Resolution-based methods for linear temporal reasoning

Martin Suda

Thesis for obtaining the title of Doctor of Engineering
of the Faculties of Natural Sciences and Technology
of Saarland University
to be awarded jointly with
the Faculty of Mathematics and Physics
of Charles University in Prague

Saarbrücken
December, 2014

Date of the colloquium:	October 16, 2015
Dean of the faculty:	Prof. Dr. Markus Bläser
Examination board:	
– Chair:	Prof. Dr. Gert Smolka
– Reviewers:	Prof. Dr. Roman Barták
	Prof. Dr. Armin Biere
	Prof. Dr. Jörg Hoffmann
	Prof. Dr. Christoph Weidenbach
– Scientific assistant:	Dr. Uwe Waldmann

Abstract

The aim of this thesis is to explore the potential of resolution-based methods for linear temporal reasoning. On the abstract level, this means to develop new algorithms for automated reasoning about properties of systems which evolve in time. More concretely, we will: 1) show how to adapt the superposition framework to proving theorems in propositional Linear Temporal Logic (LTL), 2) use a connection between superposition and the CDCL calculus of modern SAT solvers to come up with an efficient LTL prover, 3) specialize the previous to reachability properties and discover a close connection to Property Directed Reachability (PDR), an algorithm recently developed for model checking of hardware circuits, 4) further improve PDR by providing a new technique for enhancing clause propagation phase of the algorithm, and 5) adapt PDR to automated planning by replacing the SAT solver inside with a planning-specific procedure.

We implemented the proposed ideas and provide experimental results which demonstrate their practical potential on representative benchmark sets. Our system LS4 is shown to be the strongest LTL prover currently publicly available. The mentioned enhancement of PDR substantially improves the performance of our implementation of the algorithm for hardware model checking in the multi-property setting. It is expected that other implementations would benefit from it in an analogous way. Finally, our planner PDRplan has been compared with the state-of-the-art planners on the benchmarks from the International Planning Competition with very promising results.

Zusammenfassung

Das Ziel dieser Doktorarbeit ist es, das Potential resolutionsbasierter Methoden zur linearer, temporaler Beweisführung zu untersuchen. Von einem abstrakten Gesichtspunkt aus gesehen bedeutet dies, neue Algorithmen über die Eigenschaften von sich zeitlich entwicklenden Systemen im Bereich des automatischen Theorembeweisens zu entwickeln. Konkreter gesagt werden wir 1) aufzeigen, wie sich das Rahmenprogramm der Superposition so anpassen lässt, damit es Theoreme in propositionaler Linear Temporal Logic (LTL) beweist, 2) eine Verbindung zwischen der Superposition und dem CDCL-Kalkül moderner SAT-Solver nutzen, um mit einem effizienten LTL-Prover aufzuwarten, 3) das Vorangegangene auf Erreichbarkeitseigenschaften spezialisieren, und eine starke Verbindung zu der Property Directed Reachability (PDR), einem jüngst eintwickeltem Model-Checking-Algorithmus für Hardware-Schaltkreise, aufzudecken, 4) PDR durch die Einführung neuer Technik verbessern, die die Clause-Propagation-Phase des Algorithmus beschleunigt, und 5) PDR für das automatisierte Planen anpassen, indem wir den inneren SAT-Solver durch eine planungsspezifische Prozedur ersetzen.

Wir haben die vorgeschlagenen Ideen implementiert, und es werden experimentelle Ergebnisse angegeben, die das praktische Potential dieser Ideen auf repräsentativen Benchmarks aufzeigt. Es hat sich herausgestellt, dass unser System LS4 der stärkste öffentlich zugängliche LTL-Prover ist. Die erwähnte Erweiterung von PDR verbessern die Leistungsfähigkeit unserer Implementierung des Hardware-Model-Checking-Algorithmus substantiell im Bereich der Multi-Property-Einstellungen. Wir erwarten, dass andere Implementierungen in ähnlicher Weise profitieren würden. Schließlich haben wir viel versprechende Ergebnisse durch den Vergleich unser Planer PDRplan mit anderen state-of-the-art Planer auf den Benchmarks der International Planning Competition erzielt.

Shrnutí

Cílem této práce je prozkoumat potenciál metod založených na rezoluci pro uvažování s lineárním časem. To na abstraktní rovině znamená navrhnout nové algoritmy pro automatické uvažovaní o vlastnostech systémů, které se vyvíjí v čase. Konkrétně v této práci ukážeme, 1) jak adaptovat superpoziční metodu pro dokazování vět ve výrokové lineární temporální logice (LTL), 2) jak využít příbuznost mezi superpozicí a kalkulem CDCL z moderních SAT-solverů pro navržení nového LTL dokazovače, 3) jak tento specializovat pro problém dosažitelnosti a objevit tak blízkou souvislost s algoritmem Property Directed Reachability (PDR), v nedávné době vyvinutém pro model checking hardwarových obvodů, 4) jak dále vylepšit PDR novou technikou pro urychlení fáze propagace klauzulí, 5) jak PDR adaptovat pro problém automatického plánování tím, že se SAT-solver v algoritmu nahradí procedurou specifickou pro plánovací vstupy.

Navržené myšlenky byly implementovány a práce obsahuje výsledky experimentů, které na reprezentativních množinách benchmarků prokazují jejich praktický potenciál. Náš systém LS4 se ukázal býti jedním z nejsilnějších veřejně dostupných LTL dokazovačů. Zmíněné vylepšení algoritmu PDR podstatně zvyšuje výkon naší implementace při verifikaci hardware v multi-property módu. Dá se předpokládat, že ostatní implementace mohou z nové techniky benefitovat podobným způsobem. V neposlední řadě náš plánovač PDRplan uspěšně obstál při porovnání s nejmodernějšími plánovači na benchmarcích z mezinárodní plánovací soutěže IPC.

Acknowledgements

I was given the opportunity to work on this thesis under a joint supervision provided by the Saarland University, Germany and the Charles University in Prague, Czech Republic. I am grateful to these institutions for enabling and supporting such an arrangement from which I greatly benefited.

My greatest thanks go to my doctoral advisers Christoph Weidenbach, Petr Štěpánek and Roman Barták for accepting me as their student. It was their constant support, patient guidance and invaluable advice which made this work possible.

I would also like to thank my colleagues from the Automation of Logic group for creating a very friendly and inspiring work environment during my stays in Saarbrücken. Special thanks go to Arnaud Fietzke, Willem Hagemann, Matthias Horbach, Marek Košta, Evgeny Kruglov, Tianxiang Lu, Viorica Sofronie-Stokkermans, Thomas Sturm, Ching Hoo Tang, Uwe Waldmann, Daniel Wand and Patrick Wischnewski.

Moreover, I want to thank the anonymous reviewers of the publications underlying this thesis for their valuable input, Ullrich Hustadt and Viktor Schuppan for answering all my questions about LTL theorem proving, Armin Biere and Aaron Bradley for sharing their ideas on SAT-based unbounded model checking, Jörg Hoffmann and Jussi Rintanen for their quick replies to my planning related questions, and Tomáš Balyo for providing me with the SASE encoding tool.

I also want to thank Armin Biere and Jörg Hoffmann for kindly agreeing to review this thesis.

This work has been partly supported by Microsoft Research through its PhD Scholarship Programme. I am grateful for their financial support.

Finally, I would like to thank my family and friends for encouraging me in my endeavor and for providing pleasant distractions from logic when they became necessary.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

In this thesis, we explore the potential of resolution-based reasoning methods for automatically establishing properties of systems which evolve in time. Working within the *automation of logic* tradition, our main aim is to develop and analyze new algorithms for automated reasoning and to establish experimentally whether they can be successfully applied in practice. We will find relevant applications for our algorithms successively in three independent but closely related fields: theorem proving in linear temporal logic, verification of hardware circuits, and automated planning. The corresponding reasoning tasks share a common notion of time, which is modeled as a discrete linear sequence of time moments. They also share a formalism for representing the individual moments, namely propositional logic. It is this second property which will allow us to approach the tasks using resolution, a well understood rule of logical inference suitable for automation. It is the temporal aspect of the mentioned tasks, on the other hand, which will pose the main challenge.

This chapter first provides an overview of the three mentioned fields: theorem proving in linear temporal logic (Section 1.1), verification of hardware circuits (Section 1.2) and automated planning (Section 1.3) and of the corresponding reasoning tasks. It then reviews the resolution technology that we will use to approach these tasks (Section 1.4) and elaborates on the inherent challenges connected with reasoning about time and on our strategy to overcoming them (Section 1.5). The chapter ends with an outline of the main contributions of this thesis (Section 1.6).

1.1 Theorem proving in linear temporal logic

Linear-time temporal logic (LTL) is a modal logic designed for specifying temporal relations between events which occur over time. Originally conceived by Kamp (1968) to formally capture temporal relationships expressible in natural language, LTL was introduced into computer science by Pnueli (1977) as a specification language for reactive systems, i.e. systems with non-terminating computations. Nowadays, LTL is well established and widely used in practice.

LTL extends classical propositional logic by introducing temporal operators which specify the way in which a formula φ should be interpreted with respect to the flow of time. For example, the formula $\Diamond\varphi$ stands for "φ holds *eventually* in the future", $\Box\varphi$ means "φ holds *always* in the future", and $\bigcirc\varphi$ expresses that "φ holds at the *next* moment of time". Time is considered to be a linear discrete sequence of time moments represented by propositional valuations, informally also called *worlds*. Such a potentially infinite sequence forms an LTL interpretation. To prove an LTL formula φ means to establish that it is valid, i.e., that all LTL interpretations actually make φ true. For

example, the LTL formula $\Box\psi \rightarrow \bigcirc\psi$ is valid (a theorem), whereas the LTL formula $\bigcirc\psi \rightarrow \Box\psi$ is not, but is satisfiable, i.e., there is an LTL interpretation in which it is true. Similarly to classical logic, one can prove an LTL formula φ by showing that it does not have a counter-example, i.e. that its negation $\neg\varphi$ is not satisfiable. By this duality, LTL proving and satisfiability checking are essentially two sides of the same coin.

The traditional use of LTL lies in formal verification of reactive systems where the logic serves as a specification language for expressing the system's desired behavior. Such a specification is subsequently checked against a model of the system during the process of model checking (Clarke et al., 2001). More recently, the importance of LTL satisfiability checking and thus also theorem proving is becoming recognized (Rozier and Vardi, 2010; Schuppan and Darmawan, 2011). This is, for instance, essential for assuring quality of formal specifications (Pill et al., 2006). In more detail, because specifications are ultimately written by humans they may contain bugs. A priori excluding specifications which are either valid or unsatisfiable represents a useful sanity check, because in the former case the specification would be trivially satisfied by any system and in the latter by none. Another motivation comes from the fact that satisfiability is a precondition for realizability and thus a satisfiability checking procedure can be useful in debugging specifications for system synthesis (Jobstmann and Bloem, 2006).

In this thesis, we approach the problem of LTL theorem proving by building on the work of Fisher (1991) who showed how to transform any LTL formula into a certain clausal normal form. The normal form reduces the formula complexity in terms of temporal operator nestings and so removes one of the obstacles on the way to applying resolution-based methods for automated reasoning in LTL.

1.2 Verification of hardware circuits

With the growing reliance on hardware technology in our lives, ranging from the use of personal computers and mobile phones to areas such as traffic control and medicine, where human life is directly affected, the importance of the correct behavior of the used devices is becoming more than obvious. Formal verification provides methods to rigorously establish that a designed circuit meets its specification, thus greatly helping to increase our confidence in the correctness of the final device.

In this work, we focus on a particular phase in the design process in which the designed circuit obtains a representation referred to as gate-level netlist, an abstraction not unrelated to that of the Boolean circuit from theoretical computer science. Moreover, we will be interested in verifying *sequential* circuits, which in addition to logic gates for computing Boolean functions also employ state-holding elements called *latches*. Sequential circuits compute in cycles and in each cycle externally supplied inputs together with the old values stored in the latches participate in producing the circuit's output and the new values to be stored.[1] Simply put, sequential circuits have memory.

Let us have a look at Figure 1.1 for a small example. The figure depicts a simple

[1] More precisely, this describes the behavior of a sequential circuit of a *synchronous* type. There are also asynchronous circuits in which the state can change at any time in response to changing inputs.

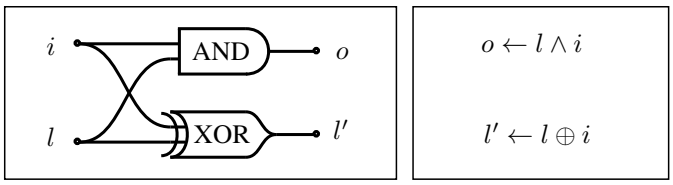


Figure 1.1: A simple circuit (left) and its imperative interpretation (right).

circuit storing a single bit of memory (latch l). If during a particular cycle the value of the input bit i is zero, the stored value is preserved ($l' = l$). If the value of the input is one, the stored value is read (output $o = l$) and at the same time toggled $l' = \neg l$. This follows from the inner working of the two employed gates, the AND gate computing logical conjunction ($\wedge$) and the XOR gate computing the exclusive or operation ($\oplus$). The semantics of a latch is to take the value l' computed at the end of one cycle and pass it back as l for the computation of the following cycle. This defines the discrete flow of time in the context of the circuit's computation.

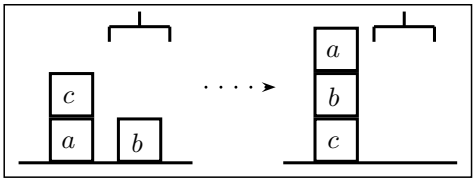
One verifies a sequential circuit with respect to a property expressing its correct behavior. In the most common setting called the verification of invariance properties, which we will consider in this work, the property is a propositional formula φ over variables corresponding to the latches. Formula φ specifies an *invariant* of the circuit which must be satisfied in all the states reachable from a fixed initial state. Dually, the invariant is violated if there is a computation of the circuit starting from the initial state, processing in cycles a particular sequence of inputs and ending in a state where φ does not hold. Such a state is usually called a *bad* state.

Sequential verification is a computationally hard problem. An intuition about this can be drawn by realizing that the size of the circuit's state space is exponential in the number of its latches. Moreover, exponentially many states may need to be traversed on the potential path from an initial state to a bad state. One way to combat this well known *state explosion problem* is to avoid explicit enumeration of the states and instead use and manipulate symbolic representation of whole sets of states. This approach was pioneered by McMillan (1993) who used Binary Decision Diagrams as such a representation.

By adapting the resolution-based methods to the problem of verification of invariance properties, we will naturally arrive in this work to a symbolic algorithm where the representation of sets of states is based on clausal propositional logic.

1.3 Automated planning

Automated planning is a classical discipline of artificial intelligence (Russell and Norvig, 2010; Ghallab et al., 2004). Operating with a given formal, high-level description of a world, its fundamental task is to look for a sequence of actions that lead to achieving a specified goal. Depending on the modeled scenario, the goal-achieving sequence, or simply the plan, may describe anything from a route for a space rover collecting samples on a remote planet, instructions for the preparation of a complex chemical from individual



Operator $unstack(X, Y)$
pre : $clear(X)$, $on(X, Y)$, $arm\text{-}empty$
add : $holding(X)$, $clear(Y)$
del : $clear(X)$, $on(X, Y)$, $arm\text{-}empty$

Figure 1.2: The initial and goal configurations of simple planning scenario (left) and an example operator (right).

simpler compounds, to a set of trajectories for conveyors loading crates in a warehouse. This generality is enabled by the expressiveness of the language used for describing the input task, which is traditionally based on first-order logic.

Figure 1.2 depicts an example situation from the emblematic *blocks world* planning domain. The task in the domain is to plan the instructions of a robotic arm to rearrange stacks of blocks. The domain description consists of a set of predicates for defining the state of the world (here, $on(X, Y)$, $clear(X)$, etc.) and a set of operators defining the possible transformations (here, $unstack(X, Y)$, $stack(X, Y)$, ...). An instance of a planning task in the domain is then specified by supplying a set of concrete objects (in our example, we have blocks a, b and c) and their initial and goal configurations (for instance, the facts $on(c, a)$, $clear(c)$, and $arm\text{-}empty$ are part of the definition of the initial configuration, whereas $on(a, b)$ and $on(b, c)$ would be part of the goal).

Operators are parametrized schemas for describing concrete actions. That means that an action is obtained by substituting concrete objects for the operator's parameters. In our example, we for instance obtain the action $unstack(c, a)$ by substituting c for X and a for Y in operator $unstack(X, Y)$. This action is *applicable* in the initial configuration, because all its three preconditions (pre): $clear(c)$, $on(c, a)$ and $arm\text{-}empty$ are satisfied there. When this action is applied, we arrive to a new configuration by adding (add) facts that are supposed to additionally hold (here $holding(c)$ and $clear(a)$) and by deleting (del) facts that now cease to hold (here the same three that were required as preconditions). Such a transition from one configuration to the next via action application is what defines a single time step within the planning semantics.

The blocks world domain is a very simple one and a specialized algorithm for efficiently solving tasks in this domain can easily be devised. The key asset of automated planning, however, lies in that it is *domain independent*. It aims to provide methods for uniformly solving tasks in any domain that can be described by its language. That is what makes automated planning versatile, but also challenging. Another interesting perspective on domain independence is to view the planning formalism as a high-level declarative programming language decoupling the problem from its solution (Hoffmann, 2011).

Thanks to the connection of the standard planning formalism to first-order logic, some of the historically first approaches to automated planning formulated the task as first-order theorem proving (Green, 1969; McCarthy and Hayes, 1969). The later discovery of Kautz and Selman (1992), who proposed to restate the same problem in terms of *propositional satisfiability*, founded a new powerful approach to planning, which is still

actively studied today. This planning as satisfiability paradigm shows how to encode a planning task into "propositional logic plus linear time" and will provide the necessary bridge to planning for the resolution-based methods developed in this thesis.

1.4 Resolution-based reasoning

In this work, we will encounter resolution in two main forms. Explicitly, as an inference rule in the context of saturation-based theorem proving, and implicitly, as a proof system underlying the computation of most of the modern propositional satisfiability (SAT) solvers. As will become eventually apparent, these two forms are, in fact, closely related.

Resolution and saturation

The history of resolution as a logical rule is at least as old as the history of the automated theorem proving research field. One of the first appearances of resolution can be found in the work of Davis and Putnam (1960), who used the principle within their theorem proving procedure for eliminating ground atomic formulas. Subsequently, Robinson (1965) showed how to use unification to lift resolution from ground formulas to formulas with variables and established saturation-based theorem proving with resolution as the main inference rule as one of the most successful approaches to automatically proving theorems in first-order logic.

Resolution operates on a formula in clause normal form which consists of a conjunctive set of clauses, each clause being a disjunction of literals. From two clauses $C \vee a$ and $D \vee \neg a$ containing complementary literals a and $\neg a$, respectively, the resolution inference rule (or, more precisely, a propositional version thereof) allows one to derive a resolvent $C \vee D$. Such an inference is typically summarized as

$$\mathcal{I}\frac{C \vee a \quad D \vee \neg a}{C \vee D}.$$

Resolution forms the basis of a refutationally complete calculus, which means that from any unsatisfiable set of clauses one can derive an obvious contradiction in the form of the empty clause by a finite number of resolution inferences.[2]

Saturation is a process of performing all available inferences in a systematic way until the empty clause is derived, or a set closed under the inferences and not containing the empty clause is obtained which signifies that the original formula was satisfiable. Because saturation typically produces a large number of clauses, techniques for controlling and restricting the process are of great practical importance.

One of the most important achievements in this area was the development of the *superposition* calculus (Bachmair and Ganzinger, 1990, 1994) and an associated set of saturation strategies which 1) allow a restriction of considered inferences to only those satisfying certain ordering constraints on the participating literals, 2) introduce a powerful notion of abstract redundancy to justify active removal of clauses that provably

[2] Combined with the explicit or implicit use of the factoring rule, which takes care of contracting multiple occurrences of the same literal in a clause to just a single occurrence.

cannot contribute to the derivation of the empty clause, 3) still guarantee overall completeness. While the ideas of superposition were originally conceived in the context of first-order theorem proving with equality, they can be restated and successfully applied already on the level of propositional logic (Bachmair and Ganzinger, 2001).

Resolution and SAT solving

By revising the already mentioned procedure of Davis and Putnam (1960) and replacing the explicit application of resolution by a splitting rule for case analysis, Davis et al. (1962) introduced an algorithm nowadays known as the DPLL procedure and started an extremely successful field of practical propositional satisfiability (SAT) checking.

DPLL is best described as a backtrack search in the space of partial truth assignments, supported by a *unit propagation* rule for early detection of conflicts. The power of modern SAT solvers still originates from DPLL, but is extended further by the use of efficient data structures and clever branching heuristics (Moskewicz et al., 2001), non-chronological backtracking with *conflict driven clause learning* (Marques-Silva and Sakallah, 1999) and many other techniques (see, e.g., Biere et al., 2009).

Although procedurally DPLL and its extensions are very different from saturation, the corresponding procedures can still be seen to implicitly generate a resolution proof of the input formula. Results of this form are not always easy to derive, but are important, for instance, for establishing the proof theoretic strength of the individual extensions (Beame et al., 2004; Pipatsrisawat and Darwiche, 2009).

In this thesis, we will exploit a particularly fine-grained connection (Weidenbach) between CDCL, a version of DPLL enhanced with the conflict driven clause learning technique, and the superposition framework (Bachmair and Ganzinger, 2001). The connection, in particular, relates unit propagation and clause learning to the concept of a *productive clause* from the completeness proof of superposition. We will be able to use this connection to transfer a proof method first developed in the saturation setting and to come up with an efficient SAT-based LTL prover.

1.5 The temporal challenge

Although resolution is a well established method for automated reasoning in general, the temporal dimension of the problems studied in this work represents an extra challenge. On an abstract level, this challenge manifests itself in at least two forms depending on whether we primarily focus on satisfiability or unsatisfiability detection.[3]

From the perspective of satisfiability detection, we deal with the challenge of large models. As mentioned before, a path from an initial state to a bad state of a circuit can be exponential in the size of the task description and a similar observation holds for planning tasks. Models in LTL are formally even infinite sequences and although they can be represented finitely, one still faces a potential exponential blowup.

[3] All the algorithms presented in this work are decision procedures in the sense that they always terminate and correctly report the satisfiability status of the input task. Nevertheless, separate aspects in their design concerning either only satisfiable or only unsatisfiable inputs can be recognized.

From the perspective of unsatisfiability detection, we face a challenge of the discovery of an inductive argument. Indeed, it seems that a form of induction is always needed to make the step from partial results of the form "a short path does not exists" to the ultimate claim "no path (of any length) exists". The option to exhaust each of the exponentially many path lengths separately does not seem to lead to a feasible solution.

Our general approach to overcoming the respective challenges rests: 1) on assigning time indexes to signature symbols to deal with the challenge of large models and 2) on a proof repetition detection and replaying technique to deal with the challenge of the discovery of an inductive argument.

Concerning the former, we take inspiration in the way time is encoded in the planning as satisfiability paradigm (Kautz and Selman, 1996) or, equivalently, in the related bounded model checking approach (Biere et al., 1999) known from verification. This solution needs to be complemented by an additional idea in the case of LTL, where we formally need infinitely many indexes to describe the whole model. We devise a way to use labels in the spirit of (Lev-Ami et al., 2007) to finitely represent clauses over the obtained infinite signature to overcome this obstacle.

Concerning the latter, we will design our proof repetition detection and replaying technique in a similar way to how the melting rule for loop detection proposed by Horbach and Weidenbach (2009) is used for inductive reasoning in the context of superposition for fixed domains (Horbach and Weidenbach, 2008). This is, however, only a high-level analogy since the details of the respective studied problems are considerably different.

1.6 Main contributions and thesis overview

The content of this thesis is structured into five main chapters. In the first three chapters (Chapters 2–4), we deal with LTL theorem proving, in the fourth (Chapter 5) we move our attention to the verification of hardware circuits and in the final chapter (Chapter 6) we consider automated planning. We develop and analyze several new algorithms, experimentally evaluate their performance, and put them into the context of related work within the respective research fields.

Although the observation that the problems studied in this work are closely related is not new, we bring it to a much more explicit level by devising a single representation for the corresponding reasoning tasks. This allows us to approach the problems in a uniform way using resolution and thus to demonstrate how close the problems can be put together and the corresponding tasks aligned on the right level of abstraction. It also paves the way for further exchange of ideas between the research fields. As such, it can be considered one of the main contribution of this thesis.

On a more fine-grained level, the five main chapters of this thesis contain the following contributions.

- In Chapter 2, we develop LPSup, a new calculus for proving theorems in LTL and use it as a basis for a new decision procedure for the logic. The main idea is to treat temporal formulas as infinite sets of purely propositional clauses over an extended signature in which symbols are indexed by time moments and to represent these

infinite sets by finite sets of labeled propositional clauses. This new representation naturally leads to the replacement of a complex temporal resolution rule, suggested by an existing resolution calculus for LTL, by a fine grained repetition check of finitely saturated labeled clause sets followed by a simple inference.

Our completeness argument is based on the standard model building idea from superposition. It inherently justifies ordering restrictions, redundancy elimination and effective partial model building. The last property can be directly used to effectively generate counter-examples to non-valid LTL conjectures out of saturated labeled clause sets in a straightforward way.

We study the computations of LPSup further by interpreting the logic-based, symbolic operations of the calculus on the semantic level of explicit valuations. This perspective later allows us to reveal interesting connections to related approaches and to identify the strengths and potential weaknesses of our method.

The material of this chapter is a revised and extended version of previously published work (Suda and Weidenbach, 2012b,c).

- Building on the understanding acquired in the previous chapter, we approach in Chapter 3 the problem of LTL theorem proving from a different angle. We abandon the saturation paradigm and design a new decision procedure based on the observation that LPSup can build partial models on the fly. We show how to use these models to drastically restrict the selection of inferences and thus to effectively guide the proof search.

  Relying on the previously mentioned connection between conflict driven clause learning and superposition, we implement the model guidance idea within the SAT solving framework. In more detail, we design our new decision procedure, which we call LS4, by using an efficient SAT solver as a subroutine. A non-trivial bookkeeping is needed, however, to maintain the correspondence with the labeled clauses of LPSup, a prerequisite for the discovery of full LTL models as well as for the detection of overall unsatisfiability.

  We prove that LS4 is correct and terminating.[4] On an extensive set of LTL benchmarks, we experimentally demonstrate that our implementation of LS4 is one of the strongest available LTL provers. In a detailed comparison with related work we then attempt to discover the key aspects behind LS4's success.

  This chapter is based on an earlier publication (Suda and Weidenbach, 2012a), but has been thoroughly revised and notably extended.

- In Chapter 4, we study preprocessing techniques for clause normal forms of LTL formulas. For that purpose, we further extend the mechanism of labeled clauses introduced in Chapter 2, which here allows us to faithfully lift simplification ideas

[4] Algorithms are presented throughout this thesis in the form of abstract programs. Therefore, they are not formally verified, e.g., as it is done in a dynamic logic or Hoare triple framework for computer programs. Instead, the key insights and invariants leading to the algorithms are formally shown on a mathematical level.

from SAT to LTL. We demonstrate this by adapting variable and clause elimination, an effective preprocessing technique used by modern SAT solvers. Our experiments confirm that even in the temporal setting substantial reductions in formula size and subsequent decrease of solver runtime can be achieved.

The results of this chapter have been published in (Suda, 2013d,a).

- Chapter 5 returns to the model guidance idea. We first show how to specialize LS4, the algorithm previously developed for proving theorems in LTL, to decide (non-)reachability in symbolically represented transition systems. The obtained algorithm Reach can be immediately used to verify invariance (and safety) properties of hardware circuits.

  We then proceed to place the new algorithm within the context of related work from the verification area. The fact that Reach builds on the SAT-solver technology and, more specifically, the form of the logical formula it indirectly evaluates make the algorithm related to the bounded model checking method of Biere et al. (1999). On the other hand, the ability of Reach to efficiently detect unsatisfiable instances can be attributed to the distinctive way, in which the algorithm generates and utilizes interpolants (McMillan, 2003).

  Ultimately, we find Reach closely related to Property Directed Reachability (PDR), also known as IC3, an algorithm recently introduced by Bradley (2011). We show how to transform Reach into PDR by a small change in the core of the algorithm and by the addition of three independent enhancements. This allows us to view PDR from the new perspective of the model guidance idea.

  We continue by proposing triggered clause pushing, an additional improvement of PDR, namely of the clause propagation phase of the algorithm. The idea is to collect models computed by the SAT solver during clause propagation and use them as witnesses for why the respective clauses could not be pushed forward. Witnesses are then used as a pre-filter on the subsequent push operations making them cheaper on average.

  The chapter is closed by a detailed experimental evaluation of Reach, of the several extensions of the algorithm leading to PDR, and of PDR improved by triggered clause pushing. We examine the relative utility of the individual improvements and its dependance on other conditions, such as the chosen search direction and the satisfiability status of the input problem.

- Given the exceptional success of PDR in hardware model checking, a natural question arises whether the algorithm could be adapted and applied in the related field of automated planning. In Chapter 6, we give a positive answer to this question.

  First, we notice that the commonly used encodings from the planning as satisfiability paradigm (Kautz and Selman, 1992) can be easily modified to yield an input for PDR. However, we also discover a non-obvious alternative to such a direct combination. We show that the SAT solver inside PDR can be replaced by

a planning-specific procedure implementing the same interface. This SAT-solver-free variant of the algorithm is not only more efficient, but also offers additional insights and opportunities for further improvements.

We confirm this claim empirically in the experimental part of Chapter 6. Then we compare our implementation of the proposed version of PDR to the state-of-the-art planners and find it highly competitive, solving the most problems on several benchmark domains. Moreover, in a separate set of experiments, we also evaluate PDR with respect to the quality of produced plans, the detection of unsatisfiable planning problems, and in the context of optimal planning.

The material of the chapter has been published in (Suda, 2014a).

The thesis ends with a concluding Chapter 7, where we provide a unifying perspective on the presented results, summarize the lessons learned and suggests directions for future research.

Setting aside Chapter 4, which only depends on Chapter 2, each of the remaining presented chapters depends on its predecessor as we transfer ideas between two subsequent approaches or transfer an approach from one field and adapt it to the next.

During our exposition, we introduce new terminology as needed throughout the whole text. The reader may refer to the index provided at the end of this text to quickly look up and recall definitions of all the introduced notions.

2 Labeled superposition for LTL

2.1 Introduction

Linear temporal logic is an extension of classical propositional logic for reasoning about time. The increased expressiveness of its language is naturally reflected by the higher complexity of the associated reasoning task. Indeed, while theorem proving is coNP-complete for propositional formulas, it is PSPACE-complete for the formulas of LTL (Sistla and Clarke, 1985). Nevertheless, a viable strategy for developing a reasoning method for LTL is to start with a well understood approach from the classical setting, such as resolution, and attempt to extend it to deal with the temporal aspects characterizing LTL. Our aim in this chapter is to follow this strategy and to extend to LTL the reasoning framework of Bachmair and Ganzinger (2001) while striving to preserve its properties which are desirable for automation.

First attempts to use clausal resolution to attack the decision problem for LTL are due to Cavalli and del Cerro (1984) and Venkatesh (1985). The most recent resolution-based approach is the Clausal Temporal Resolution calculus (CTR) of Fisher et al. (2001). It relies on a satisfiability preserving clausal translation of LTL formulas, which removes all but a core set of temporal operators and reduces certain unnecessary nestings. Classical propositional resolution is adapted to cope with "local" temporal reasoning within neighboring worlds, while an additional "global" inference rule called *temporal resolution* is introduced to deal with so called *eventuality* clauses. The temporal resolution rule is quite complex. It requires a search for certain combinations of clauses that together form a *loop*, i.e. imply that certain sets of worlds must be discarded from consideration, because an eventuality clause would be unsatisfied forever within them. This search requires specialized algorithms (Dixon, 1996) and the applicability of the rule must be verified via an additional proof task. Finally, the conclusion of the rule needs to be transformed back into the clause form.

In this chapter, we present a new resolution-based calculus for LTL called labeled superposition (denoted LPSup). It builds on a refinement (Degtyarev et al., 2002) of the above clause normal form and introduces a notion of a labeled clause in the spirit of (Lev-Ami et al., 2007). In our case, a label encodes the temporal context of a clause while the remaining standard part is kept purely propositional. Labels allow us to conceptually reduce LTL satisfiability to a set of purely propositional problems and thus to lift propositional reasoning to deal with LTL. Although the equality predicate is not present in LTL, the principles of superposition are fundamental to our calculus. Our completeness result is based on a model generation approach with an inherent redundancy concept based on a total well-founded ordering on the propositional atoms.

LPSup is turned into a decision procedure by a straightforward proof strategy which

interleaves finite saturation of certain labeled clauses with a simple Leap inference. This replaces the complex temporal resolution rule suggested by the previous work. Further advantages of our calculus include:

- its inference rules are restricted by an ordering which is known to reduce the search space considerably,
- the completeness proof justifies an abstract redundancy notion that enables strong reductions,
- if a contradiction cannot be derived, a temporal model can be extracted from the final saturated clause set in a straightforward way.

The rest of this chapter consists of the following parts with the following additional contributions. In the preliminaries (Section 2.2), we formally set up the notation and review the needed concepts. We add one final step to the adopted normal form transformation of LTL formulas and introduce the notion of a Temporal Satisfiability Task (TST) for the final product. A TST clearly separates the temporal aspect of the given satisfiability problem from the propositional basis and represents a canonical input for our decision procedure.

The next part deals with the development of LPSup. We introduce labeled clauses and their semantics in Section 2.3.1, present the inference and reduction rules of LPSup and prove their soundness in Section 2.3.2, show how to turn the calculus into a decision procedure in Section 2.3.3, and, finally, formulate and prove the completeness theorem for LPSup in Section 2.3.4. The last section also presents a model building procedure for extracting counter-examples to unprovable formulas.

The third part (Section 2.4) is devoted to a deeper analysis of LPSup and the computation of the corresponding decision procedure. First, we show that the input TST can be understood as a concise, symbolic description of a certain Büchi automaton. Inspired by this observation, we then give a new graph-based semantics to labeled clause sets and interpret the individual logical operations of the calculus as updates of an associated graph. Finally, we prove a theorem describing a similar connection for the saturation process as a whole.

The above results are important for understanding the relation of LPSup to other approaches to LTL satisfiability, which is the topic of the last part of this chapter (Section 2.5). After a general survey of the main known methods, we focus on analyzing the connection to the closest relative, the already mentioned calculus CTR by Fisher et al. (2001). We show how to align the syntax of the two calculi, compare the corresponding computations and their worst case complexity guarantees and report on an experiment with a prototype implementation of both methods.

The material of this chapter is a revised and extended version of previously published work (Suda and Weidenbach, 2012b,c).

2.2 Preliminaries

2.2.1 Resolution-based theorem proving in propositional logic

The purpose of this section is to recall the basic terminology concerning propositional logic and refutational theorem proving. After reviewing the syntax and semantics of the logic, we briefly explain in abstract terms the theorem proving methodology. We then present PSup, a propositional calculus based on the ideas of superposition (Bachmair and Ganzinger, 2001). Although superposition in the narrow sense denotes a calculus for first-order logic with equality (Bachmair and Ganzinger, 1990, 1994), in a broad sense it is seen as a particular technology for devising efficient calculi. In the case of PSup, the ideas of superposition are used to justify ordering constraints on inferences, to provide a powerful redundancy concept, and to show completeness of the calculus in a highly constructive way.

Propositional syntax and semantics

Propositional logic formulas are built over a propositional *signature* Σ of propositional *variables* using the connectives *negation* $\neg$, *conjunction* $\wedge$, and *disjunction* $\vee$. Thus, any variable $p \in \Sigma$ is a formula (we call such a formula *atomic* or simply an *atom*), and when φ and ψ are formulas then so are $\neg\varphi$, $\varphi \wedge \psi$, and $\varphi \vee \psi$. We treat the connectives *implication* $\rightarrow$ and *equivalence* $\equiv$ as abbreviations: the formula $\varphi \rightarrow \psi$ stands for $(\neg\varphi \vee \psi)$ and the formula $\varphi \equiv \psi$ abbreviates $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$.

The set of variables occurring in a formula φ is denoted $\mathit{Vars}(\varphi)$.

A *literal* is a propositional variable p or its negation $\neg p$. In the former case, the literal is called *positive* and in the latter *negative*. A *complement* $\sim l$ of a literal l is defined as $\neg p$ if $l = p$ for a variable $p \in \Sigma$, and as p if $l = \neg p$. A propositional *clause* is a finite set of literals understood as their disjunction. Under this convention, we usually write $C \vee D$ for the clause $C \cup D$. The *empty clause* is denoted $\bot$ and represents falsity. A clause is called a *tautology*, if it contains both a literal and its complement. A clause C is said to *subsume* another clause D, if $C \subseteq D$. A set of clauses is interpreted as the conjunction of its elements.

We base the semantics of propositional logic on the notion of a *valuation*, which is a mapping $V : \Sigma \rightarrow \{\mathbf{0}, \mathbf{1}\}$ from the signature Σ to the set of truth values $\{\mathbf{0}, \mathbf{1}\}$. Alternatively, we sometimes use the term *interpretation*, to refer to the set of those atoms $I \subseteq \Sigma$ that are assumed to be true. Valuations are characteristic functions of interpretations (by setting $V(p) = \mathbf{1}$ if and only if $p \in I$) and thus the two notions are equivalent. We will use interpretations in place of valuations when it is convenient.

The satisfiability relation $V \models \varphi$ between a valuation V and a formula φ is defined in the usual way. A formula φ is *satisfiable* if there is a valuation V such that $V \models \varphi$. Such a valuation V is called a *model* of φ. A formula φ is semantically *entailed* by a set of formulas Ψ, written $\Psi \models \varphi$, if every valuation V that satisfies all the formulas from Ψ also satisfies φ. A formula φ is called *valid*, written $\models \varphi$, if every valuation V satisfies φ. Note that a formula φ is valid if and only if $\neg\varphi$ is not satisfiable.

Refutational theorem proving

In refutational theorem proving we attempt to *prove* a given formula φ, i.e. to show that φ valid, by *refuting* satisfiability of its negation $\neg\varphi$. If $\neg\varphi$ is satisfiable, the corresponding model represents a *counter-example* to validity of the original formula φ.

As a first step to showing that $\neg\varphi$ does not have a model we transform the negated formula into a normal form. In propositional logic we rely on the *Conjunctive Normal Form (CNF)*, which consists of a conjunction of disjunctions of literals. Any propositional formula $(\neg)\varphi$ can be transformed in linear time (Tseitin, 1983) into an equisatisfiable formula N in CNF. Using our conventions we treat a formula in CNF simply as a set of clauses.

Given a set of clauses N, we then employ a *calculus* to derive new clauses from those of N. Formally, a calculus consists of a set *inference rules*, which are tuples of the form

$$\mathcal{I}\frac{C_1 \ \ldots \ C_n}{D},$$

where the clauses C_i are called the *premises* of the rule and the clause D its *conclusion*. If all the premises C_i of an inference are present in the set N we *derive* the conclusion D and add it to N. Inference rules are required to be *sound*, which means that the premises entail the conclusion:

$$C_1, \ldots, C_n \models D.$$

A calculus is sound if it consists of sound inference rules only. The ultimate goal of the proving process is to derive the empty clause $\bot$, which is unsatisfiable. It then follows from soundness that the original clause set N is unsatisfiable as well, and so the original formula φ is shown to be valid.

In practice, it is important to complement the addition of newly derived clauses by removal of clauses that can be shown redundant for deriving the empty clause. A *reduction rule* is a tuple

$$\mathcal{R}\frac{C_1 \ \ldots \ C_n}{D_1 \ \ldots \ D_m},$$

with premises C_i and conclusions D_j. A reduction rule can be applied if its premises are present in the current clause set N, in which case these premises C_i are deleted from N and replaced by the conclusions D_j. A reduction is sound if each of its conclusion is entailed by the set of its premises.

We turn a calculus into a theorem proving algorithm by choosing a particular *saturation strategy*, a rule which at any point selects an inference or a reduction to be applied next. The strategy should be *fair* which intuitively means that no opportunity to add a conclusion of an inference is postponed indefinitely (please consult the work of Bachmair and Ganzinger, 2001, for a formal treatment of fairness). Under such conditions the saturation process eventually derives sufficiently many clauses for us to determine satisfiability of the original clause set. A typical *completeness* result for a calculus states that any saturated clause set which does not contain the empty clause has a model. A concrete example of a complete calculus is presented next.

Propositional superposition calculus PSup

Here we present the propositional superposition calculus PSup, our adaption of the ideas of Bachmair and Ganzinger (2001). Because in our setting we treat clauses as sets of literals (as opposed to multisets) we can afford to simplify the exposition in several places. In particular, we do not have to explicitly introduce any kind of factoring.

The calculus PSup is parametrized by a strict well-founded ordering $<$ on the symbols of the signature Σ.[1] There is a standard way of extending this ordering to literals over Σ by making each negative literal slightly larger than its positive complement. This means that $p < \neg p$ for every $p \in \Sigma$, but for every $p, q \in \Sigma$ we have $\neg p < q$ whenever $p < q$. We say that a literal l is *maximal* in a clause C if there is no $l_0 \in C$ such that $l < l_0$. The ordering $<$ on literals is further extended to an ordering $<^c$ on (finite) clauses, called the *clause extension* of $<$. We define

$$C <^c D \text{ if and only if } C \neq D \text{ and } \forall l_C \in (C \setminus D)\, \exists l_D \in (D \setminus C) \text{ such that } l_C < l_D.$$

If an ordering $<$ is total (respectively, well-founded) so is its clause extension.

There is a single[2] inference rule in PSup, called *Ordered Resolution*. It is described by the tuple

$$\mathcal{I}\frac{C \vee a \quad D \vee \neg a}{C \vee D},$$

where the atom a is maximal in C and its complement $\neg a$ is maximal in D. The resolution rule is easily seen to be sound. The fact that it can be restricted to maximal literals in each clause greatly helps to keep the space of derivations manageable.

Possible reductions for PSup include *Tautology deletion*

$$\mathcal{R}\frac{C \vee l \vee {\sim l}}{},$$

where ${\sim l}$ is the complement of the literal l, and *Subsumption*

$$\mathcal{R}\frac{C \quad D}{C},$$

where the clause C strictly subsumes D. Tautology deletion and Subsumption are just example instances of the following abstract redundancy notion.

Definition 2.1. A clause C is *redundant* with respect to a set of clauses N, if there are clauses $C_1, \ldots, C_n \in N$ such that for every $i = 1, \ldots, n$, $C_i <^c C$, and $C_1, \ldots, C_n \models C$.

The advantage of abstract redundancy is that new reductions can be easily introduced to the calculus. As long as they fall under Definition 2.1, the completeness argument for the calculus does not need to be changed.

The purpose of the saturation process (see Bachmair and Ganzinger, 2001, Section 4) is to derive enough clauses to make satisfiability of the clause set obvious. This happens either when the empty clause is derived or when we "exhaust all possibilities".

[1] When we want to stress this fact, we write PSup$^<$ to denote the concrete version of the calculus.

[2] Strictly speaking, this is a single inference rule *schema* with many concrete instances: one instance for every choice of the clauses C and D, and the atom a satisfying respective the side conditions.

Definition 2.2. A set of standard clauses N is *saturated up to redundancy* (with respect to PSup), if for every PSup inference with non-redundant (with respect to N) premises in N its conclusion is either redundant with respect to N or contained in N.

The key to completeness of PSup is the following model building operator which recursively constructs an interpretation for a given set of clauses N.

Definition 2.3 (Model Operator)**.** Let $<^c$ be a clause extension of literal ordering $<$ and let N be a set of clauses. For a clause $C \in N$ we define by a well-founded recursion over $<^c$ a propositional interpretation $\mathcal{I}^{<^c}(C)$ and a clause set ϵ_C as follows. We set $\mathcal{I}^{<^c}(C) = \bigcup_{D<^cC} \epsilon_D$ and if the clause C

- is of the form $C_0 \vee a$, where the atom a is the maximal literal in C, and
- is false in $\mathcal{I}^{<^c}(C)$,

then we say the clause *produces* the atom a and set $\epsilon_C = \{a\}$; otherwise $\epsilon_C = \emptyset$. Such a clause C is called *productive*. Finally, we define $\mathcal{I}^{<^c}(N) = \bigcup_{C\in N} \epsilon_C$.

Theorem 2.1 (Bachmair and Ganzinger, 2001)**.** *Let N be a set of clauses that is saturated up to redundancy with respect to PSup$^<$ and does not contain the empty clause. Then N is satisfiable, in fact, $\mathcal{I}^{<^c}(N) \models N$.*

Proof (sketch). The proof works by reducing counterexamples (formally, by induction over $<^c$). Let us assume, for a contradiction, that $\mathcal{I}^{<^c}(N) \not\models N$ and pick D as the smallest clause in N such that $\mathcal{I}^{<^c}(N) \not\models D$. Because N does not contain the empty clause, D is of the form $D_0 \vee l$ with l being its maximal literal. The literal l cannot be positive, because then the clause D would have produced l into $\mathcal{I}^{<^c}(N)$. Thus D is of the form $D_0 \vee \neg a$ and there must be another clause C of the form $C_0 \vee a$ which produced a, making $\neg a$ false in $\mathcal{I}^{<^c}(N)$. The clauses C and D are premises of Ordered Resolution inference with a conclusion $C_0 \vee D_0$. The conclusion is smaller than D and necessarily false in $\mathcal{I}^{<^c}(N)$. As such, it cannot be contained in N (D is the smallest clause in N false in $\mathcal{I}^{<^c}(N)$), but it also cannot be redundant with respect to N (in that case there would be even a smaller clause $E \in N$ false in $\mathcal{I}^{<^c}(N)$). A contradiction. □

2.2.2 Linear temporal logic

This section covers the necessary preliminaries for theorem proving in Linear Temporal Logic (LTL). After introducing the syntax and semantics of the logic, we describe the Separated Normal Form (SNF) of LTL formulas (Fisher, 1991), which is a clause normal form particularly suited for resolution-based theorem proving. We then use the techniques by Degtyarev et al. (2002) to further refine SNF and obtain our own input format called Temporal Satisfiability Task (TST). TSTs employ "priming notation" borrowed from the verification literature to clearly separate the temporal aspects of the given satisfiability problem from the propositional background.

To keep our exposition self-contained we present a streamlined version of the SNF transformation adopted from Fisher et al. (2001) as well as a compilation of the refinements by Degtyarev et al. (2002). Because these are not needed for the understanding

$\mathcal{V}, i \models p$	iff $V_i \models p$,
$\mathcal{V}, i \models \neg\varphi$	iff not $\mathcal{V}, i \models \varphi$,
$\mathcal{V}, i \models \varphi \wedge \psi$	iff $\mathcal{V}, i \models \varphi$ and $\mathcal{V}, i \models \psi$,
$\mathcal{V}, i \models \varphi \vee \psi$	iff $\mathcal{V}, i \models \varphi$ or $\mathcal{V}, i \models \psi$,
$\mathcal{V}, i \models \bigcirc\varphi$	iff $\mathcal{V}, i+1 \models \varphi$,
$\mathcal{V}, i \models \Box\varphi$	iff for all $j \geq i$, $\mathcal{V}, j \models \varphi$,
$\mathcal{V}, i \models \Diamond\varphi$	iff there exists $j \geq i$ such that $\mathcal{V}, j \models \varphi$,
$\mathcal{V}, i \models \varphi \mathsf{U} \psi$	iff there exists $j \geq i$ such that $\mathcal{V}, j \models \psi$
	and $\mathcal{V}, k \models \varphi$ for every k, $i \leq k < j$,
$\mathcal{V}, i \models \varphi \mathsf{R} \psi$	iff for all $j \geq i$, $\mathcal{V}, j \models \psi$ or,
	there exists $j \geq i$ with $\mathcal{V}, j \models \varphi$ and for all k, $i \leq k \leq j$, $\mathcal{V}, k \models \psi$.

Figure 2.1: Recursive definition of LTL semantics.

of the further developments, the impatient reader may jump directly to the description of the priming notation on page 21 and continue to the definition of a TST, which constitutes the sole entry point of the subsequent theory.

LTL syntax and semantics

The language of LTL is an extension of the propositional language with temporal operators. The most commonly used are Next $\bigcirc$, Always $\Box$, Eventually $\Diamond$, Until U, and Release R. Formally, let $\Sigma = \{p, q, \ldots\}$ be a (finite) signature of propositional variables, then the set of LTL formulas is defined inductively as follows:

- any $p \in \Sigma$ is a formula,
- if φ and ψ are formulas, then so are $\neg\varphi$, $\varphi \wedge \psi$, and $\varphi \vee \psi$,
- if φ and ψ are formulas, then so are $\bigcirc\varphi$, $\Box\varphi$, $\Diamond\varphi$, $\varphi\mathsf{U}\psi$, and $\varphi\mathsf{R}\psi$.

In LTL we model time as an infinite discrete sequence of time points isomorphic to the set of natural numbers $\mathbb{N}$. An *LTL interpretation* is a sequence $\mathcal{V} = (V_i)_{i\in\mathbb{N}}$ where each V_i is a propositional valuation $V_i : \Sigma \to \{\mathbf{0}, \mathbf{1}\}$. The truth relation $\mathcal{V}, i \models \varphi$ between an interpretation $\mathcal{V}$, time *index* $i \in \mathbb{N}$, and a formula φ is defined recursively according to the rules given in Figure 2.1. An LTL interpretation $\mathcal{V}$ is a *model* of an LTL formula φ if $\mathcal{V}, 0 \models \varphi$. A formula φ is called *satisfiable* if it has a model, and is called *valid* if every LTL interpretation $\mathcal{V}$ is a model of φ.

Separated normal form

Formulas in the Separated Normal Form (SNF) are conjunctions of *temporal clauses*, each temporal clause assuming one of the following forms:

- an *initial* clause: $\bigvee_j k_j$,
- a *step* clause: $\Box(\bigvee_j k_j \vee \bigvee_j \bigcirc l_j)$,

$$
\begin{array}{lrcl}
(1) & \tau[\Box(\neg x \vee l)] & \longrightarrow & \Box(\neg x \vee l), \text{ if } l \text{ is a literal,} \\
(2) & \tau[\Box(\neg x \vee (\varphi \wedge \psi))] & \longrightarrow & \tau[\Box(\neg x \vee \varphi)] \wedge \tau[\Box(\neg x \vee \psi)], \\
(3) & \tau[\Box(\neg x \vee (\varphi \vee \psi))] & \longrightarrow & \Box(\neg x \vee \overline{u} \vee \overline{v}) \wedge \tau[\Box(\neg \overline{u} \vee \varphi)] \wedge \tau[\Box(\neg \overline{v} \vee \psi)], \\
(4) & \tau[\Box(\neg x \vee \bigcirc \varphi)] & \longrightarrow & \Box(\neg x \vee \bigcirc \overline{u}) \wedge \tau[\Box(\neg \overline{u} \vee \varphi)], \\
(5) & \tau[\Box(\neg x \vee \Box \varphi)] & \longrightarrow & \Box(\neg x \vee \overline{u}) \wedge \Box(\neg \overline{u} \vee \bigcirc \overline{u}) \wedge \tau[\Box(\neg \overline{u} \vee \varphi)], \\
(6) & \tau[\Box(\neg x \vee \Diamond \varphi)] & \longrightarrow & \Box(\neg x \vee \Diamond \overline{u}) \wedge \tau[\Box(\neg \overline{u} \vee \varphi)], \\
(7) & \tau[\Box(\neg x \vee (\varphi \mathsf{U} \psi)] & \longrightarrow & \Box(\neg x \vee \Diamond \overline{v}) \wedge \Box(\neg x \vee \overline{v} \vee \overline{w}) \wedge \Box(\neg \overline{w} \vee \overline{u}) \wedge \\
 & & & \Box(\neg \overline{w} \vee \bigcirc \overline{v} \vee \bigcirc \overline{w}) \wedge \tau[\Box(\neg \overline{u} \vee \varphi)] \wedge \tau[\Box(\neg \overline{v} \vee \psi)], \\
(8) & \tau[\Box(\neg x \vee (\varphi \mathsf{R} \psi)] & \longrightarrow & \Box(\neg x \vee \overline{w}) \wedge \Box(\neg \overline{w} \vee \overline{v}) \wedge \Box(\neg \overline{w} \vee \overline{u} \vee \bigcirc \overline{w}) \wedge \\
 & & & \tau[\Box(\neg \overline{u} \vee \varphi)] \wedge \tau[\Box(\neg \overline{v} \vee \psi)].
\end{array}
$$

Figure 2.2: The rules of SNF transformation. Fresh variables are typeset with overline.

- an *eventuality* clause: $\Box(\bigvee_j k_j \vee \Diamond l)$,

where k_j, l_j, and l stand for literals, i.e. propositional variables or their negation.

SNF of a given LTL formula is obtained by applying transformations that 1) introduce new variables as names for complex subformulas, 2) replace complex temporal operators by their fixpoint definitions, and 3) apply classical style rewrite operations to attain the overall structure of a conjunction of the disjunctive temporal clauses. The transformation preserves satisfiability of the input formula and it is ensured that the result does not grow in size by more than a linear factor (Fisher et al., 2001).

We start the SNF transformation by turning the given formula into Negation Normal Form (NNF), which is a form in which the negation sign only occurs in front of propositional variables in the leaves of the formula tree. This can be achieved by an operation that "pushes negations downwards" with the help of De Morgan's laws and temporal equivalences $\neg \bigcirc \varphi \equiv \bigcirc \neg \varphi$, $\neg \Box \varphi \equiv \Diamond \neg \varphi$, $\neg \Diamond \varphi \equiv \Box \neg \varphi$, $\neg(\varphi \mathsf{U} \psi) \equiv (\neg\varphi)\mathsf{R}(\neg\psi)$, and $\neg(\varphi \mathsf{R} \psi) \equiv (\neg\varphi)\mathsf{U}(\neg\psi)$. Finally, multiple negations are absorbed with the help of the classical equivalence $\neg\neg\varphi \equiv \varphi$.

