonne, IL, USA, May 1994a.

William McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, January 1994b.

William McCune and Larry Wos. Otter: The CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, April 1997. ISSN 0168-7433.

K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

Elliott Mendelson. *Introduction to Mathematical Logic.* Wadsworth, Belmont, CA, USA, third edition, 1987.

S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large-scale constraint-satisfaction and scheduling problems using a heuristic repair method. In *AAAI'90: Proceedings of the National Conference on Artificial Intelligence*, pages 17–24, 1990.

David G. Mitchell. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science*, 85:112–133, February 2005. Logic in Computer Science Column.

David G. Mitchell and Hector Levesque. Some pitfalls for experiments with random SAT. *Artificial Intelligence*, 81:111–125, 1996.

David G. Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of SAT problems. In *AAAI'92: Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, San Jose, CA, USA, July 1992. AAAI Press / MIT Press.

Rémi Monasson and Riccardo Zecchina. Statistical mechanics of the random k-SAT model. *Physical Review E*, 56:1357–1370, 1997.

Matthew H. Moskewicz, Conor F. Madigan, Yuting Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC'01: Proceedings of the 38th Conference on Design Automation*, Las Vegas, NV, USA, June 2001.

Rafiq Muhammad and Peter J. Stuckey. A stochastic non-CNF SAT solver. In Qiang Yang and Geoffrey I. Webb, editors, *PRICAI'06: Proceedings of the 9th Pacific Rim International Conference on Artificial Intelligence*, number 4099 in Lecture Notes in Computer Science, pages 120–129, Guilin, China, August 2006. Springer. ISBN 3-540-36667-9.

Juan Antonio Navarro Pérez. Generation of hard non-clausal random satisfiability problems. In *ARW'05: Proceedings of the Workshop on Automated Reasoning*, Edinburgh, Scotland, July 2005.

Juan Antonio Navarro Pérez. Translations to propositional satisfiability. In *ARW'06: Proceedings of the Workshop on Automated Reasoning*, Bristol, U.K., April 2006.

Juan Antonio Navarro Pérez. Encodings of bounded LTL model checking in effectively propositional logic. In *ARW'07: Proceedings of the Workshop on Automated Reasoning*, London, U.K., April 2007.

Juan Antonio Navarro Pérez and Andrei Voronkov. Generation of hard non-clausal random satisfiability problems. In *AAAI'05/IAAI'05: Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Conference on Innovative Applications of Artificial Intelligence*, Pittsburgh, PA, USA, July 2005. AAAI Press.

Juan Antonio Navarro Pérez and Andrei Voronkov. Encodings of bounded LTL model checking in effectively propositional logic. In *CADE-21: Proceedings of the 21st International Conference on Automated Deduction*, number 4603 in Lecture Notes in Artificial Intelligence, pages 346–361, Bremen, Germany, 2007a. Springer.

Juan Antonio Navarro Pérez and Andrei Voronkov. Planning with effectively propositional logic, 2007b. To appear at the Harald Ganzinger memorial volume.

Robert Nieuwenhuis and Albert Oliveras. Decision procedures for SAT, SAT modulo theories and beyond. The BarcelogicTools. (invited paper). In *LPAR'05: Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Montego Bay, Jamaica, December 2005.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In Franz Baader and Andrei Voronkov, editors, *LPAR'04: Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, number 3452 in Lecture Notes in Computer Science, pages 36–50, Montevideo, Uruguay, 2004. Springer.

Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Saïs. Recovering and exploiting structural knowledge from CNF formulas. In P. Van Hentenryck, editor, *CP'02: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, number 2470 in Lecture Notes in Computer Science, pages 185–199, Ithaca, NY, USA, September 2002. Springer.

Francis Jeffry Pelletier, Geoff Sutcliffe, and Christian B. Suttner. The development of CASC. *AI Communications*, 15(2):79–90, 2002. ISSN 0921-7126.

Slawomir Pilarski and Gracia Hu. SAT with partial clauses and back-leaps. In *DAC'02: Proceedings of the 39th Conference on Design Automation*, pages 743–746, New Orleans, LA, USA, June 2002. ACM Press. ISBN 1-58113-461-4.

David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986. ISSN 0747-7171.

David A. Plaisted and Yunshan Zhu. Ordered semantic hyper-linking. *Journal of Automated Reasoning*, 25(3):167–217, 2000.

Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 7:156–173, April 2005.

W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, October 1952.

I. V. Ramakrishnan, R. Sekar, and Andrei Voronkov. Term indexing. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1964. Elsevier, 2001.

Alexandre Riazanov and Andrei Voronkov. The design and implementation of Vampire. *AI Communications*, 15(2):91–110, 2002. ISSN 0921-7126.

John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965. ISSN 0004-5411.

Lawrence Ryan. Efficient algorithms for clause learning SAT solvers. Master's thesis, Simon Fraser University, February 2004.

Ashish Sabharwal. SymChaff: A structure-aware satisfiability solver. In *AAAI'05/IAAI'05: Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Conference on Innovative Applications of Artificial Intelligence*, Pittsburgh, PA, USA, July 2005. AAAI Press.

Ashish Sabharwal, Paul Beame, and Henry Kautz. Using problem structure for efficient clause learning. In *SAT'03: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, number 2919 in Lecture Notes In Computer Science, pages 242–256. Springer, January 2003.