The actual transformation is performed with the help of operator τ defined in Figure 2.2. The operator recursively reduces any formula of the form $\Box(\neg x \vee \varphi)$ into the final SNF. During the process we may need to introduce *fresh* variables, i.e., variables that did not previously occur in the formula. These variables are typeset with overline. They serve two different purposes. Some of them stand as names for subformulas, as, e.g., in the case of the rule (3) for disjunction. They may also play a role of "trackers" that influence the value of other variables not just in the current time point, but also in those to follow. This is how the semantics of, e.g., the Always operator $\Box$ is encoded in rule (5). The overall translation is triggered by the following rule

$$\varphi \quad \longrightarrow \quad \overline{i} \wedge \tau[\Box(\neg \overline{i} \vee \varphi)],$$

where the formula φ is assumed to be already in NNF and $\overline{i}$ is a fresh variable.

Example 2.1. We work out an example by Fisher et al. (2001) to demonstrate the translation procedure. Assume we would like to prove the formula $(\Diamond p \wedge \Box(p \rightarrow \bigcirc p)) \rightarrow \Diamond \Box p$.

In refutational theorem proving we proceed by negating the formula and trying to show the negation to be unsatisfiable. By taking the negation into NNF (and translating away the implication symbol) we obtain

$$(\Diamond p \wedge \Box(\neg p \vee \bigcirc p)) \wedge \Box\Diamond\neg p,$$

which is consequently translated into a conjunction of the following clauses:

i,	By the initial rule.
$\Box(\neg i \vee \Diamond u_1)$,	The first conjunct by rule (6),
$\Box(\neg u_1 \vee p)$,	terminates by rule 1.
$\Box(\neg i \vee u_2)$,	The second conjunct by rule (5),
$\Box(\neg u_2 \vee \bigcirc u_2)$,	...
$\Box(\neg u_2 \vee u_3 \vee v_3)$,	inside which there is a disjunction (rule 3),
$\Box(\neg u_3 \vee \neg p)$,	the first argument is a literal (rule 1),
$\Box(\neg v_3 \vee \bigcirc u_4)$,	the second goes by rule (4)
$\Box(\neg u_4 \vee p)$,	and terminates by rule (1).
$\Box(\neg i \vee u_5)$,	The third conjunct by rule (5),
$\Box(\neg u_5 \vee \bigcirc u_5)$,	...
$\Box(\neg u_5 \vee \Diamond u_6)$,	inside which we apply rule (6),
$\Box(\neg u_6 \vee \neg p)$.	and terminate by rule (1).

Simplifying the eventuality clauses

As part of the transformation of a general LTL formula into the desired final form, the TST, we need two further refinement steps of the intermediate SNF, which focus on eventuality clauses. In particular, we use the techniques of Degtyarev et al. (2002) to

1. turn *conditional* eventuality clauses into *unconditional* ones (of the form $\Box\Diamond l$), and
2. reduce the potentially *multiple* (unconditional) eventuality clauses into *just one* eventuality clause.

Here we present our own compilation of these two refinements, merging them into a single transformation step, and provide an informal argument of its correctness.

Assume that an SNF of a formula contains n (in general) conditional eventuality clauses

$$\Box(C_i \vee \Diamond l_i)$$

for $i = 1, \ldots, n$, where C_i, the condition part, is a disjunction of literals. We remove these clauses and replace them with a single unconditional eventuality clause

$$\Box\Diamond\overline{m} \tag{2.1}$$

together with the following five step clauses for every $i = 1, \ldots, n$:

$$\Box(C_i \vee l_i \vee \overline{t_i}), \tag{2.2}$$

$$\Box(\neg\overline{t_i} \vee \bigcirc l_i \vee \bigcirc\overline{t_i}), \tag{2.3}$$
$$\Box(\overline{s_i} \vee \neg\overline{t_i} \vee \bigcirc\neg\overline{s_i}), \tag{2.4}$$
$$\Box(\neg\overline{s_i} \vee \neg\overline{m}), \tag{2.5}$$
$$\Box(\overline{s_i} \vee \bigcirc\neg\overline{m}), \tag{2.6}$$

where again the overlined variables are meant to be fresh to the formula.

The idea behind the correctness of the transformation is as follows. If the condition $\neg C_i$ is satisfied at the current time point and the respective eventuality l_i is not satisfied at the same time point we start "tracking" the eventuality with the help of the new variable $\overline{t_i}$ (clause 2.2). The tracking variable $\overline{t_i}$ is forced to stay true also in the future time points until the eventuality l_i is finally satisfied (clause 2.3). When looking from the other side we see that the unconditional eventuality (clause 2.1) will infinitely often ensure that all the variables $\overline{s_i}$ are false at one time point (clause 2.5) and true at the previous time point (clause 2.6). Thus in the intervals between the time points where $\overline{m}$ holds, there will always be two consecutive time points where $\overline{s_i}$ changes from false to true. But this cannot happen if we are tracking that particular eventuality at that time (clause 2.4). To sum up, for each of the original eventualities we have a guarantee that in every interval between time points where $\overline{m}$ holds the eventuality was either not triggered at all ($\neg C_i$ was false in the whole interval) or the eventuality was triggered and subsequently satisfied in that interval.

We have just argued that the transformation for obtaining one unconditional eventuality clause preserves satisfiability of the formula and it is also easy to see that the formula does not grow in size more than by a linear factor. The interested reader can consult the paper by Degtyarev et al. (2002) for a formal proof.

Example 2.2. Our previous example contained two conditional eventuality clauses $\Box(\neg i \vee \Diamond u_1)$ and $\Box(\neg u_5 \vee \Diamond u_6)$. We may replace these by the following set of clauses to obtain an equisatisfiable problem with just one unconditional eventuality clause:

$$\Box\Diamond m,$$
$$\Box(\neg i \vee u_1 \vee t_1),$$
$$\Box(\neg t_1 \vee \bigcirc u_1 \vee \bigcirc t_1),$$
$$\Box(s_1 \vee \neg t_1 \vee \bigcirc\neg s_1),$$
$$\Box(\neg s_1 \vee \neg m),$$
$$\Box(s_1 \vee \bigcirc\neg m),$$
$$\Box(\neg u_5 \vee u_6 \vee t_2),$$
$$\Box(\neg t_2 \vee \bigcirc u_6 \vee \bigcirc t_2),$$
$$\Box(s_2 \vee \neg t_2 \vee \bigcirc\neg s_2),$$
$$\Box(\neg s_2 \vee \neg m),$$
$$\Box(s_2 \vee \bigcirc\neg m).$$

Priming notation

Priming notation is a simple conceptual tool that allows us to use purely propositional formulas to talk about several neighboring time points. Instead of using the temporal operator Next $\bigcirc$, we introduce "primed" copies of the basic signature. While the variables from $\Sigma = \{p, q, \ldots\}$ are reserved to describe the "current" time point, we use the variables from the first copy $\Sigma' = \{p', q', \ldots\}$ to describe the time point one step in the future and similarly $\Sigma'' = \{p'', q'', \ldots\}$ for two steps ahead, etc. Multiple primes may be shortened by parenthesized integers: e.g., p''' denotes the same symbol as $p^{(3)}$.

The convention can be extended from symbols and signatures to propositional formulas: by φ' we mean the formula obtained from φ by adding one prime to each occurrence of a variable in φ. Similarly, if V is a valuation over Σ we write V' for the valuation over Σ' such that $V'(p') = V(p)$ for every $p \in \Sigma$. If V_1 and V_2 are two valuations over Σ, we let $[V_1, V_2]$ denote the joint valuation $V_1 \cup (V_2)' : \Sigma \cup \Sigma' \to \{\mathbf{0}, \mathbf{1}\}$. That means, more explicitly

$$[V_1, V_2](x) = \begin{cases} V_1(p) & \text{if } x = p \ \in \Sigma, \\ V_2(p) & \text{if } x = p' \in \Sigma'. \end{cases}$$

Such a valuation is needed to evaluate clauses over the joint signature $\Sigma \cup \Sigma'$.

Temporal satisfiability tasks

Once we have transformed an LTL formula into an SNF with just one unconditional eventuality clause the final step in obtaining a Temporal Satisfiability Task (TST) is just a simple syntactic manipulation. Notice that we deliberately depart from the LTL syntax and instead use purely propositional clauses and the priming notation. The intended temporal semantics is, however, preserved (see below).

Definition 2.4. A *Temporal Satisfiability Tasks* (TST) is a quadruple $\mathcal{T} = (\Sigma, I, T, G)$ such that

- Σ is a finite propositional signature,
- I is a set of *initial* clauses C_i over the signature Σ,
- T is a set of *step* clauses $C_t \vee (D_t)'$ over the joint signature $\Sigma \cup \Sigma'$,
- G is a set of *goal* clauses C_g over the signature Σ.

The initial and step clauses are directly translated from SNF. The goal clauses *all together* express the single eventuality obtained in the previous transformation step. This generalization (from a single unit goal clause that we could obtain) is for free and appears to make the definition more elegant. Intuitively, a TST stands for the LTL formula

$$\left(\bigwedge_I C_i\right) \wedge \Box \left(\bigwedge_T (C_t \vee \bigcirc D_t)\right) \wedge \Box\Diamond \left(\bigwedge_G C_g\right),$$

which directly translates to the following formal definition.

Definition 2.5. An interpretation $\mathcal{V} = (V_i)_{i\in\mathbb{N}}$ is a *model* of $\mathcal{T} = (\Sigma, I, T, G)$ if

- for every $C_i \in I$, $V_0 \models C_i$,
- for every $i \in \mathbb{N}$ and every $C_t \vee (D_t)' \in T$, $[V_i, V_{i+1}] \models C_t \vee (D_t)'$, and
- there are infinitely many indexes j such that for every $C_g \in G$, $V_j \models C_g$.

A TST $\mathcal{T}$ is *satisfiable* if it has a model.

Example 2.3. We will use the following valid LTL formula as a running example that will guide us through the whole theorem proving process presented in this chapter:

$$\Box((a \to b) \to \bigcirc b) \to \Diamond\Box(a \vee b).$$

The formula is chosen in such a way that its negation, which we will try to refute, can be easily transformed[3] into an SNF $\Box(a \vee \bigcirc b) \wedge \Box(\neg b \vee \bigcirc b) \wedge \Box\Diamond(\neg a \wedge \neg b)$, which gives us the following TST $\mathcal{T} = (\{a, b\}, \emptyset, \{a \vee b', \neg b \vee b'\}, \{\neg a, \neg b\})$.

2.3 Labeled superposition

2.3.1 Labeled clauses

In this preparatory section we develop the notion of labeled clauses which will serve as syntactic objects manipulated by our calculus for LTL satisfiability. We first show that satisfiability of a TST can be restated as a hypothetical disjunction of infinitely many purely propositional satisfiability problems over an infinite signature. Labels are then introduced to *finitely* represent and control clauses within these problems, abbreviating entire clause sets. Thus labeled clauses provide a means for effectively transferring reasoning techniques from the propositional level to that of LTL.

Conservative strengthening of the notion of satisfiability

It is a known fact that when considering satisfiability of LTL formulas attention can be restricted to *ultimately periodic* (Sistla and Clarke, 1985) interpretations. These start with a particular finite sequence of valuations and then repeat another finite sequence of valuations forever. This observation motivates the following definition, which is one of the key ingredients of our approach.

Definition 2.6. Let $K \in \mathbb{N}$, and $L \in \mathbb{N}^+ = \mathbb{N} \setminus \{0\}$ be given. An LTL interpretation $\mathcal{V} = (V_i)_{i\in\mathbb{N}}$ is a (K, L)-*model* of a TST $\mathcal{T} = (\Sigma, I, T, G)$ if

- for every $C \in I$, $V_0 \models C$,
- for every $i \in \mathbb{N}$ and every $C \in T$, $[V_i, V_{i+1}] \models C$,

[3] We do not attempt to follow the general SNF transformation presented previously, which would introduce additional variables and clauses and the example would become too complex for our purposes.

- for every $i \in \mathbb{N}$ and every $C \in G$, $V_{(K+i \cdot L)} \models C$.

We will call the pair (K, L) of natural numbers $K \in \mathbb{N}$ and $L \in \mathbb{N}^+$ a *rank* of a model.

Satisfiability within a (K, L)-model for *some* values of K and L is identical to the standard semantics (Definition 2.5) except for the third point. The original condition on the goal clauses to be satisfied at infinitely many indexes is now strengthened and we require that these indexes form an arithmetic progression with K as the initial term and L the common difference. As the following lemma shows this additional restriction does not change the notion of satisfiability.

Lemma 2.1. *Every satisfiable TST $\mathcal{T} = (\Sigma, I, T, G)$ has a (K, L)-model.*

Proof. Let $\mathcal{V} = (V_i)_{i \in \mathbb{N}}$ be an LTL interpretation such that $\mathcal{V} \models \mathcal{T}$. There are only finitely many different valuations over Σ, so only finitely many that satisfy all the clauses $C \in G$. Therefore at least one of them has to appear infinitely often in $(V_i)_{i \in \mathbb{N}}$. Let $K \in \mathbb{N}$ be an index such that V_K is such a valuation and let L be the smallest number in $\mathbb{N}^+$ such that $V_K = V_{K+L}$. We now define a new LTL interpretation $\mathcal{W} = (W_i)_{i \in \mathbb{N}}$ by setting

- $W_i = V_i$ for every $i \leq K$,
- $W_i = V_{K+(i-K) \bmod L}$ for any $i > K$.

It is straightforward to verify that $\mathcal{W}$ is a (K, L)-model of $\mathcal{T}$. □

The interpretation $\mathcal{W}$ constructed in the proof of the previous lemma is, in fact, an *ultimately periodic* model of Sistla and Clarke (1985), i.e., from a certain time point the respective valuations repeat periodically. The notion of a (K, L)-model is more general as it only requires that the goal clauses be satisfied at periodically recurring indexes.

Reduction to pure propositional logic

For a particular choice of K and L, the existence of a (K, L)-model can be reduced to an infinite but purely propositional problem over the infinite signature $\Sigma^* = \bigcup_{i \in \mathbb{N}} \Sigma^{(i)}$. We devise such a reduction in two steps. First, we realize that LTL interpretations naturally correspond to propositional valuations over Σ^*.

Lemma 2.2. *There is a bijection between LTL interpretations over Σ and propositional valuations over Σ^*.*

Proof. For any given LTL interpretation $\mathcal{V} = (V_i)_{i \in \mathbb{N}}$ there is a unique valuation V^* over Σ^* defined by the equation

$$V^*(p^{(i)}) = V_i(p)$$

for every $i \in \mathbb{N}$ and every $p \in \Sigma$. This mapping is obviously surjective. □

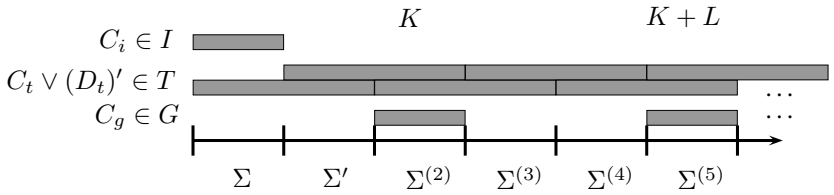


Figure 2.3: Schematic presentation of the potentially infinite set of clauses that is satisfiable if and only if a TST $\mathcal{T} = (\Sigma, I, T, G)$ has a (K, L)-model with $K = 2$ and $L = 3$. The axis represents the infinite signature Σ^*, the gray bars stand for individual copies of the initial, step, and goal clauses, respectively.

In the second step, we simply restate Definition 2.6 in propositional logic. For a given TST $\mathcal{T} = (\Sigma, I, T, G)$ we "copy" the clauses from I, T, and G and "shift them in time" using the priming notation. We obtain a (in general infinite) set of propositional clauses consisting of:

- the set of initial clauses $I = \{C^{(0)} \mid C \in I\}$,
- together with $\{C^{(i)} \mid C \in T, i \in \mathbb{N}\}$,
- and with $\{C^{(K+i \cdot L)} \mid C \in G, i \in \mathbb{N}\}$,

This set of clauses is satisfiable if and only if $\mathcal{T}$ has a (K, L)-model. See Figure 2.3 for an illustration of the situation.

The idea of lifting

We have now reduced LTL satisfiability of a TST $\mathcal{T}$ to infinitely many (for every pair of K and L) infinite propositional problems over Σ^*. We proceed by assigning labels to the clauses of $\mathcal{T}$ such that a *labeled clause* represents up to infinitely many *standard clauses* over Σ^*. Then an inference performed between labeled clauses corresponds to infinitely many inferences on the level of Σ^*. This is similar to the idea of "lifting" from first-order theorem proving where clauses with variables represent up to infinitely many ground instances. Here, however, we deal with the additional dimension of performing infinitely many reasoning tasks on the "ground level" in parallel, one for each pair (K, L).

Definition 2.7. A *label* is a pair (b, k) where b is either $*$ or 0, and k is either $*$ or an element of $\mathbb{N}$. A *labeled clause* is a pair $\mathcal{C} = (b, k) \,||\, C$ consisting of a label (b, k) and a standard clause over $\Sigma \cup \Sigma'$.

Semantics of labels is given via a map to certain sets of time indexes.

Definition 2.8. Let a rank (K, L) be given. We define a set $R_{(K,L)}(b, k)$ of indexes *represented* by the label (b, k) as the set of all $t \in \mathbb{N}$ such that

- $(b \neq * \Rightarrow t = 0)$ and

- $(k \neq * \Rightarrow \exists s \in \mathbb{N} \,.\, t + k = K + s \cdot L)$.

A standard clause of the form $C^{(t)}$ is said to be *represented by the labeled clause* $(b, k) \,||\, C$ *in* (K, L) if $t \in R_{(K,L)}(b, k)$.

The two label components stand for two independent conditions on the time indexes to which the clause relates. The first label component b relates the clause to the beginning of time and the second component relates the clause to the indexes of the form $K + i \cdot L$, where the goal should be satisfied. In both cases, $*$ stands for a "don't care" value, so if b or k equals $*$, the respective condition is trivially satisfied by any index.

New label values are computed from old ones using certain operations when labeled clauses interact in inferences, as detailed in the next section. When, initially, a labeled clause set is constructed from a TST (see Definition 2.9 below) three particular label values are used. Further values arise as results of applying the mentioned operations, and the full generality of labels reflects an entire "closure" of the three initial values under these operations.

Definition 2.9. Given a TST $\mathcal{T} = (\Sigma, I, T, G)$, the *starting labeled clause set* $N_\mathcal{T}$ for $\mathcal{T}$ is defined to contain

- labeled clauses of the form $(0, *) \,||\, C$ for every $C \in I$,
- labeled clauses of the form $(*, *) \,||\, C$ for every $C \in T$, and
- labeled clauses of the form $(*, 0) \,||\, C$ for every $C \in G$.

It is easy to check that for any particular choice of K and L the standard clauses over Σ^* represented by the labeled clauses from the starting labeled clause set $N_\mathcal{T}$ form the purely propositional problem that encodes the existence of a (K, L)-model of $\mathcal{T}$.

Example 2.4. Our example TST $\mathcal{T} = (\{a, b\}, \emptyset, \{a \vee b', \neg b \vee b'\}, \{\neg a, \neg b\})$ contains among others the single literal goal clause $\neg a$. In the starting labeled clause set $N_\mathcal{T}$ this goal clause becomes $(*, 0) \,||\, \neg a$. If we now, for example, fix the same rank $(2, 3)$ as in Figure 2.3, our labeled clause will in that rank represent all the standard clauses $(\neg a)^{(t)}$ with $t \in R_{(2,3)}(*, 0) = \{2, 5, 8, \dots\}$.

Satisfiability of labeled clause sets

Given a set of labeled clauses N let us by $N_{(K,L)}$ denote the set of all standard clauses represented in (K, L) by a labeled clause in N:

$$N_{(K,L)} = \{C^{(t)} \mid \text{labeled clause } (b, k) \,||\, C \in N \text{ and } t \in R_{(K,L)}(b, k)\}.$$

Definition 2.10. A set of labeled clauses N is called (K, L)-*satisfiable* if there is a valuation $V^* : \Sigma^* \to \{\mathbf{0}, \mathbf{1}\}$ which (propositionally) satisfies $N_{(K,L)}$. The set N is called *satisfiable* if it is (K, L)-satisfiable for some rank (K, L).

We summarize the current development of our theory in the following lemma.

Lemma 2.3. *Let $\mathcal{T}$ be a TST and $N_\mathcal{T}$ its starting labeled clause set. Then $\mathcal{T}$ is satisfiable if and only if $N_\mathcal{T}$ is. More precisely, for any rank (K, L) the starting labeled clause set $N_\mathcal{T}$ is (K, L)-satisfiable if and only if $\mathcal{T}$ has a (K, L)-model.*

2.3.2 Calculus LPSup

In this section we present our calculus for labeled clauses LPSup. We continue building on the idea that labeled clauses represent standard clauses from the "ground level" of deciding the existence of (K, L)-models, and show how to "lift" the operation of PSup, a sound and complete calculus for the ground proof tasks, and abbreviate it into a single saturation process on the level of labeled clauses.

We make sure the new calculus retains the valuable features of its originator including ordering restrictions for inferences and an abstract redundancy criterion for justifying concrete reductions. We show that the inference rules of the new calculus are sound and that the proposed reductions are instances of the new redundancy criterion. The completeness result for LPSup is postponed till Section 2.3.4, because it relies on properties of saturated clause sets, which we develop in Section 2.3.3.

Ordering and label merge

The calculus LPSup and the corresponding version of PSup which it lifts from the ground level are parametrized by a common ordering on the infinite signature Σ^*, which is uniformly derived from a given ordering on Σ.

Definition 2.11. Given an ordering $<$ over Σ, its *temporal extension* over $\Sigma^* = \bigcup_{i\in\mathbb{N}} \Sigma^{(i)}$, denoted again by $<$, is defined by

$$p^{(i)} < q^{(j)} \text{ if and only if } i < j, \text{ or } i = j \text{ and } p < q.$$

For the rest of the presentation we fix an ordering $<$ over Σ along with its temporal extension. We use the standard extension of this ordering to compare literals in clauses (recall Section 2.2.1).

Example 2.5. Assume the signature $\Sigma = \{a, b\}$ is totally ordered by $a < b$. Then Σ^* is totally ordered by the corresponding temporal extension as $a^{(0)} < b^{(0)} < a^{(1)} < b^{(1)} < a^{(2)} < \ldots$ and the corresponding standard extension orders the literals over Σ^* as $a^{(0)} < \neg a^{(0)} < b^{(0)} < \neg b^{(0)} < a^{(1)} < \neg a^{(1)} < b^{(1)} < \neg b^{(1)} < a^{(2)} < \ldots$

When two labeled clauses participate in an inference the label of the conclusion is computed by the following operation from the labels of the premises.

Definition 2.12. The *merge* of two labels (b_1, k_1) and (b_2, k_2) is the label (b, k) defined imperatively as follows:

- if $b_1 = *$ and $b_2 = *$ then $b \leftarrow *$ else $b \leftarrow 0$,
- if $k_1 = *$ then $k \leftarrow k_2$ else if $k_2 = *$ then $k \leftarrow k_1$ else if $k_1 = k_2$ then $k \leftarrow k_1$.

In the case when $k_1, k_2 \in \mathbb{N}$ and $k_1 \neq k_2$, the merge operation is *undefined*.

This idea behind label merge is that the labeled premises only interact when they represent standard clauses that interact on the ground level. Moreover, the resulting labeled conclusion represents exactly all the conclusions of the corresponding inferences from the ground level.

Ordered Resolution:

$$\mathcal{I}\frac{(b_1, k_1) \,||\, C \vee a \quad (b_2, k_2) \,||\, D \vee \neg a}{(b, k) \,||\, C \vee D},$$

where the atom a is maximal in C, its complement $\neg a$ is maximal in D, and the merge of labels (b_1, k_1) and (b_2, k_2) is defined and equal to (b, k).

Temporal Shift:

$$\mathcal{I}\frac{(*, k) \,||\, C}{(*, k') \,||\, (C)'},$$

where C is a clause over Σ, and

- $k = *$ and $k' = *$, or
- $k \in \mathbb{N}$ and $k' = k + 1$.

Leap:

$$\mathcal{I}\frac{\{(b, u + i \cdot v) \,||\, C\}_{i \in \mathbb{N}} \text{ derivable from the current clause set } N}{(b, u - v) \,||\, C},$$

where $u \geq v > 0$ are integers and C an arbitrary standard clause (see the main text for details concerning the derivability condition).

Figure 2.4: Inference rules of LPSup.

Lemma 2.4. *Let (b, k) be the merge of the labels (b_1, k_1) and (b_2, k_2), and (K, L) any rank. Then*

$$R_{(K,L)}(b, k) = R_{(K,L)}(b_1, k_1) \cap R_{(K,L)}(b_2, k_2).$$

Proof. The proof is straightforward from the definitions. We check by case analysis that

$$(b_1 \neq * \Rightarrow t = 0) \text{ and } (b_2 \neq * \Rightarrow t = 0)$$

is equivalent to $(b \neq * \Rightarrow t = 0)$, and also that

$$(k_1 \neq * \Rightarrow \exists s \in \mathbb{N} \,.\, t + k = K + s \cdot L) \text{ and } (k_2 \neq * \Rightarrow \exists s \in \mathbb{N} \,.\, t + k = K + s \cdot L)$$

is equivalent to $(k \neq * \Rightarrow \exists s \in \mathbb{N} \,.\, t + k = K + s \cdot L)$, under the condition that ($k_1 = *$ or $k_2 = *$ or $k_1 = k_2$). □

Inference rules and their soundness

The inference rules of LPSup are presented in Figure 2.4. While the Ordered Resolution rule constitutes a labeled analogue of the corresponding rule of PSup, Temporal Shift and Leap are "structural" in nature. We will show that these two latter rules only modify the syntactic form of the clauses, but the underlying set of the represented standard clauses remains the same.

It is important to note that each of the LPSup inference rules preserves the fact that the standard parts of the involved labeled clauses span only the signature $\Sigma \cup \Sigma'$. This follows, in particular, from the restriction on the premise of the Temporal Shift inference to involve clauses with literals only over the signature Σ.

Example 2.6. The starting labeled clause set $N_{\mathcal{T}}$ of our running example contains among others also clauses $(*, *) \,||\, a \vee b'$ and $(*, 0) \,||\, \neg b$. We can apply Temporal Shift to the second clause to obtain $(*, 1) \,||\, \neg b'$. Now b' is the only literal over Σ' in the first clause and therefore maximal. So the first clause and the newly derived one can participate in Ordered Resolution inference with a conclusion $(*, 1) \,||\, a$.

Further explanation is needed for the Leap rule. It is stated as an inference with infinitely many premises and so we only require their *potential* derivability from the current clause set N. The appeal to infinity is just a useful mathematical abstraction. When we discuss the saturation with LPSup in Section 2.3.3, we will show how to employ repetition detection and deduction replaying to make the Leap inference effective.

Soundness of the Ordered Resolution rule is derived from the same property of the corresponding PSup inference, as captured by the following lemma.

Lemma 2.5. *Let (K, L) be a rank. Any standard clause represented in (K, L) by the conclusion of the Ordered Resolution inference of LPSup can be derived by the corresponding PSup inference from some standard clauses represented in (K, L) by the premises of the inference.*

Proof. Let $(C \vee D)^{(t)}$ be a standard clause represented in (K, L) by the conclusion $(b, k) \,||\, C \vee D$ of the Ordered Resolution inference of LPSup with premises $(b_1, k_1) \,||\, C \vee a$ and $(b_2, k_2) \,||\, D \vee \neg a$. This means that $t \in R_{(K,L)}(b, k)$ and thus, by Lemma 2.4, also $t \in R_{(K,L)}(b_1, k_1)$ and $t \in R_{(K,L)}(b_2, k_2)$. Consider the standard clauses $(C \vee a)^{(t)}$ and $(D \vee \neg a)^{(t)}$ represented in (K, L) by the respective labeled premises. If follows from Definition 2.11 that the atom $a^{(t)}$ is maximal in $C^{(t)}$ and its complement $(\neg a)^{(t)}$ is maximal in $D^{(t)}$. Thus the clauses $(C \vee a)^{(t)}$ and $(D \vee \neg a)^{(t)}$ are valid premises of the Ordered Resolution inference of PSup with the conclusion $(C \vee D)^{(t)}$. □

As already indicated, the Temporal Shift and Leap inferences are sound, because they do not introduce any new clauses to the ground level. The next two lemmas formalize this observation.

Lemma 2.6. *Let (K, L) be a rank. Any standard clause represented in (K, L) by a conclusion of a Temporal Shift inference is represented in (K, L) by its premise.*

Proof. Let $(C')^{(t)}$ be a standard clause represented in (K, L) by a conclusion $(*, k') \,||\, (C)'$ of Temporal Shift inference. This means that $t \in R_{(K,L)}(*, k')$. We either have $k = *$ and $k' = *$ or $k \in \mathbb{N}$ and $k' = k+1$. In any case $t+1 \in R_{(K,L)}(*, k)$, and thus $C^{(t+1)} = (C')^{(t)}$ is represented in (K, L) by the premise $(*, k) \,||\, C$ of the inference. □

Lemma 2.7. *Let (K, L) be a rank. Any standard clause represented in (K, L) by a conclusion of a Leap inference is represented in (K, L) by one of its premises.*

Proof. Let $C^{(t)}$ be a standard clause represented in (K, L) by a conclusion $(b, u - v) \,||\, C$ of a Leap inference. This means we have $t \in R_{(K,L)}(b, u - v)$. We need to show that $t \in \bigcup_{i \in \mathbb{N}} R_{(K,L)}(b, u + i \cdot v)$ and thus $C^{(t)}$ is represented in (K, L) by one of the inference's premises. This is equivalent to showing that whenever

$$t + (u - v) = K + s_1 \cdot L$$

for some $s_1 \in \mathbb{N}$, we can find $i, s_2 \in \mathbb{N}$ such that

$$t + (u + i \cdot v) = K + s_2 \cdot L.$$

This can be achieved by setting $i = L - 1$ and $s_2 = s_1 + v$. □

Theorem 2.2 (Soundness of LPSup)**.** *Let N be a set of labeled clauses and $(b, k) \,||\, C$ a labeled clause derivable from N by LPSup. Then for any rank (K, L) and any $t \in R_{(K,L)}(b, k)$ the standard clause $C^{(t)}$ is derivable from $N_{(K,L)}$ by PSup. Moreover, if an empty labeled clause $(b, k) \,||\, \bot$ is derivable from N by LPSup such that $R_{(K,L)}(b, k) \neq \emptyset$, then N is not (K, L)-satisfiable.*

Proof. The first part is proved by induction on the length of the derivation, using Lemmas 2.5, 2.6, and 2.7. The second part then follows from the soundness of PSup. □

Notice that in LPSup the fact that an empty labeled clause $(b, k) \,||\, \bot$ is derived does not necessarily mean that the whole clause set is unsatisfiable. It only rules out those (K, L)-models for which $R_{(K,L)}(b, k)$ is non-empty. This motivates the following notion, which will be later used for the formulation of the completeness result.

Definition 2.13. An empty labeled clause $(b, k) \,||\, \bot$ is called *conditional* if $b = 0$ and $k \in \mathbb{N}$, and *unconditional* otherwise. A set of labeled clauses N is *obviously contradictory* if it contains an unconditional empty clause or if $(0, k) \,||\, \bot$ is in N for every $k \in \mathbb{N}$.

Lemma 2.8. *Any obviously contradictory set of a labeled clauses is unsatisfiable.*

Redundancy and reductions

Abstract redundancy for LPSup lifts the corresponding notion (see Definition 2.1 on page 15) from the ground level to the level of labeled clauses.

Definition 2.14. A labeled clause $(b, k) \,||\, C$ is *redundant* with respect to a set of labeled clauses N, if for any rank (K, L) every standard clause represented by $(b, k) \,||\, C$ in (K, L) is redundant with respect to $N_{(K,L)}$.

We present two example reductions for LPSup in Figure 2.5. These are the labeled analogues of the tautology deletion and clause subsumption, respectively. To prove that they satisfy the above redundancy criterion, we need to show in both cases that the clause missing in the conclusion of the reduction is redundant in the presence of the remaining premises. This is trivial for Tautology deletion and covered by the following lemma for Subsumption.

Tautology deletion:

$$\mathcal{R}\frac{(b,k) \,||\, C \vee l \vee \sim l}{},$$

where the literal $\sim l$ is the complement of l.

Subsumption:

$$\mathcal{R}\frac{(b_1,k_1) \,||\, C \quad (b_2,k_2) \,||\, D}{(b_1,k_1) \,||\, C},$$

where C is a strict subset of D and
the merge of labels (b_1,k_1) and (b_2,k_2) is defined and equal to (b_2,k_2).

Figure 2.5: Possible reduction rules of LPSup.

Lemma 2.9. *Let $(b_1,k_1) \,||\, C$ and $(b_2,k_2) \,||\, D$ be the premises of the Subsumption reduction, i.e., C is a subset of D and the merge of labels (b_1,k_1) and (b_2,k_2) is defined and equal to (b_2,k_2). Then $(b_2,k_2) \,||\, D$ is redundant with respect to $\{(b_1,k_1) \,||\, C\}$.*

Proof. Let (K,L) be a rank and let $D^{(t)}$ be a standard clause represented in (K,L) by $(b_2,k_2) \,||\, D$. This means that $t \in R_{(K,L)}(b_2,k_2)$ and, because the merge of labels (b_1,k_1) and (b_2,k_2) is defined and equal to (b_2,k_2), we obtain from Lemma 2.4 that $R_{(K,L)}(b_2,k_2) \subseteq R_{(K,L)}(b_1,k_1)$, and therefore $t \in R_{(K,L)}(b_1,k_1)$. Thus the standard clause $C^{(t)}$ is represented in (K,L) by $(b_1,k_1) \,||\, C$. Because C is a strict subset of D, we obtain that $C^{(t)} <^c D^{(t)}$ and that $C^{(t)} \models D^{(t)}$. □

The power of abstract redundancy lies in the modularity with which further new reduction rules can be introduced to the calculus. As long as they fit into the framework prescribed by Definition 2.14, they are guaranteed to preserve completeness (to be shown in Section 2.3.4) and its underlying proof need not be revised in any way.

2.3.3 Saturating labeled clause sets

In this section we explain how to turn the calculus LPSup into an effective decision procedure for LTL. We propose a particular saturation strategy and show that, although LPSup derivations can be potentially infinite, the strategy always derives in finitely many steps enough information to reveal whether the given clause set is satisfiable or not.

Because we use proof theoretic arguments in this section, which are "fragile" in comparison to the "static view" represented by the abstract redundancy concept, we will only show how to prove our result for one concrete set of reductions, namely those of Figure 2.5. Adapting the proof to accommodate further reductions, however, should be a straightforward task.

Motivation: a non-terminating derivation with a cycle

Example 2.7. Recall our running example with the starting labeled clause set $N_{\mathcal{T}}$ for the TST $\mathcal{T}$, from which we derived the labeled clause $(*, 1) \,||\, \neg b'$ by the Temporal Shift inference. Ordered Resolution between this clause and the clause $(*, *) \,||\, \neg b \vee b'$ yields $(*, 1) \,||\, \neg b$ to which Temporal Shift is again applicable, yielding $(*, 2) \,||\, \neg b'$. We see that the clause we started with differs from the last one only in the label where the second component k got increased by one. The whole sequence of inferences can now be repeated, allowing us to eventually derive labeled clauses $(*, k) \,||\, \neg b$ and $(*, k) \,||\, \neg b'$ for every $k \in \mathbb{N}^+$.

The example demonstrates how the Temporal Shift inference may cause non-termination when the second label component k of the generated clauses increases one by one. It also suggests, however, that from a certain point the derived clauses do not add any new information and the inferences essentially repeat in cycles. Detecting these repetitions and finitely representing the potentially infinite clause sets is the key idea for obtaining a termination result for our calculus.

Layer-by-layer saturation and repetition detection

For any set of labeled clauses N we define a decomposition of N into *layers* by grouping together clauses with the same second label component k.

Definition 2.15. Let N be a set of labeled clauses. For any $k \in \{*\} \cup \mathbb{N}$ the *k-layer* of N is defined to contain exactly those labeled clauses from N which are of the form $(b, k) \,||\, C$. The symbol k is called the *index* of the layer.

Two sets of labeled clauses L_1 and L_2, in particular, two layers, are said to be *equal up to reindexing* if for every labeled clause $(b, k_1) \,||\, C \in L_1$ there is a labeled clause $(b, k_2) \,||\, C \in L_2$ for some k_2, and, symmetrically, for every labeled clause $(b, k_2) \,||\, C \in L_2$ there is a labeled clause $(b, k_1) \,||\, C \in L_1$ for some k_1.

Lemma 2.10. *There is only finitely many different labeled clauses with a common second label component k and therefore only finitely many different layers up to reindexing.*

Proof. There are $c = 2 \cdot 4^{2|\Sigma|}$ different labeled clauses with a common second label component k in general. When tautologies are not counted the number is $2 \cdot 3^{2|\Sigma|}$: each of the $2|\Sigma|$ atoms of $\Sigma \cup \Sigma'$ is either present positively, negatively, or missing in the clause and there are two possible values, $*$ and 0, for the first label component b.

In any case, the number of different layers up to reindexing is 2^c. □

By a layer-by-layer saturation we denote a process of systematically performing inferences and reductions of LPSup (excluding the Leap inference, which we incorporate later) such that clauses from low index layers are considered before those of high index. It follows from the list of observations presented in Figure 2.6 that each individual layer, starting from the $*$-layer and continuing with $k = 0, 1, \ldots$, can be finitely saturated. Moreover, once we are done processing the clauses with labels of the form (b, k) for a particular $k \in \mathbb{N}$, the k-layer (and also all the preceding ones) will remain constant for

(1) If all the premises of Ordered Resolution or Temporal Shift inference belong to the $*$-layer, so does the conclusion of the inference.

(2) If a premise of Ordered Resolution inference belongs to the k-layer for $k \in \mathbb{N}$, so does the inference's conclusion.

(3) If a premise of Temporal Shift inference belongs to the k-layer for $k \in \mathbb{N}$, the inference's conclusion belongs to the layer with index $(k + 1)$.

(4) If a subsuming labeled clause belongs to the k-layer for $k \in \mathbb{N}$ then so does the subsumed clause.

Figure 2.6: Observations capturing the locality of LPSup with respect to layer indexes.

rest of the process. Lemma 2.10 then implies that after finitely many steps we are bound to obtain two saturated layers of different indexes that are equal up to reindexing.

A detailed pseudocode for layer-by-layer saturation is presented in Algorithm 2.1. Following a standard saturation scheme (Weidenbach, 2001) we maintain two sets of clauses: *WorkedOff* and *Usable*. The former contains clauses which have already participated on inferences and reductions with each other, the latter contains clauses yet to be processed. Initially, *WorkedOff* is empty and *Usable* equals the given clause set N (lines 1 and 2). In each iteration of the main loop a clause $\mathcal{C}$ is selected and removed from *Usable* (line 5). If $\mathcal{C}$ cannot be shown redundant with respect to *WorkedOff* (line 6) all Ordered Resolution inferences between $\mathcal{C}$ and a clause in *WorkedOff* as well as the potential Temporal Shift inference with $\mathcal{C}$ as its premise are performed and their conclusions are collected in the set *Derived* (line 12). The derived clauses then enrich the *Usable* set (line 13) and the clause $\mathcal{C}$ enters *WorkedOff*, potentially removing some of the already present clauses by subsumption (line 14). Notice that layer-by-layer saturation does not check for empty clauses to treat them in a special way. Detecting (un)satisfiability is left for the main procedure (to be described later) to which layer-by-layer saturation works as a subroutine.

The ordering used to prioritize clauses for selection (line 5) extends the natural order on $\mathbb{N}$ to treat $*$ as the smallest element. That is what makes the algorithm proceed in a layer-by-layer fashion. An attempt to detect repetition (lines 8–11) is performed each time a clause from a new layer is picked and so the preceding layers are known to be saturated already. There is also the possibility that all clauses are processed before repetition occurs. Such a case is formally treated as a repetition of two consecutive empty layers (lines 15–20). In any case, the algorithm returns the saturated clauses of *WorkedOff* together with two numbers, offset and period, which indicate the position of the repetition.

Lemma 2.11. *Algorithm 2.1 returns a set of labeled clauses M and numbers offset $o \in \mathbb{N}$ and period $p \in \mathbb{N}^+$ such that*

- *the o-layer of M is equal to the $(o + p)$-layer of M up to reindexing,*

Algorithm 2.1 Layer-by-layer saturation with LPSup

Input:
A set of labeled clauses N

Output:
A set of labeled clauses M derived from N in a layer-by-layer fashion together with an offset $o \in \mathbb{N}$ and a period $p \in \mathbb{N}^+$

1: $\mathit{WorkedOff} \leftarrow \emptyset$
2: $\mathit{Usable} \leftarrow N$
3: $k_{\mathrm{L}} \leftarrow *$ /* The value k of the last processed clause */

4: **while** $\mathit{Usable} \neq \emptyset$ **do**
5: pop a labeled clause $\mathcal{C} = (b, k) \,||\, C$ from Usable with minimal k

6: **if** $\mathcal{C}$ is a tautology or subsumed by a clause in $\mathit{WorkedOff}$ **then**
7: **continue**

8: **if** $k > k_{\mathrm{L}}$ **then** /* Entering new layer */
9: **if** the o-layer and the k_{L}-layer of $\mathit{WorkedOff}$ are equal up to reindexing for some natural number $0 \leq o < k_{\mathrm{L}}$ **then**
10: **return** $\mathit{WorkedOff}$, offset o, and period $(k_{\mathrm{L}} - o)$
11: $k_{\mathrm{L}} \leftarrow k$

12: $\mathit{Derived} \leftarrow \mathit{OrderedResolution}(\mathcal{C}, \mathit{WorkedOff}) \cup \mathit{TemporalShift}(\mathcal{C})$
13: $\mathit{Usable} \leftarrow \mathit{Usable} \cup \mathit{Derived}$
14: $\mathit{WorkedOff} \leftarrow \{\mathcal{D} \in \mathit{WorkedOff} \mid \mathcal{D} \text{ not subsumed by } \mathcal{C}\} \cup \{\mathcal{C}\}$

15: /* Saturated finitely, the repeating layers will be empty */
16: **if** $k_{\mathrm{L}} = *$ **then**
17: $o \leftarrow 0$
18: **else**
19: $o \leftarrow k_{\mathrm{L}} + 1$
20: **return** $\mathit{WorkedOff}$, offset o, and period 1

- *the clause set is saturated by LPSup without Leap, except, possibly, for Temporal Shift inferences with premises in the layer* $(o + p)$*,*
- *the layers with index larger than* $(o + p)$ *are empty.*

Example 2.8. In our running example, the $*$-layer and 0-layer are already saturated. Further layers that we obtain are

$$\{(*, 1) \,||\, \neg a', (*, 1) \,||\, \neg b', (*, 1) \,||\, a, (*, 1) \,||\, \neg b\},$$

$$\{(*, 2) \,||\, a', (*, 2) \,||\, \neg b', (*, 2) \,||\, a, (*, 2) \,||\, \neg b\},$$

$$\{(*, 3) \,||\, a', (*, 3) \,||\, \neg b', (*, 3) \,||\, a, (*, 3) \,||\, \neg b\}.$$

Because the 3-layer is equal to the 2-layer (up to reindexing), layer-by-layer saturation terminates with offset 2 and period 1.

Replaying deductions

When repetition is detected and the layer-by-layer saturation terminates, the returned clause set contains enough information for us to know how the process would continue.

Definition 2.16. Let M be a set of clauses returned by Algorithm 2.1 together with offset $o \in \mathbb{N}$ and period $p \in \mathbb{N}^+$. The *infinite extension* of M is the unique set of labeled clauses M^* such that

- for every $k = *, 0, \ldots, o$ the k-layer of M^* is equal to the k-layer of M, and
- for every $i \in \mathbb{N}^+$ the $(o + i)$-layer of M^* is equal to the $(o + i \bmod p)$-layer of M up to reindexing.

We claim that the infinite extension of M is fully saturated by LPSup without Leap. This can be shown by a deduction replaying technique. The key observation is that the two employed inference rules as well as the two reduction rules are "invariant under transfer from one layer to another". For example, if there was a Temporal Shift inference with a premise $(b, o) \,||\, C$ in the o-layer and a conclusion $(b, o{+}1) \,||\, (C)'$ in the $(o{+}1)$-layer, there is also a Temporal Shift inference with a premise $(b, o + p) \,||\, C$ in the $(o + p)$-layer and a conclusion $(b, o{+}p{+}1) \,||\, (C)'$ to go into the layer with index $(o{+}p{+}1)$. Analogous observation holds for Ordered Resolution and the two reductions.

Moreover, a clause from a k-layer for $k \in \mathbb{N}$ can *only* be derived as either

- a conclusion of a Temporal Shift inference with the premise in the $(k - 1)$-layer,
- or a conclusion of an Ordered Resolution inference where at least one of the premises is in the k-layer and the other possibly comes from the $*$-layer.

Thus, the $(o+p+1)$-layer of M^* can be obtained (from the clauses in the $(o+p)$-layer and the $*$-layer) by replaying the corresponding deductions that were used during layer-by-layer saturation to obtain the $(o+1)$-layer of M (from the clauses in the o-layer and the $*$-layer). This follows from the fact that the o-layer and $(o+p)$-layer of M are equal up to reindexing. Now the argument can be inductively generalized to a $(o+p+i)$-layer of M^* for any $i \in \mathbb{N}$ by replaying the deductions used during layer-by-layer saturation of the $(o+i \bmod p)$-layer of M.

Remark 2.1. The above observation implies that once the $*$-layer is saturated and fixed, the form of the k-layer for $k > 0$ is fully determined by the premises of the Temporal Shift inference in the $(k-1)$-layer, i.e., the clauses of the form $(*, k-1) \,||\, C$ where C is only over the signature Σ. This means that it is sufficient to consider only these, *shiftable* clauses when performing the repetition check in Algorithm 2.1. For the sake of simplicity, we do not incorporate this practical optimization into our theoretical presentation.

The decision procedure

The overall decision procedure for LTL satisfiability based on LPSup uses layer-by-layer saturation as a subroutine. Each time we obtain a clause set M saturated in a layer-by-layer fashion we reason about its infinite extension M^* in order to

- check whether M^* is obviously contradictory (see Definition 2.13), in which case we report unsatisfiability,
- find premises for Leap inferences (see Figure 2.4), such that M can be enriched with their conclusions.

Although these operations formally involve the infinite set M^*, we show that they can be implemented just by referring to its finite representation M.

Pseudocode of the procedure is presented in Algorithm 2.2. The given formula φ is first transformed into a TST $\mathcal{T}$, which in turn gives rise to a starting labeled clause set N_1 (lines 1 and 2). Recall that these transformations, described in Section 2.2.2 and Section 2.3.1, respectively, preserve satisfiability of the given formula. The procedure then runs in a loop (starting on line 3). The purpose of the loop is to 1) saturate the current clause set N_1 in a layer-by-layer fashion (line 4) producing a clause set N_2, 2) test whether N_2^* is obviously contradictory (line 5), which means the given formula φ is unsatisfiable, and 3) enrich the clause set N_2 with conclusions of Leap inferences with premises in N_2^* (lines 7 and 8). If the addition of the Leap conclusions is non-trivial and new clauses have been added (line 9) the loop needs to be run again. In the opposite case, sufficiently many inferences have been performed to allow us to conclude that the given formula φ is satisfiable (line 12).

A remark is in order concerning how the Leap inference is handled (line 7). Given a clause $(b, k) \,||\, C$ from the "periodic part" of N_2, i.e., with k satisfying $o \leq k < o+p$, its corresponding copies $(b, k + i \cdot p) \,||\, C$ for $i \in \mathbb{N}$ are elements of N_2^* and so there can be a Leap inference with these clauses as premises and with a conclusion $\mathcal{C} = (b, k-p) \,||\, C$, provided $k - p \geq 0$. Once this conclusion is added, there can be another

Algorithm 2.2 Deciding LTL satisfiability with LPSup

Input:
An LTL formula φ
Output:
Satisfiability status of φ: either SAT or UNSAT

1: $\mathcal{T} \leftarrow$ a TST obtained from φ
2: $N_1 \leftarrow$ a starting labeled clause set for $\mathcal{T}$
3: **loop**
4: $N_2 \leftarrow$ layer-by-layer saturation of N_1 with offset o and period p
5: **if** $(b, k) \,||\, \bot$ is in N_2 and ($b = *$ or $k = *$), or
$(0, k) \,||\, \bot$ is in N_2 for every $0 \leq k < o + p$ **then**
6: **return** UNSAT /* N_2^* is obviously contradictory */
7: *Leaped* $\leftarrow \{(b, k - j \cdot p) \,||\, C \mid \text{for } o \leq k < o + p, (b, k) \,||\, C \in N_2, j = 1, \ldots, \lfloor k/p \rfloor\}$
8: $N_3 \leftarrow N_2 \cup \{\mathcal{C} \in \textit{Leaped} \mid \mathcal{C} \text{ not subsumed by a clause in } N_2\}$
9: **if** $N_2 \neq N_3$ **then** /* There was a non-trivial addition by Leap */
10: $N_1 \leftarrow N_3$
11: **else**
12: **return** SAT /* N_2 saturated till repetition with respect to LPSup */

Leap inference with premises formed by the new clause $\mathcal{C}$ together with the former premises $(b, k + i \cdot p) \,||\, C$ and with a new conclusion $(b, k - 2p) \,||\, C$. Iteratively, we obtain the general form of all the conclusions: $(b, k - j \cdot p) \,||\, C$ for every $j = 1, \ldots, \lfloor k/p \rfloor$.

Example 2.9. In our example, the infinite extension of the layer-by-layer saturation contains the premises $\{(*, 1 + i) \,||\, a\}_{i \in \mathbb{N}}$ of a Leap inference with conclusion $(*, 0) \,||\, a$. This clause together with the already present $(*, 0) \,||\, \neg a$ yields the empty clause $(*, 0) \,||\, \bot$ by Ordered Resolution. This terminates the overall procedure, because the empty clause is unconditional and, therefore, the overall set becomes obviously contradictory.

Correctness

Because Algorithm 2.2 combines the initial satisfiability preserving transformations with saturation via the sound calculus LPSup, it correctly reports unsatisfiability when it computes an obviously contradictory clause set. Correctness of the satisfiable case relies on the completeness theorem for LPSup, which we present in the next section. The interface point to the completeness theorem is substantiated by the following definition.

Definition 2.17. A set of labeled clauses N is *saturated (up to redundancy) till repetition* with respect to LPSup, if there are numbers *offset* $o \in \mathbf{N}$ and *period* $p \in \mathbf{N}^+$ such that

(1) the o-layer of N is equal to the $(o + p)$-layer of N up to reindexing,

(2) the conclusion of every Ordered Resolution inference with non-redundant (with respect to N) premises in N is either redundant with respect to N or contained in N,

(3) for every Temporal Shift inference with non-redundant (with respect to N) premise $(*, k) \,||\, C$ in N such that $k = *$ or $k < o + p$, its conclusion is contained in N,

(4) for every non-redundant (with respect to N) labeled clause $(b, k) \,||\, C$ in N such that $o \leq k < o + p$ and for every $j = 1, \ldots, \lfloor k/p \rfloor$ the conclusions $(b, k - j \cdot p) \,||\, C$ of the Leap inference are contained in N.

The definition refers to the abstract redundancy notion (Definition 2.14) which is in our case instantiated within the Tautology deletion and Subsumption reductions. It is easy to establish with the help of Lemma 2.11 that when Algorithm 2.2 returns SAT the set N_2 is indeed saturated till repetition and, therefore, has a (K, L)-model according to the completeness theorem (see page 38). To conclude that the algorithm is correct, we just need to verify that this model of the final value of the pseudocode variable N_2 is also a model of the initial value of N_1.

The non-trivial step of the argument lies with the call to layer-by-layer saturation. Notice that Algorithm 2.1 returns as soon as repetition is detected without waiting for all the clauses from the input clause set N to be processed. This is not a problem during the first call, when the input is a starting labeled clause set and so it only contains clauses from the $*$-layer and the 0-layer, which all need to be processed before repetition can be detected. During the later calls, however, there can be clauses from the input clause set N of layers with an index greater than the new value of $o + p$, which are therefore still waiting to be processed when repetition occurs. With appeal to the deduction replaying argument used before we claim that these clauses can be safely ignored, because they could be re-derived in the same way as in the last call of the layer-by-layer saturation and are, therefore, logically entailed by the clauses of the lower index layers, which have been considered.

Remark 2.2. If follows from the above that we could always call layer-by-layer saturation with only the clauses of the $*$-layer and the 0-layer of the input clause set N, because the higher index layer clauses are logically entailed by these. We have chosen our current presentation, because it provides an opportunity for a more efficient implementation. Indeed, with a suitable bookkeeping between the calls each subsequent call of layer-by-layer saturation can only focus on inferences and reductions that have not been performed before, namely on those affecting the conclusions of the last Leap inferences.

Termination

We have already argued that the layer-by-layer saturation always terminates. Although the values of offset o and period p associated with N_2 may change from one loop iteration to another, the index of the repeating layer $(o + p)$ is each time bounded by the constant 2^c, which represents the number of different layers not equal up to reindexing (see Lemma 2.10) and which depends only on the size of the signature Σ.

To show the overall termination of Algorithm 2.2 we need to put a bound on the number of iterations of the loop. The key observation is that there can be only as many non-trivial additions (to any particular layer) by the Leap inference (line 8) as there is different labeled clauses with a common second label component, i.e., as many as c from Lemma 2.10: Any clause added to a particular layer either by the Leap inference or during layer-by-layer saturation, either stays in the layer till the end or is subsumed by another clause, in which case it cannot be reinserted (since the subsumption relation is transitive). Therefore, there can be at most c iterations of the loop.

2.3.4 Completeness and model building

In this section we prove the completeness theorem for LPSup. We show that any set of labeled clauses which is saturated up to redundancy till repetition with respect to LPSup (Definition 2.17) represents in a particularly chosen rank (K, L) a set of standard clauses saturated up to redundancy with respect to PSup (Definition 2.2). This way, we lift the standard completeness of PSup to the level of labeled clauses.

We also discuss the possibility of constructing models of satisfiable sets of labeled clauses. We describe how to utilize the model operator of PSup to process the potentially infinite set of standard clauses represented by a set of labeled clauses. For satisfiable sets saturated by LPSup this gives us a straightforward "backtrack-free" model building procedure.

The section ends with an extensive example which demonstrates the layer-by-layer saturation process described previously as well as the subsequent model building.

Completeness theorem

Before we prove our main result, we need a simple lemma about Leap inferences from the "periodic part" of infinite extensions (Definition 2.16).

Lemma 2.12. *Let N be a set of labeled clauses saturated till repetition with offset o and period p, let N^* be the infinite extension of N, and let $(b, k) \,||\, C$ be a non-redundant clause in N^* such that $k \geq o$. Then the clauses $(b, k - j \cdot p) \,||\, C$ are in N^* for every $j = 1, \ldots, \lfloor k/p \rfloor$.*

Proof. If $k \geq o + p$, the conclusion for $j = 1$ follows from Definition 2.16 and the rest from this very lemma with k replaced by $k - p$ (formally, this is an inductive argument). If $k < o + p$ the conclusion follows from Definition 2.17 item (4). □

Theorem 2.3 (Completeness of LPSup)**.** *Let N be a set of labeled clauses saturated till repetition with offset o and period p and N^* its infinite extension that is not obviously contradictory. Let K be the smallest natural number such that $(0, K) \,||\, \bot$ is not in N^* and let L be the smallest multiple of p that is not smaller than o. Then the set $N^*_{(K,L)}$ does not contain the (standard) empty clause and is saturated up to redundancy with respect to PSup.*

Proof. First note that since N^* is not obviously contradictory, the number K is well-defined. Moreover, if N^* contains an empty labeled clause, it must be of the form $(0, k) \,||\, \bot$ with $k \neq K$. Such a clause can only represent the (standard) empty clause in (K, L) if the equation

$$0 + k = K + s \cdot L$$

has a solution for some $s \in \mathbb{N}$. But $k = K$ is already ruled out and if we had $k = K + s \cdot L$ for some $s \in \mathbb{N}^+$, then, because L is a multiple of p not smaller than o and because an empty clause is never redundant, the labeled clause $(0, K) \,||\, \bot$ would have to be in N^* by Lemma 2.12, which is impossible. Therefore, $N^*_{(K,L)}$ does not contain the empty clause.

To show that $N^*_{(K,L)}$ is saturated up to redundancy with respect to PSup, let us take an Ordered Resolution inference of PSup with premises $C \vee a$ and $D \vee \neg a$ in $N^*_{(K,L)}$ that are non-redundant with respect to $N^*_{(K,L)}$. Recall that for such premises the atom a is maximal in C and its complement $\neg a$ is maximal in D. We claim that the labeled clauses from N^* that represent these premises in (K, L) can be chosen in such a way that they form premises of Ordered Resolution inference of LPSup. In more detail, we claim that there is a labeled clause $(b_1, k_1) \,||\, C_1 \vee a_1$ in N^* that represents $C \vee a$ in (K, L) and a labeled clauses $(b_2, k_2) \,||\, D_2 \vee \neg a_2$ in N^* that represents $D \vee \neg a$ in (K, L) such that

(i) a_1 is maximal in C_1 and $\neg a_2$ is maximal in D_2,

(ii) a_1 is identical to a_2 (this is not obvious as we could also have, e.g., $(a_1)' = a_2$),

(iii) the merge of (b_1, k_1) and (b_2, k_2) is defined and equal to some (b, k).

Because the clauses $C \vee a$ and $D \vee \neg a$ are non-redundant with respect to $N^*_{(K,L)}$, the labeled clauses $(b_1, k_1) \,||\, C_1 \vee a_1$ and $(b_2, k_2) \,||\, D_2 \vee \neg a_2$ are non-redundant with respect to N^*. It follows from Definition 2.17 item (2) that the labeled conclusion $(b, k) \,||\, C_1 \vee D_2$ is either redundant with respect to N^* or contained in N^*. By Lemma 2.4 the labeled conclusion represents the standard conclusion $C \vee D$ in (K, L), which is, therefore, either redundant with respect to $N^*_{(K,L)}$ or contained in $N^*_{(K,L)}$. Thus, to finish the proof, we just need to verify the above claim and, in particular, check the items (i)–(iii).

The ordering constraints of item (i) follow immediately from the corresponding property of the represented standard clauses, because we assume both PSup and LPSup to be parametrized by the same fixed literal ordering $<$ (see Definition 2.11).

To find the right labeled clauses for representing the premises of the PSup inference, such that they satisfy items (ii) and (iii), we will rely on the fact that the set N is saturated till repetition. Assume that the labeled clause $(b_1, k_1) \,||\, C_1 \vee a_1$ represents $C \vee a$ and the labeled clause $(b_2, k_2) \,||\, D_2 \vee \neg a_2$ represents $D \vee \neg a$. This means there are $t_1 \in R_{(K,L)}(b_1, k_1)$ and $t_2 \in R_{(K,L)}(b_2, k_2)$ such that $(a_1)^{(t_1)} = (A_2)^{(t_2)} = a$. Because all labeled clauses are over the signature $\Sigma \cup \Sigma'$, if the atom a_1 is not identical to a_2, it must be the case that $a_1 \in \Sigma$ and $(a_1)'$ equals $a_2 \in \Sigma'$, or symmetrically with a_1 and a_2 exchanged. Let us focus without loss of generality on the first case. We must have $t_1 = t_2 + 1 > 0$ and, therefore, $b_1 = *$. Moreover, because the atom a_1 is maximal in C_1, the whole clause $C_1 \vee a_1$ is necessarily only over the signature Σ. This means

the labeled clause $(b_1, k_1) \,||\, C_1 \vee a_1$ is a valid premise of the Temporal Shift inference. It follows from Definition 2.17 item (3) that the conclusion $(*, k_1') \,||\, (C_1 \vee a_1)'$ of this inference is contained in N^*. The conclusion represents the same clause $C \vee a$ in (K, L) as the premise by Lemma 2.6 and satisfies item (ii) above. We can, therefore, replace the labeled clause $(b_1, k_1) \,||\, C_1 \vee a_1$ by $(*, k_1') \,||\, (C_1 \vee a_1)'$, or, in other words, assume that item (ii) is already satisfied by the labeled clauses $(b_1, k_1) \,||\, C_1 \vee a_1$ and $(b_2, k_2) \,||\, D_2 \vee \neg a_2$.

Let us finally focus on item (iii). The only case when the merge operation is not defined for labels (b_1, k_1) and (b_2, k_2) is when $k_1, k_2 \in \mathbb{N}$ and $k_1 \neq k_2$. Because we assume item (ii) already holds, we have that the indexes t_1 and t_2 are equal, their common value lies in $R_{(K,L)}(b_1, k_1) \cap R_{(K,L)}(b_2, k_2)$, and, therefore, L divides $k_1 - k_2$. Assuming without loss of generality that $k_1 > k_2$, this implies that $k_1 \geq o$ (since $L \geq o$) and so by Lemma 2.12 there is a labeled clause $(b_1, k_2) \,||\, C_1 \vee a_1$ in N^* (since p divides L), which also represents the clause $C \vee a$ in (K, L) by Lemma 2.7. Thus, by replacing the labeled clause $(b_1, k_1) \,||\, C_1 \vee a_1$ by $(b_1, k_2) \,||\, C_1 \vee a_1$ we obtain labeled clauses for representing the premises of the PSup inference that satisfy both items (ii) and (iii). □

Model building

We obtain as a corollary of the above theorem and the completeness of PSup (Theorem 2.1) that the standard clause set $N^*_{(K,L)}$ is satisfiable. In fact, we know that

$$\mathcal{I}^{<^c}(N^*_{(K,L)}) \models N^*_{(K,L)},$$

where $\mathcal{I}^{<^c}$ is the model operator (see Definition 2.3) corresponding to the clause extension $<^c$ of the ordering $<$ on literals over $\Sigma^* = \bigcup_{i \in \mathbb{N}} \Sigma^{(i)}$. To turn this observation into an algorithm we just need to generate the clauses from $N^*_{(K,L)}$ in increasing order and incrementally build a partial interpretation I, collecting the atoms whose truth value has already been decided. This interpretation I (over the signature Σ^*) naturally corresponds to an LTL interpretation $\mathcal{V} = (V_i)_{i \in \mathbb{N}}$, i.e., to a sequence of propositional valuations over Σ (see Lemma 2.2). Because there is only finitely many possible valuations over Σ, we are eventually bound to detect a repetition $V_i = V_j$ between some of the already completed valuations V_i. At this point we stop the construction and output an ultimately periodic interpretation, which is a model[4] of N.

The pseudocode of the model building procedure is detailed in Algorithm 2.3. It iterates over the time indexes i in increasing order (line 3) and each time collects the (finite) set of those "instances of clauses in N^*", i.e., of those standard clauses over Σ^* represented in (K, L) by some clause in N^*, which are relevant for the current index i, i.e., contain a literal over $\Sigma^{(i)}$ (lines 5–8). Note that because the set N is saturated by Leap we only need to pick the labeled clauses from N and not from its infinite extension N^*. Also, because the set N is saturated by Temporal Shift, once $i > 0$, we only need to focus on clauses that contain a primed literal (see line 8).

[4] To be more precise, the corresponding (via Lemma 2.2) valuation $V^* : \Sigma^* \to \{\mathbf{0}, \mathbf{1}\}$ witnesses (K, L)-satisfiability of N (see Definition 2.10) for K equal to the length of the initial segment and L to the period of the ultimately periodic interpretation.

Algorithm 2.3 LTL model building

Input:
A set of labeled clauses N saturated till repetition
such that N^* is not obviously contradictory

Output:
An ultimately periodic LTL interpretation $\mathcal{V} = (V_i)_{i \in \mathbb{N}}$ such that $\mathcal{V} \models N$

compute the rank (K, L) as in the completeness theorem
$I \leftarrow \emptyset$ /* Set of atoms over Σ^*, working as a partial interpretation */

for $i \leftarrow 0, 1, \ldots$ **do**
  /* Collect clauses relevant for the current index i */
  if $i = 0$ **then**
    $R \leftarrow \{C^{(i)} \mid (b, k) \,||\, C \in N, C \text{ over } \Sigma, i \in R_{(K,L)}(b, k)\}$
  else
    $R \leftarrow \{C^{(i-1)} \mid (b, k) \,||\, C \in N, C \text{ has a primed literal}, (i-1) \in R_{(K,L)}(b, k)\}$

  /* Build the next valuation V_i */
  foreach $p \in \Sigma$ ordered by $<$ **do**
    if there is $(C \vee p^{(i)}) \in R$ such that
      $p^{(i)}$ is maximal in C and C is false in I **then**
      $I \leftarrow I \cup \{p^{(i)}\}$
  $V_i \leftarrow \lambda p \in \Sigma \,.\, \text{if } p^{(i)} \in I \text{ then } \mathbf{1} \text{ else } \mathbf{0}$

  /* Check for repetition */
  if $i \geq K$ and $V_i = V_j$ from some $j < K + \lfloor (i - K)/L \rfloor \cdot L$ **then**
    return $\lambda n \in \mathbb{N} \,.\, \text{if } n \leq j \text{ then } V_n \text{ else } V_{j+(n-j) \bmod (i-j)}$

The $\Sigma^{(i)}$-part of the interpretation I is then built following the definition of the standard model operator (lines 10–12). We use the lambda notation to describe the corresponding valuation V_i (line 13). Finally, the repetition check of valuations (line 15) makes sure that between the indexes j and i, which mark the repetition, there is at least one index of the form $K + s \cdot L$, where the goal clauses are satisfied. The resulting ultimately periodic interpretation is again defined using the lambda notation (line 16).

Example

Let $\Sigma = \{a, b\}$ be a signature ordered by $a < b$ and let us consider the following set of labeled clauses N

$$
\begin{array}{ll}
1: & (0, *) \,||\, \neg a \\
2: & (*, 0) \,||\, a \\
3: & (*, *) \,||\, \neg a \vee b \vee a' \\
4: & (*, *) \,||\, a \vee \neg a' \vee \neg b' \\
5: & (*, *) \,||\, \neg b \vee \neg a' \vee \neg b' \\
6: & (*, *) \,||\, \neg a \vee b \vee b' \\
7: & (*, *) \,||\, \neg a \vee \neg a' \vee b'
\end{array}
$$

We first saturate the $*$-layer of N. Ordered Resolution inferences between the pairs of clauses (4,6), (4,7), and (5,6) lead to tautologies. The last remaining pair (5,7) yields the following clause.

$$
8: \qquad (*, *) \,||\, \neg a \vee \neg b \vee \neg a' \qquad\qquad \text{OR(5,7)}
$$

Because the conclusion of the Ordered Resolution inference for the pair (3,8) is a tautology, the $*$-layer is now saturated. The layer-by-layer saturation proceeds as follows.

$$
\begin{array}{rll}
9: & (0, 0) \,||\, \bot & \text{OR(1,2)} \\
10: & (*, 1) \,||\, a' & \text{TS(2)} \\
11: & (*, 1) \,||\, \neg a \vee \neg b & \text{OR(8,10)} \\
12: & (*, 2) \,||\, \neg a' \vee \neg b' & \text{TS(11)} \\
13: & (*, 2) \,||\, \neg a \vee b \vee \neg a' & \text{OR(6,12)} \\
14: & (*, 2) \,||\, \neg a \vee \neg a' & \text{OR(7,12), subsumes 13} \\
15: & (*, 2) \,||\, \neg a \vee b & \text{OR(3,14)} \\
16: & (*, 3) \,||\, \neg a' \vee b' & \text{TS(15)} \\
17: & (*, 3) \,||\, a \vee \neg a' & \text{OR(4,16)} \\
18: & (*, 3) \,||\, \neg b \vee \neg a' & \text{OR(5,16)}
\end{array}
$$

Because both the possible Ordered Resolution inferences for the pairs (3,17) and (3,18) yield a tautology, layer-by-layer saturation terminates by exhausting the set of *Usable* clauses. We make use of Remark 2.1 and only focus on shiftable clauses when looking for layer repetition. Recall that these are clauses of the form $(*, k) \,||\, C$ where C is over Σ. There is no shiftable clause in the 3-layer and, obviously, neither in the empty 4-layer, so we report offset $o = 3$ and period $p = 1$.

Because we saturated our clause set finitely, no Leap inference is needed: the set is automatically saturated till repetition. The set is also not obviously contradictory. We learn from the completeness theorem that we should look for a (K, L)-model with $K = 1$ and $L = 3$.

Below we trace the run of the model building Algorithm 2.3. We show the values of selected variables as they change during the individual iterations. For the variable R we use one additional obvious optimization, which is not mentioned in the pseudocode: we only collect those clauses that are currently false in I and their maximal literal is positive, because only such clauses can later produce a literal into I.

iteration i	relevant clauses R	additions to I	computed V_i
0	$\emptyset$	$\emptyset$	$\{a \mapsto \mathbf{0}, b \mapsto \mathbf{0}\}$
1	$\{a'\}$	$\{a'\}$	$\{a \mapsto \mathbf{1}, b \mapsto \mathbf{0}\}$
2	$\{\neg a' \vee b' \vee a^{(2)}, \neg a' \vee b' \vee b^{(2)}, \neg a' \vee \neg a^{(2)} \vee b^{(2)}, \neg a^{(2)} \vee b^{(2)}\}$	$\{a^{(2)}, b^{(2)}\}$	$\{a \mapsto \mathbf{1}, b \mapsto \mathbf{1}\}$
3	$\{\neg a^{(2)} \vee \neg a^{(3)} \vee b^{(3)}\}$	$\{b^{(3)}\}$	$\{a \mapsto \mathbf{0}, b \mapsto \mathbf{1}\}$
4	$\{a^{(4)}\}$	$\{a^{(4)}\}$	$\{a \mapsto \mathbf{1}, b \mapsto \mathbf{0}\}$

We see repetition is detected after iteration 4 with $V_4 = V_1$. The resulting ultimately periodic interpretation $\mathcal{V}$, which is a model of N, starts with a singleton initial segment V_0 and then infinitely repeats the sequence V_1, V_2, V_3.

2.4 Semantic and syntactic aspects

We have just seen that LPSup is a complete calculus and that it can be turned into a decision procedure for LTL. In this section, we try to give more meaning to what this procedure internally computes, which will later help us relating it to other approaches.

In more detail, we first explain how a TST can be seen as a symbolic description of a Büchi automaton. We then elaborate this result, which is also of independent interest, to give a new semantic perspective to the operation of LPSup. In particular, we introduce a notion of a semantic graph, a graph theoretic analogue of the Büchi automaton. We define the meaning of labeled clauses in terms of the graph and explain how the graph changes during the saturation process. Finally, we study under what conditions does semantic equivalence on the graph side correspond to syntactic equality of sets of labeled clauses. This result is essential for understanding the semantics of repetition detection during layer-by-layer saturation.

2.4.1 TST as a symbolic description of a Büchi automaton

Büchi automata are an extension of finite automata for handling infinite inputs. Formally, a Büchi automaton is a tuple $\mathcal{A} = (Q, Q_I, \delta, Q_G)$, where Q is a finite set of states, $Q_I \subseteq Q$ is the set of initial states, $\delta \subseteq Q \times Q$ is the transition relation, $Q_G \subseteq Q$ is the set of accepting states. A run of $\mathcal{A}$ is an infinite sequence $q_0 q_1 \ldots$ of states such that

$q_0 \in Q_I$ and $(q_i, q_{i+1}) \in \delta$ for every $i \in \mathbb{N}$. A run is accepting it there are infinitely many indexes j such that $q_j \in Q_G$.

So far, we have just captured internal computation of the automaton. To enable processing of inputs, we fix an input domain $\mathcal{D}$ and equip a Büchi automaton $\mathcal{A}$ with a labeling function $l : Q \to 2^{\mathcal{D}}$ which assigns a subset of $\mathcal{D}$ to every state of $\mathcal{A}$. This way we obtain a labeled Büchi automaton $(\mathcal{A}, \mathcal{D}, l)$. Such an automaton accepts an infinite sequence $d_0 d_1 \ldots$ of elements of D if there is an accepting run $q_0 q_1 \ldots$ of the automaton $\mathcal{A}$ such that $d_i \in l(q_i)$ for every $i \in \mathbb{N}$.

Here we are, in particular, interested in automata over the input domain $\mathcal{D} = 2^{\Sigma}$ of all the propositional valuations over the signature Σ. These automata process as an input an infinite sequence of elements of 2^{Σ}, in other words, they process an LTL interpretation. It is well known (Vardi and Wolper, 1994) that for any LTL formula φ there is a Büchi automaton $\mathcal{A}_\varphi$ recognizing models of φ, i.e. an automaton which accepts exactly those LTL interpretations $\mathcal{V} = (V_i)_{i \in \mathbb{N}}$ that are models of φ. The size of such an automaton, i.e. the number of its states, is bounded by $2^{O(|\varphi|)}$, where $|\varphi|$ denotes the size of the formula.

Interestingly, we can interpret a TST $\mathcal{T} = (\Sigma, I, T, G)$ as a *symbolic representation* of such an automaton and our normal form transformation, which turns a general LTL formula to an SNF and further to TST (Section 2.2.2), as an alternative way of obtaining a Büchi automaton for the formula. The states of the represented automaton are formed by the set $Q = 2^{\Sigma}$, i.e. the set of all valuations over Σ, its transition function

$$\delta = \{(V_1, V_2) \mid [V_1, V_2] \models \bigwedge_T (C_t \vee (D_t)')\}$$

contains those pairs of valuations that satisfy the step clauses, and its initial and accepting sets are defined as $Q_I = \{V \mid V \models \bigwedge_I C_i\}$ and $Q_G = \{V \mid V \models \bigwedge_G C_g\}$, respectively. It is easy to check that the models of $\mathcal{T}$ are exactly the accepting runs of this automaton when we label each state by a singleton set containing only itself: $l(q) = \{q\} \subseteq 2^{\Sigma}$.

If we want to obtain models of the original formula φ, which are interpretations only over the original formula's signature $\Sigma_0 \subseteq \Sigma$, it is sufficient to abstract away the value of the auxiliary variables that were introduced during the transformation (i.e., the variables from $\Sigma \setminus \Sigma_0$). Formally, the labeling function then computes a restriction: $l(q) = \{q \restriction \Sigma_0\}$.

Recall how we argued that our normal form transformation does not increase the size of the formula by more than a linear factor, and thus, in particular, $|\Sigma| = O(|\varphi|)$. This means, that it is only the just described last step, when the symbolically represented automaton is made explicit which incurs the inherent exponential blowup.

Example 2.10. Recall the TST $\mathcal{T} = (\{a, b\}, \emptyset, \{a \vee b', \neg b \vee b'\}, \{\neg a, \neg b\})$ from our running example. The corresponding Büchi automaton represented by $\mathcal{T}$ is depicted in Figure 2.7.

Remark 2.3. The described transformation that turns a TST into an explicit automaton has a corresponding inverse-like mapping, which can be useful for constructing example TSTs with specific semantic properties. The mapping takes an automaton $\mathcal{A}$ and produces a TST $\mathcal{T}_\mathcal{A}$ such that $\mathcal{T}_\mathcal{A}$ symbolically describes $\mathcal{A}$.

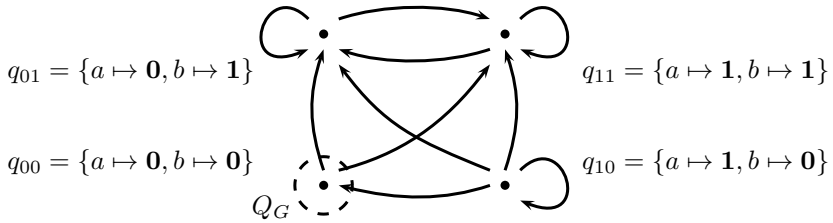


Figure 2.7: Büchi automaton represented by a TST from the running example. The dots stand for the states of the automaton, arrows for its transition relation. The set of its initial states Q_I comprises all the four states of the automaton (not visualized) and the accepting set Q_G equals the singleton set $\{q_{00}\}$.

Given a Büchi automaton $\mathcal{A} = (Q, Q_I, \delta, Q_G)$ let us assume[5] that its set of states Q is of size 2^n such that we can identify[6] Q with the set 2^Σ of valuations over a signature Σ of size n. The constructed TST $\mathcal{T}_\mathcal{A}$ has Σ for its signature. Its initial, step, and goal clause sets are defined, respectively, using the following complementation trick:

$$\begin{aligned} I_\mathcal{A} &= \{C_V \mid V \in 2^\Sigma \; V \notin Q_I\}, \\ T_\mathcal{A} &= \{C_{[V_1,V_2]} \mid V_1, V_2 \in 2^\Sigma \; (V_1, V_2) \notin \delta\}, \\ G_\mathcal{A} &= \{C_V \mid V \in 2^\Sigma \; V \notin Q_G\}, \end{aligned}$$

where the clause C_V for a valuation $V \in 2^\Sigma$ is the unique clause over Σ such that V is the only valuation over Σ that makes it false, or explicitly: $C_V = \{l \text{ literal over } \Sigma \mid V \not\models l\}$.

We see that the size of $\mathcal{T}_\mathcal{A}$ can be bounded by a polynomial in the number of states of $\mathcal{A}$.[7] This means it is exponential in the size of the signature Σ. Although the clause sets can typically be simplified and reduced, we cannot hope for a substantially better encoding in the worst case due to combinatorial reasons (counting argument).

Example 2.11. When we transform the automaton $\mathcal{A}$ from the Figure 2.7 back into a TST, we obtain $\mathcal{T}_\mathcal{A} = (\{a, b\}, I_\mathcal{A}, T_\mathcal{A}, G_\mathcal{A})$, where $I_\mathcal{A} = \emptyset$, $G_\mathcal{A} = \{\neg a \vee b, a \vee \neg b, \neg a \vee \neg b\}$), and the set of step clauses $T_\mathcal{A}$ consists of the following six elements:

$$\begin{array}{lll} a \vee b \vee a' \vee b', & a \vee b \vee \neg a' \vee b', & a \vee \neg b \vee a' \vee b', \\ a \vee \neg b \vee \neg a' \vee b', & \neg a \vee \neg b \vee a' \vee b', & \neg a \vee \neg b \vee \neg a' \vee b'. \end{array}$$

2.4.2 Semantic graphs for labeled clause sets

It is instructive keep following the above correspondence between a TST and its automaton after the TST has been transformed into a set of labeled clauses. We gain new insights by observing the operation of LPSup through the lenses of this new semantics.

[5] We can always pad Q to the nearest power of 2 by adding isolated states.
[6] Formally, we are fixing a bijection between Q and 2^Σ.
[7] The dominant term is the size of $T_\mathcal{A}$, which can contain up to $(2^n)^2$ clauses, each of size $2n$.

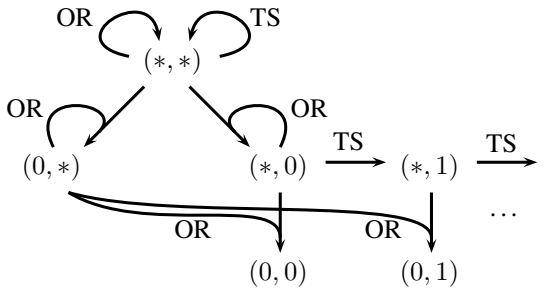


Figure 2.8: Interactions between labels during Ordered Resolution (OR) and Temporal Shift (TS) inferences. Arrows point from the labels of the inference's premises to the label of its conclusion. Three dots denote a potential continuation of the diagram.

For this purpose we gradually develop the notion of a semantic graph for a set of labeled clauses.[8] We analyze soundness of the individual inference rules of LPSup from this perspective, in which we interpret a model as an infinite path through the graph satisfying an analogy of the Büchi acceptance condition. As an interesting by-product, we explain the meaning of the different empty labeled clauses of LPSup. This will provide us with a classification of different ways in which a TST can be unsatisfiable.

Although the motivation for introducing semantic graphs is different, they are related to behavior graphs of Degtyarev et al. (2002) used in the completeness proof of Clausal Temporal Resolution (Fisher et al., 2001).

The semantic graph

Let us start by introducing two simple concepts which will help us streamline the subsequent exposition. By a (b,k)-*clause* we mean any labeled clause $\mathcal{C}$ of the form $(b,k) \,||\, C$. A labeled clause $\mathcal{C}$ is called *simple*, if its standard part C is only over the signature Σ.

The *semantic graph* for a set of labeled clauses N is a directed graph with the set of *vertexes* identified with 2^Σ, i.e. the set of all valuations over the basic signature Σ, and with the *edge relation* $\mathcal{E}_N \subseteq 2^\Sigma \times 2^\Sigma$ containing exactly those pairs of vertexes that satisfy the $(*,*)$-clauses of N:

$$\mathcal{E}_N = \{(V_1, V_2) \mid [V_1, V_2] \models C \text{ for every } (*,*) \,||\, C \in N\}.$$

We will now gradually extend the notion of the semantic graph with additional components as we go along. The plan is to start by looking at the $(*,*)$-clauses and consider the other label types one by one only after we have understood the meaning of the currently considered subset. We do this in a way that respects the interactions between the labels (as shown in Figure 2.8) such that the considered subset of label types is always closed under Ordered Resolution and Temporal Shift. Initially, we do not consider the Leap inference, the meaning which is explained as the last step.

[8] For convenience, we switch from automata theoretic to graph theoretic terminology.

Ordered Resolution and the $(*,*)$-clauses

First note that Ordered Resolution inferences of LPSup on $(*,*)$-clauses directly correspond to Ordered Resolution inferences of PSup on their respective standard parts. This allows us use results about PSup to infer properties of sets of labeled clauses.

Performing an Ordered Resolution inference on $(*,*)$-clauses of N (and adding the respective conclusion to N) does not change the edge relation $\mathcal{E}_N$ of the semantic graph, because the inference is sound. Saturation by Ordered Resolution, however, derives enough information to decide which vertexes have successors and which do not.

We define the set of *source vertexes* $\mathcal{S}_N \subseteq 2^\Sigma$ of the semantic graph as

$$\mathcal{S}_N = \{V \mid V \models C \text{ for every simple } (*,*) \,||\, C \in N\}.$$

It follows from this definition that the set $\mathcal{S}_N$ of source vertexes changes, namely shrinks, only when a new simple $(*,*)$-clause is derived from non-simple parents. The following lemma characterizes the final value of $\mathcal{S}_N$ obtained by saturation.

Lemma 2.13. *When N is saturated by Ordered Resolution, the set $\mathcal{S}_N$ consists exactly of those vertexes of the semantic graph that have out-degree at least one:*

$$\mathcal{S}_N = \{V_1 \mid \exists\, V_2 \text{ such that } (V_1, V_2) \in \mathcal{E}_N\}.$$

Proof (idea). When a pair of vertexes forms an edge $(V_1, V_2) \in \mathcal{E}_N$, the first vertex V_1 necessarily satisfies the simple $(*,*)$-clauses of N by definition. For the opposite inclusion, i.e. the claim

$$\forall\, V_1 \in \mathcal{S}_N \;\exists\, V_2 \text{ such that } (V_1, V_2) \in \mathcal{E}_N,$$

we require that N be saturated by Ordered Resolution. The argument relies on the completeness of PSup (Theorem 2.1) and on the fact that in our literal ordering any literal over Σ is smaller than a literal over Σ'. Here are its main ingredients.

- We work with a set $N^{(*,*)}$ consisting of the standard parts of the $(*,*)$-clauses of N, which we assume to be saturated up to redundancy (with respect to PSup). We aim at using the model operator (Definition 2.3) to find the required edge.
- Given $V_1 \in \mathcal{S}_N$ we enrich $N^{(*,*)}$ by all the unit clauses (clauses with just one literal) true in V_1:
$$M = N^{(*,*)} \cup \{l \text{ literal over } \Sigma \mid V_1 \models l\}.$$
- The key observation is that the set M is also saturated up to redundancy. Indeed, consider an Ordered Resolution inference between one of the added literals l and a clause $(C \vee \sim l) \in N^{(*,*)}$. Because $\sim l$ is maximal in C, the clause C is only over Σ. It must be subsumed (and therefore redundant) by some of the other added literals, because otherwise C would be false in V_1.
- By the completeness theorem we obtain a model $\mathcal{I}^{<^c}(M)$ of M which is necessarily of the form $[V_1, V_2]$ for some vertex V_2 thanks to the presence of the added literals.

□

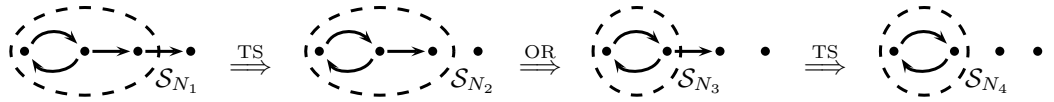


Figure 2.9: Progressive changes of the semantic graph caused by alternate saturation by the Temporal Shift (TS) and Ordered Resolution (OR) inferences. Temporal Shift effectively deletes edges to vertexes with out-degree zero, Ordered Resolution then computes the new set of source vertexes. No edge leaves $\mathcal{S}_N$ in the fixpoint.

Adding the Temporal Shift inference

The effect of a Temporal Shift inference on the semantic graph can be described as follows. We take a property C of the source vertexes in the form of a simple clause $(*,*) \,||\, C$ and assert it to also hold for the edges' targets by deriving $(*,*) \,||\, (C)'$. This way we effectively delete edges to vertexes with out-degree zero, i.e. to vertexes without an outgoing edge. Note that such vertexes cannot lie on an infinite path through the semantic graph and, therefore, can be safely discarded.

The fact that the labeled clause set N has been saturated by both Ordered Resolution and Temporal Shift means that the above operation has been run to a fixpoint and every "target vertex" is also a source vertex:

$$\mathcal{E}_N \subseteq \mathcal{S}_N \times \mathcal{S}_N.$$

See Figure 2.9 for an illustration.

What if there is no source vertex in the semantic graph of the final saturated labeled clause set N? It follows from completeness of PSup that N must in that case contain the empty clause $(*,*) \,||\, \bot$. This means, from a dynamic perspective, that the empty clause $(*,*) \,||\, \bot$ is derivable from N if and only if the semantic graph for N does not contain a cycle (or a self-loop). In this sense, LPSup on the $(*,*)$-fragment decides existence of cycles in the represented semantic graph.

Adding the $(0,*)$-clauses

Recall that the $(0,*)$-clauses of a starting labeled clause set directly correspond to the initial clauses of a TST (Definition 2.9). As such they are necessarily simple and this property is preserved during inferences of LPSup. Also note that new $(0,*)$-clauses arise as conclusions of Ordered Resolution inference either between two $(0,*)$-clauses or between a $(0,*)$-clause and a $(*,*)$-clause (see Figure 2.8). Therefore, it is natural to consider the $(*,*)$-clauses implicitly present when formalizing what the $(0,*)$-clauses represent.

We define the *initial vertexes* $\mathcal{I}_N \subseteq 2^\Sigma$ of the semantic graph for N as

$$\mathcal{I}_N = \mathcal{S}_N \cap \{V \mid V \models C \text{ for every } (0,*) \,||\, C \in N\}.$$

The initial vertexes are those source vertexes that satisfy the $(0,*)$-clauses.

It follows from soundness of Ordered Resolution that $\mathcal{I}_N$ does not change during inferences unless $\mathcal{S}_N$ does.[9] After $\mathcal{S}_N$ stabilizes, i.e. the $(*,*)$-clauses become saturated, the set of initial vertexes $\mathcal{I}_N$ may become empty. In that case we eventually derive the empty clause $(0,*) \,||\, \bot$ thanks to completeness. In fact, the empty clause $(0,*) \,||\, \bot$ is derivable from N if and only if there is no cycle reachable from an initial vertex in the semantic graph for N.

Adding the $(*,0)$-clauses

In a complete analogy to the previous case we define the *goal vertexes* $\mathcal{G}_N$ of the semantic graph for N as the subset of those source vertexes that satisfy the $(*,0)$-clauses:

$$\mathcal{G}_N = \mathcal{S}_N \cap \{V \mid V \models C \text{ for every } (*,0) \,||\, C \in N\}.$$

We again rely on the fact that the respective labeled clauses, here the $(*,0)$-clauses, are always simple: both those from the starting clause set and the derived ones.

We see, in analogy with the case of the initial vertexes, that when the $(*,*)$-clauses are saturated, the set of goal vertexes does not change during inferences and the emptiness of $\mathcal{G}_N$ is equivalent to derivability of the empty clause $(*,0) \,||\, \bot$. Its presence indicates that there is no goal vertex that can reach a cycle, i.e. a goal vertex lying on an infinite path through the semantic graph.

Unlike the case of $(0,*)$-clauses, Temporal Shift inference is applicable to the (always simple) $(*,0)$-clauses and yields a $(*,1)$-clause as its conclusion. Similarly, a simple $(*,1)$-clause gives rise to a $(*,2)$-clause, etc. (again recall Figure 2.8). This suggests we define for any $k \in \mathbb{N}$ the set of *goal vertexes of order* k as

$$\mathcal{G}^k_N = \mathcal{S}_N \cap \{V \mid V \models C \text{ for every simple } (*,k) \,||\, C \in N\}.$$

Then the above introduced notion of the set of goal vertexes $\mathcal{G}_N$ is just a short name for $\mathcal{G}^0_N$, the set of goal vertexes of order 0.

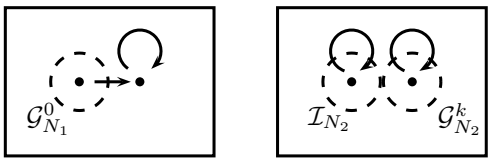
The same technique we used before to reason about the source vertexes (the proof of Lemma 2.13) can now be applied to deduce that for a saturated set of labeled clauses N the goal vertexes of order $k+1$ are exactly those vertexes that can reach a goal vertex of order k in one step:

$$\mathcal{G}^{k+1}_N = \{V_1 \mid \exists\, V_2 \in \mathcal{G}^k_N \text{ such that } (V_1, V_2) \in \mathcal{E}_N\}.$$

This is perhaps the single most interesting observation of this section. It shows that by employing Ordered Resolution (and Temporal Shift) we effectively compute the preimage operation on the symbolically represented sets of vertexes.

By induction on k we obtain that the goal vertexes of order k are exactly those vertexes that can reach a goal vertex (of order 0) in k steps. This means that an empty clause of the form $(*,k) \,||\, \bot$ signals a situation when there is no vertex that can reach a goal vertex in k steps and an empty clause of the form $(0,k) \,||\, \bot$ means that there is no initial vertex that can reach a goal vertex in k steps. See Figure 2.10 for two illustrative examples.

[9] Note that the Temporal Shift inference does not apply here, because $(0,*)$-clauses are not shiftable.



$(*, 1) \,||\, \bot$ $(0, k) \,||\, \bot$

Figure 2.10: Semantic graphs of two labeled clause sets N_1 and N_2. In the first case, there is no goal vertex of order 1 and so the empty clause $(*, 1) \,||\, \bot$ can be derived. In the second case, the set of goal vertexes of order k is non-empty for every k, in fact all the sets $\mathcal{G}^k_{N_2}$ are equal, but none of these intersects with the set of initial vertexes $\mathcal{I}_{N_2}$, and so the empty clauses $(0, k) \,||\, \bot$ are derivable for any k.

Adding the Leap inference

We have just seen that LPSup without the Leap inference analyzes single-time reachability in the semantic graph for N. It gradually answers the question: Is there a path in the semantic graph from an initial vertex to a goal vertex? Only when the Leap inference is added to the calculus the investigated question changes to reaching a goal vertex infinitely many times.

Recall that in order to apply the Leap inference we saturate the clause set in a layer-by-layer fashion and look for a repetition in our derivation (as detailed in Section 2.3.3). In particular, we look for numbers offset $o \in \mathbb{N}$ and period $p \in \mathbb{N}^+$ such that the o-layer and $(o+p)$-layer of N are equal up to reindexing. Thus on the semantic side a necessary condition for the Leap inference to apply (we will later discuss under what conditions it is also sufficient) is when the set of goal vertexes of order o and $o + p$ are equal.

Let us now study the effect of the Leap inference. We look at the "bulk" manifestation of the inference as used in Algorithm 2.2 (line 7), where full sets of clauses are copied from high index layers to strengthen low index layers. For simplicity, we only focus on the 0-layer of the clause set and so, correspondingly, to what happens to the goal vertexes $\mathcal{G}_N$ (of order zero). We obtain the following assignment as the semantic effect of the Leap inference:

$$\mathcal{G}_N \leftarrow \mathcal{G}_N \cap \mathcal{G}^r_N, \tag{2.7}$$

where r is the only multiple of p in the range $o \leq r < o + p$.

Soundness of Leap in terms of the semantic graph is the observation that the inference never deletes goal vertexes that lie on a cycle. Because only a vertex on a cycle can be reached infinitely many times, Leap never precludes a potential model path. A formal proof of this claim could go along the following lines.

Because of the detected repetition, we know that all the sets $\mathcal{G}^{r+i\cdot p}_N$ for $i \in \mathbb{N}$ are equal. Let us take a vertex $V \in \mathcal{G}_N$ that lies on a cycle of length $l \in \mathbb{N}^+$ in the semantic graph for N. For such a vertex we necessarily have $V \in \mathcal{G}^{j\cdot l}_N$ for every $j \in \mathbb{N}$. Because r is a multiple of p we may choose $i, j \in \mathbb{N}$ such that $r + i \cdot p = j \cdot l$. This shows that $V \in \mathcal{G}^{r+i\cdot p}_N$ and therefore $V \in \mathcal{G}^r_N$. We conclude that the vertex V remains in $\mathcal{G}_N$ after performing the Leap inference, which effectively executes the assignment (2.7).

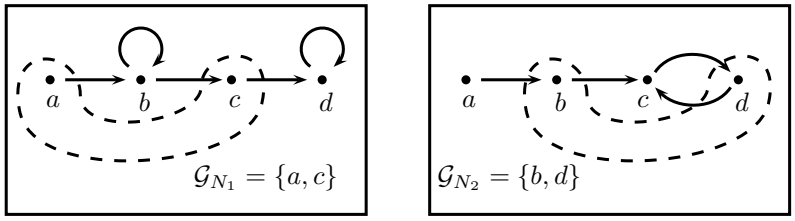


Figure 2.11: Demonstrating the Leap inference mechanics. The first example, with clause set N_1, is unsatisfiable. Its goal vertex sets are $\mathcal{G}_{N_1} = \mathcal{G}^0_{N_1} = \{a, c\}, \mathcal{G}^1_{N_1} = \{b\}, \mathcal{G}^2_{N_1} = \{a, b\}$, and then repetition occurs with $\mathcal{G}^3_{N_1} = \{a, b\}$. In a first round, the Leap inference effectively deletes the vertex c from $\mathcal{G}_{N_1}$ as it sets $\mathcal{G}_{N'_1} \leftarrow \mathcal{G}_{N_1} \cap \mathcal{G}^2_{N_1}$. After resaturation we obtain $\mathcal{G}_{N'_1} = \mathcal{G}^0_{N'_1} = \{a\}$, and $\mathcal{G}^1_{N'_1} = \emptyset$. The second example, with the clause set N_2, demonstrates that the Leap inferences does not necessarily delete every vertex that does not lie on a cycle. We obtain $\mathcal{G}_{N_2} = \mathcal{G}^0_{N_2} = \{b, d\}, \mathcal{G}^1_{N_2} = \{a, c\}$, and $\mathcal{G}^2_{N_2} = \{b, d\}$. Leap (with period 2) does not change the set of goal vertexes here.

Let us note that a single Leap inference does not suffice to detect and remove all the vertexes of the semantic graph that do not lie on a cycle. Moreover, not all vertexes that do not lie on a cycle are removed. Figure 2.11 showcases the respective situations.

2.4.3 On uniqueness of saturations

When two layers of clauses are equal up to reindexing the corresponding sets of goal vertexes (of respective orders) are necessarily also equal. We would like to establish under what conditions does also the opposite implication hold, i.e., when can we expect to obtain syntactically equal representations of semantically equivalent sets of vertexes.

Similarly to Section 2.3.3 the argument we provide here is proof theoretic and focuses solely on Ordered Resolution as the inference (Temporal Shift is handled explicitly) and on Tautology Deletion and Subsumption as reductions. The proof would need to be reexamined and revised should additional reductions be added to LPSup.

Saturation is a non-deterministic process during which we are *required* to perform certain inferences and *allowed* to perform certain reductions. Here we will insist that even reductions are performed *exhaustively*, meaning that no opportunity remains for performing a reduction in the final saturated set. Obviously, we cannot expect to obtain syntactically equal sets of clauses if we on one side, e.g., perform a certain subsumption and on the other side leave the corresponding subsumed clause in place.

Formulating a theorem

We assume a fixed ordering $<$ on Σ for constraining Ordered Resolution inferences. Let N be set of $(*, *)$-clause that is saturated by LPSup. Further, let N_1, N_2 be two sets of simple $(*, k)$-clauses such that $\mathcal{G}^k_{(N \cup N_1)} = \mathcal{G}^k_{(N \cup N_2)}$. For $i = 1, 2$, let N'_i denote the set of

corresponding shifted versions of clauses of N_i, i.e.,

$$N_i' = \{(*, k+1) \,||\, (C)' \;\mid\; (*, k) \,||\, C \in N_i\},$$

and let N_i^* denote a set of $(*, k+1)$-clauses obtained from $(N \cup N_i')$ by exhaustive saturation by the Ordered Resolution inference and the Tautology Deletion and Subsumption reductions. Finally, let $\overline{N_i^*}$ be the subset of simple clauses from N_i^*. Our claim is that despite the potentially different syntactic representations N_1 and N_2 of the same set of goal vertexes of order k and despite the non-determinism involved in the saturation process, the sets $\overline{N_1^*}$ and $\overline{N_2^*}$ are necessarily equal.

Derivability relation as a rewrite system

We split the proof of our theorem into two parts. In this first, preparatory part, we explain how to model LPSup derivations as a rewrite system (see, e.g., Baader and Nipkow, 1998, for an overview) and use the rewriting theory to obtain a normalization result for saturations, which will later on help us to focus on derivations of a particular shape. We structure this first part into several steps.

First, we define the *derivability relation* on sets of clauses. Two sets N_1, N_2 are in the relation, written $N_1 \rhd N_2$, if the set N_2 can be obtained from N_1 by adding a conclusion of the Ordered Resolution inference with premises in N_1, by deleting a tautology in N_1, or by deleting a clause from N_1 subsumed by another (different) clause in N_1. We can fix the details of the definition in such a way that the derivability relation $\rhd$ be *terminating*. This is achieved by stipulating, without the loss of generality, that tautologies and subsumed (or present) clauses cannot be (re-)derived.

Next, we show that the derivability relation is *confluent*. This means that for any sets N_1, N_2, N_3 such that $N_1 \rhd^* N_2$ and $N_1 \rhd^* N_3$, where $\rhd^*$ is the reflexive and transitive closure of $\rhd$, there is a set N_4 such that $N_2 \rhd^* N_4$ and $N_3 \rhd^* N_4$. By Newman's lemma (Newman, 1942) and relying on the fact that our relation is terminating, it is sufficient to show only *local confluence*, a property which requires that for any sets N_1, N_2, N_3 such that $N_1 \rhd N_2$ and $N_1 \rhd N_3$, there is N_4 with $N_2 \rhd^* N_4$ and $N_3 \rhd^* N_4$. This is done by case analysis over the possible ways in which the sets N_2 and N_3 can be derived (for local confluence in one step) from N_1. As an example, we present here the most interesting case, the combination of Ordered Resolution and subsumption, leaving the other cases to the reader.

We work on the level of PSup relying on the obvious correspondence to LPSup. Let us assume that the set N_2 is derived from N_1 by adding the conclusion $C \vee D$ of Ordered Resolution inference with premises $C \vee a$ and $D \vee \neg a$. Moreover, assume that N_3 arises from N_1 by deleting the premise $C \vee a$. (Note that we can choose $C \vee a$ over $D \vee \neg a$ without loss of generality. Also note that when the deleted clause is not equal to one of the two premises, the case becomes trivial.) The premise $C \vee a$ was deleted in N_3, because it is subsumed by a clause $C_1 \subset (C \vee a)$. We consider two subcases:

- $C_1 \subset C$. Then we set $N_4 = N_3$ and arrive from N_2 to N_4 in two subsumption steps using $C_1 \subset (C \vee a)$ to delete the said premise and $C_1 \subset (C \vee D)$ to delete the previously added conclusion.

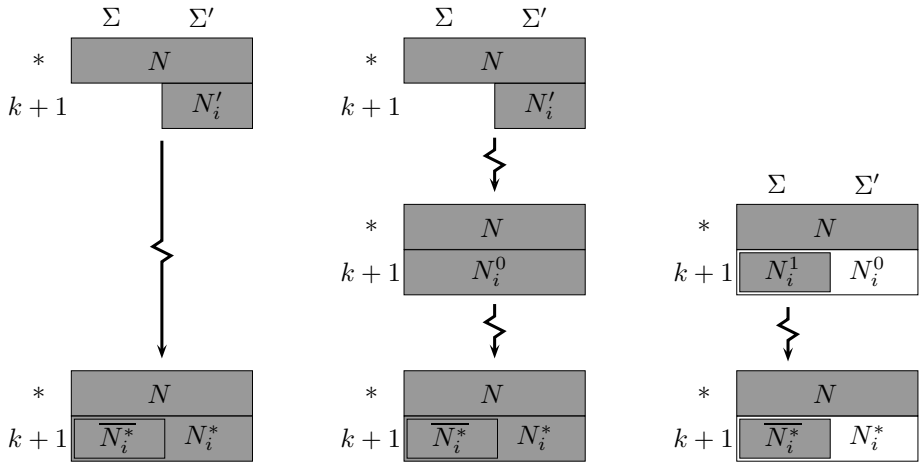


Figure 2.12: Illustrating the proof of the uniqueness of saturations.

- C_1 is of the form $C_0 \vee a$ and $C_0 \subset C$. Because the literal a must be maximal in $C \vee a$, we know it is also maximal in $C_0 \vee a$. Thus there is an Ordered Resolution inference with $C_0 \vee a$ and $D \vee \neg a$ as premises. We define N_4 as N_3 with the conclusion $C_0 \vee D$ of this inference added. We arrive from N_2 to N_4 in three steps. First, we repeat the step by which N_3 was derived from N_1, i.e., we delete $C \vee a$, which is subsumed by C_1. Then we repeat the above Ordered Resolution inference and add $C_0 \vee D$ to the set, and, finally, we delete the old conclusion $C \vee D$ as it is now subsumed by $C_0 \vee D$.

When a rewriting system is both confluent and terminating it is called *convergent*. In a convergent rewriting system every object has a unique *normal form*. For our case of the derivability relation $\rhd$ this means that for every set of clauses N_1 there is a unique set N_2 such that 1) $N_1 \rhd^* N_2$ and 2) there is no N_3 for which $N_2 \rhd N_3$. Informally, the final result of the saturation process does not depend on the order in which inferences and reductions are performed. We will slightly generalize this result for our main proof.

Definition 2.18. Let N_1, N_2 be two sets of clauses. We say that N_1 subsumes N_2, if for every $C \in N_2$ there is $C_0 \in N_1$ such that C_0 subsumes C. We say that N_1 and N_2 are subsumption equivalent, if N_1 subsumes N_2 and N_2 subsumes N_1.

Lemma 2.14. *Subsumption equivalent sets of clauses have identical normal form.*

Proof. Let the sets N_2 and N_3 be subsumption equivalent. Set $N_1 = (N_2 \cup N_3)$ and check that $N_1 \rhd^* N_2$ and $N_1 \rhd^* N_3$. □

Proof tree analysis

Let us now focus back on the theorem about uniqueness of saturations. We start with a set N of $(*,*)$-clause that is saturated by LPSup and two sets N_1, N_2 of simple $(*,k)$-clauses such that $\mathcal{G}^k_{(N \cup N_1)} = \mathcal{G}^k_{(N \cup N_2)}$. We then, for $i = 1, 2$, separately saturate $(N \cup N'_i)$

to obtain $(N \cup N_i^*)$, where N_i' contains the shifted versions of clauses from N_i, and N_i^* denotes the $(*, k+1)$-part of the saturated set (the set of $(*,*)$-clauses N does not change anymore). We want to study the sets $\overline{N_i^*}$ of simple clauses from N_i^*. See Figure 2.12 on the left.

Because we now know that the result of saturation is unique, we can assume that the derivation $(N \cup N_i') \rhd^* (N \cup N_i^*)$ is of a particular shape. We imagine a saturation strategy that first performs all the resolutions and only then deletes all the tautologies and subsumed clauses. This way we split the saturation into two parts

$$(N \cup N_i') \rhd^* (N \cup N_i^0) \rhd^* (N \cup N_i^*),$$

where the first part $(N \cup N_i') \rhd^* (N \cup N_i^0)$ consist of additions only and the second part $(N \cup N_i^0) \rhd^* (N \cup N_i^*)$ consist only of deletions. This is illustrated in Figure 2.12 in the middle column.

We now analyze parent-child relations between the clauses of N_i^0. We say that a clause C is a *parent* of a clause D (and D is a *child* of C) if C is a premise and D the conclusion of Ordered Resolution inference in our derivation. Note that some clauses from N_i^0 have only one parent in N_i^0, the other parent being a $(*,*)$-clause from N. Clauses from N_i^0 without any parent are those from the starting set $N_i' \subseteq N_i^0$.

Of special interest to us are simple $(*, k+1)$-clauses of N_i^0 that do not have a simple parent. Let us denote their set N_i^1. Note that if a premise of Ordered Resolution is simple, then so is the other premise and the conclusion. This means that $(N \cup N_i^1) \rhd^* (N \cup \overline{N_i^*})$, i.e., the set N_i^1 already contains all the clauses needed to derive the simple $(*, k+1)$-clauses $\overline{N_i^*}$ we are interested in (as indicated in Figure 2.12 on the right).

Our current plan is to show that the sets N_1^1 and N_2^1 are subsumption equivalent and from that to conclude that the sets $\overline{N_1^*}$ and $\overline{N_2^*}$ are equal. In what follows we assume the literals of every labeled clause $\mathcal{D}$ to be separated, as in $\mathcal{D} = (b, k) \,||\, D_l \vee (D_u)'$, into a *lower*, non-primed part D_l and *upper*, primed part $(D_u)'$, where where D_l and D_u are standard clauses over Σ.

Let us consider a clause $\mathcal{C} \in N_1^1$ and let us follow the parent links from $\mathcal{C}$ to construct the whole *proof tree for* $\mathcal{C}$ with $\mathcal{C}$ as its root and some subsets $M \subseteq N$ and $M_1' \subseteq N_1'$ forming the leaves. We now observe that:

- The set of upper parts of clauses from M and M_1' is unsatisfiable. Indeed, if we just "forget" the lower parts of the clauses in the proof tree for $\mathcal{C}$, we obtain a proof of the empty clause.