Rocío Santillán Rodríguez. New upper bounds for the SAT/UNSAT threshold for shapes. Master's thesis, Vienna University of Technology, June 2007.

JMP, Version 7. SAS Institute Inc., Cary, NC, USA, 2007.

L. K. Schubert. Monotonic solution of the frame problem in the situation calculus: An efficient method for worlds with fully specified actions. In H. Kyburg, R. Loui, and G. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer Academic Publishers, Dordrecht, Netherlands, 1990.

Stephan Schulz. A comparison of different techniques for grounding near-propositional CNF formulae. In Susan Haller and Gene Simmons, editors, *FLAIRS'02: Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference*, pages 72–76. AAAI Press, 2002.

Bart Selman, Hector Levesque, and David G. Mitchell. A new method for solving hard satisfiability problems. In *AAAI'92: Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–460, San Jose, CA, USA, July 1992. AAAI Press / MIT Press.

Bart Selman, Henry Kautz, and Bram Cohen. Noise strategies for improving local search. In *AAAI'94: Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA, USA, July 1994. AAAI Press.

Maria Sorea. Bounded model checking for timed automata. In *MTCS'02: Proceedings of the Third Workshop on Models for Time-Critical Systems, heldt at CONCUR 2002*, volume 68 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

Zbigniew Stachniak. Going non-clausal. In *SAT'02: Proceedings of the 5th International Symposium on Theory and Applications of Satisfiability Testing*, Cincinnati, OH, USA, 2002.

P. Stephan, Robert K. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15:1167–1176, September 1996.

O. Strichman. Accelerating bounded model checking for safety properties. *Formal Methods in System Design*, 24(1):5–24, 2004.

Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. NiVER: Non increasing variable elimination resolution. In *SAT'04: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, number 3542 in Lecture Notes in Computer Science, pages 351–356, Vancouver, Canada, May 2004. Springer.

Geoff Sutcliffe. The CADE-20 Automated Theorem Proving Competition. *AI Communications*, 19(2):173–181, 2006.

Geoff Sutcliffe. The CADE-21 ATP system competition website, 2007. URL `http://www.cs.miami.edu/~tptp/CASC/21/`.

Geoff Sutcliffe and Christian B. Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006.

Geoff Sutcliffe and Christian B. Suttner. Website of the TPTP problem library for automated theorem proving, 2007. URL `http://www.cs.miami.edu/~tptp/`.

Geoff Sutcliffe and Christian B. Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

Geoff Sutcliffe, Christian B. Suttner, and Francis Jeffry Pelletier. The IJCAR ATP System Competition. *Journal of Automated Reasoning*, 28(3):307–320, 2002.

Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving non-clausal formulas with DPLL search. In *SAT'04: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, number 3542 in Lecture Notes in Computer Science, pages 663–678, Vancouver, Canada, May 2004. Springer.

Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In Sergio Flesca, Sergio Greco, Giovambattista Ianni, and Nicola Leone, editors, *JELIA'02: Proceedings of the European Conference on Logics in Artificial Intelligence*, number 2424 in Lecture Notes in Computer Science, pages 308–319, Cosenza, Italy, September 2002. Springer. ISBN 3-540-44190-5.

Dave A. D. Tompkins and Holger H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *SAT'04: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, number 3542 in Lecture Notes in Computer Science, Vancouver, Canada, May 2004. Springer.

Grigori S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, Part II*, 1968.

Allen van Gelder. Combining preorder and postorder resolution in a satisfiability solver. In Henry Kautz and Bart Selman, editors, *SAT'01: Proceedings of the 4th International Workshop on Theory and Applications of Satisfiability Testing, held at the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, volume 9 of *Electronic Notes in Discrete Mathematics*, Boston, MA, USA, June 2001. Elsevier.

Miroslav N. Velev and Randal E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar VLIW microprocessors. In *DAC'01: Proceedings of the 38th Conference on Design Automation*, pages 226–231, Las Vegas, NV, USA, June 2001.

Christoph Weidenbach, Bernd Gaede, and Georg Rock. SPASS & FLOTTER version 0.42. In *CADE-13: Proceedings of the 13th International Conference on Automated Deduction*, pages 141–145, London, U.K., 1996. Springer. ISBN 3-540-61511-3.

Poul Frederick Williams, Armin Biere, Edmund M. Clarke, and Anubhav Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In E. A. Emerson and A. P. Sistla, editors, *CAV'00: Proceedings of the 12th International Conference on Computer Aided Verification*, number 1855 in Lecture Notes in Computer Science, pages 124–138. Springer, July 2000. ISBN 3-540-67770-4.

Hanato Zhang and Mark E. Stickel. An efficient algorithm for unit-propagation. In *ISAIM'96: Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics*, Ft. Lauderdale, FL, USA, 1996.

Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *CAV'02: Proceedings of the 14th International Conference on Computer Aided Verification*, number 2404 in Lecture Notes in Computer Science, pages 17–36, London, U.K., 2002. Springer. ISBN 3-540-43997-8.

Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE'03: Proceedings of the Conference on Design, Automation and Test in Europe*, Munich, Germany, March 2003.

Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD'01: Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, pages 279–285, San Jose, CA, USA, 2001. IEEE Computer Society. ISBN 0-7803-7249-2.