- If we, on the other hand, track these lower parts of the clauses in the proof tree for $\mathcal{C}$, we realize they are being collected to constitute the standard part of $\mathcal{C}$. Specifically, the simple labeled clause $\mathcal{C}$ is necessarily of the form $(*, k+1) \,||\, C$, where

  $$C = \bigvee \{D_l \;\mid\; (*,*) \,||\, D_l \vee (D_h)' \in M\}.$$

  The clauses from M_1' do not directly influence the form of C, because their lower parts are empty.

We now use the assumption that $\mathcal{G}^k_{(N\cup N_1)} = \mathcal{G}^k_{(N\cup N_2)}$ to discover a clause $\mathcal{C}_0 \in N_2^1$ that subsumes $\mathcal{C}$. The details follow.

- The assumption $\mathcal{G}^k_{(N\cup N_1)} = \mathcal{G}^k_{(N\cup N_2)}$ means that the sets $(\overline{N}\cup N_1)$ and $(\overline{N}\cup N_2)$ are logically equivalent, where $\overline{N}$ stands for the simple $(*,*)$-clauses from N. To focus on the main idea, let us first formulate our result for the case when the equivalence holds already between N_1 and N_2.

- Then, since the upper parts of clauses from $(M \cup M_1')$ are unsatisfiable, so are the upper parts of clauses from $(M \cup N_2')$: we first strengthen $(M \cup M_1')$ to $(M \cup N_1')$, which preserves unsatisfiability, and than exchange N_1' with N_2' using our equivalence.

- Because the set of the upper parts of the clauses $(M \cup N_2')$ is unsatisfiable, by completeness of PSup the empty clause can be derived from the set. In analogy to what we have seen above, there is a corresponding proof tree from the clauses $(M \cup N_2')$ as leaves. The root of the tree is formed by a clause $\mathcal{C}_0 = (*, k+1) \,||\, C_0$, where
$$C_0 = \bigvee \{D_l \;\mid\; (*,*) \,||\, D_l \vee (D_h)' \in M_0\},$$
and where M_0 is the subset those clauses of M that were actually used in the derivation of the empty clause. From $M_0 \subseteq M$ it follows that $C_0 \subseteq C$ and we are done.

- In the general case, where we assume equivalence of $(\overline{N} \cup N_1)$ and $(\overline{N} \cup N_2)$, we can only obtain unsatisfiability of $(M \cup \overline{N}' \cup N_2')$ instead of $(M \cup N_2')$, where
$$\overline{N}' = \{(*,*) \,||\, (C)' \;\mid\; (*,*) \,||\, C \in \overline{N}\}$$
is the set of shifted versions of clauses from $\overline{N}$. If a clause $\mathcal{D} \in \overline{N}'$ is not in N, it must be subsumed by another clause $\mathcal{D}_0 \in N$, because we assume N to be saturated by LPSup and so, in particular, by Temporal Shift. This means each of the additional assumptions from $\overline{N}'$ (or its stronger version) is available for the derivation of the empty clause. Moreover, these additional assumptions cannot influence the form of the clause $\mathcal{C}_0$, because their lower parts are empty.

We have just shown that the set N_1^1 is subsumed by N_2^1. Because the argument is symmetrical, we obtain that sets N_1^1 and N_2^1 are subsumption equivalent, and so the sets $\overline{N_1^*}$ and $\overline{N_2^*}$ are equal. This concludes the proof.

Final remarks

As a corollary of our theorem we obtain that under the suitable conditions imposed on the saturation process the exact syntactic form of the derived layers is uniquely determined by their semantics, i.e., by the set of goal vertexes the layer represents. The only layer which can have a "non-compatible" syntactic form is the 0-layer, which our decision procedure obtains as an input.

Example 2.12. We close this section with an example showing that subsumption is important for the proof of our theorem to go through. The following three $(*,*)$-clauses

$$(*,*) \,||\, a \vee \neg a',$$
$$(*,*) \,||\, a \vee b \vee b',$$
$$(*,*) \,||\, a \vee \neg b \vee \neg b'$$

are saturated by LPSup (we may assume that $a < b$ in the ordering). If we now use the following (redundant) representation of the 0-layer

$$(*,0) \,||\, a \vee \neg b \qquad\qquad (*,0) \,||\, a$$

and altogether refrain from applying subsumption, the subsequent saturation will generate the following simple clauses in the 1-layer and 2-layer:

$$(*,1) \,||\, a \vee b \qquad\qquad (*,1) \,||\, a$$
$$(*,2) \,||\, a \vee \neg b \qquad\qquad (*,2) \,||\, a$$

We see that without subsumption the "syntactic period" of the saturation is 2 while the "semantic period" is just 1.

2.5 Related work

2.5.1 Approaches to LTL satisfiability checking: an overview

There are three basic approaches to checking satisfiability of LTL formulas: one is based on automata theory, one uses semantic tableaux, and, finally, there are the resolution-based methods. This section provides a quick overview of the methods less related to LPSup before we present a detailed comparison with the closest relative, Clausal Temporal Resolution (Fisher et al., 2001).

Automata and model checking

Vardi and Wolper (1986, 1994) were the first to describe a translation from LTL into Büchi automata. Satisfiability of the translated formula corresponds to non-emptiness of the language accepted by the automaton and can be efficiently checked by analyzing the underlying graph structure (Courcoubetis et al., 1992). Because the size of the automaton is exponential in the size of the formula, generating its states in an on-demand fashion (Gerth et al., 1995) is of practical importance. In the unsatisfiable case, however, all the reachable states need to be visited.

An appealing alternative to such an *explicit*, enumerative approach, is to use Binary Decision Diagrams (Bryant, 1986) to represent whole sets of states at once and analyze the state space *symbolically* (Burch et al., 1992). In the context of LTL, one typically uses the translation described by Clarke et al. (1997) and the analysis algorithm of Emerson and Lei (1986). We have shown in Section 2.4 that our normal form, TST, also

represents a Büchi automaton in a symbolic way (although the core data structure is a set of propositional clauses instead of a BDD) and that the saturation process corresponds to the analysis of the underlying semantic graph. It would be interesting to further follow and investigate this connection between the two approaches.

It needs to be noted that the primary interest of all the mentioned automata based work is on verification of systems with non-terminating computations. There the LTL formula plays the role of a specification of the system and the task is to verify that the system complies with the specification. The verification process is then usually referred to as *model checking* (Clarke et al., 2001). As explained by Rozier and Vardi (2010), LTL satisfiability checking can, however, be easily reduced to LTL model checking. One just needs to check the given LTL formula against a *universal model*, i.e. a model that admits all possible computations. This observation makes the model checking approach directly related to our work.

Tableaux methods

Semantic tableaux pioneered by Beth (1955) represent a well established method for deriving decision procedures for various logics. A tableau calculus consists of a collection of rules which specify how to break down the given formula into its constituent parts. The rules are used to systematically generate subcases until an elementary contradiction is reached. If the process runs out of options without discovering the contradiction, the final tableau can be analyzed to yield a model of the original formula.

The early publications on tableaux-based calculi for LTL (Gough, 1984; Wolper, 1985) describe a decision procedure which proceeds in two phases: first, the procedure creates a pre-model for the given formula by applying the tableaux rules, then, in the second phase, the procedure checks whether the pre-model satisfies all the eventuality sub-formulas. In contrast, a later work of Schwendimann (1998) presents a one-phase tableau calculus, which checks for the fulfillment of the eventuality formulas locally and on-the-fly. Recently, (Gaintzarain et al., 2008) proposed a tableaux calculus in which the handling of the eventualities is fully transferred to the syntax level. The method remembers the formula context whenever a fulfillment of an eventuality is postponed and then asserts the negation of the context to ensure progress. The authors argue that this feature makes their approach especially suited for completely automatic theorem proving. The practical relevance of this claim, however, still waits to be established.

Despite the differences in the initial viewpoint as well is in the theory employed, a closer look reveals that operationally the automata theoretic approach and the tableaux method for LTL satisfiability are closely related. Indeed, Gerth et al. (1995) already present their contribution as “a tableau-based algorithm”. This connection should be always kept in mind as it allows for a transfer of ideas between the two approaches.

Resolution

Following the rise of interest in LTL in the mid-eighties several resolution-based methods for LTL satisfiability were proposed. These include the non-clausal approach of Abadi

and Manna (1985) and the works of (Cavalli and del Cerro, 1984) and (Venkatesh, 1985). The methods differ in the subset of the language considered, in the normal form used, as well as in suitability for mechanization.

In the next section we thoroughly compare LPSup with Clausal Temporal Resolution (CTR) introduced by (Fisher, 1991). This is the single most developed approach preceding our work, with a robust implementation of the corresponding decision procedure available (Hustadt and Konev, 2003). CTR has been also been extended to a fragment of first-order linear temporal logic (Degtyarev et al., 2006). A detailed overview of the early resolution-based methods listed above can be found the in main reference on CTR (Fisher et al., 2001).

2.5.2 Comparison with Clausal Temporal Resolution

In this section we show that operationally there is a close connection between LPSup and the Clausal Temporal Resolution (CTR) of Fisher et al. (2001). From this perspective, our formalism of labeled clauses can be seen as a new way to derive completeness of CTR that justifies the use of ordering restrictions and redundancy elimination in a transparent way. This has not been achieved yet in full by previous work: Hustadt et al. (2005) provide a proof theoretic argument, but only for the use of ordering restrictions, Konev et al. (2005) sketch the idea how to justify tautology removal and subsumption, but do not consider the abstract redundancy notion in the style of Bachmair and Ganzinger (2001), which we provide.

Moreover, there is also a correspondence between our layer-by-layer saturation followed by the application of the Leap inference and the BFS-Loop search of CTR as described by Gago et al. (2002); see also Ludwig and Hustadt (2009a). Apart from being interesting in its own right, this view sheds new light on explaining BFS-Loop search, as it gives meaning to the intermediate clauses generated in the process, and we thus do not need to take the detour through the DNF representation of Dixon (1996, 1998). Even here, the idea of labels clearly separates logical content of the clauses from the meta-logical one (c.f. the ad hoc marker literal of Gago et al., 2002).

Despite these similarities between LPSup and CTR, the calculi are by no means identical. As discussed before, a temporal model can be extracted in a straightforward way from a satisfiable set of labeled clauses saturated by LPSup. This does not hold for CTR, where a more complex procedure, which simulates the model construction of Bachmair and Ganzinger (2001) only locally, needs to be applied (Ludwig and Hustadt, 2009b). In particular, because saturation by CTR does not give the model building procedure any guidance towards satisfying the goal clauses, the procedure needs to try out at every time point in a fair way all the possible orderings on the signature (in the worse case) to make sure the goal is eventually reached. As each change of the ordering calls for a subsequent re-saturation of the clause set in question (so that the local model construction still works), it obviously diminishes the positive effect orderings in general have on reducing the search space.

Finally note that since we eventually rely on propositional superposition, we can also take into account the explicit use of partial models to further guide the search for a proof

Table 2.1: Clause alignment between CTR and LPSup. l_a, l_b, and l_c denote literals.

name	CTR	LPSup
initial	$\mathbf{start} \to \bigvee_a l_a$	$(0,*) \,\|\|\, \bigvee_a l_a$
step	$\bigwedge_b l_b \to \bigcirc \bigvee_c l_c$	$(*,*) \,\|\|\, \bigvee_b \overline{l_b} \vee \bigvee_c l'_c$

or saturation. The idea is to build a partial model based on the ordering on propositional literals. Then it can be shown that resolution can be restricted to premises where one is false and the other true in the partial model (Bachmair and Ganzinger, 1990). This superposition approach on propositional clauses is closely related to the state-of-the-art CDCL algorithm for propositional logic (Marques-Silva et al., 2009). The missing bit is to "lift" this setting to our labeled clauses. We explore this idea in Chapter 3.

Aligning the syntax

The calculus CTR operates on temporal clauses of the Separated Normal Form (SNF) (see Section 2.2.2). The classical exposition (Fisher et al., 2001) adopts implicative notation for the temporal clauses and introduces a special temporal constant **start** interpreted to indicate the initial time point.

Recall that there are three kinds temporal clauses in SNF: initial, step, and eventuality clauses. The initial and step clauses used by CTR correspond to labeled counterparts of LPSup in a straightforward way (see Figure 2.1). While CTR in general works with several eventuality clauses of the form

$$\Box \left(\bigwedge_{b \in B} k_b \to \Diamond l \right), \tag{2.8}$$

LPSup uses the ideas of Degtyarev et al. (2002) to obtain a problem with only a single eventuality that is unconditional (with the set of the antecedent literals B being empty). On the other hand, LPSup relaxes the requirement that the eventuality be represented by a single literal l. Instead, the eventuality is described by a whole set of the $(*,0)$-clauses, understood conjunctively:

$$\Box\Diamond \left(\bigwedge_{(*,0) \,||\, C} C \right).$$

Note that the technique of Degtyarev et al. (2002) allows us to obtain a formulation of a problem that contains only a single unconditional eventuality in a form of a single literal. That is an "intersection" format directly accessible to both LPSup and CTR.

Comparing deductions

There are two *step resolution* rules in CTR (Fisher et al., 2001) dealing with initial and step clauses, respectively. When equipped with ordering constraints (Hustadt et al.,

2005) these can be seen to be equivalent to the Ordered Resolution inference of LPSup acting on the corresponding labeled clauses. Similarly, the upper half of the following *clause conversion* rule of CTR

$$\mathcal{R}\frac{\phi \to \bigcirc\mathbf{false}}{\begin{array}{l}\mathbf{true} \to \bigcirc\neg\phi \\ \mathbf{start} \to \neg\phi\end{array}}$$

can be matched by the Temporal Shift inference. The other half of the rule, which turns the step clause with unsatisfiable succedent into an initial clause is not needed in LPSup, where the assumption is kept instead and allowed to interact with initial clauses directly (by relying on merge of the respective labels).

Let us now compare how the two calculi deal with eventualities. The inference in CTR dedicated to this purpose is called *temporal resolution*. It combines several step clauses into groups (so called merged-SNF clauses) and resolves those against one eventuality clause. There is a nontrivial side condition to be verified that amounts to proving that the step clauses involved form a so called *loop*, meaning that they together conditionally imply that the eventuality may become false forever. As a last step, the inference's conclusion, which is not a temporal clause in general, must be translated into SNF after it is derived.

Several methods have been proposed how to actually implement temporal resolution (Dixon, 1996). Here we focus on breadth first search for the loop as described by Gago et al. (2002). The idea is to perform the loop search by iteratively applying step resolution inferences to certain clauses and to organize the individual iterations by enriching the participating clauses with a special marking literal (see also Ludwig and Hustadt, 2009a). The marking literals are numbered by the iteration index. This helps to separate the clauses of the individual iterations and allows for their reuse in subsequent loop searches for the same eventuality literal.

Interestingly, we can map this form of loop search to the layer-by-layer saturation process of LPSup. We identify the marker literal with the label $(*, k)$, i.e. the label of the clauses associated with the goal, and interpret k as the iteration index. There is, however, a small but important difference in how the clauses with a new value of the index k are created. In LPSup they arise as conclusions of the Temporal Shift inference

$$\mathcal{I}\frac{(*, k) \,||\, C}{(*, k+1) \,||\, (C)'}.$$

The corresponding inference of CTR, when adopted to our notation, becomes

$$\mathcal{I}\frac{(*, k) \,||\, C}{(*, k+1) \,||\, (C \vee l)'}, \tag{2.9}$$

where l is the respective eventuality literal from (2.8). Weakening the derived clause by l' is not sound with respect to the semantics based on (K, L)-models, but can be interpreted and justified with the help of semantic graphs.

As explained in Section 2.4.2, in LPSup the $(*, k)$-clauses of a given clause set can be seen to represent a set of those vertexes of the semantic graph that can reach a goal

vertex in *exactly* k steps. As an effect of the added eventuality literal in (2.9) above, the corresponding $(*, k)$-clauses in CTR represent the vertexes that can reach a goal vertex in *at most* k steps. Intuitively, the goal vertexes of order zero are effectively reinserted to the computed preimage after each iteration. As a side-effect, the sequence of the represented vertex sets grows monotonically with respect to the subset relation.

In LPSup, layer-by-layer saturation ends when a repetition is detected. We then invoke the Leap inference and potentially derive additional $(*, k)$-clauses. Analogously, a successful repetition check concludes the loop search in CTR. There, the new clauses collected by an equivalent of Leap obtain the status of simple step clauses, i.e., they effectively become $(*, *)$-clauses. These clauses stand for a fixpoint result of the iterative loop search and represent the set of those vertexes that can reach a goal vertex in any number of steps. It is therefore sound to assert these clauses to hold universally, because only the vertexes they represent can be part of any potential model path.

Remark 2.4. Because, unlike in LPSup, the sequence of vertexes represented during loop search by the respective clause sets grows monotonically, we can for CTR derive a better theoretical bound on the maximal number of iterations before repetition occurs. Indeed, under similar conditions on the saturation process as those of Section 2.4.3, there is at most $|2^{\Sigma}|$ iterations for CTR, because each iteration must include at least one new vertex unless it stabilizes. On the other hand, our current best bound for LPSup derives from $2^{|2^{\Sigma}|}$ – the number of all subsets of the set of vertexes 2^{Σ}.

On the other hand, the inconspicuous addition of the eventuality literal seems to have a negative effect on the performance of CTR in practice, because it means that more inferences typically need to be performed before the computation proceeds from one layer to the next. We explore this phenomenon empirically in the next section.

2.5.3 Experimental comparison

We implemented a simple prototype of both LPSup and CTR in order to compare the two calculi on non-trivial input problems. In this section we briefly introduce the implementation, describe our problem set, which consists of both adopted and newly devised formula families, and report on the results of our experiment. The prototype as well as the test problems are publicly available (Suda, 2012b).

Implementation

Our prototype, written in SWI-Prolog, is a straightforward implementation of the satisfiability checking procedure of Algorithm 2.2, using layer-by-layer saturation (Algorithm 2.1) as its main subroutine. It takes for an input an LTL formula in SNF and normalizes it further (following the simplification described in Section 2.2.2) to an SNF with a single unconditional eventuality clause consisting of a single literal. Such a formula constitutes a valid input for CTR and, at the same time, can be treated as a TST to be handled by LPSup.

Building on the observations of Section 2.5.2 we interpret CTR (with the BFS loop-search in the style of Gago et al., 2002) in the framework of labeled clauses of LPSup and

so a large part of the code is shared by the two calculi. The differences that distinguish LPSup from CTR in our implementation can be summarized as follows:

- CTR does not have $(0, k)$-clauses. A modified version of the merge operation blocks for CTR inferences between initial clauses and the clauses derived from the goal.
- For CTR we modify the Temporal Shift inference (a.k.a. clause conversion rule) for $(*, k)$-clauses such that the conclusion is additionally extended by the unique eventuality literal.
- In CTR the layer-by-layer saturation always finishes by detecting a repetition with period $p = 1$. This is checked at runtime by an assertion.
- Finally, the conclusion of the Leap inference takes the form of $(*, 0)$-clauses in the case of LPSup (see Remark 2.2) and the form of $(*, *)$-clauses for CTR.

Adopted formulas

We adopted two formula families from a previous paper on LTL satisfiability (Hustadt and Schmidt, 2002). They are referred to as $\mathbf{C}_n^1$ and $\mathbf{C}_n^2$ and parametrized by a natural number $n \in \mathbb{N}$. Both families originally consist of a certain pattern of temporal clauses together with a set of essentially standard clauses that encode a random k-SAT problem. Because the tested calculi treat initial and step clauses identically, we decided to drop the random part and only compare how effectively they deal with the temporal aspects of the formulas. This way we obtained the following two families of temporal formulas:

$$\mathbf{C}_n^1 = \Box(\neg p_1 \vee \Diamond p_2) \wedge \Box(\neg p_2 \vee \Diamond p_3) \wedge \cdots \wedge \Box(\neg p_n \vee \Diamond p_1),$$

$$\begin{aligned}\mathbf{C}_n^2 = {} & r_1 \wedge (\neg r_1 \vee q_1) \wedge (\neg r_1 \vee \neg q_n) \wedge \\ & \Box(\neg r_n \vee \bigcirc r_1) \wedge \Box(\neg r_{n-1} \vee \bigcirc r_n) \wedge \cdots \wedge \Box(\neg r_1 \vee \bigcirc r_2) \wedge \\ & \Box(\neg r_n \vee \bigcirc \neg q_n) \wedge \Box(\neg r_{n-1} \vee \bigcirc \neg q_n) \wedge \cdots \wedge \Box(\neg r_1 \vee \bigcirc \neg q_n) \wedge \\ & \Box(\neg q_1 \vee \Diamond s_2) \wedge \Box(\neg s_2 \vee q_2 \vee \bigcirc q_n \vee \cdots \vee \bigcirc q_3) \wedge \\ & \vdots \\ & \Box(\neg q_{n-1} \vee \Diamond s_n) \wedge \Box(\neg s_n \vee q_n)\end{aligned}$$

Note that while $\mathbf{C}_n^1$ are trivially satisfiable, $\mathbf{C}_n^2$ is unsatisfiable.

New formulas

In addition to the above, we also tested the calculi on formulas from two new families, specifically constructed to highlight the respective strengths and weaknesses of LPSup and CTR. They are both based on the idea of putting together several independent "cycles", and are both parametrized by lists of integers that correspond to the cycles'

lengths. The cycles, however, play different conceptual roles in each family, and the resulting problems are, in fact, very different.

Explicit cycles problem In the *explicit cycles* problem $\mathbf{E}_{(l_1,\ldots,l_k)}$ there are k "syntactic" cycles composed of variables connected by implications. For any $i = 1,\ldots,k$, cycle i consists of variables $p^i_1,\ldots,p^i_{l_i}$ together with the step clauses

$$\Box(\neg p^i_1 \vee \bigcirc p^i_2),\ \Box(\neg p^i_2 \vee \bigcirc p^i_3),\ \ldots,\ \Box(\neg p^i_{l_i-1} \vee \bigcirc p^i_{l_i}),\ \Box(\neg p^i_{l_i} \vee \bigcirc p^i_1).$$

Additionally, for each cycle there is also a step clause $\Box(\bigcirc\neg p^i_1 \vee \bigcirc\neg g)$, where g is an independent "goal" variable. Finally, the cycles are "connected together" by the eventuality clause $\Box\Diamond g$. In total, the explicit cycles problem $\mathbf{E}_{(l_1,\ldots,l_k)}$ consists of $1+k+l_1+\cdots+l_k$ temporal clauses and the corresponding signature contains $1+l_1+\cdots+l_k$ variables. The problem is satisfiable.

The explicit cycles problem is designed in such a way that LPSup processes the individual cycles independently from each other and derives only unit clauses during layer-by-layer saturation. In CTR, on the other hand, the addition of the goal literal triggers an interaction between the cycles and the calculus derives clauses of increasing size which mix literals from different cycles.

Example 2.13. The behavior of the two calculi on problem $\mathbf{E}_{(2,3)}$ is demonstrated in Table 2.2. We formulate the problem using variables a, b and c, d, e for the respective cycles and reserve variable g for the goal. This means we start with the following set of labeled clauses

$$\begin{array}{lll}
(*,*) \,||\, \neg a \vee b', & (*,*) \,||\, \neg c \vee d', & (*,0) \,||\, g,\\
(*,*) \,||\, \neg b \vee a', & (*,*) \,||\, \neg d \vee e', & \\
 & (*,*) \,||\, \neg e \vee c', & \\
(*,*) \,||\, \neg a' \vee \neg g', & (*,*) \,||\, \neg c' \vee \neg g'. &
\end{array}$$

We assume that g is the largest symbol in the used ordering, which makes the $(*,*)$-clauses already saturated. Table 2.2 shows the standard parts of the simple $(*,k)$-clauses derived by the respective calculi during layer-by-layer saturation process.[10] To capture more details of the process, there are two columns displayed for CTR. The first column shows the simple $(*,k)$-clauses as they are generated, before repeated literals are factored out and subsumption is applied. The resulting reduced clause set is displayed in the other column. We see that repetition occurs with layer 7 for both calculi. In the case of LPSup, layer 7 is equal to layer 1, for CTR it is equal to layer 6.

Given an explicit cycles problem $\mathbf{E}_{(l_1,\ldots,l_k)}$ let $m = \mathrm{LCM}^k_{i=1}\ l_i$ be the *least common multiple* of the cycles' lengths. It can be shown that for both LPSup and CTR repetition occurs when saturating layer of index $m+1$. As Example 2.13 suggests, however, in the case of CTR the saturation process is much more expensive both with respect to the size and the number of involved clauses.

[10] To save space, we omit the 0-layer, which contains for both calculi just the goal clause $(*,0) \,||\, g$.

Table 2.2: Simple $(*, k)$-clauses generated by LPSup and CTR on $\mathbf{E}_{(2,3)}$.

Layer index k	LPSup	CTR generated	CTR reduced
1	$\neg b$	$\neg b$	$\neg b$
	$\neg e$	$\neg e$	$\neg e$
2		$\neg a \vee \neg b$	$\neg a \vee \neg b$
	$\neg a$	$\neg a \vee \neg e$	$\neg a \vee \neg e$
	$\neg d$	$\neg d \vee \neg b$	$\neg b \vee \neg d$
		$\neg d \vee \neg e$	$\neg d \vee \neg e$
3		$\neg b \vee \neg a \vee \neg b$	
		$\neg b \vee \neg a \vee \neg e$	
		$\neg b \vee \neg d \vee \neg b$	$\neg a \vee \neg b$
	$\neg b$	$\neg b \vee \neg d \vee \neg e$	$\neg b \vee \neg d$
	$\neg c$	$\neg a \vee \neg c \vee \neg b$	$\neg a \vee \neg c \vee \neg e$
		$\neg a \vee \neg c \vee \neg e$	$\neg c \vee \neg d \vee \neg e$
		$\neg c \vee \neg d \vee \neg b$	
		$\neg c \vee \neg d \vee \neg e$	
4		$\neg b \vee \neg a \vee \neg b$	
		$\neg b \vee \neg a \vee \neg e$	
		$\neg a \vee \neg c \vee \neg b$	$\neg a \vee \neg b$
	$\neg a$	$\neg a \vee \neg c \vee \neg e$	$\neg a \vee \neg c \vee \neg e$
	$\neg e$	$\neg b \vee \neg e \vee \neg d \vee \neg b$	$\neg b \vee \neg d \vee \neg e$
		$\neg b \vee \neg e \vee \neg d \vee \neg e$	$\neg c \vee \neg d \vee \neg e$
		$\neg e \vee \neg c \vee \neg d \vee \neg b$	
		$\neg e \vee \neg c \vee \neg d \vee \neg e$	
5		$\neg b \vee \neg a \vee \neg b$	
		$\neg b \vee \neg a \vee \neg e$	
		$\neg b \vee \neg e \vee \neg d \vee \neg b$	$\neg a \vee \neg b$
	$\neg b$	$\neg b \vee \neg e \vee \neg d \vee \neg e$	$\neg b \vee \neg d \vee \neg e$
	$\neg d$	$\neg a \vee \neg c \vee \neg d \vee \neg b$	$\neg c \vee \neg d \vee \neg e$
		$\neg a \vee \neg c \vee \neg d \vee \neg e$	
		$\neg e \vee \neg c \vee \neg d \vee \neg b$	
		$\neg e \vee \neg c \vee \neg d \vee \neg e$	
6		$\neg b \vee \neg a \vee \neg b$	
		$\neg b \vee \neg a \vee \neg e$	
	$\neg a$	$\neg a \vee \neg c \vee \neg d \vee \neg b$	$\neg a \vee \neg b$
	$\neg c$	$\neg a \vee \neg c \vee \neg d \vee \neg e$	$\neg c \vee \neg d \vee \neg e$
		$\neg e \vee \neg c \vee \neg d \vee \neg b$	
		$\neg e \vee \neg c \vee \neg d \vee \neg e$	
7		$\neg b \vee \neg a \vee \neg b$	
	$\neg b$	$\neg b \vee \neg a \vee \neg e$	$\neg a \vee \neg b$
	$\neg e$	$\neg e \vee \neg c \vee \neg d \vee \neg b$	$\neg c \vee \neg d \vee \neg e$
		$\neg e \vee \neg c \vee \neg d \vee \neg e$	

Implicit cycles problem In the *implicit cycles* problem the cycles emerge on the conceptual level of semantics: the problem is constructed using the ideas of Remark 2.3 such that its corresponding semantic graph has a particular shape.

We define the problem $\mathbf{I}_{(l_1,\ldots,l_k)}$ only when the sum of the cycles' lengths $l = \sum_{i=1}^{k} l_i$ is equal to a power of two, i.e., $l = 2^n$ for some n. We then construct the problem over a signature Σ of size n. The clauses of $\mathbf{I}_{(l_1,\ldots,l_k)}$ are selected in such way that the semantic graph of the problem consists of a disjoint union of k (oriented) cycles of lengths $l_1, \ldots, l_k$, respectively, there is exactly one goal vertex on each cycle, and, finally, every vertex of the graph is an initial vertex. As discussed in Remark 2.3 the number of clauses needed to construct such a problem can be polynomially bounded in 2^n, i.e. in the number of vertexes of the graph. The problem is satisfiable.

The idea behind the implicit cycles problem is to force the layer-by-layer saturation of LPSup to explore many layers before a repetition can be detected. Using the results of Section 2.4.2 we can show that LPSup needs at least m iterations, where $m = \mathrm{LCM}_{i=1}^{k} l_i$ is again the least common multiple of the cycles' lengths. In CTR, on the other hand, the number of iterations before a repetition can be detected is proportional to the length of the longest cycle only.

Remark 2.5. Let us consider the problem $\mathbf{I}_{(1,2,\ldots,k)}$, where the cycles' lengths are formed by the first k positive integers.[11] It follows from the prime number theorem (Weisstein, 2013) that the least common multiple $m = \mathrm{LCM}_{i=1}^{k} i$ grows asymptotically as $e^{k(1+o(1))}$ when $k \to \infty$. Because the number of vertexes of the corresponding semantic graph grows quadratically in k, this number k is of order $\sqrt{2^n}$, where n is the size of the signature Σ needed to express such a problem. Thus the number of iterations before LPSup detects a repetition on $\mathbf{I}_{(1,2,\ldots,k)}$ is of order $e^{\sqrt{2^n}}$, i.e. doubly exponential in n (cf. Remark 2.4). This, however, does not (yet) imply doubly exponential lower bound on the complexity of LPSup, because we are currently only able to describe the input problem using exponentially many clauses in n.

Experimental results

In our experiment we compared our prototype implementations of LPSup and CTR on several instances of the formula families described above. Additionally, we also supplied the test problems to the the temporal prover TRP++ (Hustadt and Konev, 2003)[12], which also implements the CTR calculus, to provide evidence that our experimental results are not biased. We used the default mode (no extra options) for running TRP++ and collected the information on the number of generated and subsumed clauses from the output which the prover by default provides.

For a variable ordering for restricting resolution inferences in the prototype we experimentally selected an ordering that gave good results for both calculi on the tested examples.[13] The variable ordering used by TRP++ was determined automatically by

[11] To make the sum $\sum_{i=1}^{k} l_i$ equal to a power of two, we use additional cycles of length one as a padding.

[12] We used version 2.1 available at `http://www.csc.liv.ac.uk/~konev/software/trp++/`.

[13] We optimized the order of groups of variables, where one group was formed by all the variables of the input formula and several other groups contained different kinds of auxiliary variables introduced

Table 2.3: Experimental comparison of our prototype implementations of LPSup and CTR and the LTL prover TRP++. The number of clauses in the input (size) and the number of derived and subsumed clauses for each prover are reported.

		LPSup		CTR		TRP++	
problem	size	derived	subsumed	derived	subsumed	derived	subsumed
$\mathbf{C}^1_{10}$	56	53	100	174	110	363	300
$\mathbf{C}^1_{15}$	81	78	145	334	240	688	595
$\mathbf{C}^1_{20}$	106	103	190	544	420	1113	990
$\mathbf{C}^2_3$	22	442	324	984	909	1146	968
$\mathbf{C}^2_4$	30	1937	1612	5298	5047	3560	3053
$\mathbf{C}^2_5$	38	6287	5635	18724	18134	7925	6922
$\mathbf{E}_{(2,3)}$	8	23	4	131	78	177	77
$\mathbf{E}_{(2,3,4)}$	13	52	6	1061	595	1597	627
$\mathbf{I}_{(3,5)}$	62	406	368	203	194	86	160
$\mathbf{I}_{(3,5,8)}$	253	8010	7356	1087	1145	390	745

the prover.

The results of the experiment are summarized in Table 2.3. For each system and input problem the table shows the number of derived and subsumed clauses. We decided not to report on running times as our aim here is to compare the calculi rather than the implementations. We believe that the number of considered clauses provides a good measure of the amount of data that needs to be processed by any saturation-based implementation of the respective calculus, which is, moreover, independent on the choice programming language or the use of particular data structures.

We can see that LPSup consistently needs to generate fewer clauses than CTR to draw its conclusion on both $\mathbf{C}^1_n$ and $\mathbf{C}^2_n$ problems. The behavior on the explicit and implicit cycles problems corresponds to our expectations based on the theoretical analysis. While the explicit cycles are trivial for LPSup, they require considerable amount of computation before the results of obtained by CTR. This trend is reversed on the implicit cycles, where LPSup performs worse than CTR.

Although the inferior performance of CTR on $\mathbf{C}^1_n$ and $\mathbf{C}^2_n$ could possibly be stemming from the translation to single eventuality formulation of the problems (while heuristics for efficiently treating the individual eventualities separately can be imagined), the other two families $\mathbf{I}_{(l_1,\ldots,l_k)}$ and $\mathbf{E}_{(l_1,\ldots,l_k)}$ contain single eventuality from the outset. Only further tests on examples from practice may reveal which of the two phenomena exemplified by the families $\mathbf{I}_{(l_1,\ldots,l_k)}$ and $\mathbf{E}_{(l_1,\ldots,l_k)}$, respectively, have a higher impact on practical utility.

during the translation of multiple eventualities into a single unconditional one.

2.6 Conclusion

In this chapter we have presented LPSup, a new resolution-based calculus for Linear Temporal Logic (LTL). The main idea behind LPSup is to interpret an LTL formula as a set of purely propositional problems over an infinite signature and to use labeled clauses to finitely represent reasoning within these problems. This enables us to lift PSup, a well understood calculus for propositional logic to the temporal setting and to formally transfer its favorable features such as ordering restrictions on inferences and abstract redundancy concept along the way to LPSup. We described a saturation decision procedure based on LPSup and explained how it can be extended to build models for satisfiable inputs in a straightforward way.

We uncovered a connection between our formula normal form and Büchi automata which lead us to propose an alternative semantics for the operation of LPSup. In the light of this semantics, the saturation process can be seen as a symbolic computation of the preimage operation on sets of graph vertexes represented in CNF. This observation, in addition to being of independent interest, allows us to easily expose the relation of LPSup to other approaches to LTL satisfiability checking.

Finally, we studied the relation between LPSup and its closest relative, the Clausal Temporal Resolution (CTR) calculus by Fisher et al. (2001). Although the underlying principles behind the two calculi are different, the ensuing procedures can be almost aligned on the computational level. Theoretical analysis favors CTR, whose worst case complexity is better than that of LPSup by an exponential factor. On the other hand, our experiments indicate that in practice, LPSup is able to outperform CTR on many examples. Moreover, unlike LPSup, CTR cannot be easily extended to perform model construction.

3 LTL proving with partial model guidance

3.1 Introduction

In this chapter we describe a new algorithm for LTL satisfiability checking. We call the algorithm LS4 as a pseudo-acronym for Labeled Superposition for LTL with partial model guidance. As the name suggests, LS4 can be seen as a continuation and extension of our work on the LPSup calculus developed in Chapter 2. The main difference is that instead of relying on saturation, LS4 constructs a partial model on the fly and uses it to effectively guide the selection of inferences. As we will show, this idea leads to a very successful algorithm both for finding full LTL models and for showing unsatisfiability.

Let us start our exposition here by explaining partial model guidance in more detail.

Partial model guidance

In Chapter 2 we have presented a simple algorithm (Algorithm 2.3 in Section 2.3.4) for building models of sets of labeled clauses saturated by LPSup. The algorithm conceptually reduces the labeled clause input to a set of purely propositional clauses over an infinite signature and then employes the standard model operator for propositional logic (Definition 2.3).

It considers individual propositional variables in a prescribed order and incrementally constructs a partial valuation V, which eventually becomes the desired model. In each step the considered variable p is set to true if and only if there is a productive clause for p, i.e. a clause $C \vee p$ such that all its literals except for p are already assigned a value in V and the value makes them false: $V \not\models C$. In this situation, setting p to true is the "last chance" for the clause to become satisfied in the constructed valuation.

Because the propositional clause set is saturated and does not contain the empty clause (as follows from Theorem 2.3), the algorithm cannot reach a "conflict", which means it cannot arrive to a situation in which setting p to true because of a productive clause $C \vee p$ would make another clause $D \vee \neg p$ inevitably false in the constructed valuation. This is what the completeness theorem for PSup (Theorem 2.1) tells us. In other words, for a saturated set of clauses the model construction is backtrack free.

However, saturation is an expensive process and may compute more than what is actually needed. The proof of Theorem 2.1 gives us a hint on what resolution inferences are the essential ones for deriving the potential empty clause. They involve a productive clause $C \vee p$ and a clause $D \vee \neg p$ false in the currently constructed valuation V, as in the imagined conflict situation above. When we resolve these two clauses on the variable p we obtain a conclusion $C \vee D$, which is smaller than the two premises and necessarily also false in V. Thus such an inference reduces the conflict to a smaller clause. By resetting

Table 3.1: Comparing superposition-based model building with CDCL

	model operator	CDCL
variable ordering	fixed beforehand	dynamic; based on heuristics
variable polarity	defaults to **0**	based on heuristics
implied assignments	productive clause forces **1**	via unit propagation
resolution inferences	reduce the minimal false clause	derive the learned clause
reductions	based on abstract redundancy	(pre/in-processing)

V to a state before the truth status of $C \vee D$ would be determined, we may discover that this new clause now becomes productive and a subsequent update of V will make it true. In the opposite case (if the last unassigned literal of $C \vee D$ is not positive), the conflict cascades further.

The just outlined principle for selecting inferences guided by the current partial valuation is the key idea behind LS4. It drives the algorithm forward to build the partial valuation greedily and only to perform an inference when addressing an immediate conflict. The overhead connected with maintaining the valuation is by far compensated by that fact that we do not need to fully saturate the given clause set.

SAT solver inside

Instead of directly building our algorithm on the above model guidance idea, we exploit a connection (Weidenbach) between the propositional model operator and the Conflict Driven Clause Learning (CDCL) algorithm for propositional satisfiability checking (Marques-Silva et al., 2009). Weidenbach shows that CDCL, an algorithm which powers the current state-of-the-art SAT solvers, is closely related to the superposition model building framework. This means that we can obtain the benefits of model guidance by employing a modern CDCL SAT solver as an underlying inference engine.

The CDCL algorithm advances the well known DPLL procedure for propositional satisfiability (Davis et al., 1962) by a non-chronological backtracking scheme based on conflict analysis and clause learning (Marques-Silva and Sakallah, 1999; Bayardo and Schrag, 1997). The algorithm constructs a partial valuation V by alternating between *deciding* a value for a unassigned variable and exhaustively applying the *unit propagation* rule. Unit propagation extends the partial valuation further by setting an unassigned literal l to true whenever there is a clause $C \vee l$ such that $V \not\models C$. A conflict arises when this process reaches a point where a clause D has all literals assigned to false. By analyzing the conflict and the propagations which lead to it, the algorithm can derive a clause summarizing which decisions contributed to the conflict. The clause is then learned and the algorithm backtracks the partial assignment to a state where the conflict inducing decision can be repaired. Interestingly, the clause learning procedure can be defined by a sequence of resolution inferences between the conflict clause and the clauses that triggered the relevant propagations (Beame et al., 2004).

We can now say a little more about the relation between the superposition model

operator and CDCL (see also Table 3.1). Both approaches incrementally construct a partial valuation. While the model operator follows a fixed ordering for assigning values to variables and prefers setting them to false unless there is a productive clause, CDCL is more flexible both in the order of assigning the variables and in choosing their polarity and typically relies in this respect on heuristics. On the other hand, a closer look reveals that the definition of a productive clause follows from the same principle as the unit propagation rule. Moreover, the way conflicts are dealt with and new clauses derived is essentially equivalent in both approaches. It applies the resolution inference to the conflict clause and backtracks the partial model to a point where the conflicting assignment can be revised. Lastly, we note that CDCL per se does not employ any reductions. However, redundant clauses are typically removed during independent preprocessing (e.g., Eén and Biere, 2005) or in-processing (Järvisalo et al., 2012) stages.

In what follows, we will describe LS4 as an algorithm that uses a CDCL SAT solver as a subroutine. We will, however, draw on the connection to the model operator to strengthen our intuition on what the computation means from the perspective of LPSup.

Step-by-step construction

LS4 attempts to show satisfiability of a given Temporal Satisfiability Task (TST) by iteratively constructing a partial model until a full, ultimately periodic model is discovered. The partial model is a finite sequence of valuations over the basic signature Σ, which we informally refer to as *worlds*. In each iteration, the algorithm prepares and poses a query to the underlying SAT solver, to find out whether the current partial model can be extended by one more world. If the query is satisfiable, a new world is extracted from the satisfying assignment and the partial model is extended. In the opposite case, a new clause is learned from the solver and the last world of the partial model is removed. LS4 backtracks over the last world, because that world (by construction) does not satisfy the new clause, which must hold (again by construction) at that particular position in the final model. In the following iteration, the new clause will help to guide the extension of the updated partial model in the right direction.

We see that LS4 manipulates the partial model with a granularity corresponding to individual worlds. This is achieved by forming the query for the SAT solver over the joint signature $\Sigma \cup \Sigma'$. The algorithm uses the lower part of the signature to encode the current last world, it supplies the step clauses to the query to relate the last world to the potential new world, and asserts all additional constraints on the new world as clauses over the primed variables. Thus, in the satisfiable case, the new world is to be extracted from the Σ'-part of the satisfying assignment.

The current last world is encoded into the query using a mechanism of solving under assumptions. This is a natural extension of the standard interface provided by any SAT solver based on the CDCL algorithm and has been first introduced by Eén and Sörensson (2003a) in their solver Minisat. Assumptions are literals supplied as additional one-time constraints to the solving function. If a model is found, it is required to satisfy these literals. Additionally, in the unsatisfiable case the solver is able to return a subset of those assumptions that were actually used for showing the inconsistency. The new clause

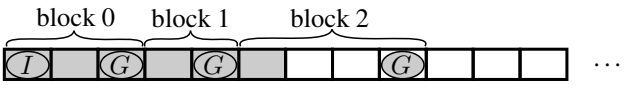


Figure 3.1: Illustrating blocks in LS4: The first world of block 0 satisfies the initial clauses I, the last world of every block satisfies the goal clauses G. The grayed part represents the current partial model.

to be learned by LS4 is computed by collecting negations of the used assumptions. As such, the new clause is automatically false in the current last world and so backtracking follows.

By relying on the mechanism of assumptions, the model guidance paradigm is effectively split onto two levels. It is applied by CDCL on the level of individual variables *within* each call to the SAT solver and by LS4 itself on the level of whole worlds *between* the individual calls. The mechanism of assumptions connects the two levels seamlessly, preserving the overall efficiency, but allowing LS4 to treat the inner workings of the SAT solver in a black-box manner, such that only the clauses learned on the "macroscopic" level of worlds need to be explicitly registered and processed by the algorithm.

Dealing with the goal clauses

To construct a full model for the given TST, LS4 needs to ensure that the goal clauses are satisfied infinitely many times. Unlike Algorithm 2.3, which relies on a previous saturation and picks a rank (K, L) in advance using the completeness theorem of LPSup (Theorem 2.3), LS4 does not have any prior information about the indexes were the goal clauses could be satisfied in the final model. Instead of explicitly looking for a rank and constructing a (K, L)-model, the strategy of LS4 is to greedily assume that every new world along the model sequence could be a goal world, i.e. a world that satisfies the goal clauses, and to update this assumption only when it leads to a conflict.

In more detail, we imagine that the model sequence constructed by LS4 is separated into consecutive *blocks* of worlds, such that the last world of every block is meant to be a goal world (see Figure 3.1). When the partial model grows to the end of the last block, LS4 allocates a new block of length one and places it at the end of the sequence. This means that it creates a new requirement for satisfying the goal clauses positioned at the immediate next index. When, on the other hand, the current configuration of blocks becomes inconsistent, the algorithm extends the last block by adding one more world to it, thus changing the distance between the last block's goal world and the other goal worlds. This is the smallest update to the configuration of blocks that can help to dismiss the inconsistency.

Detecting unsatisfiability

Most of the clauses LS4 learns from unsuccessful extensions of the partial model correspond to labeled clauses of LPSup. As will be explained in detail in Section 3.2, LS4 relies on so called marker literals to track dependencies between the learned clauses. For

instance, a clause depending on the goal clauses corresponds to a $(*, k)$-clause of LPSup. The exact value of k follows in LS4 implicitly from the distance between the index where the clause is learned and the respective goal world.

Inconsistencies are manifested as empty clauses. Like in LPSup, there are several kinds of empty clauses and while deriving an *unconditional* empty clause immediately signals unsatisfiability of the input, a conditional empty clause only implies inconsistency of the current assumption about the configuration the goal worlds and blocks. As already explained, the latter case triggers update of the configuration and the model construction is resumed. What must ensured, however, is that the algorithm does not keep deriving conditional empty clauses indefinitely when the input is unsatisfiable.

To prevent this from happening, LS4 follows the same strategy as LPSup. It organizes the equivalent of $(*, k)$-clauses into layers and checks for a repetition whenever a new layer clause is derived. Depending on the situation, detected repetition may either already signal overall unsatisfiability, or it at least triggers the Leap inference. Leaped clauses globally strengthen the goal formula which forces a backtrack of the partial model from an unpromising part of the search space and ensures overall progress.

Chapter overview

In the following sections we describe the workings of LS4 in full detail. First we explain how to use the mechanism of SAT solving under assumptions to build models in a step-by-step fashion (Section 3.2.1) and how to utilize so called marker literals for tracking dependencies in derivations (Section 3.2.2). We then start the actual presentation of LS4 by declaring the global variables and stating the main invariants maintained by the algorithm (Section 3.3.1). We proceed by giving a detailed pseudocode (Section 3.3.2) and close by correctness (Section 3.3.3) and termination (Section 3.3.4) proofs. The latter proof implicitly entails a doubly-exponential worse case running time estimate for the algorithm.

We have implemented LS4 (Section 3.4.1) and compared it to alternative approaches to LTL satisfiability on a large set of benchmarks. Our experiments show (Section 3.4.2) that our implementation of LS4 is one of the strongest LTL satisfiability checker currently available. We close the chapter by discussing the relation of LS4 to saturation with LPSup (Section 3.5.1), to other SAT-based LTL checkers (Section 3.5.2), and to algorithms recently developed for model checking hardware circuits (Section 3.5.3).

This chapter is based on our earlier publication (Suda and Weidenbach, 2012a), but has been thoroughly revised and notably extended.

3.2 SAT solving under assumptions

Assumptions are literals supplied to a SAT solver along with the standard input and they are meant to further constrain the search to only those models that satisfy them. Given a CNF formula N and a set of assumptions A the solver then decides satisfiability of the conjunction $A \wedge N$. In the satisfiable case, we obtain as the expected result a *satisfying assignment* of the given formula $A \wedge N$, i.e. a valuation V over the introduced

variables such that $V \models A \wedge N$. In the opposite case, however, the SAT solver is able to return more than just the unsatisfiability claim. It returns a subset of those assumptions $A_0 \subseteq A$ that were actually used to show the inconsistency.

The support for solving under assumptions can be easily incorporated into the CDCL algorithm (Eén and Sörensson, 2003a). The assumptions are treated as artificial decisions performed at the beginning of the propagation process. These decisions cannot be backtracked over and the standard conflict resolution mechanism is responsible for collecting the used subset A_0 in the case of an inconsistency.

In this section we describe two different applications of the mechanism of solving under assumptions which are employed in LS4. First, we explain how to build models by parts, in a step-by-step fashion. LS4 uses this idea to construct the partial model of a TST by extension steps which concern one world at a time.

Then we show how to employ the so called marker literals to track dependencies in derivations without the need of proof recording. We define the abstraction of marked clauses to be used in the pseudocode of LS4. Marked clauses are an equivalent of labeled clauses of LPSup implemented within the SAT-solving framework.

New terminology and notation

Above, we have treated the set of assumption literals A in the formula context as a conjunction of its elements. Let us extend our conventions to properly accommodate this concept. In accord with the verification terminology, we will denote a conjunctive set of literals as a *cube*. Syntactically, cubes do not differ from clauses, whose interpretation is disjunctive. The distinction will, however, always be clear from the context. By complementing a cube A one obtains a clause $\sim A = \{\sim l \mid l \in A\}$ and also vice versa.

3.2.1 Solving by parts

Imagine we are given a CNF formula $N = A \wedge T \wedge (B)'$ over a joint signature $\Sigma \cup \Sigma'$, such that the clauses in A and B are only over the signature Σ. We know that formula A alone is satisfiable and we have found a valuation $V : \Sigma \to \{\mathbf{0}, \mathbf{1}\}$ that satisfies it: $V \models A$. We ask ourselves a question. Can V be extended to a full valuation over $\Sigma \cup \Sigma'$ that satisfies the whole formula N? In other words, is there a valuation $W : \Sigma \to \{\mathbf{0}, \mathbf{1}\}$ such that $[V, W] \models N$? We can answer this question using solving under assumptions.

Let us denote by $\mathit{Lits}(V)$ the following encoding of the valuation V into a cube of assumptions:

$$\mathit{Lits}(V) = \{l \mid l \text{ is a literal over } \Sigma \text{ and } V \models l\}.$$

We can now look for the valuation that would extend V and satisfy the whole formula N by posing the query

$$SAT?[\, \mathit{Lits}(V) \wedge T \wedge (B)' \,].$$

If the query is satisfiable, the satisfying assignment will be of the form $[V, W]$ for some W, because the assumptions $\mathit{Lits}(V)$ admit only one possible value for the unprimed variables. In the unsatisfiable case, we obtain a sub-cube $A_0 \subseteq \mathit{Lits}(V)$ of assumptions that were sufficient for detecting the inconsistency.

By complementing the literals from the assumption sub-cube, we can construct a clause C which captures in logical terms the discrepancy between the valuation V and the formula $T \wedge (B)'$. We call it an *explaining* clause for the failed extension attempt and write

$$C = {\sim}A_0 = \{{\sim}l \mid l \in A_0\}.$$

On the one hand, we have, by definition, that $V \not\models C$. On the other hand, because the SAT solver has shown that the formula $A_0 \wedge T \wedge (B)'$ is unsatisfiable, we see that

$$T \wedge (B)' \models C.$$

In other words, only those models of A that additionally satisfy the explaining clause C can be possibly extended to a full model of the formula N.

LS4 is based on the idea of building models by parts and learning explaining clauses from failed extension attempts. Intuitively, the amount of information that an explaining clause carries is inversely proportional to its size. While a clause mentioning every literal over Σ only excludes the model V itself from future considerations, shorter clauses tend to generalize the situation and help to also exclude other, in some sense similar models. Typically, the SAT solver returns a subset with a reasonable generalizing power.[1]

3.2.2 Tracking dependencies with markers

Assume we have an unsatisfiable CNF formula built up as a conjunction of several independent parts:

$$N = N_1 \wedge \ldots \wedge N_k.$$

We would like to know which of the constituents N_i truly contribute to the unsatisfiability of the whole formula N. This is a form of an unsatisfiable core extraction problem and can be solved with the help of marker literals (Achá et al., 2010).

We allocate new variables $v_1, \ldots, v_k$, which do not appear in the formula N, and extend the clauses of the constituent sets by marker literals as follows:

$$\overline{N_i} = \{\neg v_i \vee C \mid C \in N_i\}.$$

Solving the original formula N is equivalent to solving a conjunction of the extended clauses

$$\overline{N_1} \wedge \ldots \wedge \overline{N_k}$$

under a set of assumptions $A = \{v_1, \ldots, v_k\}$. If we receive from the SAT solver a subset $A_0 \subseteq A$ of those assumption that were actually needed to derive contradiction from the given formula, we learn that the possibly weaker conjunction

$$\bigwedge_{v_i \in A_0} N_i$$

is already unsatisfiable.

[1] There are techniques for explicitly minimizing the size of the set of used assumptions and thus the size of the explaining clause. However, we do not consider them in this chapter.

The abstraction of marked clauses

Marker literals are used in LS4 to track dependencies between the learned clauses and the different parts of the input TST. To simplify the pseudocode of the algorithm we introduce the following abstraction of marked clauses.

Formally, a *marked clause* C^m is a pair consisting of a standard clause C and a finite, possibly empty, set of markers m. We use sans-serif symbols like $\mathsf{I}, \mathsf{T}, \mathsf{G}, \ldots$ to denote sets of marked clauses. In logical expressions, the meaning of a marked clause C^m is identical to the meaning of the clause C. However, we adopt a convention that a call to a SAT solver automatically collects markers of all the supplied marked clauses, allocates new variables for marker literals and silently inserts assumptions about the marker literals as describe above. The explaining clause returned in the unsatisfiable case is a marked clause, whose markers correspond to the used assumptions. The explaining clause can be non-empty, if there were additional assumptions explicitly supplied to the solver.

Example 3.1. In the above situation with the conjunctive formula $N = N_1 \wedge \ldots \wedge N_k$, the individual conjuncts can be encoded as sets of marked clauses

$$\mathsf{N}_i = \{C^{\{i\}} \mid C \in N_i\},$$

where we mark each clause in N_i by the index i itself. Because we assume the conjunction N to be unsatisfiable, the query

$$SAT?[\,\mathsf{N}_1 \wedge \ldots \wedge \mathsf{N}_k\,]$$

must return an explaining empty clause of form $\bot^m$ for some $m \subseteq \{1, \ldots, k\}$. Then the formula

$$\bigwedge_{i \in m} N_i$$

is the unsatisfiable core of N automatically extracted by the SAT solver.

3.3 The algorithm LS4

3.3.1 Global variables and invariants

In this section we declare the global variables of LS4 and state the algorithm's main invariants. We assume we are given a temporal satisfiability task $\mathcal{T} = (\Sigma, I, T, G)$ to serve as an input of the algorithm. To simplify our subsequent analysis, we make sure that the input TST $\mathcal{T}$ and its constituent parts are considered read-only by LS4 and thus maintain only one value throughout the run of the algorithm.

Marking the input clauses

The individual sets I, T, and G of initial, step, and goal clauses, respectively, do not directly enter the computation, but are first preprocessed and equipped with markers

for tracking proof dependencies. We mark the initial clauses with a single marker $\circ$ and denote the resulting set I:

$$\mathsf{I} \leftarrow \{C^{\{\circ\}} \mid C \in I\}.$$

We split the set of step clauses T into two subsets, based on whether the respective clauses are or are not simple, i.e. only over the basic signature Σ. The simple step clauses are called *universal* clauses and go into set U, and the remaining "proper" step clauses form the set T. We mark both kind of step clauses with an empty set of markers.

$$\mathsf{U} \leftarrow \{C^{\emptyset} \mid C \in T,\, C \text{ simple}\},$$
$$\mathsf{T} \leftarrow \{C^{\emptyset} \mid C \in T,\, C \text{ not simple}\},$$

We also declare a variable G for storing the goal clauses. Variable G is just an intermediate repository and does not directly participate on forming queries to the SAT solver. That is why we use a dummy marker "_" for the clauses in G:

$$\mathsf{G} \leftarrow \{C^{\{\text{-}\}} \mid C \in G\}.$$

While the sets I and T remain constant during the run of the algorithm, LS4 adds new universal clauses into U when learning from (certain) unsuccessful extensions and new goal clauses into G during the Leap inference. Clause are, however, never deleted from the sets, which means their logical strength is non-decreasing during the run.

This *monotonicity* property of the values of I, T, U and G with respect to time helps to simplify reasoning about LS4. For instance, as we later show, when LS4 learns a new universal clause $C^{\emptyset}$, the clause satisfies

$$\mathsf{T} \wedge (\mathsf{U})' \models C^{\emptyset}, \tag{3.1}$$

with respect to the original value of U. Thanks to monotonicity, relation (3.1) also holds after $C^{\emptyset}$ is added to U and, in general, from that point on. We will rely on monotonicity when proving correctness of LS4.

The partial model

Our algorithm stores the partial model it has built so far in the variable $\mathcal{V}$. Mathematically, $\mathcal{V}$ is a finite sequence $(V_i)_{0 \le i < |\mathcal{V}|}$ of valuations $V_i : \Sigma \to \{\mathbf{0}, \mathbf{1}\}$ over the given signature Σ. We denote by $|\mathcal{V}|$ the length of the sequence and also use this expression for indexing. For example, $V_{|\mathcal{V}|-1}$ stands for the current last valuation in the sequence. Initially, the partial model is empty. There are operations **add** and **remove** for adding an element to the end and removing an element from the end of the sequence.

The configuration of blocks

The configuration of blocks in LS4 determines the current set of time points where the goal clauses are supposed to be satisfied. The algorithm maintains a finite set $\mathcal{B}$ of currently allocated blocks and assigns to each block $b \in \mathcal{B}$ the following two numbers:

- the block's size $s_b \in \mathbb{N}^+$ and
- the index of the block's goal clauses $i_b \in \mathbb{N}$.

The indexes are unique across blocks, which means that they impose a natural ordering. This allows us to treat $\mathcal{B}$ also as a finite sequence $\mathcal{B} = (b_j)_{0 \leq j < |\mathcal{B}|}$, where

$$i_{b_j} < i_{b_{j+1}}$$

for every $0 \leq j < |\mathcal{B}|$. The indexes and sizes of the blocks are related by the following two equations:

$$\begin{aligned} i_{b_0} &= s_{b_0} - 1, \\ i_{b_j} &= s_{b_j} + i_{b_{j-1}}, \end{aligned}$$

where the second equation holds for every $0 < j < |\mathcal{B}|$. LS4 initializes the configuration of blocks $\mathcal{B}$ to contain just one block b of size $s_b = 1$, positioned at index $i_b = 0$.

Associated layers

LS4 collects clauses derived (transitively) from the goal clauses within so called layers. Each block $b \in \mathcal{B}$ is associated with two sequences of sets of marked clauses, the *proper layers* L_i^b or simply layers, and the *dirty layers* D_i^b. For convenience of notation, the index i formally ranges over the whole set of integers $\mathbb{Z}$. However, during computation only finitely many of these sets are non empty and, in particular, L_i^b and D_i^b are always empty for $i < 0$.

Each layer L_i^b is a set of clauses marked by a single marker, the block b itself. For each block b, LS4 maintains that the layer L_0^b contains exactly the goal clauses marked with the marker b:

$$\mathsf{L}_0^b = \{C^{\{b\}} \mid C^{\{\text{-}\}} \in \mathsf{G}\}. \tag{3.2}$$

To preserve (3.2), the layer L_0^b is updated after each Leap inference. The layers L_i^b for $i > 0$ are initialized empty and the algorithm gradually adds clauses to these layers when learning from (certain) unsuccessful extensions.

Semantically, the proper layers satisfy the following two properties

$$\mathsf{G} \models \mathsf{L}_0^b, \tag{3.3}$$

$$\mathsf{T} \wedge (\mathsf{U} \wedge \mathsf{L}_i^b)' \models \mathsf{L}_{i+1}^b. \tag{3.4}$$

We call (3.3) and (3.4) the *initial* and *progress layer property*, respectively. While the initial property is an immediate consequence of (3.2), the progress property depends on the learning process and we will justify it in full later on. Similarly to I, T, U and G the values of layers change only monotonically during the run of the algorithm.[2]

A dirty layer D_i^b contains clauses marked not only by b, but also by some other blocks. Those other blocks always have an index smaller than b. Formally, every element of D_i^b is

[2] Unless a block is deleted, in which case its layers are destroyed.

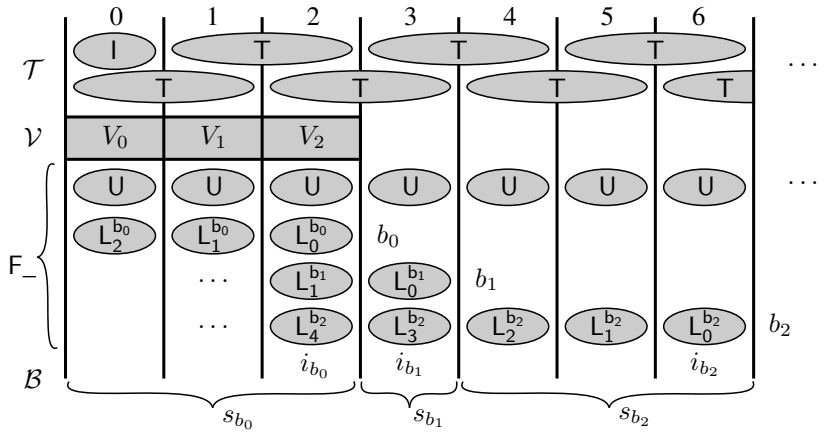


Figure 3.2: Alignment between the clauses of the given TST $\mathcal{T}$, the partial model $\mathcal{V}$, and the blocks $\mathcal{B}$ in LS4. The goal clauses G reside within L_0^b of every block $b \in \mathcal{B}$, marked by the respective block. The dirty layers D_i^b follow the same pattern as L_i^b and are not shown. LS4 will next attempt to compute valuation V_3. If the attempt fails and the derived conflict clause depends on both b_1 and b_2, it will be inserted into $\mathsf{D}_4^{b_2}$.

a marked clause C^m such that 1) $m \subseteq \mathcal{B}$, 2) $b \in m$, 3) $|m| > 1$, and 4) $\max_{b' \in m} i_{b'} = i_b$. Dirty layers of a new block start empty and get populated when new clauses are learned.

The semantics of dirty layers is more complex than that of the proper layers as it also depends on the current configuration of blocks. Because the dirty layers only serve to guide the search of the partial model, but do not directly influence the correctness of LS4, we refrain from explicitly formulating their semantics at this point.

In Figure 3.2, a more elaborate version of Figure 3.1 from the introduction, we can see the intended alignment between the configuration of blocks $\mathcal{B}$, the partial model $\mathcal{V}$, and the sets of marked clauses operated by LS4. The figure shows that the indexes of layers are meant to be interpreted in reversed order to those of the individual valuations of the partial model $\mathcal{V}$. This means that a step from V_i to V_{i+1} in the model corresponds to a step from L_{j+1}^b to L_j^b in the layers. This interpretation reflects the flow of information inside LS4: either the partial model gets extended from V_i to V_{i+1} or the model is backtracked and a new clause derived from L_j^b is learned and added to L_{j+1}^b.

Two key invariants

We close this section by stating two invariants, which will later help us show that LS4 is correct. The first invariant relates the constructed partial model to the input TST.

Invariant 3.1. *Let $\mathcal{T} = (\Sigma, I, T, G)$ be the input TST, $\mathcal{V} = (V_i)_{0 \le i < |\mathcal{V}|}$ the partial model, and $\mathcal{B}$ the current configuration of blocks at any moment during the run of LS4. Then*

1. *$V_0 \models I$, provided $\mathcal{V}$ is not empty,*
2. *$[V_i, V_{i+1}] \models T$ for every $0 \le i < |\mathcal{V}| - 1$, and*

3. $V_{i_b} \models G$ *for every* $b \in \mathcal{B}$ *for which* $i_b < |\mathcal{V}|$.

Invariant 3.1 expresses the standard requirements on any model of a TST restricted in range to the currently constructed part of $\mathcal{V}$. Notice how the configuration of blocks $\mathcal{B}$ dictates (item 3) at what indexes are the goal clauses are supposed to be satisfied.

The second invariant captures a relation between the marked clauses maintained by LS4 and an arbitrary (hypothetical) model of the given TST $\mathcal{T}$. Essentially, it claims that the learned clauses are logically entailed (in their respective contexts) and, therefore, LS4 never discards a potential model by learning them.

Invariant 3.2. *Let* $\mathcal{T} = (\Sigma, I, T, G)$ *be the input TST,* $\mathcal{B}$ *a configuration of blocks, and* I, T, U, G, L_i^b, *and* D_i^b *the values of the respective global variables of LS4 at any moment during the run of the algorithm. Moreover, let* $\mathcal{W} = (W_i)_{i \in \mathbb{N}}$ *be any model of* $\mathcal{T}$. *Then*

1. $W_0 \models \mathsf{I}$,
2. $[W_i, W_{i+1}] \models \mathsf{T}$ *for every* $i \in \mathbb{N}$,
3. $W_i \models \mathsf{U}$ *for every* $i \in \mathbb{N}$,
4. $W_i \models C$ *for every* $C^{\{b\}} \in \mathsf{L}_j^b$ *and every* $i \in \mathbb{N}$ *such that* $W_{i+j} \models \mathsf{G}$, *and*
5. $W_i \models C$ *for every* $C^{\{b_1,\ldots,b_k\}} \in \mathsf{D}_j^b$ *and every* $i \in \mathbb{N}$ *such that* $W_{i+j+(i_{b_l}-i_b)} \models \mathsf{G}$ *for* $l = 1, \ldots, k$.

Items 1 and 2 follow trivially from how LS4 initializes the sets I and T and from the fact that the sets are never modified. Item 3 expresses soundness of learning universal clauses and can be easily derived from relation (3.1). Item 4 states that any clause $C^{\{b\}}$ from a layer L_j^b is bound to hold j steps before the set of goal clauses G in any model. This follows (by induction on j) from the layer properties (3.3) and (3.4) relying on item 3 and on monotonicity. Analogically, we learn from item 5 that any clause $C^{\{b_1,\ldots,b_k\}} \in \mathsf{D}_j^b$ must hold at an appropriate distance to a set of indexes at which the goal clauses hold, where the respective indexes correspond to the involved blocks $b_1, \ldots, b_k$. Item 5 is not needed for showing correctness of LS4 and is mentioned here only for completeness.

3.3.2 Pseudocode

This section presents a detailed pseudocode of LS4 and describes the workings of the algorithm. The code is split into three parts. Algorithm 3.1 defines a procedure for initializing the global variables and another one for creating new blocks, Algorithm 3.2 contains the main body of LS4, i.e. the overall iterative construction of the partial model, and, finally, Algorithm 3.3 provides two additional auxiliary procedures, one for updating the configuration of blocks and the other implementing the Leap inference. The section ends with a demonstration run of LS4 on a simple example.

Initialization

Before LS4 enters the main loop, it calls the procedure InitializeGlobalVariables (Algorithm 3.1 lines 1–7), which sets the initial values for the global variables I, T, U and G by copying and appropriately marking the clauses of the input TST as described in the previous section. The procedure also resets the partial model $\mathcal{V}$ to an empty sequence and initializes the configuration of blocks $\mathcal{B}$ to contain exactly one block, positioned at index 0. The latter is achieved by a call to procedure NewBlock (Algorithm 3.1 lines 8–14), which, in general, allocates a new block b to a given index i and initializes the associated layers L_j^b and D_j^b to empty sets for every $j \in \mathbb{Z}$. An exception is the layer L_0^b, which receives the goal clauses from the set G marked by b itself.

Algorithm 3.1 LS4 – Auxiliary procedures I

```
1: procedure InitializeGlobalVariables(TST 𝒯 = (Σ, I, T, G))
2:     I ← {C^{∘} | C ∈ I} /* Initial clauses marked by ∘ */
3:     U ← {C^{∅} | C ∈ T, C is simple} /* Universal clauses; only over Σ */
4:     T ← {C^{∅} | C ∈ T, C is not simple} /* Proper step clauses; spanning Σ ∪ Σ′ */
5:     G ← {C^{-} | C ∈ G} /* Goal clauses with a dummy marker */
6:     𝒱 ← ∅ /* The partial model is initially empty */
7:     ℬ ← ∅, NewBlock(0) /* Start with one block of size 1 */

8: procedure NewBlock(index i ∈ ℕ)
9:     add new block b to the end of ℬ
10:    i_b ← i /* Index of b's goal time point */
11:    s_b ← 1 /* Size of b */
12:    L_0^b ← {C^{b} | C^{-} ∈ G} /* L_0^b contains goal clauses marked by b */
13:    foreach j ∈ ℤ : j ≠ 0 ⇒ L_j^b ← ∅ /* Remaining layers are empty */
14:    foreach j ∈ ℤ : D_j^b ← ∅ /* All dirty layers are empty */
```

Main loop overview

Every iteration of the main loop of LS4 (see Algorithm 3.2) attempts to extend the partial model $\mathcal{V}$ by one more world, i.e. by one more valuation over the basic signature Σ. This is done by formulating a query formula and calling a SAT solver. If the query formula is satisfiable, the extension succeeds and a new valuation V is extracted from the satisfying assignment and added to $\mathcal{V}$. If, on the other hand, the query formula is unsatisfiable, the extension fails and the solver returns a explaining clause. The clause is analyzed and learned to guide future extension attempts.

The extension query

A query formula is formed and the SAT solver is called on line 3 of the pseudocode. There are two versions of the query depending on whether the current partial model

is empty or not. They both refer to a formula "macro" F_i defined to collect all the clauses that the new valuation should satisfy based on its index i and on the current configuration of blocks:

$$\mathsf{F}_i = \mathsf{U} \wedge \left(\bigwedge_{b \in B} \mathsf{L}^b_{(i_b - i)} \right) \wedge \left(\bigwedge_{b \in B} \mathsf{D}^b_{(i_b - i)} \right).$$

In the previously mentioned Figure 3.2, the collected clauses F_i occupy the same column.

The first version of the query, called the *initial query*, is formed when the partial model $\mathcal{V}$ is currently empty. The query consists of the initial clauses I in conjunction with the formula F_0 and reflects LS4's current requirement on the first valuation V_0 of the partial model:

$$(\mathsf{I} \wedge \mathsf{F}_0)'. \tag{3.5}$$

The query uses priming to align with the second version: in both versions the query is formed over the joint signature $\Sigma \cup \Sigma'$ and we want to extract the new valuation from the Σ'-part of the satisfying assignment.

We call the second version of the query *proper extension query*. It consists of three parts and is formed when the length $|\mathcal{V}|$ of the partial model is greater than zero. It encodes the current last valuation $V_{|\mathcal{V}|-1}$ into an assumption cube over Σ. It supplies the step clauses T to express the desired relation between $V_{|\mathcal{V}|-1}$ and the expected new valuation. And, finally, it expresses the constraints on the new valuation via the formula $\mathsf{F}_{|\mathcal{V}|}$:

$$\mathit{Lits}(V_{|\mathcal{V}|-1}) \wedge \mathsf{T} \wedge (\mathsf{F}_{|\mathcal{V}|})'. \tag{3.6}$$

Observe that $|\mathcal{V}|$ is the index the new valuation will obtain if it is successfully constructed and inserted into the partial model.

Successful extension

When the query formula is satisfiable and the extension succeeds, we execute the following steps (see lines 4–11 in Algorithm 3.2). First, we extract the new valuation V from the Σ'-part of the satisfying assignment. Then, we perform a *model repetition check* to test whether a valuation identical to V does already appear in the partial model $\mathcal{V}$. If this is indeed the case and V equals V_i for some $0 \leq i < |\mathcal{V}|$ and if, moreover, there is a block $b \in \mathcal{B}$ such that

$$i \leq i_b < |\mathcal{V}| \tag{3.7}$$

we terminate the computation reporting that the given TST $\mathcal{T}$ is satisfiable. In this situation, we can complete $\mathcal{V}$ to an ultimately periodic model of $\mathcal{T}$ (see line 7). We must insist on the existence of the block b whose index i_b satisfies (3.7) to make sure that the repeating part of the ultimately periodic model, which will consist of the valuations $V_i, V_{i+1}, \ldots, V_{|\mathcal{V}|-1}$, contains at least one valuation that satisfies the goal clauses. In our case, this will be the valuation V_{i_b}, which satisfies L^b_0 and therefore G.

The model repetition check is succeeded by a *new block check* (lines 8–10). As the partial model grows during the computation, LS4 ensures that there is at least one block

Algorithm 3.2 LS4 – Main loop

Input:
A TST $\mathcal{T} = (\Sigma, I, T, G)$
Output:
An ultimately periodic model $\mathcal{V}$ of $\mathcal{T}$, or a guarantee that none exists

```
1:  InitializeGlobalVariables(T)

2:  loop
3:    if |V| = 0 and SAT?[ (I ∧ F_0)' ] or
          |V| > 0 and SAT?[ Lits(V_{|V|-1}) ∧ T ∧ (F_{|V|})' ] then
4:      V ← valuation extracted from the Σ'-part of the satisfying assignment
5:      /* Model repetition check */
6:      if ∃ i ∈ ℕ, b ∈ B such that V_i = V and i ≤ i_b < |V| then
7:        return λj ∈ ℕ . if j < i then V_j else V_{i+(j-i) mod (|V|-i)}
8:      /* New block check */
9:      if |V| = max_{b∈B} i_b then /* Just completed the last block */
10:       NewBlock(|V| + 1)
11:     add V to the end of V /* |V| increased by 1 */
12:   else /* Unsuccessful extension */
13:     C^m ← the explaining marked clause from unsuccessful extension
14:     if C = ⊥ then /* Empty clause */
15:       if m = ∅ or m = {∘} or m = {b} for some b ∈ B then
16:         return UNSAT /* "Unconditional" empty clause derived */
17:       else
18:         ExtendLastInvolvedBlock(m)
19:     else /* A non-empty clause (over Σ); cannot be marked with ∘ */
20:       remove V_{|V|-1} from V /* |V| decreased by 1 */
21:       if m = ∅ then /* A new universal clause */
22:         add C^m to U
23:       else if m is of the form {b} then /* Depends only on one block */
24:         add C^m to L^b_{(i_b - |V|)}
25:         /* Layer repetition check */
26:         if b = b_{|B|-1} and L^b_i = L^b_j for some 0 < i < j ≤ s_b then
27:           if B = {b} then /* Repetition in the first and only block */
28:             return UNSAT /* Enough empty clauses derivable */
29:           else /* There is more than one block */
30:             Leap(b, i, j - i)
31:       else /* Depends on several blocks: |m| > 1 */
32:         b* ← argmax_{b∈m} i_b /* The last involved block */
33:         add C^m to D^{b*}_{(i_{b*} - |V|)}
```

b with its index "in front of" the last valuation $V_{|\mathcal{V}|-1}$ so that the extensions are always "guided" towards satisfying the goal clauses in some future valuation. Formally, the algorithm maintains the condition

$$|\mathcal{V}| \leq i_{b_{|\mathcal{B}|-1}}. \tag{3.8}$$

This is done by allocating a new block whenever the size of the partial model $|\mathcal{V}|$ equals the index of the last block $i_{b_{|\mathcal{B}|-1}}$ and the condition (3.8) would otherwise be violated after the addition of the new valuation to $\mathcal{V}$. The index of the new block is the smallest possible, namely $|\mathcal{V}| + 1$.

The final step of successful extension is to actually add the new valuation V to $\mathcal{V}$ (line 11). Note that the addition also implicitly increments the value of $|\mathcal{V}|$. This should be taken into account especially when interpreting the usage of $|\mathcal{V}|$ as an index.

By examining the extension queries from the perspective of Invariant 3.1, it is straightforward to establish that the invariant is preserved when a new valuation is added to $\mathcal{V}$ as a result of a successful extension.

Unsuccessful extension

An unsuccessful extension (Algorithm 3.2 lines 13–33) occurs when the query formula is unsatisfiable. In such a case, the SAT solver returns a marked explaining clause C^m. Subsequent execution of LS4 depends on whether the explaining clause is empty or not.

The algorithm first analyzes the case when the explaining clause is empty (lines 14–18). If the set of involved markers m is either empty, equals the singleton set $\{\circ\}$ or the singleton set $\{b\}$ for a block $b \in \mathcal{B}$, LS4 terminates with result UNSAT. Intuitively, this corresponds to an unconditional empty labeled clause being derived by LPSup (recall Definition 2.13) and it means that the input TST $\mathcal{T}$ cannot have model.[3] We will justify this formally in the next section.

If, on the other hand, the set of involved markers m is more complex (and contains both the initial clause marker $\circ$ and a block marker, or two different block markers), the derived empty clause does not entail unsatisfiability of $\mathcal{T}$. Instead, it only signals that the current configuration of blocks cannot yield a model and must be updated accordingly. To this end LS4 calls the procedure ExtendLastInvolvedBlock (Algorithm 3.3 lines 1–5), which

1. extracts from the set of markers m the last block b^* involved in the conflict,
2. discards all the blocks whose index lies further away than i_{b^*},
3. extends b^* by increasing the index i_{b^*} and the size s_{b^*} by one, and
4. clears all the dirty layers of b^*.

Deleting the far blocks (item 2) as well as clearing the dirty layers of b^* (item 4) is a simple way to ensure that all the dirty layers which depend on the block b^*'s old position i_{b^*} will be removed. This is needed to maintain Invariant 3.2 item 5.

[3] The clauses, in this case, would be, respectively, $(*, *) \,||\, \bot$, $(0, *) \,||\, \bot$, and $(*, k) \,||\, \bot$ for $k = |\mathcal{V}| - 1 - i_b$.

Algorithm 3.3 LS4 – Auxiliary procedures II

1: **procedure** ExtendLastInvolvedBlock(marker m)
2: $b^* \leftarrow \operatorname{argmax}_{b \in (m \cap \mathcal{B})} i_b$ /* The last involved block */
3: **remove** $\{b \in B \mid i_b > i_{b^*}\}$ from (the end of) $\mathcal{B}$ /* Delete blocks further away */
4: $i_{b^*} \leftarrow i_{b^*} + 1$, $s_{b^*} \leftarrow s_{b^*} + 1$ /* Extend the last involved block */
5: **foreach** $i \in \mathbb{Z} : \mathsf{D}_i^{b^*} \leftarrow \emptyset$ /* Discard clauses depending on b^* and its old index */

6: **procedure** Leap(block b, offset $o \in \mathbb{N}$, period $p \in \mathbb{N}^+$)
7: /* Assuming b is the last block, but not the first */
8: /* Moreover, $o + p \leq s_b$ and $\mathsf{L}_o^b = \mathsf{L}_{o+p}^b$ */
9: $r \leftarrow p \cdot \lceil o/p \rceil$ /* The only multiple of p such that $o \leq r < o + p$ */
10: $\mathsf{L} \leftarrow \{C^{\{\text{-}\}} \mid C^{\{b\}} \in \mathsf{L}_r^b\}$ /* Clauses to leap; with a dummy marker */
11: **remove** b from (the end of) $\mathcal{B}$ /* Deletes also every L_j^b and D_j^b */
12: $\mathsf{G} \leftarrow \mathsf{G} \cup \mathsf{L}$ /* Strengthen the goal formula ... */
13: **foreach** $b' \in \mathcal{B} : \mathsf{L}_0^{b'} \leftarrow \mathsf{L}_0^{b'} \cup \{C^{\{b'\}} \mid C^{\{\text{-}\}} \in \mathsf{L}\}$ /* ... and the existing blocks */
14: $b^* \leftarrow b_{|\mathcal{B}|-1}$ /* The new last block */
15: **remove** $\{V_i \mid i \geq i_{b^*}\}$ from (the end of) $\mathcal{V}$ /* Backtrack the partial model */

Let us now consider the part of the code which handles the case of a non-empty explaining clause C^m (lines 20–33). To derive a non-empty clause, the solver must have been supplied with explicit assumptions. This means that the unsatisfiable query was necessarily a proper extension query (3.6) and we have $|\mathcal{V}| > 0$ and are dealing with an extension of an existing valuation $V_{|\mathcal{V}|-1}$. It also means that the set of markers m does not contain the initial marker $\circ$ and thus $m \subseteq \mathcal{B}$.

LS4 first backtracks and removes the valuation $V = V_{|\mathcal{V}|-1}$ from the partial model $\mathcal{V}$ (line 20), which decreases the value of $|\mathcal{V}|$ by one. Then it proceeds to learning the explaining clause C^m. Depending on the involved markers m, the clause is either added

- to the set of universal clauses U in the case the set m is empty (line 22),
- to the proper layer $\mathsf{L}_{(i_b - |\mathcal{V}|)}^b$ of the respective block in the case of a singleton set $m = \{b\}$ (line 24), and
- to the dirty layer $\mathsf{D}_{(i_{b^*} - |\mathcal{V}|)}^{b^*}$ of the last involved block b^* in the case of more than one involved block (line 33).

The general relation (recall Section 3.2) between the explaining clause and the proper extension query can be expressed as

$$\mathsf{T} \wedge (\mathsf{U} \wedge \bigwedge_{b \in m} \mathsf{L}_{(i_b - i)}^b \wedge \bigwedge_{b \in m} \mathsf{D}_{(i_b - i)}^b)' \models C^m, \tag{3.9}$$

where $i = |\mathcal{V}|$ is the orignal length of the partial model before backtracking. Notice, however, that (3.9) simplifies to

$$\mathsf{T} \wedge (\mathsf{U} \wedge \mathsf{L}_{(i_b - i)}^b)' \models C^{\{b\}}, \tag{3.10}$$

when there is only one marker involved, i.e. when $m = \{b\}$, and further to

$$\mathsf{T} \wedge (\mathsf{U})' \models C^{\emptyset}, \tag{3.11}$$

when the set of involved markers m is empty. Recall that relation (3.11) has already been mentioned as the key to showing that learning universal clauses is sound (Invariant 3.2 item 3) and also notice that (3.10) entails the progress layer property (3.4) from which the soundness of proper layers (Invariant 3.2 item 4) follows. Similarly, one can use (3.9) to infer the analogous property of dirty layers (Invariant 3.2 item 5).

If the explaining clause involves only one block b and this block is the current last block, i.e. $b = b_{|\mathcal{B}|-1}$, LS4 executes a *layer repetition check* (lines 26–30). The purpose of the check is to look for two layers of the block b that have distinct indexes but are equal as sets of clauses: $\mathsf{L}_i^b = \mathsf{L}_j^b$ for some $0 < i < j \leq s_b$. In analogy to the calculus LPSup, the layer repetition check is a key to detecting unsatisfiability and to initiating the Leap inference.

Unsatisfiability is reported, if the block b is currently the only allocated block (line 27). This corresponds in LPSup to the case when the conditional empty clauses $(0, k) \,||\, \bot$ are derivable for every $k \in \mathbb{N}$. Also in LS4, unsatisfiability is formally shown using a derivation replaying argument (see the next section).

If there is more than one block currently allocated, i.e. $|\mathcal{B}| > 1$, the Leap inference is executed via a call to the Leap procedure (Algorithm 3.3 lines 6–15). Similarly to LPSup, the algorithm first identifies an offset $o \in \mathbb{N}$ and period $p \in \mathbb{N}^+$ of the repetition (in our formulation $o = i$ and $p = j - i$), it then computes an index r of the layer to leap as the only multiple of p in the range $o, o + 1, \ldots, o + p - 1$ (line 9), and finally uses the clauses from layer L_r^b to strengthen the set of goal clauses G (marked by the dummy marker) and the layers $\mathsf{L}_0^{b'}$ for every block $b' \in \mathcal{B} \setminus \{b\}$ (marked by the respective block b' itself, as usual). The last block b is removed from $\mathcal{B}$ and the partial model $\mathcal{V}$ is backtracked to such a state that the next iteration of the main loop will attempt to construct a valuation for the index i_{b^*}, where b^* is the new last block after the removal of b from $\mathcal{B}$. Formal analysis of the Leap inference is postponed to the next section.

Example execution

Let us demonstrate execution of LS4 on a simple example defined by the following unsatisfiable TST over a two element signature

$$\mathcal{T} = (\{a, b\}, \{a \vee \neg b\}, \{\neg a \vee a', \neg b \vee a'\}, \{\neg a, b\}).$$

The execution is presented in Table 3.2. Each line in the table represent an event in the run of the algorithm, which is either a call to the SAT solver or to one of the auxiliary procedures. Corresponding updates to the global variables are given in the rightmost column. The example uses natural numbers to represent blocks ($\mathcal{B} \subset \mathbb{N}$), which removes one level of indirection from index expressions. Because the set of universal clauses U remains empty during the whole execution, it was omitted from the query formulas.

After necessary initialization, LS4 learns that the initial clauses and the goal clauses cannot be satisfied together (id 1). The algorithm then succeeds in satisfying the goal

Table 3.2: Example execution of LS4.

id	call	update
–	InitializeGlobalVariables	$\mathsf{I} \leftarrow \{(a \vee \neg b)^{\{\circ\}}\}$, $\mathsf{G} \leftarrow \{(\neg a)^{\{\text{-}\}}, (b)^{\{\text{-}\}}\}$, $\mathsf{U} \leftarrow \emptyset$, $\mathsf{T} \leftarrow \{(\neg a \vee a')^{\emptyset}, (\neg b \vee a')^{\emptyset}\}$, $\mathcal{V} \leftarrow \emptyset$, $\mathcal{B} \leftarrow \emptyset$, add 0 to $\mathcal{B}$, $i_0 \leftarrow 0$, $s_0 \leftarrow 1$, $\mathsf{L}_0^0 \leftarrow \{(\neg a)^{\{0\}}, (b)^{\{0\}}\}$, other layers empty
1	$SAT?[\,(\mathsf{I} \wedge \mathsf{L}_0^0)'\,]$ false	derived $\bot^{\{\circ,0\}}$
	ExtendLastInvolvedBlock	$i_0 = 1, s_0 = 2$
2	$SAT?[\,(\mathsf{I} \wedge \mathsf{L}_1^0)'\,]$ true	add $V_0 = \{a \mapsto \mathbf{0}, b \mapsto \mathbf{0}\}$ to $\mathcal{V}$
3	$SAT?[\,Lits(V_0) \wedge \mathsf{T} \wedge (\mathsf{L}_0^0)'\,]$ true	add $V_1 = \{a \mapsto \mathbf{0}, b \mapsto \mathbf{1}\}$ to $\mathcal{V}$
	NewBlock	add 1 to $\mathcal{B}$, $i_1 = 2$, $s_1 = 1$, $\mathsf{L}_0^1 = \{(\neg a)^{\{1\}}, (b)^{\{1\}}\}$, other layers empty
4	$SAT?[\,Lits(V_1) \wedge \mathsf{T} \wedge (\mathsf{L}_0^1)'\,]$ false	add $\{\neg b\}^{\{1\}}$ to L_1^1, remove V_1 from $\mathcal{V}$
5	$SAT?[\,Lits(V_0) \wedge \mathsf{T} \wedge (\mathsf{L}_0^0 \wedge \mathsf{L}_1^1)'\,]$ false	derived $\bot^{\{0,1\}}$
	ExtendLastInvolvedBlock	$i_1 = 3, s_1 = 2$
6	$SAT?[\,Lits(V_0) \wedge \mathsf{T} \wedge (\mathsf{L}_0^0 \wedge \mathsf{L}_2^1)'\,]$ true	add $V_1 = \{a \mapsto \mathbf{0}, b \mapsto \mathbf{1}\}$ to $\mathcal{V}$
7	$SAT?[\,Lits(V_1) \wedge \mathsf{T} \wedge (\mathsf{L}_1^1)'\,]$ true	add $V_2 = \{a \mapsto \mathbf{1}, b \mapsto \mathbf{0}\}$ to $\mathcal{V}$
8	$SAT?[\,Lits(V_2) \wedge \mathsf{T} \wedge (\mathsf{L}_0^1)'\,]$ false	add $\{\neg a\}^{\{1\}}$ to L_1^1, remove V_2 from $\mathcal{V}$
9	$SAT?[\,Lits(V_1) \wedge \mathsf{T} \wedge (\mathsf{L}_1^1)'\,]$ false	add $\{\neg b\}^{\{1\}}$ to L_2^1
10	$SAT?[\,Lits(V_0) \wedge \mathsf{T} \wedge (\mathsf{L}_0^0 \wedge \mathsf{L}_2^1)'\,]$ false	derived $\bot^{\{0,1\}}$
	ExtendLastInvolvedBlock	$i_1 = 4, s_1 = 3$
11	$SAT?[\,Lits(V_0) \wedge \mathsf{T} \wedge (\mathsf{L}_0^0 \wedge \mathsf{L}_3^1)'\,]$ true	add $V_1 = \{a \mapsto \mathbf{0}, b \mapsto \mathbf{1}\}$ to $\mathcal{V}$
12	$SAT?[\,Lits(V_1) \wedge \mathsf{T} \wedge (\mathsf{L}_2^1)'\,]$ true	add $V_2 = \{a \mapsto \mathbf{1}, b \mapsto \mathbf{0}\}$ to $\mathcal{V}$
13	$SAT?[\,Lits(V_2) \wedge \mathsf{T} \wedge (\mathsf{L}_1^1)'\,]$ false	add $\{\neg a\}^{\{1\}}$ to L_2^1, layers repeat: $\mathsf{L}_1^1 = \mathsf{L}_2^1$
	Leap	remove 1 from $\mathcal{B}$, add $\{\neg b\}^{\{0\}}$ and $\{\neg a\}^{\{0\}}$ to L_0^0, remove V_2 and V_1 from $\mathcal{V}$
14	$SAT?[\,Lits(V_0) \wedge \mathsf{T} \wedge (\mathsf{L}_0^0)'\,]$ false	derived $\bot^{\{0\}}$, return UNSAT

clauses at index $i_0 = 1$, after which it allocates a second block (id 3). Satisfying the goal clauses for a second time is, however, impossible, because the only goal vertex in the semantic graph for $\mathcal{T}$ (recall Section 2.4.2), namely $V = \{a \mapsto \mathbf{0}, b \mapsto \mathbf{1}\}$, does not lie on a cycle. LS4 recognizes this situation by detecting a repetition in layers of the second block (id 13). A subsequent Leap inference makes the set of goal clauses inconsistent, which is immediately detected in the next call to the solver (id 14).

3.3.3 Correctness

In this section we argue that LS4 is correct, meaning that the algorithm only returns UNSAT when the given TST $\mathcal{T}$ does not have a model, and that the algorithm only returns an ultimately periodic interpretation $\mathcal{V}$ when this interpretation is indeed a model of $\mathcal{T}$. Together with a termination proof, which we provide in the next section, this will show that LS4 is a decision procedure for TSTs and thus for LTL.

Our strategy to proving the correctness is the following. First, we analyze layer repetitions of LS4 and define infinitely repeating layers, a semantic version of derivation replaying argument from LPSup. We then use this concept in two places: 1) to prove soundness of the Leap inference and 2) to justify the case of "conditional" empty clauses in the main theorem.

All the line numbers below refer to the pseudocode of the main loop (Algorithm 3.2).

Layer repetitions

When LS4 detects a repetition of layers (line 26) we know there is a block $b \in \mathcal{B}$ and indexes $0 < i < j \leq s_b$ such that $\mathsf{L}_i^b = \mathsf{L}_j^b$. We define the following sequence of (marked) clauses L_k, called the *infinitely repeating layers* derived from the repetition $\mathsf{L}_i^b = \mathsf{L}_j^b$ in block b, by setting

$$\mathsf{L}_k = \begin{cases} \mathsf{L}_k^b & \text{for } 0 \leq k < i, \\ \mathsf{L}_{i+(k-i) \bmod (j-i)}^b & \text{for } k \geq i. \end{cases}$$

It is straightforward to check that the infinitely repeating layers L_k satisfy the initial and progress layer properties (3.3) and (3.4), and, therefore, the following analogy of Invariant 3.2 item 4 holds for the infinitely repeating layers.

Lemma 3.1. *Let $\mathcal{W} = (W_l)_{l \in \mathbb{N}}$ be any model of the input TST $\mathcal{T}$ and let L_k be the infinitely repeating layers derived from a repetition. Then for every $k, l \in \mathbb{N}$*

$$\text{if } W_{k+l} \models \mathsf{G} \text{ then } W_l \models \mathsf{L}_k.$$

Soundness of Leap

Although LS4 does not explicitly attempt to construct a (K, L)-model (see Definition 2.6), it relies on the (K, L)-model semantics to justify soundness of Leap. The Leap inference may remove some standard models from consideration, but guarantees to preserve the existence of at least one (K, L)-model.

Lemma 3.2. *Let $\mathcal{T} = (\Sigma, I, T, G)$ be the input TST of LS4. Assume the algorithm has just detected a layer repetition $\mathsf{L}_i^b = \mathsf{L}_j^b$ for a block $b \in \mathcal{B}$ and indexes $0 < i < j \leq s_b$. Let $o = i$ and $p = (j - i)$ be the offset and period passed to the* Leap *procedure. Furthermore, let $r = p \cdot \lceil o/p \rceil$ be the only multiple of p such that $o \leq r < o + p$ and let $G^+ = \{C \mid C^{\{\text{-}\}} \in \mathsf{G}\}$ and $H = \{C \mid C^{\{b\}} \in \mathsf{L}_r^b\}$. If the TST $\mathcal{T}^+ = (\Sigma, I, T, G^+)$ is satisfiable then so is the TST $\mathcal{T}^{++} = (\Sigma, I, T, G^+ \cup H)$.*

Proof. Let us assume that the TST $\mathcal{T}^+$ is satisfiable. By Lemma 2.1 (page 23) it must have a (K, L)-model $\mathcal{W} = (W_k)_{k \in \mathbb{N}}$ for some $K \in \mathbb{N}$ and $L \in \mathbb{N}^+$. In such a model

$$W_{K+l \cdot L} \models G^+ \tag{3.12}$$

for every $l \in \mathbb{N}$. We will prove the lemma by showing that at the indexes of the form $K + l \cdot L$ the model $\mathcal{W}$, in fact, also satisfies the formula H.

Let $K + l \cdot L$ for $l \in \mathbb{N}$ be such an index. We consider the following linear Diophantine equation

$$l' \cdot L = r + k' \cdot p, \tag{3.13}$$

which must have a solution pair $l', k' \in \mathbb{N}$, because r is a multiple of p. It follows from (3.12) that $W_{K+(l+l') \cdot L} \models G^+$ or, equivalently, $W_{(K+l \cdot L)+l' \cdot L} \models \mathsf{G}$. By Lemma 3.1, we obtain

$$W_{K+l \cdot L} \models \mathsf{L}_{l' \cdot L},$$

where L_k is the sequence of the infinitely repeating layers derived from our repetition. Now $\mathsf{L}_{l' \cdot L} = \mathsf{L}_{r+k' \cdot p}$ by (3.13) and $\mathsf{L}_{r+k' \cdot p} = \mathsf{L}_r = \mathsf{L}_r^b$ by the definition of the infinitely repeating layers. Therefore, $W_{K+l \cdot L} \models H$, since the sets L_r^b and H contain the same clauses up to the markers. □

Correctness theorem

Below we prove the main theorem of this section. When justifying correctness in the unsatisfiable case, the presented contradictions depend on the values of U and G at the moment when they are detected. We implicitly rely on the soundness of learning universal clauses (Invariant 3.2 item 3) and the soundness of the Leap inference (Lemma 3.2) to relate these contradictions back to the original input TST $\mathcal{T}$.

Theorem 3.1. *LS4 only returns UNSAT when the input TST $\mathcal{T}$ does not have a model, and it only returns an ultimately periodic interpretation $\mathcal{W}$ when $\mathcal{W}$ is a model of $\mathcal{T}$.*

Proof. Let us first consider the case when LS4 returns an ultimately periodic interpretation. This happens when the model repetition check (line 6) succeeds and there is an index i and a block $b \in \mathcal{B}$ such that $i \leq i_b < |\mathcal{V}|$ and $V_i = V$. Recall that V is the just extracted valuation to be added to $\mathcal{V}$ after a successful extension. The ultimately periodic interpretation $\mathcal{W} = (W_j)_{j \in \mathbb{N}}$ to be returned is defined (see also line 7) by

$$W_j = \begin{cases} V_j & \text{for } 0 \leq j < i, \\ V_{i+(j-i) \bmod (|\mathcal{V}|-i)} & \text{for } i \leq j. \end{cases}$$

It follows from Invariant 3.1 that $\mathcal{W}$ is indeed a model of $\mathcal{T}$. In particular, the goal clauses G are satisfied at every index of the form $i_b + j \cdot (|\mathcal{V}| - i)$ for $j \in \mathbb{N}$.

When LS4 returns UNSAT on line 16, it means it has just derived an empty clause $\bot^m$ from an unsuccessful extension. Depending on the set of markers m and on the kind of the extension query, which could have been either the initial query (3.5) or the proper extension query (3.6), there are the following entailments to consider:

- $\mathsf{U} \models \bot^{\emptyset}$ or $\mathsf{T} \wedge (\mathsf{U})' \models \bot^{\emptyset}$,
- $\mathsf{I} \wedge \mathsf{U} \models \bot^{\{\circ\}}$, or
- $\mathsf{U} \wedge \mathsf{L}_i^b \models \bot^{\{b\}}$ or $\mathsf{T} \wedge (\mathsf{U} \wedge \mathsf{L}_i^b)' \models \bot^{\{b\}}$ for some $i \in \mathbb{N}$ and $b \in \mathcal{B}$.

The first two options imply contradiction in the set of universal clauses, the third option the same between the initial and universal clauses, and with the last two options the contradiction arises i (or $i + 1$) steps before the goal clauses can be satisfied (see Invariant 3.2 item 4). We can see that the input TST $\mathcal{T}$ cannot have a model in any of the listed cases.

When LS4 returns UNSAT on line 28, it has just detected a repetition in the first block (line 27), which means that $\mathcal{B} = \{b\}$ and $\mathsf{L}_i^b = \mathsf{L}_j^b$ for some indexes $0 < i < j \leq s_b$. Previously, there must have been $(s_b - 1)$ moments during the run of the algorithm when the block b was extended. Because b is the first block, the extensions happened as a result of deriving the empty explaining clause $\bot^{\{\circ,b\}}$ from an unsuccessful extension, which means that the conjunction

$$\mathsf{I} \wedge \mathsf{U} \wedge \mathsf{L}_l^b, \tag{3.14}$$

was (and remained thanks to monotonicity) unsatisfiable for every $0 \leq l < s_b$.

Let us now recall the sequence L_k of infinitely repeating layers derived from our repetition. It is easy to see that every L_k for $k \in \mathbb{N}$ is equal to some L_l^b for $0 \leq l < s_b$. This implies there cannot be a model of the input TST $\mathcal{T}$. Indeed, suppose otherwise. Let $\mathcal{W} = (W_l)_{l \in \mathbb{N}}$ be such a model and let $k \in \mathbb{N}$ be the first index at which the goal clauses G are satisfied in $\mathcal{W}$:

$$W_k \models \mathsf{G}.$$

It follows by Lemma 3.1 that $W_0 \models L_k$ in that model and so $W_0 \models \mathsf{L}_l^b$ for some $0 \leq l < s_b$. But this in not possible, because (3.14) is unsatisfiable. A contradiction. □

3.3.4 Termination

In this section we demonstrate that any computation of LS4 terminates. We do this by stating and proving four lemmas and putting them together in a final theorem, which provides an upper bound on the running time.

Central to our proof is an idea of a "proper strengthening" of a clause set. Because LS4 uses a SAT solver, deriving and learning a new clause always means that the current partial model must be changed and can never again assume the same form. We use this idea directly in the proof of Lemma 3.5 and in a much more subtle way in Lemma 3.6, which deals with the Leap inference and its role in the termination argument.

Unless explicitly stated otherwise, the line numbers below refer to the pseudocode of the main loop of LS4 (Algorithm 3.2).

Lemma 3.3. *The number of active blocks $|\mathcal{B}|$ is bounded by $2^{|\Sigma|}$ during the run of LS4.*

Proof. Let $\mathcal{B}$ be a configuration of blocks and let $b = b_{|\mathcal{B}|-1}$ be the current last block. Every time a new block is about to be created (line 9) the new valuation V satisfies L_0^b and therefore G. At that moment, LS4 also registers $|\mathcal{B}|-1$ other valuations satisfying G, namely the valuations $V_{i_{b'}}$ for $b' \in \mathcal{B} \setminus \{b\}$. By inspecting the model repetition condition (line 6) at the moments when these valuations were generated, we can observe that they are all distinct. Thus, $|\mathcal{B}| \leq 2^{|\Sigma|}$ since there are altogether at most as many valuations over Σ, let alone valuations satisfying G. □

Lemma 3.4. *Let $\mathcal{B}$ be a configuration of blocks and $b = b_{|\mathcal{B}|-1}$ the last block. The size of the last block s_b is bounded by 2^c, where $c = 3^{|\Sigma|}$ denotes the number of different non-tautological clauses over the signature Σ.*

Proof. Every block b is created with $\mathsf{L}_i^b = \emptyset$ for every $i \in \mathbb{N}^+$ and whenever b is extended, its layers $\mathsf{L}_1^b, \ldots, \mathsf{L}_{s_b}^b$ are necessarily non-empty, because the marker b could otherwise not be involved in the proof of the respective empty clause. Therefore, continual extensions of the block b must be interleaved by additions of new clauses to the layers L_i^b and thus also by layer repetition checks (line 26). But there is at most 2^c different sets of clauses over the signature Σ and so a repetition is bound to occur before s_b exceeds 2^c. □

The following lemma estimates the maximal number of iterations of the main loop for an unchanging configuration of blocks $\mathcal{B}$. Note that the three mentioned procedures are the only points where a configuration gets modified.

Lemma 3.5. *Let $\mathcal{B}$ be a configuration of blocks and let $b = b_{|\mathcal{B}|-1}$ be the last block. LS4 performs at most $O(i_b \cdot 2^{|\Sigma|})$ iterations of the main loop before updating $\mathcal{B}$ by calling the procedures* NewBlock, ExtendLastInvolvedBlock *or* Leap.

Proof. Unless NewBlock is called, the length $|\mathcal{V}|$ of the partial valuation cannot exceed the index of the last block i_b. Thus, the number of successful extensions minus the number of unsuccessful extensions is at any moment bounded by $i_b + 1$. Moreover, every unsuccessful extension properly strengthens either the set U or a layer $\mathsf{L}_j^{b'}$ or a dirty layer $\mathsf{D}_j^{b'}$ for some block $b' \in \mathcal{B}$ and $j \in \mathbb{N}$, which means that the removed valuation $V_{|\mathcal{V}|-1}$ (see line 20) cannot be later generated again at that particular index. □

Lemma 3.6. *Each invocation of the Leap inference properly strengthens the set of goal clauses G. Therefore, LS4 performs at most $O(2^{|\Sigma|})$ invocations of the Leap inference.*

Proof. Let $\mathcal{B}$ be a configuration of blocks, let $b = b_{|\mathcal{B}|-1}$ be the last block that is not a first block, and let $b_1 = b_{|\mathcal{B}|-2}$ be the second last block. Furthermore, let V be the valuation that LS4 added to $\mathcal{V}$ just after the block b was created (see lines 9–11). At that moment, which we denote $\mathfrak{m}_1$, we have $V \models \mathsf{L}_0^{b_1}$ and so $V \models \mathsf{G}$.

Let us consider a later moment $\mathfrak{m}_3$ when the Leap inference is invoked (line 30) for b as the last block. At that moment, there are numbers $0 < i < j \leq s_b$ such that $\mathsf{L}^b_i = \mathsf{L}^b_j$. The Leap procedure (Algorithm 3.3) computes an index r in the range $i \leq r < j$ and globally strengthens G by the clauses from L^b_r. To complete the proof we show that $V \not\models \mathsf{L}^b_r$ and thus, after the strengthening, $V \not\models \mathsf{G}$.

We consider a moment $\mathfrak{m}_2$, between $\mathfrak{m}_1$ and $\mathfrak{m}_3$, when the size s_b of the block b was equal to r and the block b was to be extended because of an empty clause (line 18). By deriving the empty clause, LS4 has proven that the configuration of blocks $\mathcal{B} \setminus \{b\}$ with an additional requirement that L^b_r be satisfied at index $i_{b_1} = i_b - r$ does not allow any partial model. Yet, at moment $\mathfrak{m}_1$, there was a partial model $\mathcal{V}$ with V as the last valuation at index i_{b_1} that complied with the configuration $\mathcal{B} \setminus \{b\}$ except for the additional requirement. Therefore, V cannot satisfy L^b_r. This holds already for the value of L^b_r at moment $\mathfrak{m}_2$, and so all the more for the value at moment $\mathfrak{m}_3$ after potential strengthenings of L^b_r that could happen in between. □

Theorem 3.2. *For any input TST $\mathcal{T} = (\Sigma, I, T, G)$ LS4 terminates. Its running time is at most doubly-exponential in the size of the signature Σ.*

Proof. By Lemmas 3.3 and 3.4, there is at most $d = 2^{|\Sigma|} \cdot 2^{3^{|\Sigma|}}$ different configurations of blocks that the algorithm can consider. Also, d is a bound on the maximal value of the last block's index i_b. Thus, by Lemma 3.5, LS4 can spend at most $O(d \cdot 2^{|\Sigma|})$ iterations of the main loop in one configuration. By treating each configuration $\mathcal{B}$ as a sequence $(i_{b_0}, \ldots, i_{b_{|\mathcal{B}|-1}})$ and ordering the sequences lexicographically, we can observe that LS4 always transitions to a strictly larger configuration during calls to NewBlock and ExtendLastInvolvedBlock. This means a configuration cannot be reconsidered unless Leap is called. But by Lemma 3.6, there can be at most $2^{|\Sigma|}$ calls to Leap. The time spent during one iteration of the main loop is dominated by the call to the SAT solver and can be bounded by $O(2^{|\Sigma|})$. In total, we obtain

$$O(d^2 \cdot (2^{|\Sigma|})^3) = 2^{2^{O(|\Sigma|)}}$$

as our bound on the running time of LS4. □

Although a doubly-exponential theoretical upper bound on the running time of an algorithm may seem discouraging, we demonstrate in the next section that the practical performance of LS4 is actually quite good.

3.4 Practical experience

3.4.1 Implementation

We implemented LS4 using Minisat (Eén and Sörensson, 2003a) version 2.2 as the underlying SAT solver. In this section we describe our implementation, focusing on how it utilizes an incremental interface which Minisat provides.

Incremental SAT solving

The incremental interface of Minisat (Eén and Sörensson, 2003b) allows allocating new variables and adding new clauses between individual solver calls. Although the option to explicitly delete clauses is not available, a similar effect can be achieved via marking. We can selectively deactivate a set of clauses marked by a marker literal $\neg v$ by omitting the corresponding assumption v from the particular call. Permanent deletion can be emulated by adding the unit clause $\{\neg v\}$, which will subsume the respective marked clauses. From that moment on, however, the marking variable v cannot be reused.

Our implementation of LS4 relies on the incremental interface of Minisat to minimize copying of clauses. We allocate as many instances of the solver as there are currently considered indexes of the partial model (more precisely, the number is equal to the sum of the sizes of the currently allocated blocks) and assign each solver to handling calls corresponding to a particular index. In this setup, each new explaining clause needs only to be inserted into one solver. Another advantage is that clauses learned internally by the CDCL algorithm are preserved and get reused between the calls.

Clause deletion is only needed when extending blocks. In particular, we need to delete clauses that correspond to the dirty layers of the extended block and to move clauses that correspond to the proper layers of that block. The latter can be partially achieved by instead moving the corresponding solvers and assigning a new solver for handling the index of the gap thus created. Thus, for example, when the extended block is the first block, we need to deactivate the initial clauses in the originally first solver, which will next be replaced by the new solver for handling calls at index 0.

Design choices

We have implemented an abstraction layer on top of the Minisat code to encapsulate the marking mechanism and the above described clause deactivation and deletion. We use the `vector` container for storing the partial model and the standard `set` container to represent layers. A simple linear scan realizes both the model repetition check and the layer repetition check. We compute hash signatures of valuations and use them as a pre-filter in the model repetition check. We found out that most of the overall running time is typically spent inside individual calls to Minisat and therefore did not attempt any further optimizations of the checks.

Our implementation, including the marking abstraction layer, comprises little less then 1K lines of C++ code. The implementation is publicly available (Suda, 2012a).

3.4.2 Experimental evaluation

In 2011, Schuppan and Darmawan (2011) performed a comprehensive experimental evaluation of off-the-self satisfiability solvers for LTL. They collected an extensive set of benchmarks and identified mature tools within each of the three main approaches to LTL satisfiability: 1) reduction to model checking, 2) tableaux methods, and 3) resolution. We evaluated LS4 on the mentioned benchmark set and compared its performance

to the most successful solvers from each category. In this section we report on our experiment.

Solvers

Here we list the solvers that we selected for our evaluation. For each solver we choose one or two configurations of command line options (typeset in `teletype`) which yielded the best results on the used benchmark set.

- NuSMV (Cimatti et al., 2002) is a symbolic model checker implementing several techniques. In our experiments, we used the following two configurations of NuSMV version 2.5.4. Configuration `NuSMV-BDD` performs symbolic model checking of LTL using fixed point computation with BDDs (Clarke et al., 1997). We enabled forward computation of reachable states and dynamic variable reordering. Configuration `NuSMV-BMC` performs incremental simple bounded model checking (Heljanko et al., 2005) using Minisat as a SAT solver. The configuration uses a completeness check and so it can detect unsatisfiable inputs.

- PLTL (Goré, 2012) is an Ocaml implementation of two tableau-based algorithms for LTL checking by Florian Widmann. The first, which we will denote `PTLT-tree`, is based on the "one-pass" method outlined by Schwendimann (1998). The second, `PTLT-graph`, is an LTL version of the "on-the-fly" algorithm proposed by Goré and Widmann (2009).

- TRP++ (Konev, 2012) is a temporal resolution prover by Hustadt and Konev (2003). It implements a saturation strategy within the framework of the CTR calculus (Fisher et al., 2001). We used TRP++ version 2.1 in the default mode (to be denoted `TRP++`).

- STRP (Williams and Konev, 2013) is a recently developed LTL prover based on the simplified temporal resolution calculus (Degtyarev et al., 2002). It reduces the search for premises in the calculus to the minimal unsatisfiable subsets (MUS) extraction problem and uses external tools for solving it. The configuration `STRP` which we used in our experiment delegates the extraction to the tool `shd`.

Remark 3.1. We did not include our implementation of the LPSup saturating decision procedure (Section 2.5.3) in this experiment. Since it is just a prototype, which was not optimized for speed in any way, it would probably end up last in the comparison. A proper experimental comparison of LPSup and CTR in the saturating setting would require a new implementation comparable in engineering discipline to TRP++ and as such is left for future work. However, in Section 3.5 we provide further theoretical insights concerning the saturation approach on the one hand and model guidance on the other.

Benchmarks, normal form transformations, and the experimental setup

The set of benchmarks collected by Schuppan and Darmawan (2011) consists of a total of 3723 problems from various sources (mostly previous papers on LTL satisfiability) and of various flavors (application, crafted, random), and represents the most comprehensive collection of LTL problems we are aware of. The set is available online[4] and contains several syntactic variations of each problem. The variations correspond to individual input formats supported by the respective solvers.

The provers TRP++ and STRP require that the input LTL formula be first transformed into SNF before the main algorithm can be started. We used the `translator`[5] tool provided with TRP++ to transform the formulas. Because `translator` performs some non-trivial simplifications, it can take substantial time on larger inputs. (In some cases, it did not even terminate within our time limit.) For fairness of the comparison, we decided to include the time spent within `translator` into the overall run time of TRP++ and STRP reported below.

We created a simple tool `TST-translate` for transforming the input into TST as required by LS4. The tool is written in SWI-prolog and implements the linear time algorithm described in Section 2.2.2. Similarly to the case of `translator`, the run time of `TST-translate` is included in the reported performance of LS4. In what follows, we denote by `LS4` the configuration combining the two programs.

We performed our experiments on servers with 3.16 GHz Intel Xeon CPU, 16 GB RAM, running Debian 6.0. We measured the run time by a simple Python script harness. The measurement precision was 0.1 seconds.

Results

We ran `LS4` and all the other selected solvers on each of the 3723 problems with a time limit of 300 seconds per problem. Figure 3.3 plots the number of problems solved within a given time. We can see that `LS4` solves the most problems out of all the solvers and does so very fast. Indeed, `LS4` solves 3461 problems when given just one second per problem. This is more than what the runner-up, `NuSMV-BDD`, achieves within the full limit of 300 seconds per problem.

The perspective of Figure 3.3 attributes similar performance to the related solvers within the pairs `NuSMV-BDD` and `NuSMV-BMC`, `STRP` and `TRP++`, and `PLTL-graph` and `PLTL-tree`, respectively. In each of the pairs, the first solver solves eventually more problems than the second, but the final difference is small and the crossover point lies further than the 50-seconds-per-problem mark. This is probably most interesting in the case of the closely related tableau-based algorithms `PLTL-graph` and `PLTL-tree`, where we see that the theoretically superior `PLTL-graph`, which is only of singly exponential worst case complexity, eventually beats the initially faster `PLTL-tree`, whose worst case complexity is doubly exponential.

[4] `http://www.schuppan.de/viktor/atva11/`
[5] `http://cgi.csc.liv.ac.uk/~konev/software/trp++/translator/`

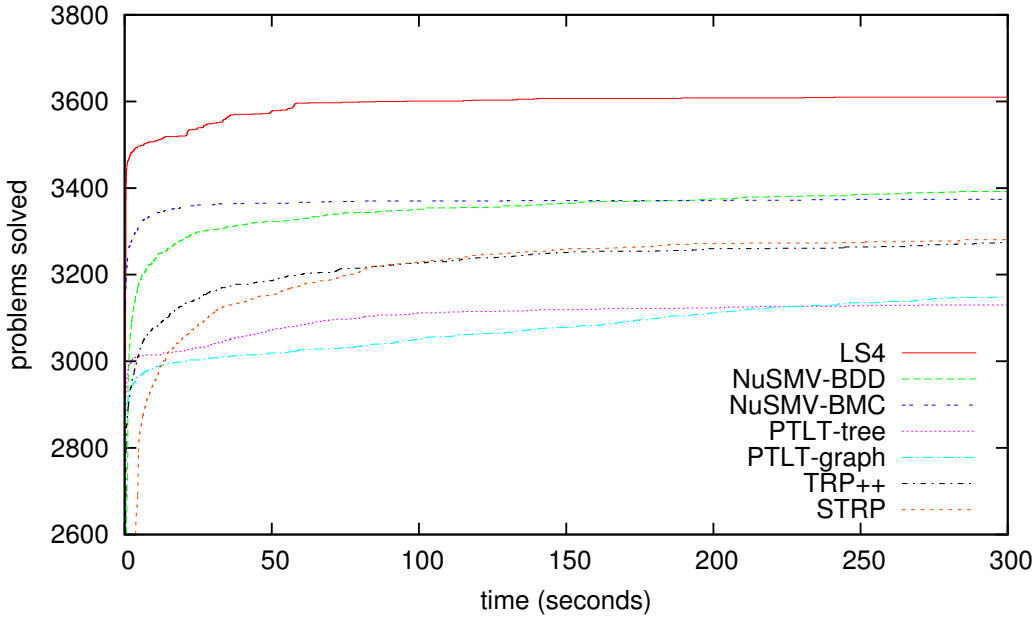


Figure 3.3: Comparison of LS4 and the other LTL solvers – I. Showing the number of problems out of the whole benchmark set that were solved within a given time limit.

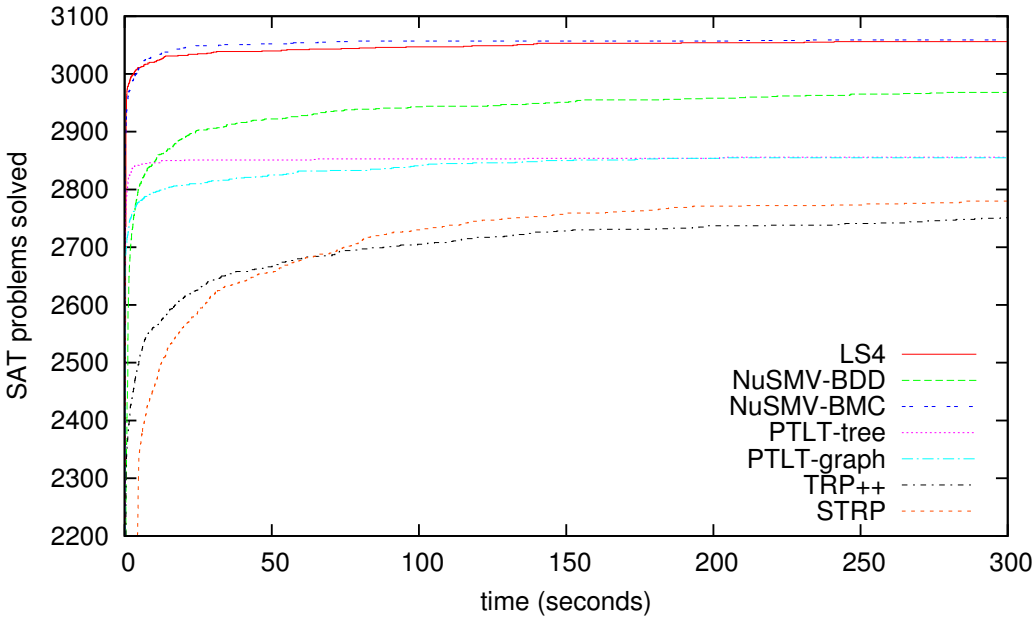


Figure 3.4: Comparison of LS4 and the other LTL solvers – II. Showing the number of *satisfiable* problems solved within a given time limit.

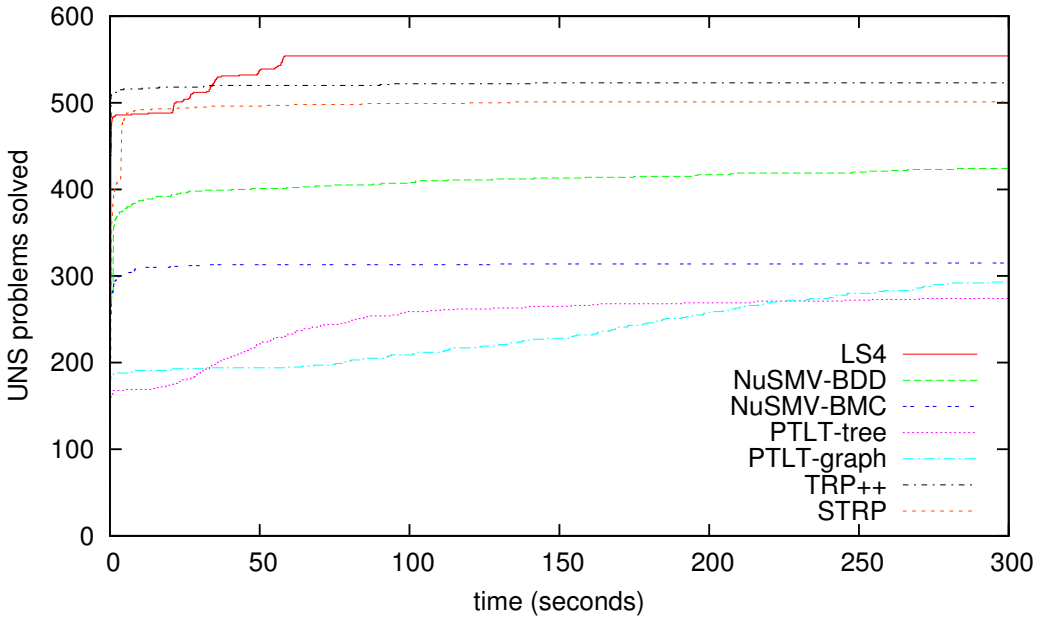


Figure 3.5: Comparison of LS4 and the other LTL solvers – III. Showing the number of *unsatisfiable* problems solved within a given time limit.

The performance plot of Figure 3.3 is decomposed between the satisfiable and unsatisfiable problems in Figure 3.4 and Figure 3.5, respectively. Most satisfiable problems, namely 3059, are solved by `NuSMV-BMC`, which is closely followed by `LS4` with 3056 solved problems. In this satisfiable category, the tableau solvers `PLTL-graph` and `PLTL-tree` perform better than the resolution-based provers `STRP` and `TRP++`. This trend is reversed, however, in the unsatisfiable category, where `TRP++` solves 523 and `STRP` 501 problems, which places the provers on the second and third place, just behind `LS4` with 554 solved unsatisfiable problems.

The benchmark set that we used can be split into several problem families of various sources and characteristics (see Schuppan and Darmawan, 2011). In Table 3.3, we show the number of problems solved within 300 seconds grouped by the individual problem families. We see that no solver dominates across all the families and some solvers are complementary in their strengths and weaknesses. Nevertheless, if we count the number of families on which a particular solver achieves the best performance, we can identify `LS4` and `NuSMV-BDD` as the most successful solvers, each being able to perform best on 12 families in total. The second best is `TRP++`, excelling on 11 problem families.

Overall, we believe that our results provide sufficient evidence that LS4 is a strong algorithm and its implementation, `LS4`, a successful tool for checking LTL satisfiability.

Table 3.3: Comparing LS4 and the other LTL solvers. Showing the number of problems solved with 300 seconds grouped by formula families. The best results for each family are typeset in bold.

family	size	LS4	NuSMV-BDD	NuSMV-BMC	PTLT-tree	PTLT-graph	TRP++	STRP
acacia_demo-v22	10	**10**	**10**	**10**	**10**	**10**	**10**	**10**
acacia_demo-v3	36	**36**	**36**	**36**	29	31	**36**	33
acacia_example	25	**25**	**25**	**25**	**25**	**25**	**25**	**25**
alaska_lift	136	117	**136**	112	40	44	32	28
alaska_szymanski	4	**4**	**4**	**4**	**4**	**4**	**4**	**4**
anzu_amba	51	**47**	9	45	35	35	12	3
anzu_genbuf	60	46	33	**49**	26	28	17	3
forobots	39	**39**	**39**	21	1	16	**39**	**39**
rozier_counter	76	34	47	14	34	62	**76**	34
rozier_formulas	2000	**2000**	**2000**	1991	1993	1997	1787	1866
rozier_pattern	244	**244**	182	**244**	**244**	**244**	227	225
schuppan_O1formula	27	**27**	22	**27**	10	10	**27**	**27**
schuppan_O2formula	27	9	**14**	2	9	9	6	9
schuppan_phltl	18	5	**9**	4	4	3	5	5
trp_N5x	240	**240**	**240**	231	226	**240**	**240**	**240**
trp_N5y	140	**140**	**140**	94	94	94	**140**	**140**
trp_N12x	400	397	256	342	223	173	**400**	**400**
trp_N12y	190	**190**	**190**	123	123	123	**190**	**190**
total	3723	3610	3392	3374	3130	3148	3273	3281

Table 3.4: Correspondence between marked (LS4) and labeled (LPSup) clauses.

LS4	I	U	T	L_k^b	D_k^b
LPSup	$(0,*)$	$(*,*)$	$(*,*)$	$(*,k)$	—

3.5 Discussion and related work

3.5.1 Semantic graphs and the relation to LPSup

In Section 2.4 of the previous chapter, we explained that a TST of a given LTL formula φ can be seen as a symbolic representation of the Büchi automaton for recognizing models of φ. We defined the semantic graph, a graph theoretic equivalent of the automaton, and argued that the saturation-based decision procedure of LPSup essentially computes the preimage operation on the sets of graph vertexes represented by the involved layer clauses. Let us now try to understand the workings of LS4 from this perspective.

The approach of LS4 can be characterized as a hybrid between explicit and symbolic exploration of the semantic graph. The partial model built by LS4 corresponds to an explicit path through the graph. Like an explicit search algorithm, LS4 constructs the path starting from a concrete initial vertex and always extending it vertex by vertex in the direction of the goal vertexes. However, the information about unsuccessful extensions or, in other words, about the already explored but unpromising parts of the graph, is given a symbolic representation within the learned clauses.

Let us return to the already mentioned correspondence between the marked clauses learned by LS4 and the labeled clauses of LPSup. As shown in Table 3.4, the correspondence is almost one to one. The only marked clauses that do not have a labeled clause equivalent are the clauses of the dirty layers D_k^b, which depend on two or more different copies of the goal clauses. Each new clause learned by LS4 belongs to one of the sets U, L_k^b or D_k^b and is always simple, i.e. just over the basic signature Σ. This is a consequence of the way the mechanism of solving under assumptions is employed in passing the last valuation of the partial model into the extension query. Non-simple marked clauses are only learned internally by the CDCD algorithm inside the individual SAT solvers. They are, however, not explicitly registered by LS4 itself.

The correspondence between the marked clauses of the proper layers L_k^b and $(*,k)$-clauses of LPSup allows us to justify correctness of LS4 in the unsatisfiable case using essentially the same derivation replaying argument as in LPSup (see Section 3.3.3). An interesting difference is that while the layer-by-layer saturation of LPSup is exhaustive, designed to derive all the possible (non-redundant ordered) resolvents, LS4 selectively learns only those layer clauses which show that a particular reachable vertex cannot be extended towards the goal. From the semantic perspective, we can say that the preimage computation performed by the layer-by-layer saturation of LPSup is precise, whereas the corresponding operation on the side of LS4 only over-approximates the true preimage. LS4 only learns enough information to repair the last wrong choice, but does not spend time discovering properties of the preimage that may never be needed.

3.5.2 Two other solvers relying on SAT

In Section 3.4.2 we experimentally evaluated LS4 and six other LTL satisfiability solvers. Similarly to LS4, two other tools, `STRP` and `NuSMV-BMC`, internally rely on the SAT-solving technology, although each of them in a different way. It is interesting to compare and contrast these approaches from a theoretical perspective.

STRP

STRP is a saturation prover based on the simplified temporal resolution calculus (Degtyarev et al., 2002). This calculus essentially reformulates Clausal Temporal Resolution (CTR) (Fisher et al., 2001) on an abstract level with multi-premise "macro" inference rules and purely classical side conditions. The main idea behind STRP is to reduce the search for premises in the calculus to the minimal unsatisfiable subsets (MUS) extraction problem (Marques-Silva, 2012). Solving this problem is then delegated to an external tool, which eventually relies on a SAT solver.

If we abstract away the differences between LPSup and CTR (see Section 2.5.2), which indirectly carry over to LS4 and STRP, respectively, it can be said that the two latter systems have the opportunity to derive the same clauses. As mentioned, new clauses learned by LS4 are always simple and the same holds for the conclusions of the macro-inferences of the simplified temporal resolution in STRP. The saturation paradigm, however, forces STRP to derive all the possible conclusions that follow from the current clause set, while LS4 only learns a new clause to record the reason of an unsuccessful extension.

This is again the difference between the exhaustive and selective approach to clause generation. It gives LS4 a huge advantage in the satisfiable case, where STRP needs to finish generating all the implied clauses, while LS4 only derives enough of them to be guided to a model. However, even in the unsatisfiable case, there are typically many implied clauses which do not contribute to the final contradiction. LS4 may be able to ignore these altogether, whereas STRP can only avoid generating them, if the ultimate empty clause is derived first.

Bounded Model Checking (BMC)

Compared to STRP, the use of a SAT solver in the Bounded Model Checking (Biere et al., 1999) algorithm, like the one implemented inside `NuSMV-BMC`, is much more direct. Given an LTL formula φ, BMC encodes the existence of a model for φ of size k into a propositional formula F_φ^k, which is then passed to the SAT solver. If the formula F_φ^k is satisfiable, the algorithm terminates successfully and a model of φ is recovered from the satisfying assignment. In the opposite case, BMC increases the bound k on the model size and starts again from the beginning.

The encoding of BMC relies basically on the same principles as our normal form transformation. We can identify building blocks similar to the formulas I, T and G of a TST and the main part of the encoding F_φ^k expresses that the model of size k must

start in a world satisfying I and respect the transition relation represented by T:

$$I \wedge T^{(0)} \wedge T^{(1)} \wedge \ldots \wedge T^{(k-2)}.$$

Additionally, the chain of worlds is forced to form a *loop*, meaning that the last world W_{k-1} is equal to one of the preceding worlds W_l. Finally, the goal condition G is required to hold in one of the worlds of the repeating part of the loop, between W_l and W_{k-1}.

We can see that from the point of view of finding models, LS4 is very similar to BMC. The main difference is that in LS4 the formula expressing model existence is constructed in a step-by-step fashion and the model is built one world at a time. Moreover, LS4 does not decide in advance the size of the model nor the exact shape of the loop. Indeed, it relies on the model repetition check without imposing any explicit condition for the repetition to occur. As our experimental results indicate this passive approach is surprisingly successful in practice. Although BMC solved the most satisfiable problems of the tested solvers, LS4 ended very close on the second place (see Figure 3.4, page 96).[6]

Bounded model checking *per se* does not offer an efficient way for recognizing unsatisfiable formulas. The implementation `NuSMV-BMC` (Heljanko et al., 2005) performs a separate *completeness check* after each iteration. The idea is to combine those parts of F_φ^k which actually do not depend on k with a *simple path* condition that forces the involved worlds to be distinct. If such a formula is unsatisfiable, there can be no models larger than the already considered size k and the algorithm terminates with UNSAT.[7] There is not much similarity between the completeness check of BMC and the way unsatisfiable instances are handled by LS4. It is easy to construct LTL formulas for which the completeness check is satisfiable for large values of k, but which will be easily recognized as unsatisfiable by LS4. This could explain why `LS4` performed so much better than `NuSMV-BMC` on unsatisfiable problems in our experiments (cf. Figure 3.5, page 97).

3.5.3 Recent advances in hardware model checking

In 2010, Aaron Bradley proposed a new SAT-based algorithm for checking safety properties of finite state systems (Bradley, 2011). The algorithm, called Property Directed Reachability (PDR), also known as IC3, has been shown to perform remarkably well in practice and was given the title of the "most important contribution to bit-level formal verification in almost a decade" (Eén et al., 2011). Recently, two new algorithms for checking *liveness*, an equivalent of our TST (un)satisfiability, have been developed, each building on the success of PDR and using it as a subroutine.

In Chapter 5, we will specialize LS4 to (single time) reachability[8] and will obtain an algorithm very similar to PDR. It is, therefore, appropriate to briefly compare LS4 to those two liveness checking algorithms. The reader may want to return to this section after the relation of LS4 and PDR has been presented in full detail in Section 5.3.

[6] It should be noted that, unlike LS4, `NuSMV-BMC` guarantees to find models of minimal size.

[7] This is an LTL analogue of the k-induction method, originally proposed by Sheeran et al. (2000).

[8] Instead of looking for an infinite path with infinitely many goal worlds, reachability is concerned about the existence of just a finite path from an initial world to a goal world. Specializing LS4 to reachability essentially corresponds to limiting its computation to the first block.

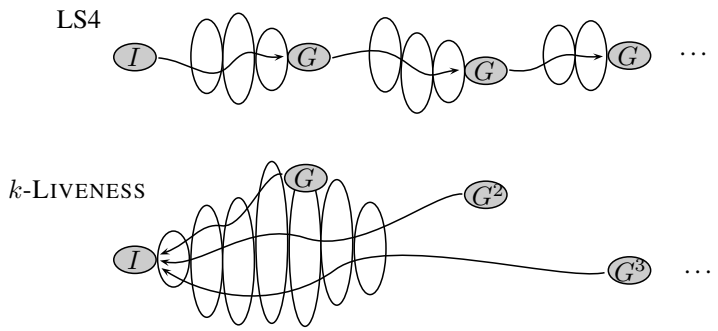


Figure 3.6: Comparing state space exploration of LS4 and k-LIVENESS. Constructed paths grow along the arrows, guiding layers (denoted by ovals) in the opposite direction.

Fair

The less related of the two algorithms is FAIR by Bradley et al. (2011). It uses a SAT solver to iteratively pick a selection of worlds, called a *skeleton*, and attempts to connect these worlds by paths using reachability queries delegated to PDR. A successfully connected skeleton represents a witness for satisfiability of the given problem: an initial world connected to a loop with a goal world on it. A failure to connect two worlds of a skeleton leads to a discovery a new *wall* in the state space, such that new skeletons must lie entirely on one side of the wall. Walls are extracted from clausal reachability information maintained by PDR, an almost exact equivalent of LS4's layers.

The algorithm that counts

The idea behind the k-LIVENESS algorithm by Claessen and Sörensson (2012) is to reduce liveness checking to safety checking (i.e., reachability). This has already been proposed previously in a form of a one-time encoding (see Biere et al., 2002), but in k-LIVENESS, the reduction is incremental.

To show that the given problem is unsatisfiable, the algorithm counts and bounds the number of times the goal condition G can be satisfied along an infinite path that starts in an initial world. If the goal condition cannot be reached even once, the given problem is obviously unsatisfiable and k-LIVENESS terminates. Otherwise it constructs a strengthened condition G^2, which expresses that the goal G should be satisfied twice in a row, and runs the safety check again. In general, if the given problem is unsatisfiable, the goal cannot be satisfied more than k times for some $k \in \mathbb{N}$ and the algorithm will terminate after failing to reach a world that satisfies G^{k+1}. In the setting of verifying hardware circuits considered by Claessen and Sörensson (2012), the transformation from condition G^i to G^{i+1} can be realized by adding a simple one bit memory element to the circuit.

The algorithm relies on PDR for answering the reachability queries. While in LS4, we construct the model path from an initial world towards the goal, in PDR the default direction is reversed. This means that the equivalent of LS4's layers in PDR encode

over-approximation of the image of the set of initial worlds, as opposed to the preimage of the goal worlds as in LS4 (see Figure 3.6). This has an important consequence for k-Liveness. Because these "layers" do not depend on the goal conditions G^i, they can be shared and reused for guidance by the individual reachability queries. This was shown to be a key to a good performance of the algorithm.

Performance comparison estimate

In order to obtain a rough comparison of the relative performance of LS4, Fair, and k-Liveness, we have extended LS4 to parse circuits in the AIGER format (Biere, 2012) of the Hardware Model Checking Competition (HWMCC) and to translate the corresponding liveness problems to equisatisfiable TST's. Using the same time limit of 900 seconds per problem as in the competition, we then ran the extended LS4 on the 118 problems of HWMCC 2012 (Biere et al., 2012). The results we obtained can be summarized as follows.

If LS4 participated in the competition, it could have[9] placed third with 66 problems solved (44 satisfiable and 19 unsatisfiable) just after the system `iimc2011`, which implements Fair and solved 70 problems (27 satisfiable and 43 unsatisfiable), and the winner `TIP`, which implements k-Liveness and solved 92 problems (46 satisfiable and 46 unsatisfiable). There are, however, several reasons why this is only a fair comparison of the implementations rather than of the algorithms themselves.

- k-Liveness is not suitable for recognizing satisfiable instances, therefore, it was complemented in `TIP` by a Bounded Model Checker running in lock-step.

- Both `iimc2011` and `TIP` employ circuit specific preprocessing of the input, while our extension of LS4 relies only on a straightforward encoding followed by variable and clause elimination (see also Chapter 4).

- Unlike LS4, both Fair in `iimc2011` and k-Liveness in `TIP` incorporate special heuristic techniques for efficiently dealing with problems involving counters.[10] About 10 of the unsatisfiable problems were solved thanks to this technique by each of the systems (see Claessen and Sörensson, 2012, Fig. 9).

For a better estimate of LS4's potential as a liveness model checking algorithm, a full-fledged implementation including the mentioned techniques would be needed. This is left as a future work.

[9] Actually, our hardware configuration is stronger than the one used in the competition (3.16 GHz CPU and 16 GB RAM versus 2.6 GHz CPU and 8 GB RAM), but because all the problems solved by LS4 were solved before the 300 second mark, this should not make a difference.

[10] Look for "skeleton-independent proofs" (in Bradley et al., 2011, Section III-E) and for "stabilizing constraints" (in Claessen and Sörensson, 2012, Section IV).

3.6 Conclusion

We have presented LS4, a new algorithm for LTL satisfiability. Building on the calculus LPSup and, in particular, on its ability to construct (partial) models on the fly, LS4 departs from the saturation paradigm and instead employes a modern SAT solver to efficiently drive the search and select inferences. This gives rise to a hybrid between explicit and symbolic exploration of the space of potential models.

LS4 was shown to perform remarkably well in practice, staying on par with or even improving over the state-of-the-art LTL satisfiability checkers. In particular, the model guidance approach clearly outperforms saturation. This is more evident on the satisfiable instances, where an explicit model is typically discovered long before a full saturation of the clause set can confirm satisfiability indirectly. But a performance gain can also be observed on the unsatisfiable problems. Its likely explanation is the lazy nature in which LS4 derives clauses from failed attempts of model extension thus only focusing on the relevant reasons for unsatisfiability.

4 Variable and clause elimination for LTL

4.1 Introduction

When developing practically useful tools for satisfiability checking of formulas in a particular logic, one is on a constant lookout for techniques that would improve performance and help to fight the typically high inherent computational complexity of the decision problem. One possibility for speeding up such a tool lies in simplifying the input formula before the actual decision method is started.

In the context of resolution-based methods for LTL, where the given formula is first translated into a clausal normal form, simplification means reducing the number of clauses and variables while preserving satisfiability of the formula. Such a preprocessing step may have a significant positive impact on the subsequent running time.

In this chapter we take inspiration from the SAT community where a technique called variable and clause elimination (Eén and Biere, 2005) has been shown to be particularly effective. It combines exhaustive application of the resolution rule over selected variables with subsumption and other reductions. Our main contribution lies in showing that variable and clause elimination can be adapted from SAT to the setting of LTL.

Preprocessing and normal forms

It was observed by Eén and Biere (2005) that work on preprocessing techniques can be seen as a viable alternative to optimizing normal form transformation procedures. Let us recall the SNF transformation presented in Figure 2.2 (Section 2.2.2) and consider the LTL formula

$$\neg p \vee \Box p. \tag{4.1}$$

In order to produce an equisatisfiable SNF, our transformation will introduce several new variables and eventually end up with the following set of temporal clauses

$$\overline{i},\ \Box(\neg\overline{i} \vee \overline{u} \vee \overline{v}),\ \Box(\neg\overline{u} \vee \neg p),\ \Box(\neg\overline{v} \vee \overline{w}),\ \Box(\neg\overline{w} \vee \bigcirc\overline{w}),\ \Box(\neg\overline{w} \vee p).$$

With variable and clause elimination for LTL, as described in this chapter, we will be able to reduce this set to

$$\neg p \vee \overline{w},\ \Box(\neg\overline{w} \vee \bigcirc\overline{w}),\ \Box(\neg\overline{w} \vee p), \tag{4.2}$$

and, if we also eliminate the original variable p, further to just the single clause

$$\Box(\neg\overline{w} \vee \bigcirc\overline{w}). \tag{4.3}$$

By enlarging the set of rules of Figure 2.2 and introducing additional side conditions, it would be possible to optimize the transformation such that it directly produces (4.2) as the SNF of (4.1). We believe that investing the effort into a general purpose preprocessing is more worthwhile. For one thing, it is applicable even when we are not in control of the normal form transformation and obtain the input already in SNF. Moreover, it typically allows us to reduce the input further, as shown by our example and its final form (4.3).

Strategy and chapter overview

We start our exposition by reviewing propositional variable and clause elimination in Section 4.2.1. To lift the technique from SAT to LTL, we reuse the idea of labels and labeled clauses (recall Section 2.3.1). Because the labels we introduced in Chapter 2 are not sufficiently expressive to justify the lift of variable elimination, we extend them by a third component and correspondingly update the semantics and operations on labels (Section 4.2.2). The intuitive explanation is that we need a correspondence between labeled resolution and the represented propositional resolution that is not only sound, as was sufficient in the case of LPSup, but also complete, in the sense that every propositional resolution is lifted by some labeled resolution.

We develop the actual variable and clause elimination for labeled clauses in Section 4.2.3. Interestingly, we will be able show that when the elimination is completed, clauses with non-trivial third label component can be removed from the resulting set without affecting satisfiability. This means that the third label component, although an important part of the theoretical argument, does not need to be explicitly realized in an actual implementation.

As a proof of concept, we implemented (a restricted version of) variable and clause elimination for LTL, building on top of the simplification capabilities of the SAT solver Minisat (Section 4.3.1). We then experimentally evaluated the effect of the preprocessing on the performance of resolution-based LTL provers LS4 and TRP++ (Section 4.3.2). Our results confirm that even in the temporal setting substantial reductions in formula size and subsequent decrease of running time can be achieved.

The results of this chapter have been published in (Suda, 2013d,a).

4.2 Theory

4.2.1 Variable and clause elimination in SAT

By variable and clause elimination we understand the preprocessing technique described by Eén and Biere (2005) for simplifying propositional SAT problems. It consists of a combination of a controlled version of variable elimination and subsumption-based reductions for strengthening and removing clauses, as described below. These two are alternated in a saturation loop until no further immediate improvement is possible.

Propositional variable elimination

Propositional variable elimination was originally proposed by Davis and Putnam (1960) under the name "Rule for Eliminating Atomic Formulas". It relies on performing the standard (unordered) resolution inference for every possible pair of clauses that mention a given variable and replacing these clauses by the obtained resolvents.

Given two (standard) clauses $C = p \vee C_0$ and $D = \neg p \vee D_0$, their *propositional resolvent* over the variable p, denoted $C \otimes_p D$, is defined as $C_0 \vee D_0$. Now, given a propositional problem in CNF consisting of a set of clauses N and a variable p, one separates N into three disjoint subsets $N = N_p \cup N_{\neg p} \cup N_0$ of clauses. The first set, N_p, is a set of clauses containing the variable p positively, the clauses from $N_{\neg p}$ contain p negatively, and N_0 is a set of clauses without variable p. A new clause set $\overline{N}$ is obtained as $(N_p \otimes_p N_{\neg p}) \cup N_0$, where

$$N_p \otimes_p N_{\neg p} = \{C \otimes_p D \mid C \in N_p, D \in N_{\neg p}\}.$$

The variable p does not occur in the set $\overline{N}$ and the set is satisfiable if and only if N is.

The obtained set $\overline{N}$ may contain tautological clauses, which are redundant and should be removed. Then the sizes of N and $\overline{N}$ are compared. In general, eliminating a single variable may incur a quadratic blowup of the number of clauses in the set. An elimination step is only considered an *improvement* and should be committed to when the size of $\overline{N}$ is not greater than that of N (possibly up to an additive[1] constant).

Subsumption-based reductions

By subsumption we here mean the standard reduction which allows us to remove a clause D from the clause set N in the presence of another clause $C \in N$ such that $C \subseteq D$. Eén and Biere (2005) also propose to employ *self-subsuming resolution*, a resolution inference in which the conclusion subsumes one of the premises. In our notation we write self-subsuming resolution as

$$\mathcal{R}\frac{C \vee p \quad D \vee \neg p}{C \vee p \qquad D},$$

under the condition that $C \subseteq D$. Effectively, the clause $D \vee \neg p$ is getting strengthened to D in the presence of the clause $C \vee p$.

It is advantageous to alternate variable elimination attempts with application of subsumption and self-subsuming resolution. That is because removing a subsumed clause may turn elimination of a particular variable into an improvement and, on the other hand, new clauses generated during elimination may be subject to subsumption. For a detailed description on how to organize and efficiently implement this process we refer the reader to Eén and Biere (2005).

4.2.2 Adapting the mechanism of labeled clauses

In this section we adapt the mechanism of labeled clauses introduced in Chapter 2 to prepare for transferring variable and clause elimination to LTL. We aim at reusing the

[1] To ensure that the size of the clause set does not grow more than linearly in the number of eliminations.

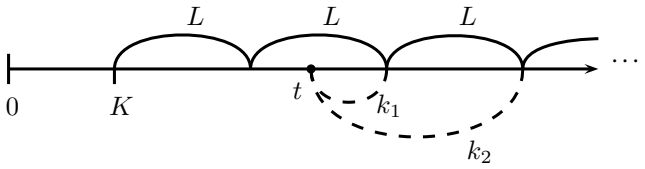


Figure 4.1: Imagine two labeled clauses $\mathcal{C}_1 = (*, k_1) \,||\, p \vee C_1$ and $\mathcal{C}_2 = (*, k_2) \,||\, \neg p \vee C_2$. Given a rank (K, L) such that L divides $k_2 - k_1$, the clauses $\mathcal{C}_1$ and $\mathcal{C}_2$ represent standard clauses $(p \vee C_1)^{(t)}$ and $(\neg p \vee C_2)^{(t)}$, resp., which can be resolved on the variable $p^{(t)}$.

idea of lifting and employing labeled clauses to finitely represent the elimination process happening on the "ground level" of the signature Σ^*.

Unfortunately, labeled clauses used by LPSup are not sufficiently expressive to allow for lifting of *every* possible resolution inference. This may be a little surprising given that LPSup is a complete calculus. The phenomenon is illustrated in Figure 4.1. It concerns pairs of labeled clauses with distinct second label components $k_1, k_2 \in \mathbb{N}$, for which the merge operation is not defined. We fix the situation by extending the label by one additional component and capture the neglected overlap by defining the merge operation on labels for all possible inputs.

We also change the role of temporal shift: from a standalone inference to a prefix operation which precedes resolution. We do this to avoid generating the intermediate clauses, which were harmless from the perspective of the calculus LPSup, but are undesirable during variable and clause elimination process, whose sole purpose is to reduce the size of the given clause set.

Extended labeled clauses

In order to record all the necessary information about a clause we extend the Definition 2.7 of a label by one new label component (the domains of the first and second label component are preserved) and allow the standard part C of the labeled clause to contain literals with arbitrary many primes.

Definition 4.1. An *extended label* is a triple $(b, k, l) \in \{*, 0\} \times (\{*\} \cup \mathbb{N}) \times \mathbb{N}$. An *extended labeled clause* is pair $\mathcal{C} = (b, k, l) \,||\, C$, where (b, k, l) is an extended label and C is a standard clause over $\Sigma^* = \bigcup_{i \in \mathbb{N}} \Sigma^{(i)}$. We will omit the qualifier "extended" whenever confusion cannot arise.

The additional label component imposes an additional independent condition on the semantics of the label. If the condition is satisfied, the label's semantics reduces to the original Definition 2.8. In the opposite case, the label does not represent any index.

Definition 4.2. Given a rank (K, L), the set $R_{(K,L)}(b, k, l)$ of indexes represented by the extended label (b, k, l) is defined as

$$R_{(K,L)}(b, k, l) = \begin{cases} R_{(K,L)}(b, k) & \text{if } L \text{ divides } l, \\ \emptyset & \text{otherwise,} \end{cases}$$

where $R_{(K,L)}(b,k)$ is the set of indexes represented by the label (b,k). A standard clause $C^{(t)}$ is represented by the labeled clause $(b,k,l) \,||\, C$ in (K,L) if $t \in R_{(K,L)}(b,k,l)$.

Because the natural number zero is divisible by any natural number, when the third label component l equals zero, the corresponding divisibility condition from the above definition is trivially satisfied independently of the value of the rank (K,L). Thus when extending the notion of the starting labeled clause set (Definition 2.9) we uniformly supply the additional label component with value $l = 0$ to preserve the original semantics.

Definition 4.3. The extended starting labeled clause set $N_{\mathcal{T}}$ for a TST $\mathcal{T} = (\Sigma, I, T, G)$ consists of the extended labeled clauses of the form

- $(0,*,0) \,||\, C$ for every $C \in I$,
- $(*,*,0) \,||\, C$ for every $C \in T$, and
- $(*,0,0) \,||\, C$ for every $C \in G$.

The definition of (K,L)-satisfiability and satisfiability of sets of labeled clauses (Definition 2.10) as well as Lemma 2.3 on the satisfiability of labeled clause sets carry over to extended labeled clauses in a straightforward way.

New label merge

The merge operation on extended labels takes care of the previously undefined case (recall Definition 2.12) and the corresponding divisibility condition under which the overlap of indexes occurs (see Figure 4.1) is recorded via the third label component.

Definition 4.4. The *merge* of labels (b_1,k_1,l_1) and (b_2,k_2,l_2) is a label (b,k,l) defined imperatively as follows:

- if $b_1 = *$ then $b \leftarrow b_2$ else if $b_2 = *$ then $b \leftarrow b_1$ else $b \leftarrow 0$,
- if $k_1 = *$ then $k \leftarrow k_2$ else if $k_2 = *$ then $k \leftarrow k_1$ else $k \leftarrow \min(k_1,k_2)$,
- if $k_1 = *$ or $k_2 = *$ then $l \leftarrow \gcd(l_1,l_2)$ else $l \leftarrow \gcd(l_1,l_2,|k_1-k_2|)$,

where gcd stands for the greatest common divisor operation and $\gcd(0,0) = 0$.

Example 4.1. Merge of $(*,2,0)$ and $(*,5,0)$ is $(*,2,3)$: we compute the minimum of the second components k and the greatest common divisor of their difference and of the original values of the third components l. Merge of $(*,2,3)$ and $(*,2,3)$ is $(*,2,3)$; merge is, in fact, an idempotent operation. Merge of $(*,2,3)$ and $(*,*,0)$ is $(*,2,3)$; merge has, in fact, a neutral element $(*,*,0)$. Merge of $(*,2,3)$ and $(0,1,4)$ is $(0,1,1)$.

Lemma 2.4 asserts that the merge operation on (non-extended) labels corresponds to the intersection operation on the represented sets of indexes. The definition of the merge operation on extended labels is designed in a such way that an analogue of this lemma still holds. Extension of the proof needs to take into account the added label component and relies, in particular, on the equivalence

$$L \text{ divides } l_1 \text{ and } L \text{ divides } l_2 \text{ if and only if } L \text{ divides } \gcd(l_1,l_2).$$

Lemma 4.1. *Let (b, k, l) be the merge of labels (b_1, k_1, l_1) and (b_2, k_2, l_2). Then for any rank (K, L)*

$$R_{(K,L)}(b, k, l) = R_{(K,L)}(b_1, k_1, l_1) \cap R_{(K,L)}(b_2, k_2, l_2).$$

Temporal shift and labeled resolution

The intuitive role of temporal shift is to align one labeled clause with another such that the variable over which we subsequently plan to resolve the two clauses occurs in both of them under the same number of primes. In Chapter 2 we introduced temporal shift as a standalone inference. Here we instead package it together with resolution to avoid generating the intermediate clauses.

We define the temporal shift operation on extended labeled clauses by the following two equations

$$TS(\ (*, *, l) \,||\, C\ ) = (*, *, l) \,||\, (C)', \tag{4.4}$$
$$TS(\ (*, k, l) \,||\, C\ ) = (*, k + 1, l) \,||\, (C)'. \tag{4.5}$$

Note that the operation is *undefined* for labeled clauses with the first component $b = 0$, because these only represent standard clauses fixed to the first time index.

Soundness of temporal shift operation is the statement that all the standard clauses represented by the right hand side of (4.4) and (4.5) are also represented by the respective left hand sides in any (K, L).[2] The soundness is shown analogously to the corresponding property of the LPSup inference (Lemma 2.6).

We now move to defining the (unordered) labeled resolution operation. Because we allow arbitrarily many primes in extended labeled clauses, the operation in general involves iterated application of temporal shift.

Definition 4.5. Let $\mathcal{C}_1 = (b_1, k_1, l_1) \,||\, p^{(i)} \vee C_1$ and $\mathcal{C}_2 = (b_2, k_2, l_2) \,||\, \neg p^{(j)} \vee C_2$ be two extended labeled clauses. Their labeled resolvent over the variable p, denoted $\mathcal{C}_1 \otimes_p \mathcal{C}_2$, is computed as follows:

1. If $i = j$ then $\mathcal{C}_1 \otimes_p \mathcal{C}_2$ equals the labeled clause $(b, k, l) \,||\, C_1 \vee C_2$ where (b, k, l) is the merge of the labels (b_1, k_1, l_1) and (b_2, k_2, l_2).

2. If $i < j$ and $b_1 = *$ then $\mathcal{C}_1 \otimes_p \mathcal{C}_2$ equals $TS^{j-i}(\mathcal{C}_1) \otimes_p \mathcal{C}_2$, where TS^n stands for the n-fold application of the temporal shift operation. This reduces the computation to the previous case.

3. If $i > j$ and $b_2 = *$ then $\mathcal{C}_1 \otimes_p \mathcal{C}_2$ equals $\mathcal{C}_1 \otimes_p TS^{i-j}(\mathcal{C}_2)$; analogously to case 2.

4. If either $i < j$ and $b_1 = 0$ or $i > j$ and $b_2 = 0$ then the resolvent is undefined.

Strictly speaking, the resolvent $\mathcal{C}_1 \otimes_p \mathcal{C}_2$ does not depend just on the variable p, but more specifically on the occurrences of the literals $p^{(i)}$ and $\neg p^{(j)}$ in the respective clauses.

[2] The converse does not hold. For instance, the labeled clause $(*, *, 0) \,||\, p' = TS(\ (*, *, 0) \,||\, p\ )$ does not represent the standard clause $p^{(0)}$ in any (K, L), although the original clause $(*, *, 0) \,||\, p$ does.

We will, however, only use Definition 4.5 in situations where there is just one occurrence of a literal mentioning the variable p (possibly primed) in each of the two clauses, and therefore, confusion will not arise.

Example 4.2. Let two labeled clauses $(*, 0, 0) \,||\, \neg p \vee q$ and $(*, 0, 0) \,||\, r \vee p'$ be given. They cannot directly (as in case 1 above) generate a labeled resolvent, although in $(K, L) = (0, 1)$ there are (for every t) standard clauses $\neg p^{(t+1)} \vee q^{(t+1)}$ and $r^{(t)} \vee p^{(t+1)}$ represented, respectively, by the two labeled clauses, which resolve on $p^{(t+1)}$. The first labeled clause first needs to be shifted to $(*, 1, 0) \,||\, \neg p' \vee q'$ (case 2), and the clauses then resolve on p' and a labeled resolvent $(*, 0, 1) \,||\, r \vee q'$ is obtained.

In analogy to soundness of the Ordered Resolution Inference of LPSup (Lemma 2.5) we have a lemma stating soundness of the labeled resolution operation.

Lemma 4.2. *Let (K, L) be a rank. Any standard clause represented in (K, L) by the labeled resolvent $\mathcal{C}_1 \otimes_p \mathcal{C}_2$ of clauses $\mathcal{C}_1$ and $\mathcal{C}_2$ is a propositional resolvent over some "instance" $p^{(i)}$ of the variable p of clauses represented in (K, L) by $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively.*

Proof. Follows from soundness of temporal shift and from Lemma 4.1. □

More interestingly, labeled resolution operation is also designed to be complete, i.e. to lift every possible propositional resolution from the "ground level".

Lemma 4.3. *Let $\mathcal{C}_1 = (b_1, k_1, l_1) \,||\, p^{(i)} \vee C_1$ and $\mathcal{C}_2 = (b_2, k_2, l_2) \,||\, \neg p^{(j)} \vee C_2$ be two labeled clauses and let (K, L) be a rank. If there are numbers $n, t_1, t_2 \in \mathbb{N}$ such that $n = i + t_1 = j + t_2$ and the clauses $p^{(n)} \vee (C_1)^{(t_1)}$ and $\neg p^{(n)} \vee (C_2)^{(t_2)}$ are represented in (K, L) by $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively, then the labeled resolvent $\mathcal{C} = \mathcal{C}_1 \otimes_p \mathcal{C}_2$ is defined and the propositional resolvent $(C_1)^{(t_1)} \vee (C_2)^{(t_2)}$ over $p^{(n)}$ is represented in (K, L) by $\mathcal{C}$.*

Proof. The labeled resolvent is only undefined when the corresponding temporal shift operation would be undefined. Let us (without loss of generality) focus on one of the symmetrical cases when this happens and assume that $i < j$ and $b_1 = 0$. Because $b_1 = 0$ we must have $t_1 = 0$, but then $i + t_1 = i < j \leq j + t_2$ for any $t_2 \in \mathbb{N}$, and so there can be no $n = i + t_1 = j + t_2$. In other words, whenever there are numbers $n, t_1, t_2 \in \mathbb{N}$ such that $n = i + t_1 = j + t_2$ and the clauses $p^{(n)} \vee (C_1)^{(t_1)}$ and $\neg p^{(n)} \vee (C_2)^{(t_2)}$ are represented in (K, L) by $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively, then the labeled resolvent $\mathcal{C} = \mathcal{C}_1 \otimes_p \mathcal{C}_2$ is defined.

To prove the second part of the lemma, we focus on case 2 of Definition 4.5. The remaining cases are simpler (case 1) or analogous (case 3). We, therefore, assume that $i < j$ and $b_1 = *$. The clause $TS^{j-i}(\mathcal{C}_1)$ must be of the form $(b_1, k_1', l_1) \,||\, p^{(j)} \vee (C_1)^{(j-i)}$, where either $k_1' = k_1 = *$ or $k_1' = k_1 + (j - i)$. Either way, by Definition 4.2 (and, indirectly, by Definition 2.8)

$$t_1 \in R_{(K,L)}(b_1, k_1, l_1) \text{ implies } t_2 = t_1 - (j - i) \in R_{(K,L)}(b_1, k_1', l_1).$$

Thus, by Lemma 4.1 $t_2 \in R_{(K,L)}(b, k, l)$, where (b, k, l) is the merge of labels (b_1, k_1', l_1) and (b_2, k_2, l_2), i.e. the label of the labeled resolvent $\mathcal{C}_1 \otimes_p \mathcal{C}_2 = \mathcal{C}$. Thus $\mathcal{C}$, which is of the form $(b, k, l) \,||\, (C_1)^{(j-i)} \vee C_2$, in (K, L) represents

$$((C_1)^{(j-i)} \vee C_2)^{(t_2)} = (C_1)^{(j-i)+t_2} \vee (C_2)^{(t_2)} = (C_1)^{(t_1)} \vee (C_2)^{(t_2)}.$$

Table 4.1: Example labeled clauses translated to first-order logic

	labeled clause	first-order clause
(1)	$(0, *, 0) \,\|\|\, \neg p \vee q$	$\neg p(0) \vee q(0)$
(2)	$(*, *, 0) \,\|\|\, p' \vee r'$	$\forall X.\ p(s(X)) \vee r(s(X))$
(3)	$(*, *, 0) \,\|\|\, \neg r \vee q$	$\forall Y.\ \neg r(Y) \vee q(Y)$

□

Remark 4.1. The way we package the temporal shift operation with resolution is reminiscent to the use of most general unifiers in first-order theorem proving. Indeed, we can understand temporal shift as a form substitution on time indexes.

There is a straightforward encoding of LTL formulas and, in particular, of TSTs into first-order logic. The encoding models time by first-order variables understood to range over the natural numbers. On the syntax level, we have the constant 0 for the initial time point and the successor function s for modeling the single step from the current time point X to the next $s(X)$. Also, each signature symbol $p \in \Sigma$ is translated to a monadic predicate $p(X)$, parametrized by the time point where it is supposed to hold. Table 4.1 shows three labeled clauses translated this way.

In the light of this translation, the fact that temporal shift cannot be applied to initial clauses corresponds to non-unifiability of the constant 0 with any term starting with the successor function s: that is the reason why clauses (1) and (2) in Table 4.1 cannot be unified and resolved on p. The successful use of temporal shift corresponds to a substitution: we can substitute $Y \mapsto s(X)$ to unify clauses (2) and (3) of Table 4.1 and resolve them. Lemma 4.3 effectively states that the substitutions corresponding to temporal shift operation employed in Definition 4.5 are, in fact, most general unifiers.

4.2.3 Elimination in LTL

In this section we describe how to eliminate variables and clauses in LTL. To lift the corresponding technique from the propositional level, we rely on the presented connection between extended labeled clauses on the one side and the standard clauses over the infinite signature Σ^* on the other. For inherent expressibility reasons not every variable can be eliminated from a set of extended labeled clauses. We will discus the restrictions under which eliminating a particular variable is feasible and practical.

When the elimination process is complete, some of the obtained clauses may carry extended labels with non-zero third label component. Such clauses are undesirable, because a subsequent decision procedure (e.g. a one based on LPSup) would not be able to deal with them. We prove a theorem showing that these labeled clauses can be removed from the set while preserving satisfiability. This means that extended labels provide a useful tool for theoretically justifying the correctness of our approach, but from the algorithmic perspective they can be dispensed with.

Lifting variable elimination

Our strategy for eliminating a single variable p from the basic signature Σ is to use extended labeled clauses to lift propositional elimination steps of all the variable's "instances" $p, p', p^{(2)}, \ldots$ from the "ground level" of the signature Σ^*. To be able to represent the result after elimination, all these instances need to be eliminated from the ground level uniformly, in one step. This seems to be a difficult task when the given set of labeled clauses contains a clause that mentions the variable p in two different time contexts, like, for example, in $\neg p \vee q \vee p'$. In this case the individual eliminations cannot be done independently from each other and we rule the case out from further considerations.

Remark 4.2. There are some interesting subcases where eliminating such a variable would, in theory, be possible and would yield useful results. Consider a clause set containing the following three labeled clauses

$$(0, *, 0) \,||\, p, \;\; (*, *, 0) \,||\, \neg p \vee p', \;\; (*, *, 0) \,||\, \neg p \vee r,$$

from which p can be "semantically"eliminated and one obtains a single clause $(*, *, 0) \,||\, r$. On the other hand, eliminating p from a clause set containing

$$(0, *, 0) \,||\, p, \;\; (*, *, 0) \,||\, \neg p \vee \neg p', \;\; (*, *, 0) \,||\, p \vee p', \;\; (*, *, 0) \,||\, \neg p \vee a,$$

should give us a formula whose models $\mathcal{V} = (V_i)_{i \in \mathbb{N}}$ satisfy the condition

$$(i \bmod 2 = 0 \Rightarrow V_i \models a),$$

which is a property known (Wolper, 1983) not to be expressible by an LTL formula over the single variable a.

Let us now, therefore, assume that we are given a set of labeled clauses N, perhaps a starting labeled clause set for a TST $\mathcal{T}$, and a variable $p \in \Sigma$ such that no clause in N contains more than one possibly primed occurrence of p. We separate N into $N_p \cup N_{\neg p} \cup N_0$, a subset containing p positively (possibly primed), a subset containing p negatively (possibly primed), and a subset not containing p at all. A new set of labeled clauses $\overline{N}$ is constructed as $(N_p \otimes_p N_{\neg p}) \cup N_0$, where

$$N_p \otimes_p N_{\neg p} = \{\mathcal{C}_1 \otimes_p \mathcal{C}_2 \mid \mathcal{C}_1 \in N_p, \mathcal{C}_2 \in N_{\neg p}\}$$

stands for the set of all the labeled resolvents (Definition 4.5) over the variable p between labeled clauses from N_p and $N_{\neg p}$, respectively.

Example 4.3. Let us assume that a set N contains the following labeled clauses

$$(0, *, 0) \,||\, p \vee q \vee r, \tag{4.6}$$
$$(0, *, 0) \,||\, \neg p \vee \neg r, \tag{4.7}$$
$$(*, *, 0) \,||\, r \vee \neg p', \tag{4.8}$$
$$(*, 0, 0) \,||\, \neg p \vee q, \tag{4.9}$$

and these are the only labeled clauses of N mentioning the variable p. Then eliminating p from N means removing the above labeled clauses and replacing them by all the possible labeled resolvents over p. Notice that, actually,

- the tautology (4.6) $\otimes_p$ (4.7) $= (0, *, 0) \,||\, q \vee r \vee \neg r$ can be immediately dropped,
- and (4.6) $\otimes_p$ (4.8) is undefined, because temporal shift does not apply to (4.6).

Thus we replace in N the above four clauses by the only nontrivial resolvent (4.6) $\otimes_p$ (4.9) $= (0, 0, 0) \,||\, q \vee r$.

We now show that variable elimination preserves satisfiability of the given clause set.

Theorem 4.1. *Let $N = N_p \cup N_{\neg p} \cup N_0$ and $\overline{N} = (N_p \otimes_p N_{\neg p}) \cup N_0$ be sets of labeled clauses as described above. Then N is (K, L)-satisfiable if and only if $\overline{N}$ is.*

Proof. Let us first assume that N is (K, L)-satisfiable. This means there is a valuation $V : \Sigma^* \to \{\mathbf{0}, \mathbf{1}\}$ such that $V \models N_{(K,L)}$. In order to show that $\overline{N}$ is (K, L)-satisfiable, we construct a new valuation $\overline{V} : (\overline{\Sigma})^* \to \{\mathbf{0}, \mathbf{1}\}$, where $\overline{\Sigma} = \Sigma \setminus \{p\}$ is the reduced basic signature. Similarly to the propositional case, we do this by simply forgetting the value of eliminated variable's instances:

$$\overline{V}(q^{(i)}) = V(q^{(i)})$$

for every $q \in \overline{\Sigma}$ and every $i \in \mathbb{N}$. Now we need to show that $\overline{V} \models \overline{N}_{(K,L)}$.

Because the variable p does not occur in the clauses of N_0, we have directly that $\overline{V} \models (N_0)_{(K,L)}$. Let us now take a clause $E \in (N_p \otimes_p N_{\neg p})_{(K,L)}$. By Lemma 4.2, E must be of the form $C \vee D$ for some $p^{(i)} \vee C \in (N_p)_{(K,L)}$ and $\neg p^{(i)} \vee D \in (N_{\neg p})_{(K,L)}$. Because these two clauses are true in V by assumption, their resolvent E is true in V by soundness of propositional resolution. Because the variable p does not occur in E, the clause is also true in $\overline{V}$.

To show the opposite direction let us assume that the set $\overline{N}$ is (K, L)-satisfiable, i.e., that there is a valuation $\overline{V} : (\overline{\Sigma})^* \to \{\mathbf{0}, \mathbf{1}\}$ such that $\overline{V} \models \overline{N}_{(K,L)}$. We extend $\overline{V}$ to a valuation V over Σ^* by defining for every $i \in \mathbb{N}$

$$V(p^{(i)}) = \mathbf{1} \text{ if and only if there is a clause } p^{(i)} \vee C \in (N_p)_{(K,L)} \text{ such that } \overline{V} \not\models C.$$

Note that the definition is correct, because by our restriction on variable elimination no instance of p occurs in the clause C above. Next we show that $V \models N_{(K,L)}$.

The fact that $V \models (N_0)_{(K,L)}$ follows again trivially from the assumption. Also, from the way we extended $\overline{V}$ to V, we have $V \models (N_p)_{(K,L)}$. Let us prove by contradiction that also $V \models (N_{\neg p})_{(K,L)}$. Assume there is a clause $\neg p^{(i)} \vee D \in (N_{\neg p})_{(K,L)}$ false in V. This means $V(p^{(i)}) = \mathbf{1}$ and, therefore, there is another clause $p^{(i)} \vee C \in (N_p)_{(K,L)}$ such that C is false in $\overline{V}$. We must have that the propositional resolvent $C \vee D$ is false in $\overline{V}$. But by Lemma 4.3, we have $C \vee D \in (N_p \otimes_p N_{\neg p})_{(K,L)}$. A contradiction. □

Apart from the previously explained restriction, there is another limitation on practical variable elimination. Consider a clause set containing the following two labeled clauses:

$$(*, *, 0) \,||\, \neg x \vee p' \text{ and } (*, *, 0) \,||\, \neg p \vee y'.$$

Eliminating p from the set would yield (possibly among other clauses) the labeled clause $(*, *, 0) \,||\, \neg x \vee y''$. This could be a useful simplification in some contexts, but notice that it got us outside SNF and TSTs, because y now occurs doubly primed.

Although we normally avoid eliminating variables like p above, there is, nevertheless, an advantage in knowing that such a step has a proper meaning and can be performed. If the problematic resolvent could be, for instance, shown redundant in the clause set (e.g. by subsumption), it would be sound to remove it and the desired syntactic simplicity of the clause set would be preserved.

Eliminating labeled clauses by subsumption

Let us now turn to reductions and in particular to showing how to extend subsumption[3] to work with (extended) labels. Unlike in Chapter 2, where we relied on an abstract redundancy concept, here we directly lift propositional subsumption: any standard clause represented by the subsumed labeled clause must be subsumed by a standard clause represented by the subsuming labeled clause. This is achieved by the following:

Definition 4.6. We say that a labeled clause $(b_1, k_1, l_1) \,||\, C$ *subsumes* a labeled clause $(b_2, k_2, l_2) \,||\, D$, if C subsumes D and the merge of the labels (b_1, k_1, l_1) and (b_2, k_2, l_2) is equal to (b_2, k_2, l_2).

In analogy to resolution, the subsumption relation on labeled clauses can be made stronger if we allow the subsuming clause (but not the subsumed one) to be potentially shifted in time. For example, the clause $(*, *, 0) \,||\, q$ subsumes $(*, 1, 0) \,||\, p \vee q'$ in this sense. On the other hand, the clause $(*, *, 0) \,||\, q'$ cannot subsume $(*, *, 0) \,||\, p \vee q$, because there is a standard clause represented by the latter, namely $(p \vee q)^{(0)} = p \vee q$, which is not subsumed by any standard clause represented by the former.

The following theorem states soundness of labeled clause elimination by subsumption.

Theorem 4.2. *Let N and $\widetilde{N}$ be sets of labeled clauses, such that $\widetilde{N} \subseteq N$ and for every $\mathcal{D} \in N \setminus \widetilde{N}$ there exists $\mathcal{C} \in \widetilde{N}$ such that $\mathcal{C}$ subsumes $\mathcal{D}$. Then N is (K, L)-satisfiable if and only if $\widetilde{N}$ is.*

Elimination of "exotic" labels

We know that only the clauses labeled by $(0, *, 0), (*, *, 0)$ and $(*, 0, 0)$, which are the labels used in the definition of the starting labeled clause set, directly correspond to initial, step and goal clauses of a TST, respectively. When clauses with other labels arise during elimination, the subsequent procedure for deciding satisfiability of the resulting set needs to know how to deal with them. Interestingly, according to the theorem below, we may drop several kinds of labeled clauses just after they are created without affecting satisfiability of the clause set.

We will need a simple lemma, which follows directly from the definitions.

[3] A labeled version of self-subsuming resolution can be derived by combining labeled resolution and subsumption in a straightforward way.

Lemma 4.4. *Let (K, L) be an arbitrary rank, $i \in \mathbb{N}$, and $j \in \mathbb{N}^+$. Then $(K + i \cdot L, j \cdot L)$ is a rank and for any label (b, k, l) we have $R_{(K+i\cdot L, j\cdot L)}(b, k, l) \subseteq R_{(K,L)}(b, k, l)$.*

Theorem 4.3. *Let N be a finite set of labeled clauses and let N^- be a subset of N obtained be removing all the clauses with a label (b, k, l) such that either $(b = 0$ and $k \neq *)$ or $(l \neq 0)$. Then N^- is satisfiable if and only if N is.*

Proof. One implication is trivial since $N^- \subseteq N$. For the other implication, we need an auxiliary definition. We say that a label (b, k, l) is *relevant* for a rank (K, L) if $R_{(K,L)}(b, k, l) \neq \emptyset$. Let us now assume that N^- is (K_0, L_0)-satisfiable, i.e. that there is a valuation V such that $V \models (N^-)_{(K_0,L_0)}$. We may choose K_1 of the form $K_0 + i \cdot L_0$ and L_1 of the form $j \cdot L_0$ large enough such that none of the removed clauses, i.e. none of the clauses from $N \setminus N^-$, is relevant for the rank (K_1, L_1). This is possible, because a clause with a label satisfying $(b = 0$ and $k \neq *)$ is only relevant for a rank (K, L) if $k = K + s \cdot L$ for some $s \in \mathbb{N}$, and a clause with a label satisfying $(l \neq 0)$ is only relevant for a rank (K, L) when L divides l. If we now write

$$N_{(K_1,L_1)} = (N \setminus N^-)_{(K_1,L_1)} \cup (N^-)_{(K_1,L_1)},$$

we can observe that $(N \setminus N^-)_{(K_1,L_1)} = \emptyset$ by the choice of (K_1, L_1) and $(N^-)_{(K_1,L_1)} \subseteq (N^-)_{(K_0,L_0)}$ by Lemma 4.4. Therefore, $V \models N_{(K_1,L_1)}$ and so N is (K_1, L_1)-satisfiable. □

Example 4.4. Deriving an empty labeled clause during elimination does not immediately imply that the current clause set is unsatisfiable. For instance, the label of the empty clause $(*, 0, 2) \,||\, \bot$ is only relevant for (K, L) when L divides 2 and thus the current clause set may still be (K, L)-satisfiable for $L > 2$. Moreover, thanks to Lemma 4.4 we can always avoid dealing with the problematic ranks for which the empty clause is relevant by "typecasting" a potential model to a higher rank.

After filtering a clause set with the help of Theorem 4.3, it will only contain clauses with the familiar labels of the starting labeled clause set and possibly also clauses labeled by $(*, k, 0)$, $k \in \mathbb{N}$. These do not pose any further complications, because they arise naturally in our calculus LPSup for LTL satisfiability.

4.3 Implementation and experiment

4.3.1 Variable and clause elimination via Minisat

For our evaluation of the effectiveness of variable and clause elimination in LTL, we adapted the preprocessing capabilities of Minisat (Eén and Sörensson, 2003a) version 2.2. We decided to keep Minisat's main simplification loop, which efficiently combines propositional variable elimination with subsumption and self-subsuming resolution, relying on a fine-tuned heuristics for deciding which variables to eliminate and in what order. To realize LTL elimination by this procedure, we emulated labels by extending the respective clauses with marker literals as detailed below. Although this does not exploit the full potential of variable and clause elimination with labeled clauses, we already obtained encouraging results with this setup.

Marking and freezing

We have seen in Chapter 3 how to use marker literals for tracking dependencies between clauses derived by a SAT solver. Here we show how to employ the same technique for emulating labels during variable and clause elimination in Minisat.

To prepare for simplification of a TST $\mathcal{T} = (\Sigma, I, T, G)$, we allocate variables for the joint signature $\Sigma \cup \Sigma'$ and two extra variables i and g for marking. The following clauses are then inserted into the solver:

- $(C)' \vee i$ for every initial clause $C \in I$,
- $(C)'$ for every simple step clause $C \in T$, i.e. a clause only over Σ,
- $C \vee (D)' \vee \neg i$ for every non-simple, or proper, step clause $C \vee (D)' \in T$, and
- $(C)' \vee g$ for every goal clause $C \in G$.

Note that all the simple clause are shifted to the signature Σ', where all the elimination steps will happen. By marking the shifted initial clauses with i and the proper step clauses with the complement $\neg i$ we ensure that these, incompatibly aligned, clauses will not generate an unsound resolvent: any resolvent between a clause $(C)' \vee i$ and a clause $D \vee \neg i$ over a variable from Σ' will be recognized as a tautology (containing both i and $\neg i$) and therefore discarded.

We keep Minisat from eliminating the marker variables themselves by "freezing" them. The interface of Minisat allows the user to *freeze* selected variables, which means the solver will not attempt to eliminate them. To ensure that the simplification of the TST $\mathcal{T}$ is sound, we freeze the marker variables i and g and also all the lower part variables of proper step clauses both in their Σ and Σ' versions:

$$\bigcup_{C \vee (D)' \in T,\ (D)' \neq \emptyset} var(C) \cup var(C').$$

Intuitively, this protects the "ground level overlap" between $I^{(0)}$ and $T^{(0)}$ and also between every $T^{(i)}$ and $T^{(i+1)}$ from being disrupted by the simplification. A formal proof based on the ideas of the previous section is left to the reader.

Simplified clause set

After the simplification procedure is run, we retrieve the resulting clauses from Minisat and separate them back into sets I, T, and G, based on the present marker literals. Clauses marked with just i belong to I, those marked with just g belong to G, and clauses without a marker belong to T (all of these need to be shifted back to the signature Σ). Also clauses marked with $\neg i$ belong to T (but do not get shifted back). There can also be clauses marked with both i and g, which correspond to clauses with label $(0, 0, 0)$ and can be discarded based on Theorem 4.3, and clauses marked with both $\neg i$ and g which should be treated as $(*, 1, 0)$-clauses or, equivalently, as $(*, 1)$-clauses of LPSup.

Remark 4.3. In the experiment we present below, we needed to extract simplified clause sets compatible with SNF, to be passed as an input to the temporal prover TRP++. For that purpose, we additionally froze the goal clause variable (there was always only one goal clause consisting of a single variable in the experiment) and so neither the $(0, 0, 0)$-clauses nor the $(*, 1, 0)$-clauses were ever generated.

Related work

The simplification process we just described is similar in spirit to a method proposed by Kupferschmid et al. (2011) for simplifying inputs used in bounded model checking. Kupferschmid et al. define "Don't Touch" variables which correspond to the variables frozen in our approach. However, they perform elimination separately on the (equivalent of the) sets I, T, and G. By using the marking literals and, in particular, the complementary markers i and $\neg i$ for marking the initial and proper step clauses, respectively, we can implement the elimination by a single invocation of the simplification loop.

4.3.2 An experiment

We adopted a very similar experimental setup as in Chapter 3. We took all the 3723 LTL formulas of the benchmark set collected by Schuppan and Darmawan (2011) and used the same machines with 3.16 GHz Intel Xeon CPU, 16 GB RAM and Debian 6.0.

The testing proceeded in three stages. First, all the benchmarks were translated by our tool, `TST-translate`, from LTL into TSTs. `TST-translate` is a straightforward SWI-Prolog implementation of the the linear time algorithm described in Section 2.2.2. This means, the output it produces contains a single single-literal goal clause and thus can be treated both as a TST and as a formula in SNF. Second, we applied our Minisat-based elimination tool and obtained a set of simplified TSTs. Finally, we ran two resolution-based LTL provers, namely LS4 (see Section 3.4.1) and TRP++ (Hustadt and Konev, 2003) version 2.1, on both the original and simplified TSTs to measure the effect of the simplification on prover running time. Performing the experiments on two independent implementations should allow us to draw more general conclusion about the effects of variable and clause elimination.

Elimination statistics

For each input TST, we recorded the number of variables and clauses that we were able to eliminate in the second stage. We distinguish variables from the *original* formula and *auxiliary* variables that were introduced by `TST-translate` during the transformation in stage one. In total, 39% of the variables (7% original, 32% auxiliary) and 32% of the clauses were eliminated.

The numbers vary greatly over individual families of the benchmark set. For example, the family `schuppan_phltl` allowed for almost no simplification: only 3% of the variables (just auxiliary), and 2% of the clauses could be removed. On the other hand, 99% of the variables (almost all of them original) and 98% of the clauses were removed on the family `schuppan_O1formula`. While the former extreme can be explained by a concise

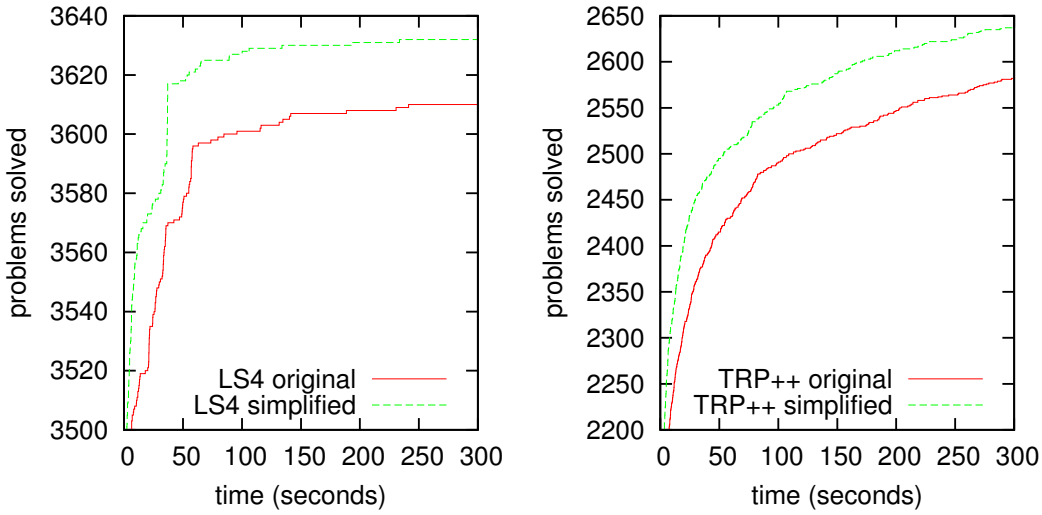


Figure 4.2: Comparing the number of problems solved, simplified and original, within a given time limit. Showing the data for LS4 (on the left) and TRP++ (on the right).

and already almost clausal structure of the original formulas from `schuppan_phltl`, the latter follows from the fact that most of the variables in `schuppan_O1formula` occur in just one polarity, i.e., are pure. Eliminating a pure variable amounts to a removal of all the clauses in which the variable appears.[4]

We observed that the time required to simplify a particular TST was negligible for most of the inputs, with maximum of 0.3s for the largest instance.

Introducing the main results

The results of the third stage, in which we measured the effect of simplification on the performance of the two selected provers, are summarized in Table 4.2 (and at the same time represented graphically in Figure 4.2). The table compares the number of *original* and *simplified* problems solved by LS4 and TRP++, respectively, within the time limit of 300 seconds. It also shows the accumulated times spent during the solution attempts. Note that unsolved problems contribute 300.0s to the accumulated time and the solved ones at least 0.1s due to our measurement technique. The data are grouped by formula families and the last line of the table gives summary totals.

The effect of simplification

We can see from the summary line (and also from Figure 4.2) that both LS4 and TRP++ clearly benefit from the simplification, both in the number of solved instances and the overall running time. In detail, LS4 solved 22 extra problems thanks to the simplification

[4] If p is a pure variable (literal) then $N_{\neg p}$ is empty and so $N_p \otimes_p N_{\neg p}$ is empty as well.

Table 4.2: Performance of LS4 and TRP++ on original (o) and simplified (s) problems.

family	size		LS4		TRP++	
			solved	time	solved	time
acacia_demo-v22	10	o	10	1.0s	10	1.7s
		s	10	1.0s	10	1.1s
acacia_demo-v3	36	o	36	3.6s	36	10.2s
		s	36	3.6s	36	3.8s
acacia_example	25	o	25	2.5s	25	27.4s
		s	25	2.5s	25	6.4s
alaska_lift	136	o	117	6601.6s	6	39108.3s
		s	135	879.5s	8	38714.5s
alaska_szymanski	4	o	4	0.4s	3	314.9s
		s	4	0.4s	4	3.0s
anzu_amba	51	o	47	1432.8s	0	15300.0s
		s	48	1231.5s	0	15300.0s
anzu_genbuf	60	o	46	4314.6s	0	18000.0s
		s	46	4240.1s	0	18000.0s
forobots	39	o	39	4.2s	39	1198.8s
		s	39	3.9s	39	194.2s
rozier_counter	76	o	34	13040.1s	49	8247.4s
		s	34	13018.8s	49	8315.3s
rozier_formulas	2000	o	2000	200.0s	1771	85462.0s
		s	2000	200.0s	1830	66845.9s
rozier_pattern	244	o	244	67.7s	243	2584.3s
		s	244	48.1s	241	1759.9s
schuppan_O1formula	27	o	27	2.7s	27	2.7s
		s	27	2.7s	27	2.7s
schuppan_O2formula	27	o	9	5401.5s	6	6493.8s
		s	9	5401.2s	7	6157.3s
schuppan_phltl	18	o	5	3928.2s	3	4693.3s
		s	5	3916.5s	3	4581.0s
trp_N5x	240	o	240	24.5s	154	27066.8s
		s	240	24.3s	155	26424.4s
trp_N5y	140	o	140	14.3s	0	42000.0s
		s	140	14.0s	0	42000.0s
trp_N12x	400	o	397	1723.4s	210	62978.4s
		s	400	657.6s	204	64713.9s
trp_N12y	190	o	190	2477.1s	0	57000.0s
		s	190	1514.6s	0	57000.0s
total	3723	o	3610	39240.2s	2582	370490.0s
		s	3632	31160.3s	2638	350023.4s

(all of them satisfiable) and the accumulated time for processing the whole set was reduced by 20%. TRP++ solved 80 extra problems (72 satisfiable and 8 unsatisfiable), but also lost 24 problems[5] (10 satisfiable and 14 unsatisfiable), gaining 56 problems overall. The accumulated time was improved by more than 5 hours for TRP++, which means approximately by 5%.

Notice that on many of the families the performance of LS4 was already perfect (all of `acacia`, `alaska_szymanski`, `rozier_formulas`, and `schuppan_O1formula`) or almost perfect (`forobots`, `trp_N5x`, `trp_N5y`) before the simplification (recall that each problem adds at least 0.1s to the accumulated time). Some (scalable) families, on the other hand, seem to be inherently difficult for LS4 (`anzu_genbuf`, `rozier_counter`, `schuppan_O2formula`, and `schuppan_phltl`). This left the "gray zone" where improvement is possible relatively small. LS4 solved more problems from families `alaska_lift` and `trp_N12x` and substantially improved time also on `rozier_pattern` and `trp_N12y`.

The results are more fluctuating for TRP++. There was a huge improvement in the running time on all the `acacia` families and on `forobots`, but the time deteriorated on `rozier_counter` and `trp_N12x`. More problems were solved from both `alaska`s, `rozier_formulas`, `schuppan_O2formula`, and `trp_N5x`. Problems were, however, lost from `rozier_pattern` and `trp_N12x`.

Relation to the previous experiment

Let us now comment on the relation of the presented results for the original problems to the experiment performed in Chapter 3 (Table 3.3), which used the same set of benchmarks, the same time limit and also evaluated both LS4 and TRP++. In the experiment of Chapter 3, each prover relied on its own tool for translating the input formula into the required normal form: LS4 used `TST-translate` and TRP++ used `translator`[6]. Moreover, the translation time was counted into the overall running time. Here, the translation was done beforehand by `TST-translate` and both provers then ran on the same input TSTs. We can see that this did not change the number of problems solved by LS4, which shows that the running time of `TST-translate` itself is negligible.

On the other hand, the overall performance of TRP++ was considerably impaired by exchanging `translator` with `TST-translate`. As an extreme example, notice that TRP++ was not able to solve any problems from the families `anzu_amba`, `anzu_genbuf`, `trp_N5y`, and `trp_N12y` as translated by `TST-translate`. We attribute this to the fact that these problems originally contain many eventualities, which `TST-translate` merges into one. Apparently, this puts an extra load on TRP++, which the prover cannot deal with efficiently. There is, however, also an example in the opposite direction. The performance of TRP++ improved on the family `rozier_pattern`, which contains 17 formulas on which `translator` ran out of memory in the experiment of Chapter 3.

[5] We currently do not have a better explanation for this phenomenon than that a seemingly innocuous change in the presentation of the problem caused by the elimination, like, e.g., the order in which variables appear in the input, steered TRP++ to a different part of the search space. These effects are common in theorem proving in general.

[6] `http://cgi.csc.liv.ac.uk/~konev/software/trp++/translator/`

These observations indicate that in the case of TRP++, we can currently only interpret the positive effects of elimination in relative rather than in absolute terms, because the penalty we pay for transforming the problems to contain just one eventuality, a precondition of our simplification method in its current form, is too high in general. We expect that an alternative elimination procedure that would not merge the eventualities would give TRP++ a similar boost without incurring the corresponding penalty. Implementing such a procedure is left as a future work.

Conclusion

To conclude, the results of our evaluation demonstrate that variable and clause elimination represents a useful preprocessing technique of TSTs. Simplifying a clause set not only removes redundancies introduced by a previous, potentially sub-optimal normal form transformation (when auxiliary variables get eliminated), but usually reduces the input even further. This ultimately decreases the time needed to solve the problem. Further improvements are expected from an independent implementation that would harness the full potential of the mechanism of labels.

4.4 Discussion and related work

We are not aware of any related work directly focusing on simplifying clause normal forms for LTL. Some interesting connections, however, can be drawn with the help of the observations made in Section 2.4.1, which shows that a TST can be viewed as a symbolic representation of a Büchi automaton. For instance, Gerth et al. (1995) in their classical paper construct an automaton accepting the models of an LTL formula φ such that its states are identified with sets of φ's subformulas. A closer look reveals an immediate connection between these subformulas and the variables introduced to represent them in the SNF for φ. The paper also suggests several improvements of the basic algorithm. For instance, it is advocated that subformulas of the form $\mu_1 \wedge \mu_2$ need not be stored, because the individual conjuncts μ_1 and μ_2 will be later added as well and they already imply the conjunction as a whole. We can restate this on the symbolic level as an observation that a variable introduced to represent a conjunctive subformula can always be eliminated. Such a claim is easy to verify.

We believe this connection deserves further exploration, as one could possibly use it to bring some of the numerous techniques for optimizing explicit automata construction (see, e.g., Rozier and Vardi, 2010, for an overview) to the symbolic level. Note, however, that the main application of the explicit automata construction approach lies in model checking and so the resulting automaton is required to be *equivalent* to the original formula. On the other hand, our clausal symbolic approach is meant for satisfiability testing only and so more general *satisfiability preserving* transformations are allowed. An elimination of a variable from the original signature of the formula φ, or the "forgetting step" justified by Theorem 4.3, are examples of transformations that do not have a counterpart on the automata side.

While the explicit notion of a symbolic representation of a Büchi automaton via a clause normal form has received relatively little attention[7] so far, symbolic approaches to LTL model checking and satisfiability based on Binary Decision Diagrams are well known (Clarke et al., 1997). Again, it seems possible that some optimization techniques could be shared between the two approaches. For instance, the various BDD encodings recently studied by Rozier and Vardi (2011), could correspond to different ways of turning a formula into a TST. Finally, an inspiration could also be taken from the techniques for translating LTL formulas into logical circuits (Claessen et al., 2013).

4.5 Conclusion

We have shown that variable and clause elimination, a practically successful preprocessing technique for propositional SAT problems, can be adapted to the setting of linear temporal logic. For that purpose we have utilized the mechanism of labeled clauses, a method for interpreting an LTL formula as finitely represented infinite sets of standard propositional clauses. The ideas were implemented and tested on a comprehensive set of benchmarks with encouraging results. In particular, variable and clause elimination has been shown to significantly improve subsequent runtime of resolution-based provers LS4 and TRP++.

We would like to stress here that labeled clauses provide a general method for transferring resolution-based reasoning from SAT to LTL. It is therefore plausible that other preprocessing techniques, like, for example, the blocked clause elimination (Järvisalo et al., 2010), can be adapted along the same lines. Exploring this possibility will be one of the directions for future work.

[7] A correspondence between SNF and Büchi automata has been shown by Bolotov et al. (2002). The relevant theorem of the paper, however, does not establish an equivalence between models of the formula and accepting runs of the automaton. Its value for translating techniques between the symbolic and explicit approaches is, therefore, limited.

5 Reachability, model checking, and triggered clause pushing for PDR

5.1 Introduction

While studying satisfiability of LTL formulas in the previous chapters, we have learned that the problem translates on the semantic side to finding an infinite sequence of worlds which reaches a certain set of goal worlds infinitely many times. From this chapter on, we shift our attention to the conceptually simpler notion of (single time) reachability, where the task is to look for a finite sequence which reaches a goal world once.

Reachability is a canonical problem studied in formal verification under the name *model checking of invariance properties* (Clarke et al., 2001; Baier and Katoen, 2008). Given a system to be verified, an invariance property φ holds in the system if it is satisfied by every state of the system that is reachable from an initial state. A state that does not satisfy the invariance property is a called a *bad state*. Thus the property φ holds in the system if and only if there is no bad state reachable from an initial state.

Sometimes one also talks in this context about model checking of *safety* properties. While an invariance property can be decided locally in a state, safety properties are properties of computations. Informally, they are characterized by expressing that "something bad will never happen". This means that each violation of a safety property can be recognized already from a finite prefix of a potentially infinite computation. Although checking safety is more general than checking invariance, the former can be reduced to the latter by known techniques (see, e.g. Kupferman and Vardi, 2001).

Roadmap and contributions

In this chapter we take the model-guidance idea we previously developed in the context of LTL-satisfiability checking and show how to specialize it and apply it to reachability. Our aim is to explore its practical utility for model checking of invariance properties and to relate the obtained algorithm to other known approaches for solving the problem. The chapter consists of four main parts.

In the first part (Section 5.2), we formalize reachability for systems which can be symbolically represented using propositional logic. This is an immediate adaptation of the setting we have been using in the previous chapters. Such systems have finitely many states, but their number can be exponential in the size of the representation. We then specialize LS4, our algorithm for LTL-satisfiability checking, to dealing with the reachability task. We call the obtained algorithm Reach. In the first approximation, Reach can be understood as LS4 without Leap restricted to computations in the first

block. In reality, however, one needs to be more careful to take into consideration the so called *finite path semantics*: a new complication which arises because of finite paths that cannot be extended to infinite ones. We close the first part by comparing and contrasting Reach with related work, primarily with bounded model checking (Biere et al., 1999) and interpolation-based model checking (McMillan, 2003).

The second part (Section 5.3) focuses on relating Reach to one of the most successful recent advancements in the field, an algorithm called Property Directed Reachability (PDR) but also known as IC3 (Bradley, 2011; Eén et al., 2011).[1] Although derived from different initial perspectives, namely the model guidance idea in the case of Reach and the notion of a relatively inductive clause (Bradley and Manna, 2007) in the case of PDR, we observe that the cores of the two algorithms are, in fact, very similar. We pinpoint the main difference as a slight change in the treatment of the layers. When this change is applied to Reach, all one needs to do to obtain PDR is to add three independent enhancements on top of the altered core. We describe all these transformation steps in detail, present PDR in full along with a standalone proof of its correctness and close the section by a review of some recent attempts to improve the algorithm further.

In the third part (Section 5.4), we propose *triggered clause pushing*, our own improvement of PDR. We focus on clause propagation phase of the algorithm, one of the three enhancements mentioned above. The aim of propagation is to "push" clauses from one layer to the next, by which PDR primarily speeds up the occurrence of layer repetition, i.e. the termination condition for unsatisfiable inputs. An attempt to push a clause is successful if the SAT solver confirms a certain formula to be unsatisfiable. In the opposite case, the solver computes a model, which is normally thrown away. We propose to keep the model instead and use it as a witness for why the push attempt failed. Because the next pushing of the same clause can only succeed after the witness has been subsumed by a generated layer clause, we may save time by keeping the witnesses and avoiding futile push attempts. Moreover, with triggered clause pushing we can afford to keep clauses pushed as far as possible at all times, which potentially leads to earlier detection of layer repetition.

We have implemented Reach, PDR, and PDR with triggered clause pushing and coupled them with a parser for And-Inverter Graphs, a canonical representation of hardware model checking problems (Biere, 2012). In the fourth part (Section 5.5), we report on experimental evaluation of the obtained tools on invariance property benchmarks of the Hardware Model Checking Competition (HWMCC) (Biere et al., 2012). We compare the strengths of the two basic algorithms, establish the effect of the three independent enhancements, which mainly distinguish PDR from Reach, and, finally, evaluate the merits of the triggered clause pushing technique.

In the concluding Section 5.6, we wrap up and propose directions for future work.

[1] IC3 is the name Aaron Bradley, the originator of the algorithm, gave to the first implementation (Bradley, 2011). The more descriptive name Property Directed Reachability, which we will mostly prefer here, was coined by Eén et al. (2011).

5.2 Specializing LS4 to reachability

5.2.1 Formalizing reachability

The conceptual transition from LTL satisfiability, as embodied by our notion of Temporal Satisfiability Task, to single time reachability amounts to a simple change. Instead of requiring the goal formula to be satisfied infinitely many times along an infinite sequence of valuations, with reachability we only focus on the existence of a finite sequence such that the goal is satisfied by its last element.

Below we formalize reachability by introducing the notion of a Symbolic Transition System (STS). Syntactically, an STS does not differ from a TST. We choose the new name to stress that the intended semantics is now (single time) reachability and to bring the terminology closer to that of verification.

Symbolic transition systems

Definition 5.1. A *Symbolic Transition System* (STS) is a tuple $\mathcal{S} = (\Sigma, I, G, T)$, where Σ is a finite propositional signature, I, called the *initial* formula, and G, the *goal* formula, are sets of clauses over Σ, and T, the *transition* formula, is a set of clauses over $\Sigma \cup \Sigma'$.

An STS $\mathcal{S}$ symbolically represents an explicit transition system $\mathcal{T}_\mathcal{S} = (S, S_I, S_G, R_T)$, which we describe next. We emphasize that the symbolic representation can be exponentially more succinct than the explicit system.[2] The system $\mathcal{T}_\mathcal{S}$ consists of

- the set of *states*[3] S, identified with the set of all valuations over Σ:

$$S = \{s \mid s : \Sigma \to \{\mathbf{0}, \mathbf{1}\}\},$$

- a subset $S_I \subseteq S$ of the initial states, which are states that satisfy the initial formula:

$$S_I = \{s \in S \mid s \models I\},$$

- a subset $S_G \subseteq S$ of the goal states, which are states that satisfy the goal formula:

$$S_G = \{s \in S \mid s \models G\},$$

- and the transition relation $R_T \subseteq S \times S$ of pairs of states (also called transitions) which jointly satisfy the transition formula:

$$R_T = \{(s, t) \mid s, t \in S \text{ and } [s, t'] \models T\}.$$

A *path* in $\mathcal{T}_\mathcal{S}$ is a finite sequence $s_0, \ldots, s_n$ of states such that $(s_j, s_{j+1}) \in R_T$ for every $j = 0, \ldots n-1$. We will be interested in the existence of paths connecting an initial state with a goal state.

[2] The explicit system is an analogue of the semantic graph introduced in Section 2.4.2.
[3] We will prefer using the term "state" instead of "world" from now on.

Definition 5.2. An STS $\mathcal{S}$ is *satisfiable* if there is a path $s_0, \ldots, s_n$ in $\mathcal{T}_\mathcal{S}$ such that $s_0 \in S_I$ and $s_n \in S_G$. We call such a path a *witnessing* path for $\mathcal{S}$.

Remark 5.1. It is useful to notice that the definition of an STS is symmetrical in the following sense. Given an STS $\mathcal{S} = (\Sigma, I, G, T)$ an *inverted* STS is defined as $\mathcal{S}^{-1} = (\Sigma, G, I, T^{-1})$, where T^{-1} is obtained from T by simultaneously removing primes from all the occurrences of primed variables and adding primes to all the occurrences of originally unprimed variables. This corresponds, on the explicit side, to exchanging the initial and goal states and inverting the direction of all the transitions. Therefore, an STS $\mathcal{S}$ is satisfiable if and only $\mathcal{S}^{-1}$ is. Moreover, a witnessing path for $\mathcal{S}$ can be recovered from a witnessing path for $\mathcal{S}^{-1}$ (also vice versa) by reading the sequence backwards.

Although the STS $\mathcal{S}$ and $\mathcal{S}^{-1}$ are equisatisfiable they may be of a different degree of difficulty for solving by a particular algorithm. As a simple example, consider an STS $\mathcal{S} = (\{a, b\}, I, G, T)$, where $I = \{a\}$, $G = \{b\}$, and $T = \{\neg a \vee b'\}$. When starting from an initial state of $\mathcal{S}$ one necessarily reaches a goal state in one step (without any guidance), this does not hold for the initial state $s = \{a \mapsto \mathbf{0}, b \mapsto \mathbf{1}\}$ of $\mathcal{S}^{-1}$ in which one can stay "looping", since $[s, s'] \not\models T^{-1}$. We will explore the practical impact of *search direction*, implicitly defined by using either $\mathcal{S}$ or $\mathcal{S}^{-1}$, in our experimental evaluation in Section 5.5.

5.2.2 The Reach algorithm

In this section we introduce Reach, a specialization of LS4 to reachability. The new semantics allows us to simplify the algorithm in several places. First of all, since we are looking for a finite path, the whole computation happens within the first block. We thus only need one set of the proper layers and the dirty layers can be dispensed with completely. When the constructed path grows to the end of the first block, the algorithm terminates with the result SAT.

Secondly, since the goal is supposed to be reached only once, we can get rid of the Leap inference. Indeed, removing Leap from LPSup is almost all that is needed to turn the calculus into one that decides reachability. A special treatment is required, however, for dealing with witnessing paths that cannot be extended to infinite paths. As we explain in detail below, this aspect of the *finite path semantics* complicates the way in which universal clauses are handled.

Pseudocode

Given an STS $\mathcal{S} = (\Sigma, I, G, T)$, the algorithm Reach (Algorithm 5.1) decides whether there exists a witnessing path for $\mathcal{S}$. Let us first have a look at the global variables of the algorithm and their initialization (lines 1–7).

Similarly to LS4, Reach relies on the marking abstraction (recall Section 3.2.2) to track dependencies between derived clauses. It maintains a set I of the initial clauses marked by the initial marker $\circ$ and a set T of the transition clauses marked by the empty set of markers. Notice that Reach does not separately treat the simple transition clauses, which in LS4 initialize the set of universal clauses U. As detailed later on, in order to respect the finite path semantics, when Reach derives a new universal clause, i.e. a clause

Algorithm 5.1 Reach

Input:
An STS $\mathcal{S} = (\Sigma, I, G, T)$

Output:
A witnessing path for $\mathcal{S}$ or a guarantee that none exists

1: $\mathsf{I} \leftarrow \{C^{\{\circ\}} \mid C \in I\}$ /* Initial clauses marked by $\circ$ */
2: $\mathsf{T} \leftarrow \{C^{\emptyset} \mid C \in T\}$ /* Transition clauses marked by an empty set of markers */
3: **foreach** $j \geq 0 : \mathsf{U}_j \leftarrow \emptyset$ /* Universal clause layers */
4: $\mathsf{L}_0 \leftarrow \{C^{\{\bullet\}} \mid C \in G\}$ /* Goal clauses marked by $\bullet$ go to layer 0 */
5: **foreach** $j > 0 : \mathsf{L}_j \leftarrow \emptyset$ /* All other proper layers start empty */
6: $k = 0$ /* The current index for the goal state */
7: $\mathcal{V} \leftarrow \emptyset$ /* The constructed path is initially empty */
8:
9: **loop**
10: **if** $|\mathcal{V}| = 0$ and $SAT?[\,(\mathsf{I} \wedge \mathsf{F}_0)'\,]$ or
$|\mathcal{V}| > 0$ and $SAT?[\,Lits(V_{|\mathcal{V}|-1}) \wedge \mathsf{T} \wedge (\mathsf{F}_{|\mathcal{V}|})'\,]$ **then**
11: $V \leftarrow$ valuation extracted from the Σ'-part of the satisfying assignment
12: **add** V to the end of $\mathcal{V}$ /* $|\mathcal{V}|$ increased by 1 */
13: **if** $|\mathcal{V}| > k$ **then** /* Completed the full path */
14: **return** $\mathcal{V}$
15: **else** /* Unsuccessful extension */
16: $C^m \leftarrow$ explaining marked clause from the unsuccessful extension
17: **if** $C = \bot$ **then** /* Empty clause */
18: **if** $m = \emptyset$ or $m = \{\circ\}$ or $m = \{\bullet\}$ **then**
19: **return** UNSAT /* "Unconditional" empty clause derived */
20: **else** /* Necessarily $m = \{\circ, \bullet\}$ and $|\mathcal{V}| = 0$ */
21: $k \leftarrow k + 1$
22: **else** /* A non-empty clause (over Σ); cannot be marked with $\circ$ */
23: **remove** $V_{|\mathcal{V}|-1}$ from $\mathcal{V}$ /* $|\mathcal{V}|$ decreased by 1 */
24: **if** $m = \emptyset$ **then**
25: **add** C^m to $\mathsf{U}_{(k-|\mathcal{V}|)}$ /* A new universal clause */
26: **else** /* Depends on the goal: $m = \{\bullet\}$ */
27: **add** C^m to $\mathsf{L}_{(k-|\mathcal{V}|)}$ /* A new proper layer clause */
28: /* Layer repetition check */
29: **if** $\mathsf{L}_i = \mathsf{L}_j$ for some $0 < i < j \leq k$ **then**
30: **return** UNSAT /* Enough empty clauses derivable */

marked by the empty set of markers, this clause is put into one of the *universal layers* U_i. These sets of clauses are all initially empty.[4]

Since there is only one block to consider in Reach, there is only one set of *proper layers* L_i. Clauses in these layers are marked by a goal marker $\bullet$. The layers L_i for $i > 0$ are initially empty and the layer L_0 is initialized to contain the goal clauses. The configuration of the block is represented by a single number k, which denotes the current index at which the goal clauses are to be satisfied. The goal index k is initialized to 0. Finally, there is a variable $\mathcal{V}$, which stores a sequence of valuations. The variable represents the currently constructed path and is initialized by an empty sequence.

The main loop of Reach (lines 9–30) is a straightforward simplification of its equivalent in LS4 (cf. Algorithm 3.2 on page 83). The loop is driven by an adaptation of the two extension queries (line 11), which look identical to the initial and proper extension queries of LS4, but differ internally due to a new definition of the formula "macro" F_i. As detailed below (see equations (5.1) and (5.2)), the F_i's now take into account the finite path semantics and the new way of dealing with universal clauses. However, the intuition is the same as in LS4. The "macro" F_i collects those clauses that must be satisfied by the i-th member V_i of the of the currently constructed path $\mathcal{V}$.

In Reach, there is no need for a model repetition check nor for a new block check: as soon as the constructed path reaches the goal state, the computation terminates with the SAT result (line 14). Also the branch of the unsuccessful path extension (lines 15–30) is simpler than in LS4, because the dirty layers and the Leap inference are missing and a block extension of the only block amounts to a simple increment of the index k (line 21). The parts of the code dedicated to detecting unsatisfiability, i.e. the derivation of an "unconditional" empty clause (line 18) and the layer repetition check (line 29), are both immediate adaptations from LS4. The only interesting difference is thus the handling of the universal clauses.

Universal clauses under the finite path semantics

Example 5.1. Let us consider an STS $\mathcal{S} = (\Sigma, I, G, T)$ with $\Sigma = \{p, q\}$, $I = \{\neg q\}$, $G = \{\neg p, q\}$, and $T = \{p\}$. This STS is satisfiable as witnessed, for example, by the path

$$\mathcal{V} = \{p \mapsto \mathbf{1}, q \mapsto \mathbf{0}\}, \{p \mapsto \mathbf{0}, q \mapsto \mathbf{1}\}.$$

Notice that although the transition clause p spans only the signature Σ, it cannot be treated universally, because the algorithm would then immediately obtain a conflict with the goal formula G, which contains the clause $\neg p$.[5] But even if p is initially put into the set of proper transition clauses T, as actually done by Reach, the algorithm may still derive the clause in a universal context, for instance, as the explaining clause $p^{\emptyset}$ corresponding to the unsuccessful extension of the initial state $\{p \mapsto \mathbf{0}, q \mapsto \mathbf{0}\}$.

[4] As in the presentation of LS4, we introduce universal layers U_i (and also the proper layers L_i) as infinite sequences of clauses. This notational abstraction cannot sabotage computability, since only finitely many of the mentioned sets are non-empty at any moment during the computation.

[5] This would correspond to deriving the unconditional empty clause $(*, 0) \,||\, \bot$ of LPSup.

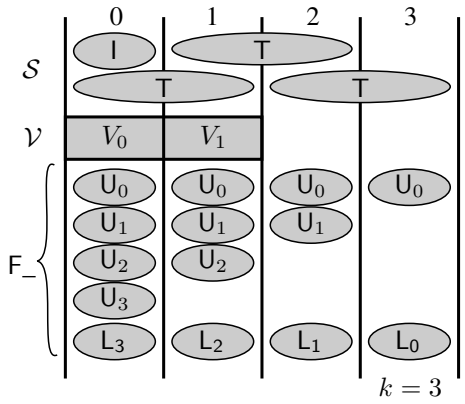


Figure 5.1: Alignment between the constructed path and the clause sets in Reach.

We see that in order to prevent Reach from classifying the STS $\mathcal{S}$ as unsatisfiable, the universal clauses must be treated in a more refined way than in LS4.

The solution adopted by Reach is the following. Similarly to the proper layers, the algorithm maintains the universal layers and populates them based on the "distance" of the currently derived clause to the index of the goal (line 25). By storing the universal clauses in the layers U_i, the algorithm keeps track of the number of "unfoldings" of the transition formula that were needed to derive the particular clause. It is not sound to assert a universal clause closer to the goal than where it was derived, because that would correspond to reasoning about path segments that extend beyond the goal index k. However, as a useful optimization, we can assert the clause further away from the goal. These considerations are reflected in the definition of the formula "macro" F_i.

Formally, we could define the macro as the following conjunction:

$$\mathsf{F}_i = \mathsf{L}_{(k-i)} \wedge \mathsf{U}_{(k-i)}, \tag{5.1}$$

and only use each universal clause for guidance at the respective index at which it was derived. The more efficient approach, which still makes use of universality in a sound way, is to define the macro as

$$\mathsf{F}_i = \mathsf{L}_{(k-i)} \wedge \bigwedge_{0 \leq j \leq k-i} \mathsf{U}_j. \tag{5.2}$$

The corresponding alignment between the involved clauses and the constructed path is depicted in Figure 5.1. We can see how clauses from the universal layer U_j contribute to the formula F_i whenever $j \leq k - i$. Thus in the shown configuration with $k = 3$, when extending the current path $\mathcal{V}$ of length $|\mathcal{V}| = 2$ the new valuation is required to satisfy the formula $\mathsf{F}_2 = \mathsf{L}_1 \wedge \mathsf{U}_1 \wedge \mathsf{U}_0$.

The soundness of our approach is formally captured by the following lemma.

Lemma 5.1. *Let $\mathcal{S} = (\Sigma, I, G, T)$ be an STS given as an input to Reach and let U_i be the values of universal layers at any moment during the run of the algorithm. Moreover, let $s_0, \ldots, s_n$ be a witnessing path for $\mathcal{S}$. Then for any $0 \leq i \leq n$*

$$s_i \models \bigwedge_{0 \leq j \leq n-i} \mathsf{U}_j. \tag{5.3}$$

Proof. We prove the lemma by induction along the run of the algorithm. Because the universal layers are initialized as empty, the statement trivially holds at the beginning of the run of the algorithm.

Now let us assume that the statement holds just before an addition of a new universal clause $C^{\emptyset}$ to a universal layer U_{l+1} for some $0 \leq l < n$ in order to show it will also hold afterwards. This clause $C^{\emptyset}$, being an explaining clause corresponding to the proper extension query marked by an empty set of markers, satisfies

$$\mathsf{T} \wedge \left(\bigwedge_{0 \leq j \leq l} \mathsf{U}_j \right)' \models C^{\emptyset}. \tag{5.4}$$

To prove that (5.3) holds after the insertion it is sufficient to show that $s_i \models C^{\emptyset}$ for every $0 \leq i \leq n - l - 1$. Let us pick such an i and consider the states s_i and s_{i+1}. They naturally satisfy $[s_i, s'_{i+1}] \models \mathsf{T}$ and, by the induction hypothesis, $s_{i+1} \models \bigwedge_{0 \leq j \leq l} \mathsf{U}_j$ because $l \leq n - i - 1$. It follows from (5.4) that $s_i \models C^{\emptyset}$. □

Correctness and termination of Reach

The correctness and termination proofs for Reach can be obtained by reviewing and simplifying the corresponding results for LS4. We focus here on explaining what needs to be changed in order to accommodate the finite path semantics and the ensuing different treatment of universal clauses. Affected is the correctness proof in the unsatisfiable case.

Similarly to how we used entailment (5.4) to prove Lemma 5.1 about universal clauses, one can use

$$\mathsf{T} \wedge \left(\mathsf{L}_l \wedge \bigwedge_{0 \leq j \leq l} \mathsf{U}_j \right)' \models C^{\{\bullet\}}, \tag{5.5}$$

a property satisfied by any proper layer clause $C^{\{\bullet\}}$ derived into L_{l+1}, to prove a similar lemma expressing soundness of the layer clauses:

Lemma 5.2. *Let $\mathcal{S} = (\Sigma, I, G, T)$ be an STS given as an input to Reach and let U_i and L_i be the values of the universal and proper layers at any moment during the run of the algorithm. Moreover, let $s_0, \ldots, s_n$ be a witnessing path for $\mathcal{S}$. Then for any $0 \leq i \leq n$*

$$s_i \models \mathsf{L}_{n-i}. \tag{5.6}$$

These two lemmas, in fact analogues of items 3 and 4 of Invariant 3.2 of LS4 (see page 80), together essentially tell us that any state s lying l steps from the last state on a witnessing path for $\mathcal{S}$ necessarily satisfies all the clauses from $\mathsf{U}_0, \ldots, \mathsf{U}_l$ and L_l. Equipped by this knowledge, it is straightforward to prove that:

1. When Reach derives the empty clause $\bot^{\{\circ,\bullet\}}$ and is about to increment the goal index k (line 21), it has just shown that there is no witnessing path of length k.

2. When the algorithm derives one of the unconditional empty clauses $\bot^{\emptyset}$, $\bot^{\{\circ\}}$ or $\bot^{\{\bullet\}}$ (line 18) and the distance of the current last state of the constructed path to the goal index is $d = k - |\mathcal{V}|$, it has just shown that, respectively,
   - there is no path in the transition system of length $L \geq d$,
   - there is no path starting in an initial state of length $L \geq d$,[6]
   - there is no path ending in a goal state of length $L \geq d$.

Let us by *iteration* i denote the period of the run of Reach during which the goal index k equals i. The algorithm starts with iteration 0 and transitions from one iteration to the next when the goal index is incremented on line 21. When an unconditional empty clause is derived in iteration k, the algorithm has already shown during the previous iterations that there are no witnessing paths of length $0, \ldots, k-1$ (item 1.) and now we also know that there is no witnessing path of length $L \geq d$ (item 2.). Since $d \leq k$, we conclude that there is no witnessing path of any length. This shows correctness of the algorithm in the "unconditional" empty clause case (UNSAT on line 19).

Also the treatment of the case of repeating layers (UNSAT on line 30) is analogous, if not identical, to how its dealt with in LS4 (Theorem 3.1). Assume the algorithm detects repeating layers $\mathsf{L}_i = \mathsf{L}_j$ for some $0 < i < j \leq k$. We construct the infinitely repeating layers $\overline{\mathsf{L}}_l$ by defining

$$\overline{\mathsf{L}}_l = \begin{cases} \mathsf{L}_l & \text{for } 0 \leq l < i, \\ \mathsf{L}_{i+(l-i) \bmod (j-i)} & \text{for } l \geq i, \end{cases} \tag{5.7}$$

and show, using the "derivation replaying argument", that an analogy of Lemma 5.2 holds for the infinitely repeating layers $\overline{\mathsf{L}}_l$ in place of L_l and for every $l \in \mathbb{N}$. This is the place where the new definition of the universal clauses is reflected. But notice that as we move from small indexes l to those greater than j while proving (5.7) by "replaying" the entailment (5.5) there, we need the validity of $\bigwedge_{0 \leq j \leq l'} \mathsf{U}_j$ for some $i \leq l' < j$ to justify each step while already a stronger $\bigwedge_{0 \leq j \leq l} \mathsf{U}_j$ is available by Lemma 5.1. We conclude the proof by using the infinitely repeating layers to "progress" the already established non-existence of witnessing paths of lengths $0, \ldots, k-1$ towards arbitrary length $l \geq k$.

[6] This case, in fact, can only occur when $|\mathcal{V}| = 0$ (and thus $d = k$), because only then does the set of initial clauses I participate on the extension query.

5.2.3 Related work

In this section we focus on approaches most closely related to Reach, in particular, on Bounded Model Checking (BMC) and Interpolation-based Model Checking (IMC). Before the advent of modern SAT solvers, the state of the art relied on representations based on BDDs (Bryant, 1986) to symbolically analyze systems (McMillan, 1993). Although this method still complements the more recent approaches in modern checkers, it is less related to our work and will not be mentioned here anymore. We also postpone discussing the connection to PDR (Bradley, 2011), which will be the main topic of the next section.

Bounded Model Checking

The relation between LS4 and BMC (Biere et al., 1999), which we discussed in Section 3.5.2 in the context of LTL satisfiability, becomes even more apparent when the two approaches are specialized to reachability.

Let us have a look at Algorithm 5.2. Given an STS $\mathcal{S}$ BMC works by iteratively constructing a sequence of formulas which encode the existence of a witnessing path for $\mathcal{S}$ of increasing lengths. The formulas are successively supplied to a SAT solver and when a satisfiable one is detected a witnessing path is recovered from the corresponding satisfying assignment.

Algorithm 5.2 Bounded Model Checking of Symbolic Transition Systems

Input:
An STS $\mathcal{S} = (\Sigma, I, G, T)$
Output:
A witnessing path for $\mathcal{S}$

1: **for** $k \leftarrow 0, 1, \ldots$ **do**
2: **if** $SAT?[\, I^{(0)} \wedge \bigwedge_{j=0}^{k-1} T^{(j)} \wedge G^{(k)} \,]$ **then**
3: extract the satisfying assignment in the form $[s_0^{(0)}, \ldots, s_k^{(k)}]$
4: **return** $s_0, \ldots, s_k$

As we can see, in its pure form BMC is only concerned with finding witnessing paths, not with showing their non-existence. Traditional approaches to making BMC complete (see Biere, 2009, Section 4) attempt to (automatically) establish an upper bound on the length of the longest witnessing path. This bound, called the *completeness threshold*, could be as simple as the size of the state space $2^{|\Sigma|}$, but only more refined estimates can be useful in practice. Examples are the *recurrence diameter* (Kroening and Strichman, 2003), which is the length of longest simple path (a path with no repeating states) in the transition system, and the forward and backward *recurrence radii*, which correspond to the longest simple paths starting in an initial state or ending in a goal state, respectively.

One particular complete approach, which can be seen as establishing the backward recurrence radius, is known under the name *temporal induction* (Sheeran et al., 2000).

There the SAT queries expressing the existence of a witnessing path of length k:

$$I^{(0)} \wedge \bigwedge_{0 \leq j < k} T^{(j)} \wedge G^{(k)} \tag{5.8}$$

are interleaved with queries for the existence of a simple path ending in a goal state:

$$\left(\bigwedge_{0 \leq i < j \leq k} \bigvee_{p \in \Sigma} \neg(p^{(i)} \leftrightarrow p^{(j)}) \right) \wedge \bigwedge_{0 \leq j < k} T^{(j)} \wedge G^{(k)}.$$

Unsatisfiability of the latter query for a particular value of k, conditioned on unsatisfiability of the former query for all the previous values of k, implies there is no witnessing path for the system of any length. Because most of the clauses can be shared between the queries, temporal induction naturally benefits from incremental SAT solving (Eén and Sörensson, 2003b).

It is easy to see that Reach, in fact, analyzes that same sequence of formulas as BMC. Our algorithm essentially cuts the formula (5.8) into pieces corresponding to the consecutive time moments (or, equivalently, to the individual shifted copies of the basic signature Σ) and relies on the mechanism of solving by parts (recall Section 3.2.1) to solve the formula as a whole. By imposing a fixed order on how the satisfying assignment can be constructed, namely by "growing it" from an initial state towards the goal, Reach potentially loses some of the efficiency in comparison to the unrestricted SAT solver in BMC. On the other hand, the extra control over the explaining clauses, which necessarily span only one copy of the basic signature Σ (in comparison to arbitrary clauses the CDCL algorithm learns internally in BMC), allow Reach to realize the repetition detection, which makes the algorithm complete in the UNSAT case.

Actually, the idea of repetition detection has recently been studied right in the context of BMC. Fuhrmann and Hoory (2009) propose to analyze the proof that the formula (5.8) is unsatisfiable and look for so called Δ-invariant cuts. The presence of such a cut in the proof implies that there are no witnessing paths of length $k + \Delta$, $k + 2\Delta$, etc. The authors also present a proof manipulation algorithm that rearranges the proof by the natural temporal order to expose cuts that would otherwise not be detected. The method, however, remains incomplete as there is no guarantee that the SAT solver will generate a proof of the required form. Note that the Reach algorithm does not require explicit proof generation from the SAT-solver side and achieves completeness through controlled clause generation.

Another interesting reference in this context is the work of Stoffel and Kunz (1997) who propose a method for sequential equivalence checking based on circuit unwinding combined with learning certain transformations for establishing observable equivalence over k-steps. Although the connection to Reach cannot be traced so far as in the case of the previously mentioned approach, the authors face the same task of detecting repetitions, here within the transformation sets, upon which the argument can be run for arbitrary large values of k and thus a true equivalence is shown.

We would like to close the list of similarities in the context of BMC by mentioning the work of Strichman (2001), whose method of *internal constraints replication* resembles

our treatment of universal clauses. Constraint replication means to actively copy learned clauses within the SAT solver between individual time frames. To justify soundness of copying a learned clause C to a frame, say, i steps in the future, one ensures that the original formula's clauses $D_1, \ldots, D_n$ from which C was (transitively) derived are in (5.8) also present in their shifted incarnation $D_1^{(i)}, \ldots, D_n^{(i)}$. A sufficient condition for this is that $D_1, \ldots, D_n$ are already part of $\bigwedge_{0 \leq j < k-i} T^{(j)}$. There is also a symmetrical case for copying "into the past" (with a negative value of i). This latter case is the one implicitly realized within Reach by our treatment of universal clauses. Strichman (2001) implements the replicability check by propagating certain tags along with the learned clauses – one can recognize the idea of labels in disguise.

Interpolation-based Model Checking

Let us state the propositional version of Craig's interpolation theorem (Criag, 1957) in the form typically used in verification. For any two propositional formulas A and B for which the conjunction $A \wedge B$ is unsatisfiable there exists a formula P over the common variables of A and B (i.e., $\mathit{Vars}(P) \subseteq \mathit{Vars}(A) \cap \mathit{Vars}(B)$) called the *interpolant* of A and B such that $A \models P$ and $P \wedge B$ is unsatisfiable.

In the seminal work on Interpolation-based Model Checking (IMC), McMillan (2003) showed that such an interpolant can be efficiently extracted from any resolution proof of unsatisfiability of the conjunction $A \wedge B$ (see also Krajícek, 1997; Pudlák, 1997). IMC uses this idea to extend the BMC framework into an approximate reachability analysis procedure with a completeness guarantee.

Imagine we decompose the BMC query formula (5.8) into A and B as follows:

$$A = I \wedge T, \qquad B = \bigwedge_{0<j<k} T^{(j)} \wedge G^{(k)}. \tag{5.9}$$

Then the corresponding interpolant P is a formula over the primed signature Σ' satisfying $I \wedge T \models P$ while making the conjunction $P \wedge B$ unsatisfiable. In other words, P over-approximates the set of states reachable in one step from an initial state, but is at the same time strong enough to guarantee that no P-state can reach the goal in $k-1$ steps.

McMillan (2003) uses decomposition (5.9) to define a monotone operator $\mathcal{F}_k$ on propositional formulas where what we have just described is essentially the evaluation of $\mathcal{F}_k$ on the formula I. To evaluate $\mathcal{F}_k(X)$ on a general input X one just replaces I by X in the definition of the formula A above. And to ensure that the operator is monotone one takes $(P^{(-1)} \vee X)$ as its final result.[7]

The overall IMC procedure attempts to obtain a fixed point of $\mathcal{F}_k$ on the initial formula I. It computes $\mathcal{F}_k(I), \mathcal{F}_k(\mathcal{F}_k(I)), \ldots$ until one of the examined formulas is revealed to be satisfiable (and thus does not provide an interpolant) or the sequence stabilizes on the fixed point. The former case means that the over-approximation was too coarse for proving unreachability and the procedure needs to be restarted with an increased value of k (in hope for better precision). Notice, however, that if satisfiability is discovered

[7] By $P^{(-1)}$ we mean that the interpolant P is reinterpreted over the variables of the basic signature Σ.

during the computation of $\mathcal{F}_k(I)$ we know there is a witnessing path for the input STS and the procedure will terminate.

In the latter case, the obtained fixed point F satisfies $I \to F$, $F \wedge T \to F'$, and no F-state can reach the goal in $k-1$ steps. Thus F allows us to conclude that the given STS $\mathcal{S}$ is unsatisfiable. McMillan (2003) proved that for an unsatisfiable STS the later case is bound to occur at the latest with k equal to the backward radius of the system (the maximal length of a shortest path from any state to a goal state).

We will now explain that similarly to IMC, Reach also computes interpolants. Recall the formula "macros" F_i defined by equation (5.2) on page 131, which guide the path construction in Reach. It is easy to see that when the algorithm discovers that there is no witnessing path of length k by deriving the empty clause $\bot^{\{\circ,\bullet\}}$ (line 21) then for every $0 \leq i \leq k$ the formula $(\mathsf{F}_i)^{(i)}$ is an interpolant for the pair

$$A_i = \bigwedge_{i \leq j < k} T^{(j)} \wedge G^{(k)}, \qquad B_i = I \wedge \bigwedge_{0 < j < i} T^{(j)}.$$

In particular, for $P = (\mathsf{F}_{k-1})^{(k-1)}$ we have $T^{(k-1)} \wedge G^{(k)} \models P$ and $I \wedge \bigwedge_{0<j<k-1} T^{(j)} \wedge P$ is unsatisfiable. Thus the set of P-states over-approximates the one step preimage of the set of goal states and, at the same time, no P-state is reachable from an initial state in $k-1$ steps. We see that the roles of the formulas A_i and B_i with respect to the time flow are exchanged. To get a prefect correspondence with IMC it would be enough, however, to run Reach on the inverted STS $\mathcal{S}^{-1}$ or, in other words, to change the algorithm to construct the path backwards from a goal state towards the initial states.

McMillan's (2003) convergence argument for IMC translates to Reach in the following form. For an unsatisfiable STS, when the index k becomes larger than the forward radius l of the system, the algorithm will no longer update the "low index" layers L_j (and U_j) for $0 < j \leq k - l$. Thus in analogy to IMC, which still needs to wait for the fixed point to be reached even when k is already as large as the backward radius, Reach needs to wait for the repetition to occur within the layers. Once the index k exceeds the value of l, however, convergence is in some sense already guaranteed. A similar observation holds also for the PDR algorithm (Bradley, 2011), the topic of the next section.

Reach differs from IMC substantially in the way the interpolants are derived. Interpolation-based model checking relies on the extraction and subsequent processing of resolution proofs generated by the SAT solver. Although this is relatively easy to implement in the CDCL setting (Zhang and Malik, 2003), there is some performance penalty connected with the necessary bookkeeping and not all SAT solvers support the feature. Moreover, the extracted interpolants tend to be highly redundant (McMillan, 2003) and typically need to be simplified afterwards. In contrast, all we need for our method from the SAT solver is the mechanism for solving under assumptions (Section 3.2) and the invariants obtained are already in CNF and contain very little redundancy. Note that each new clause derived into a particular layer properly strengthens it.

The essence of interpolation in Reach can be cast as an algorithm for obtaining interpolants for a general formula pair A, B. Algorithm 5.3 showcases its pseudocode.

This "interpolation without proofs" paradigm has already been described by other researchers (Chockler et al., 2012). We believe it could also be applied in a more general

Algorithm 5.3 Interpolation without proofs

Input:
Propositional formulas A and B
Output:
A CNF interpolant P of A and B, unless the conjunction $A \wedge B$ is satisfiable

```
1: P ← ∅
2: while SAT?[ B ∧ P ] do
3:    V ← the corresponding satisfying assignment restricted to Vars(A) ∩ Vars(B)
4:    if not SAT?[ Lits(V) ∧ A ] then
5:       C ← the corresponding explaining clause
6:       add C to P /* Note that Vars(C) ⊆ Vars(A) ∩ Vars(B) */
7:    else
8:       return "A ∧ B is SAT" /* Obtain the satisfying assignment if necessary. */
9: return P
```

setting than purely propositional, such as Satisfiability Modulo Theories (Nieuwenhuis et al., 2006).

5.3 Towards Property Directed Reachability

Property Directed Reachability (PDR), also known as IC3, is a recently proposed algorithm for deciding reachability in symbolically represented transition systems (Bradley, 2011; Eén et al., 2011). Since its discovery in 2010, it has already established itself as one of the strongest model checking algorithms used in hardware verification.

In this section, we show that our algorithm Reach and PDR are closely related. In fact, we explain how to *transform* Reach into PDR in several simple steps. This provides a new perspective on the famous algorithm by relating it to the model guidance idea. A perspective that should be novel to those only familiar with the standard exposition, which primarily builds on a notion of inductiveness (Bradley and Manna, 2007).

Another advantage of discretizing the transition from Reach to PDR is that one can then separately measure the effect of each step on the practical performance of the transformed algorithm. We will report on exactly that experiment later on, in Section 5.5. Here we provide a detailed presentation of the transformation (Sections 5.3.1 and 5.3.2), the final pseudocode of PDR together with a proof of its correctness (Section 5.3.3) and we also comment on the work related to the algorithm (Section 5.3.4).

5.3.1 Monotone layers

The main difference between Reach and PDR, which is also the only difference in the core of the algorithms, is an extra condition that PDR imposes on the layers. PDR requires that its layers are *monotone* or, more precisely, linearly ordered by set inclusion.

While this immediately entails a theoretical improvement of the algorithm's time complexity upper bound, namely from doubly to singly exponential, its observed practical performance, as we later show, actually deteriorates. The monotone layers, however, subsequently motivate and enable three independent enhancements of the algorithm, whose practical benefits are significant. In this sense, PDR can be understood as Reach enhanced.

We will now set out to show how to keep the layers monotone by just a slight change in the usual update of the layers that happens after an unsuccessful extension. To keep our exposition simple and also to conform with the standard presentation of PDR, we will only consider a single sequence of sets of clauses $L_0, L_1, \ldots$ (corresponding to the proper layers of Reach). Thus we will here ignore the possibility of using the dependency tracking technique for the discovery of universal clauses. Omitting this optimization does not influence correctness of the algorithms in any way.[8]

Unsuccessful extensions in Reach and PDR

Let us recapitulate (taking into account the simplified treatment of layers) that the extension query for a state s lying in a position i steps from the goal asks whether there is a successor of s with respect to the transition formula T that satisfies the clauses of layer L_{i-1}. Formally, the query is expressed by the formula

$$SAT?[\, Lits(s) \wedge T \wedge (L_{i-1})' \,]. \tag{5.10}$$

An unsuccessful extension occurs when no such successor exists and the formula (5.10) is unsatisfiable. In that case, we expect our SAT solver to return a sub-cube $r \subseteq Lits(s)$ of those assumptions $Lits(s)$ that were actually used in the unsatisfiability proof. Let us give this cube r a name and call it a *reason* for the unsuccessful extension. This notion will become useful mainly later on. As we know from Section 3.2.1, by complementing the reason cube we obtain a corresponding explaining clause $C = \sim r = \{\sim l \mid l \in r\}$. It follows that this clause is a property of the preimage (with respect to T) of set of states represented by L_{i-1}:

$$T \wedge (L_{i-1})' \models C \tag{5.11}$$

and that the state s fails to satisfy the clause: $s \not\models C$.

Now, in Reach we insert the explaining clause C into layer L_i and backtrack over the state s to look for a different state that additionally satisfies C. The main difference in PDR is that we insert the new clause into all the layers $L_0, \ldots, L_i$. This is all that is needed to ensure monotonicity. However, to maintain overall correctness of the algorithm we must additionally require that the explaining clause C does not rule out any goal state or, in symbols, that $G \Rightarrow C$. We will discuss in detail how to satisfy this *weaker-than-goal* requirement after formally stating PDR's main invariants.

[8] Two further remarks are in order. First, with the monotonicity condition imposed in PDR there is no need to organize the universal clauses in separate layers (like in Reach) and we can store them just in one universal set U (cf. the set $\mathbf{F}_\infty$ in the presentation of Eén et al., 2011, Section IV). Second, although dependency tracking is not explicitly shown in Pseudocode 5.5, our implementation of PDR does use the technique and so its effects are reflected in the reported experiments.

Layer invariants in PDR

At any moment during the run of the algorithm the layers in PDR satisfy the following three properties:

1) L_0 is equivalent to G,

2) $L_{j+1} \subseteq L_j$ and thus $L_j \Rightarrow L_{j+1}$ for any $j \geq 0$,

3) $(L_j)' \wedge T \Rightarrow L_{j+1}$ for any $j \geq 0$.

Invariant 1) generalizes a similar claim from Reach, where L_0 is initialized by the goal clauses G and then remains constant during the whole subsequent computation. Since PDR inserts explaining clauses even to L_0, it relies on the mentioned weaker-than-goal requirement (details still further below) to keep 1) valid.

Invariant 2) is the discussed monotonicity of layers, which distinguishes PDR from Reach. It is satisfied at the beginning of the algorithm's run, because PDR initializes all the layers except L_0 to be empty, and it is also preserved when a new explaining clause is derived, as the algorithm inserts the clause to all the layers $L_0, \ldots, L_i$ for some i. Notice that it follows from monotonicity that when a layer repetition occurs in PDR, it must be between two neighboring layers, i.e., $L_j = L_{j+1}$ for some j.

Finally, invariant 3), which PDR shares with Reach, informally states that the layer L_{j+1} over-approximates the preimage (with respect to T) of layer L_j. The key to showing preservation of invariant 3) is property (5.11) of explaining clauses. The preservation is immediate in Reach, where the new explaining clauses C is added just to L_i. In PDR, where the clause is added to $L_0, \ldots, L_i$, we combine the same argument with monotonicity of layers (in particular, we use the fact that $L_j \Rightarrow L_i$ whenever $j < i$) to justify preservation also for $j < i - 1$.

Explaining clauses weaker than the goal

As we have seen, preservation of invariant 1) requires that every derived explaining clause C satisfies $G \Rightarrow C$ or, in other words, C must be weaker than the goal formula G. Here we show that this requirement can always be met, although we may be forced to weaken the explaining clause a bit, which goes against the heuristic of preferring short explaining clauses for their better generalizing power.

Notice that the algorithm never attempts to extend a goal state (because reaching the goal is a reason for termination) and thus we always have $s \models \neg G$ when extending a state s. This means that there is always an explaining clause weaker than the goal, namely the clause obtained by complementing the maximal reason $r = \mathit{Lits}(s)$. Typically, however, much smaller clauses are available.

There is a simple way to satisfy the weaker-than-goal requirement in practice, which can be employed whenever we work with a goal formula G expressed in the form of a set of unit clauses.[9] We can then scan the explaining clause C looking for a literal $l \in C$ such that the unit clause $\{l\}$ is in G. If we find such a literal, the weaker-than-goal

[9]This is assumption holds, for instance, when a logical circuit is encoded into an STS in a standard

requirement is already met: because $G \Rightarrow l$, the presence of l in C ensures that the clause is weaker than G. If we do not find it, there must still be at least one literal l with $\{l\} \in G$ such that $\sim l \notin C$ (otherwise $s \models G$). In this case, we weaken C by adding a single such literal l into it to satisfy the weaker-than-goal requirement.

Our approach for dealing with the situation of a goal formula G in its general clausal form is to reduce it to the above case by translating the given STS $\mathcal{S} = (\Sigma, I, G, T)$. The idea of the translation, inspired by the technique of Gago et al. (2002) for implementing loop search in CTR (recall Section 2.5.2), is to represent the formula G by a single unit goal clause $\overline{g}$ (where $\overline{g}$ is a fresh variable) and add a set of defining clauses into the "universal part" of the STS to express the desired relation between $\overline{g}$ and G. In detail, we define the transformed STS as $\mathcal{S}^* = (\Sigma \cup \{\overline{g}\}, I \cup H, \{\{\overline{g}\}\}, T \cup (H)')$, where

$$H = \{\neg\overline{g} \vee C \mid C \in G\}$$

functions as the definition of the original goal formula G. We insert H both to the new initial formula (to properly constrain a potential witnessing paths of length 0) and put the primed version $(H)'$ to the transition formula (to constrain all the longer paths). It is straightforward to verify that the STS $\mathcal{S}$ and $\mathcal{S}^*$ are equisatisfiable.

5.3.2 Three enhancements

Having modified Reach to use monotone layers, we are still short of three individual enhancements before the algorithm is turned into full-fledged PDR. We first explain obligation rescheduling, by which the algorithm generalizes Reach's strict backtracking scheme of dealing with the partial model path.

Obligation rescheduling

Recall that the computation of Reach can be separated into iterations such that during iteration k the algorithm tries to construct a witnessing path of length k while it has already shown in the previous iterations that there are no witnessing paths of lengths $0, 1, \ldots, k-1$. In PDR, we separate the individual states V_i of the currently constructed path $\mathcal{V} = (V_i)_{0 \leq i < |\mathcal{V}|}$ into so called *obligations*, where an obligation corresponding to the state V_i during iteration k is the ordered pair $(V_i, k-i)$ and $k-i$ can be understood as an estimate on the distance of the state V_i towards the goal. This simple reindexing is an essential step towards the following observation.

When the extension query fails for a state s from an obligation (s, j), we do not need to immediately discard the state (as we do in Reach), but we can retry it later at positions $j+1, j+2, \ldots$, which are further from the goal. This *obligation rescheduling* technique allows the algorithm to discover witnessing paths of a length greater than the current iteration k. Intuitively, this helps to boost the performance of the algorithm in the satisfiable case as it can then avoid completing a potentially computationally expensive

way and then inverted (see Remark 5.1). Then the initial formula I describes the negated invariance property and the goal formula G is a conjunction of unit clauses that initializes the latches. This is the most common encoding setup of circuits in PDR, as it typically yields the best results.

proof that a short path does not exist. Obligation rescheduling incurs no overhead and can be turned off if a witnessing path of guaranteed minimal length is required.

Clause propagation

While obligation rescheduling could be added already to Reach, the second enhancement, *clause propagation*, relies in an essential way on the monotonicity of layers maintained by PDR. Recall that with monotone layers the layer repetition always occurs between two neighboring layers L_i and L_{i+1} for some i. Clause propagation can be understood as a way of bringing about layer repetition sooner by actively copying clauses from L_i to L_{i+1} whenever this preserves invariant 3), which states that L_{i+1} over-approximate the preimage of L_i. In detail, one checks for every $C \in L_i \setminus L_{i+1}$ by a call to the SAT solver whether

$$(L_i)' \wedge T \Rightarrow C.$$

If the implication holds, the clause C can be "pushed forward" and added to L_{i+1}.[10] This makes the two layers "more equal" by reducing the size of $L_i \setminus L_{i+1}$.

We note that making the layers stronger has a positive effect also on the performance on satisfiable problems, because the strengthened layers subsequently provide for a better guidance towards the goal.

Small explaining clauses

When the extension query fails in Reach, the algorithm relies on the SAT solving under assumptions technique (Section 3.2) to obtain an explaining clause and enrich one of the layers. The explaining clause should ideally be as small as possible to generalize the most from the current failure and guide well during future extensions. So far, we have been relying on the SAT solver to provide a small explaining clause by itself. It turns out that one can typically remove further literals afterwards and thus improve performance. Let us first explain a simple version of an explicit minimization technique, which could already be added into Reach, before moving to a more advanced "inductive" version, which relies on monotone layers and is thus PDR-specific.

Recall the general form of a extension query for a state s, transition formula T and a layer L_{i-1}:

$$SAT?[\, Lits(s) \wedge T \wedge (L_{i-1})' \,].$$

In the unsatisfiable case, we obtain a reason cube $r \subseteq Lits(s)$ for the unsuccessful extension and compute the corresponding explaining clause C as a complement of r. The reason r returned by a SAT solver can typically be further reduced, which results in a smaller explaining clause. We can *explicitly minimize* r by trying to remove literals one by one. If the respective query remains unsatisfiable we leave the literal out. Otherwise we put it back. In a number of steps proportional to $|r|$ we obtain a final reason set $r^* \subseteq r$ minimal with respect to subset relation such that the query

$$SAT?[\, r^* \wedge T \wedge (L_{i-1})' \,],$$

[10] Because C is already present in layers $L_0, \ldots, L_i$, adding it to L_{i+1} preserves invariant 2).

is unsatisfiable. The order in which the literals are tried out influences the final result and may be subject to heuristical tuning. Although reason minimization is an expensive operation (we need one extra SAT-solver call per literal), it pays off on average and, as experiments show, it is an important ingredient for solving hard problems.

Inductive reason minimization, a more powerful version of the above, relies on the specific way in which the layers are updated in PDR. Since after a reason r is computed we are in the next step going to strengthen the layers $L_0, \ldots, L_i$ (and, in particular, the layer L_{i-1}) with the explaining clause $C = \sim r$, we may already assume C to hold "on the primed side" when minimizing r. This means, we can use the stronger query

$$SAT?[\, r \wedge T \wedge (L_{i-1} \wedge \sim r)' \,].$$

Having r on both sides of the transition breaks monotonicity of the minimization process: as r gets weaker, $\sim r$ gets stronger. Satisfiable query may become unsatisfiable again when more literals are removed from r. This makes the task of finding subset-minimal "inductive reason" computationally difficult (Bradley and Manna, 2007).

Algorithm 5.4 Inductive reason minimization:

Input:
A set of clauses L and a cube r such that
the formulas $r \wedge T \wedge (L)'$ and $r \wedge G$ are unsatisfiable

Output:
Minimized inductive reason $r^* \subseteq r$, i.e., a cube r^* such that
the formulas $r^* \wedge T \wedge (L \wedge \sim r^*)'$ and $r^* \wedge G$ are unsatisfiable

```
1:  r_0 ← r
2:  loop
3:    foreach l ∈ r do /* Check each literal of r once */
4:      if there is l_0 ∈ (r_0 \ {l}) such that {∼l_0} ∈ G then /* Can try removing l */
5:        r_0 ← (r_0 \ {l})
6:        if SAT?[ r_0 ∧ T ∧ (L ∧ ∼r_0)' ] then
7:          r_0 ← (r_0 ∪ {l}) /* Put the literal back */
8:    if r = r_0 then /* No removal in the last iteration */
9:      return r
10:   r ← r_0
```

In Algorithm 5.4, we present a simple version of inductive reason minimization with no minimality guarantee, which was, however, successfully applied in hardware model checking (Eén et al., 2011). To ensure the the weaker-than-goal requirement for the final explaining clause, the procedure assumes that the goal formula G is in the form of a set of unit clauses. It then keeps $G \wedge r_0$ unsatisfiable, which is equivalent to maintaining $G \Rightarrow C$ for the corresponding explaining clause $C = \sim r_0$. Given the non-monotone flavor of inductive minimization it makes sense to retry all the literals once a single literal has been successfully removed. That is why the procedure employes the outer loop to continue minimizing till a true "fixed point" is reached.

5.3.3 Pseudocode and correctness

Let us now have a look at the pseudocode of PDR (Algorithm 5.5). The algorithm is presented in full, with both obligation rescheduling and clause propagation techniques. However, inductive reason minimization is only implicitly assumed to be called by the pseudocode.

To better understand the connection of PDR to Reach the following should be noted.

- As stated before, we do not explicitly consider dependency tracking in this exposition and thus only work with the sequence of proper layers $L_0, L_1, \ldots$ The proper layers are initialized as in Reach (line 1), but no marking is employed.
- To bring the pseudocode closer to the standard presentation (Eén et al., 2011), *iterations* of the algorithm are driven explicitly by a for-cycle (line 2) and the initial and proper extension queries have been set apart (lines 4 and 11, respectively). This is just an equivalent way of presenting the same overall control flow.
- Each individual iteration consist of a *path construction* phase (lines 4–20) and a *clause propagation* phase (lines 23–30). The former corresponds to the computation within the main loop of Reach, while the latter captures the clause propagation enhancement, which is unique to PDR.
- The decisive difference between Reach and PDR, which causes the layers of the latter to be monotonically ordered by inclusion, happens during the path construction phase when the new explaining clause C is added to all the layers $L_0, \ldots, L_i$ (line 16) and not just to L_i.
- Layer repetition check appears in PDR in the clause propagation phase (line 29) and is thus executed only once per iteration. This does not seem to affect performance either negatively or positively.[11]
- Instead of maintaining the constructed path as an explicit sequence of states (recall variable $\mathcal{V}$ from Reach), a set Q is introduced to store obligations. Had it not been for obligation rescheduling (imagine lines 19 and 20 removed), the content of the container Q would faithfully correspond to the partial path stored in $\mathcal{V}$ by Reach.[12] However, with obligation rescheduling active, PDR works in general on more than one path at once and the set Q functions as a priority queue.

To get a better idea of how obligation rescheduling works, let us have a look at a small example. In Figure 5.2, PDR is in the middle of the path construction phase of iteration 2. The algorithm is attempting to extend the obligation $(t, 1)$ and to reach a goal state

[11] In any case, our implementation of PDR tests for layer repetition also after each addition of an explaining clause. Thus the experimental comparison was not biased by this change.

[12] As already noted, state V_i would correspond to obligation $(V_i, k-i)$ during iteration k. One can verify this by checking that there would always be at most one obligation stored in Q with a particular index j and that the set $\{j \mid \exists(s, j) \in Q\}$ of indexes of obligation stored in Q would at any moment be equal to an interval of indexes $\{k, k-1, \ldots, i\}$ for some index i.

Algorithm 5.5 PDR

Input:
A symbolic transition system $\mathcal{S} = (\Sigma, I, G, T)$
Output:
A witnessing path for $\mathcal{S}$ or a guarantee that no path exists

1: $L_0 \leftarrow G$; **foreach** $j > 0 : L_j \leftarrow \emptyset$ /* Initialize the layers */
2: **for** $k = 0, 1, \ldots$ **do**
3: /* Path construction: */
4: **while** $SAT?[\, I \wedge L_k \,]$ **do**
5: extract state s from the model
6: $Q \leftarrow \{(s, k)\}$
7: **while** Q not empty **do**
8: pop some (s, i) from Q with minimal i
9: **if** $i = 0$ **then**
10: **return** WITNESSING PATH FOUND
11: **if** $SAT?[\, Lits(s) \wedge T \wedge (L_{i-1})' \,]$ **then**
12: extract a successor state t from the model
13: $Q \leftarrow Q \cup \{(s, i), (t, i-1)\}$
14: **else**
15: compute a (small) explaining clause C such that $G \Rightarrow C$
16: **foreach** $0 \leq j \leq i : L_j \leftarrow L_j \cup \{C\}$
17:
18: /* Obligation rescheduling: */
19: **if** $i < k$ **then**
20: $Q \leftarrow Q \cup \{(s, i+1)\}$
21:
22: /* Clause propagation: */
23: **for** $i = 1, \ldots, k+1$ **do**
24: **foreach** $C \in L_{i-1} \setminus L_i$ **do**
25: /* Clause push check */
26: **if** not $SAT?[\, \sim C \wedge T \wedge (L_{i-1})' \,]$ **then**
27: $L_i \leftarrow L_i \cup \{C\}$
28: /* Layer repetition check */
29: **if** $L_{i-1} = L_i$ **then**
30: **return** NO PATH POSSIBLE

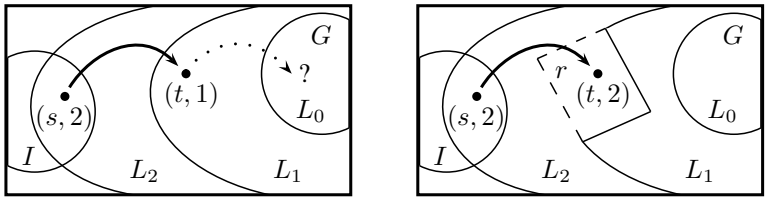


Figure 5.2: Layers, obligations and rescheduling.

in one step (left). When the attempt fails (right), PDR generalizes from t, obtains a reason r, and learns a new explaining clause $C = \sim r$ to strengthen the layers L_1 and L_0. Notice how the obligation is rescheduled to $(t, 2)$ and PDR can now attempt to extend it and satisfy the new L_1 in one step. Without rescheduling, PDR would forget t and would go back to extending $(s, 2)$ instead.

Remark 5.2. When popping obligations from the set Q (line 8) we make sure we select among those estimated closest to the goal. This is necessary for ensuring termination of the algorithm (see below). Otherwise, however, we are free to choose any obligation with the minimal index i. Two prominent strategies for resolving this "don't-care" non-determinism are

- to select the most recently added obligation first, which we call the *stack* strategy,
- to select the least recently added obligation first, the *queue* strategy.

In the above example, continuing to work on $(t, 2)$ corresponds to the stack strategy, while going back to $(s, 2)$ corresponds the queue strategy. Even in the latter case, however, $(t, 2)$ remains stored in Q and will be considered before the end of the iteration 2 (unless a full witnessing path is discovered first).

The stack strategy prefers exploring longer paths before short ones, while the queue strategy does the opposite. Eén et al. (2011) report a small performance gain with the stack strategy on hardware model checking benchmarks. We used the stack strategy as the default in our experiments.

Correctness and termination

Although the correctness and termination result for PDR follows essentially from the same ideas as in Reach, it is worthwhile presenting its proof in full. There is a new complication to be dealt with stemming from the generalized treatment of paths separated into obligations. Another interesting point to focus on is a better theoretical upper bound on PDR's running time enabled by the monotonicity of layers.

In Section 5.3.1, we have stated three main invariants satisfied by the layers in PDR and sketched why they are preserved during algorithm's run. Now we add a simple observation and derive an auxiliary lemma, before stating and proving the main result.

Observation 5.1. *When the path construction phase of iteration k finishes there is no initial state satisfying L_k.*

The observation follows from the fact that the query on line 4 must be unsatisfiable for the path construction to finish.

Lemma 5.3. *When PDR creates (either on line 6 or on line 13) a new obligation (s, i) then $s \models L_i$. Moreover, $s \not\models L_j$ for any $j < i$. This latter property is maintained throughout the run of the algorithm and, in particular, holds also after obligation rescheduling (line 20).*

Proof. First note that it is sufficient to show the second part only for $j = i - 1$ and then use invariant 2). Also note that during the run of PDR clauses are only *added to* and never *removed from* the layers. This means it is sufficient to focus on the moments when a new obligation is created: if $s \not\models L_j$ when the obligation (s, i) is created, this must also hold later, after the layer L_j has been strengthened by addition of new clauses.

Let us now consider iteration k. When creating a new obligation (s, k) on line 6, we have $s \models L_k$ by construction and $s \not\models L_{k-1}$ by Observation 5.1. When creating a new obligation $(t, i-1)$ on line 13, we assume that its parent (s, i) already satisfies our lemma and, in particular, that $s \not\models L_{i-1}$. We now have $t \models L_{i-1}$, again by construction, and if $i > 1$ we infer $t \not\models L_{i-2}$ from our assumption about s and from invariant 3). Finally, an obligation (s, i) is only rescheduled to $(s, i+1)$ after the addition of an explaining clause $C = \sim r$ into L_i for some reason $r \subseteq \mathit{Lits}(s)$. This means that $s \not\models L_i$ at the time of the rescheduling. □

Lemma 5.3 captures the intuition that a state s of an obligation (s, i) is always at least i steps from reaching the goal.

Theorem 5.1 (Bradley, 2011)**.** *Given an STS $\mathcal{S} = (\Sigma, I, G, T)$ the algorithm terminates and returns a witnessing path for $\mathcal{S}$ if and only if $\mathcal{S}$ is satisfiable.*

Proof. It is easy to see that if PDR returns a path (line 10) it is a witnessing path for $\mathcal{S}$.[13] Indeed, for every considered obligation (s, i) the state s is reachable from an initial state and when $i = 0$ the state s satisfies L_0, which is equivalent to G by invariant 1).

If PDR terminates claiming that no witnessing path exists (line 30) the path construction phase of iteration k has finished and there is an index $0 \leq j \leq k$ such that $L_j = L_{j+1}$. By combining invariants 1)–3) and the detected equality we obtain $G \Rightarrow L_j$ and $(L_j)' \wedge T \Rightarrow L_j$. This together with Observation 5.1 rules out the existence of a witnessing path of any length.

To address termination we first show that the path construction phase of iteration k cannot run indefinitely. Recall that PDR always selects for extension an obligation with minimal index i (line 8). Thus it follows from Lemma 5.3 that after a successful

[13] Strictly speaking, line 10 of our pseudocode only reports on the existence of a witnessing path. To be able to recover the path one extends the structure of an obligation to store a pointer to its parent, i.e. to the obligation from which it was derived. A witnessing path can be then read (in reverse order) by following these pointers from the last obligation $(s, 0)$.

extension of obligation (s, i) the new extracted state t is not equal to any other state previously considered during iteration k (t is currently the only state that satisfies L_{i-1}). On the other hand, after an unsuccessful extension of obligation (s, i) the addition of the new clause C to the layer L_i ensures that $s \not\models L_i$ anymore (recall that $C = \sim r$ for some $r \subseteq \mathit{Lits}(s)$). This means there cannot be more than $k \cdot 2^{|\Sigma|}$ repetitions of the "Q-processing" while-loop (line 7) and more than $2^{|\Sigma|}$ repetitions of the outer while-loop of path construction (line 4).

We are left to bound the maximal number of iterations of PDR. By invariant 2) the sets of states represented by the individual layers are ordered by inclusion and after the clause propagation phase of iteration k finishes the first $k+1$ of these sets are necessarily distinct. Thus there cannot be more than $2^{|\Sigma|}$ iterations before PDR terminates. □

5.3.4 PDR – related work

Let us close this section by providing an overview of the work related to PDR. This should mainly serve as a guidepost for anyone interesting in further study of the algorithm.

IC3/PDR invented

Although the official seminal publication on PDR (Bradley, 2011) dates back to 2011, historians will discover that the idea was published already in March 2010 (Bradley, 2010). Bradley's way of arriving to the algorithm was to take the already existing method for learning clauses by inductive generalization from states (Bradley and Manna, 2007) and provide for it a context relative to which the generalizations could not fail. This context took the form of clausal k-step reachability over-approximations, which we decided to call layers in our presentation.

Although the work of Lu et al. (2005) did not serve as an inspiration for the author of PDR (Bradley, 2012), the SAT-based model checking algorithm described there bears several similarities to PDR. One can, for instance, recognize analogues of obligation rescheduling, clause minimization or lifting of states (see below). However, the algorithm by Lu et al. (2005) does not operate with layers and so its search is only guided towards the unexplored part of the search space, but not directly to the goal.

Lifting states

Apart from coining the name "property directed reachability", the main contribution of the paper by Eén et al. (2011) is a method for improving the algorithm's performance by *lifting states*. The idea is to identify cases when it can be efficiently shown that not only the currently discovered state s but all the states satisfying a particular cube $x \subseteq \mathit{Lits}(s)$ are reachable and to use x instead of s within the new obligation (essentially preparing to reason about all those represented states at once). The payoff comes when learning from a failure to extend the obligation, because minimization (already the implicit one within the SAT solver) then starts from x, which is a potentially much smaller set of assumptions than the full $\mathit{Lits}(s)$.

Eén et al. (2011) propose to use ternary simulation for obtaining x. There is also a different method described by Chockler et al. (2011), which requires only one additional SAT-solver call (and that call only performs unit propagation). However, both methods essentially rely on the input being in a circuit form,[14] and thus do not seem to generalize to STS. That is why we did not consider lifting of states in our experiments.

Extensions, generalizations and further improvements

Further followup work on PDR can be classified into three main groups. First, extensions to checking more complex properties such as liveness checking (Bradley et al., 2011) or CTL model checking (Hassan et al., 2012). Second, generalizations beyond the domain of finite state systems, for instance, to handle theories within the context of SMT (Cimatti and Griggio, 2012) or nonlinear fixed-point operators (Hoder and Bjørner, 2012).

Last but not least, research continues on how to improve the performance of the algorithm itself. Hassan et al. (2013) present a new method for enhancing the power of inductive minimization by focusing on and learning from states which hinder the actual minimization attempt (so called counter-examples to generalization). The "SAT Modulo SAT" approach proposed by Bayless et al. (2013), on the other hand, explores the idea of allowing unit propagation to "cross the boundaries" of the individual SAT-solver calls, as if there was an unrolling of the transition relation. This leads to a faster discovery of conflicts (essentially, of unsuccessful extensions) and improves the success rate of PDR both for satisfiable and unsatisfiable inputs.

Our own proposed improvement of the algorithm is the topic of the next section.

5.4 Triggered clause pushing

There are two main reasons for why the clause propagation phase (recall Algorithm 5.5, lines 23–30) is an important part of PDR. First, as pushing clauses reduces the difference between two neighboring layers, it speeds up the convergence of the algorithm towards a successful layer repetition check (on unsatisfiable problems). Second, it generally strengthens the layers, which then provide better guidance for the path construction.[15] Clause propagation is, however, also quite costly, requiring one SAT-solver call per layer clause.

In this section, we describe a new method for speeding up propagation. We notice that during each failed attempt to push a clause the SAT solver computes a model, which is normally thrown away. We propose to keep the model instead and use it as a witness for why the respective clause could not be pushed forward. It only makes sense to recheck a particular clause for pushing when its witnessing model falsifies a newly added clause.

[14] And the search direction being from the goal (more precisely, from the set of bad states, i.e. states violating the given property) towards an initial state.

[15] Clause propagation of iteration k is also an opportunity to insert clauses into the till then empty layer L_{k+1} before the start of iteration $k+1$. Sometimes, thanks to pushed clauses, iterations pass off without actually entering the path construction loop.

Because this test can be implemented via subsumption, we obtain an effective necessary condition which allows us to often skip the expensive SAT-solver call.

After explaining the idea of witnesses in full detail (Section 5.4.1), we describe a scheme in which subsumption is not only used to prune layers (as already proposed by Bradley, 2011), but also to trigger obligation rescheduling and, more importantly, clause pushing (Section 5.4.2). Because utilization of witnesses makes the pushing cheap, it allows us to essentially merge the clause propagation phase with path construction. Thus all clauses can be pushed as far as possible at all times, which makes the guidance more precise and allows for earlier discovery of layer repetitions.

Experimental evaluation of triggered clause pushing is part of Section 5.5.

5.4.1 Witnesses for failed push attempts

Consider a clause $C \in L_i \setminus L_{i+1}$ that could not be pushed forward. This means the query on line 26 Algorithm 5.5 returned SAT. We may now inspect the model computed by the SAT solver and extract a state w_C which satisfies L_{i-1} and to which there is a transition from a state satisfying $\sim C$. Notice that as long as w_C remains to satisfy L_{i-1} during the potential strengthenings of the layer, the query in question cannot become UNSAT. The state w_C, therefore, represents a *witness* for why C cannot be pushed forward from L_{i-1} to L_i.

But how do we efficiently recognize whether w_C still satisfies L_{i-1} after a new clause D has been added to the layer? The answer is: via subsumption! It is only when

$$D \subseteq \sim Lits(w_C)$$

that w_C ceases to be a witness, because it does not satisfy the strengthened L_{i-1}. When this happens we may directly retry the pushing query of line 26 and either discover a new witness or finally push the clause C to L_i.

It may seem expensive to perform the subsumption test against every witness whenever a new clause is derived. Note, however, that efficient implementations of PDR already use subsumption routinely to test each new clause against all other clauses (as detailed below) and that by also considering the witnesses, one per each clause, the overhead is at most doubled. Moreover, in our proposed scheme the subsumption check is only applied within the context of those layers where the new clause itself is known to be sufficiently strong. This typically helps to further reduce this cost.

5.4.2 Implementing triggered clause pushing via subsumption

It has been observed that PDR often derives a clause C to be inserted into layer L_i while L_i already contains a weaker clause $D \supseteq C$. Bradley (2011) proposes to remove such clauses during propagation; the implementation described by Eén et al. (2011) is more eager and clears layers via subsumption each time a new clause is derived. Removing subsumed clauses pays off, because they do not contribute to better guidance and only

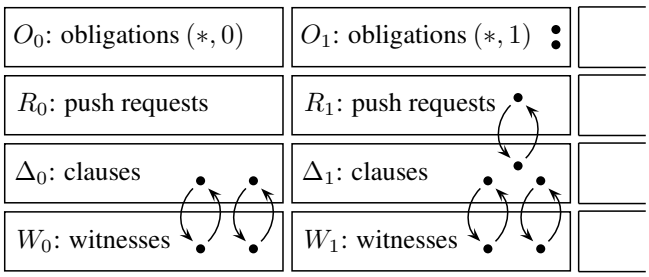


Figure 5.3: Organizing the data structures of PDR with triggered clause pushing. A bi-directional link is maintained between a clause and its witness or its push request.

make the layers unnecessarily large.[16]

Once subsumption is implemented for reducing layers it can also be used for pruning obligations. By construction, an explaining clause C learned from an unsuccessful extension of an obligation (s, i) satisfies $C \subseteq \sim Lits(s)$. But the clause may also subsume other obligations (t, i) currently on Q. These can be directly rescheduled to index $i + 1$, each saving us one SAT-solver call.

Now we describe a new way to organize the data structures of PDR such that 1) subsumption by newly derived clauses can be used to prune layers and obligations, 2) clause pushing triggered by subsuming a witness is integrated into the path construction phase to keep clauses pushed as far as possible at all times. This will require three main updates of the algorithm's data structures.

First, to avoid duplicating clauses we use the delta encoding of layers proposed by Eén et al. (2011). A delta layer Δ_i consist of clauses appearing last in L_i. Thus $\Delta_i = L_i \setminus L_{i+1}$ and $L_i = \bigcup_{j \geq i} \Delta_j$. Next, for each layer clause C, we either associate it with its witness w_C or store a *push request* for it, which means it will need to be considered for pushing. Finally, instead of using the priority queue Q, we explicitly separate obligations into sets O_i based on their index. The whole situation is depicted in Figure 5.3.

The algorithm now works as follows. It picks the smallest index i such that there is either an obligation in O_i or a request in R_i. If both sets are non empty, obligations are picked first.[17] Handling an obligation corresponds to asking the query from line 11 in Algorithm 5.5 and either creates a new obligation or derives a new clause to be added to Δ_i. Similarly, handling a push request corresponds to the query of line 26 and either generates a new witness, which is stored to W_i, or pushes the clause from Δ_i to Δ_{i+1}. In both cases a new clause may be added to a layer, which is where subsumption comes into play.

When a clause C is added into Δ_i we put a push request for it into R_i and then

[16] Although a subsumed clause could potentially be pushed to a higher layer than the subsuming clause and become useful there, implementations prefer to keep the layers small and get rid of the subsumed clause immediately.

[17] Because by learning from an unsuccessful extension of an obligation from O_i we may further strengthen L_i. Then we consider the requests from R_i. If a clause is successfully pushed to L_{i+1} it may subsume obligations waiting in O_{i+1}.

do the following: 1) we remove all the clauses from Δ_i subsumed by C (along with their witnesses or associated push requests), 2) we remove the subsumed witnesses from W_i and insert push requests for the respective clauses into R_i, 3) we reschedule the subsumed obligations from O_i to O_{i+1}. If the clause C was pushed to Δ_i from Δ_{i-1}, we are done. If, on the other hand, C was derived from an unsuccessful extension, it formally strengthened all $L_0, \ldots, L_i$. We therefore continue towards lower indexes performing 1) and 2)[18] for $j = i-1, i-2, \ldots$ A key observation is that the iteration can be stopped as soon as the clause C is itself subsumed by some clause D from Δ_j. Since layers of low index are stronger than those further on, the iteration typically terminates way before reaching $j = 0$. This way a lot of time spent on futile subsumption tests can be saved.

Two notes on related work

Eén et al. (2011) use subsumption to tests whether an obligation has a chance of getting successfully extended, just before attempting the corresponding SAT call (look for method ***isBlocked***). We perform the same check, but already when the potential subsuming clause gets derived. The advantage of our approach is that an obligation is only tested against new clauses and not those that were already in the layers when the obligation was created and for which the test must fail by construction.

To mitigate the quadratic cost (in the number of performed iterations) of clause propagation, Bayless et al. (2013) propose, as one of their improvements of PDR, to only start propagating clauses from the index of the least delta layer which received a new clause during the last path construction phase. As the authors admit, this modification may lead to a loss of PDR's convergence guarantees. Our approach is similar in spirit, but does not have the corresponding drawback. We save time by only reattempting to push clauses which lost their witnesses, but never fail to discover all of them, because we only stop our search when the new clause is itself subsumed.

5.5 Practical part

In this section, we present results of an experimental evaluation of the previously described algorithms for deciding reachability performed on hardware model checking benchmarks. We start from an implementation of Reach as a baseline model checking tool and progressively transform it into PDR, following the individual steps detailed in Section 5.3. Each tool obtained in succession (or a configuration of a tool), is compared to its predecessor to demonstrate the effect of the change on practical performance. The sequence is closed by PDR enhanced with the triggered clause pushing technique.

[18] The sets O_j of obligations are empty for $j < i$ at this point so there is now no work to be done regarding 3).

5.5.1 Experimental setup

Input problems, encoding, and search directions

As input problems we collected and merged the safety (invariance) property benchmarks from the Hardware Model Checking Competition (HWMCC) of years 2007–2012 (Biere et al., 2012). This yielded 1161 problems altogether.

Each input problem describes a hardware circuits together with a corresponding property in the AIGER format (Biere, 2012), which our tools internally translate to STS. We rely on the standard Plaisted-Greenbaum encoding (Plaisted and Greenbaum, 1986) and map the initial state of the given circuit to the initial formula I and the negation of the tested property to the goal formula G of the STS.

This gives rise to a search direction we refer to as *forward*, in which the explicit path is effectively constructed from the initial state of the circuit towards a potential violation of the property. We also tested the tools in the opposite, *backward* direction, achieved by inverting the obtained STS (recall Remark 5.1). Note that our backward direction is the one in which PDR is traditionally presented.

Implementation

The implemented tools share a common front-end consisting of a parsing and encoding module followed by a variable elimination procedure for simplifying the obtained STS. The simplification is closely related to the method described in Chapter 4, but adapted to reachability.

Similarly to LS4, our reachability checking tools rely on Minisat (Eén and Sörensson, 2003a) version 2.2 as the underlying SAT solver. Also here, we allocate as many SAT-solver instances as there are currently considered indexes of the constructed path. This means there are k instances of the solver during iteration k and the i-th instance is assigned to dealing with queries corresponding to layer L_i. Although this approach goes against the initial selling point of PDR that the transition relation need not be "unrolled" (Bradley, 2011), it has the advantage that the low index solvers are not polluted by the weaker[19] clauses derived further on. Other recent implementations of PDR adopt this approach as well (Bayless et al., 2013).

The source code of the tested tools is publicly available (Suda, 2013e).

Testing environment

As with our previous experiments, the computations were run on servers with 3.16 GHz Intel Xeon CPU, 16 GB RAM, running Debian 6.0. In accord with the setup of HWMCC, each tool was given the time of 900 seconds per problem. However, we did not impose any explicit memory limit.

[19] The clauses are weaker in the monotone setup of PDR. In Reach, their logical strength compared to the clauses of higher index layers is in general arbitrary.

5.5.2 Incremental evaluation

Let us first review the performance of the individual tested tools visualized as a function of the number of solved problems within a given time limit. For each tool, we present separate data for the forward (FWD) and backward (BWD) directions. We always provide a summarizing plot and also a decomposition of the data for satisfiable (SAT) and unsatisfiable (UNS) problems. Starting with a plot for Reach, the performance of each subsequent tool is visualized along with the data of its predecessor to highlight the impact of the respective change.

We present the data, in succession, for:

1) the Reach algorithm, as described in Section 5.2.2 (Algorithm 5.1),

2) Reach with monotone layers, denoted Monot, which satisfies the layer invariants of PDR, but is not enhanced in any other way (see Section 5.3.1),

3) Monot implementing obligation rescheduling,

4) Monot implementing obligation rescheduling and clause propagation,

5) the same as above, additionally enhanced by explicit reason minimization,

6) the same as above, but employing inductive reason minimization instead,[20]

7) the above, which is a full-fledged PDR, extended by triggered clause pushing (Section 5.4).

Although some of the above steps are independent from each other (e.g., non-inductive minimization could be tried already as an extension of Reach), others can only be introduced in a specific order (e.g., clause propagation relies on the layers being monotone). We decided to present the experiment as a sequence of extensions mainly to keep the number of the performed tests manageable. This means, however, that in the cases where two extensions could be introduced in a different order (like obligation rescheduling and clause propagation) we have to tacitly assume that their effects are sufficiently independent when drawing general conclusions about the respective techniques per se.

Reach

The performance of Reach can be examined in Figures 5.4 and 5.5 on page 155. To demonstrate the benefit of reusing a universal clause at indexes different from the one where it was derived (recall Section 5.2.2), there are two versions plotted in the backward direction: Reach `u0` corresponds to defining the formula "macro" F_i as in (5.1, page 131), whereas Reach `u1` represents the more refined version of (5.2, page 131), which reuses universal clauses. There is no distinction made in the forward direction, because the transition relation naturally arising from encoding circuits is left-total and so the algorithm can never derive a universal clause on circuits in the forward direction.

[20] For details on enhancements leading to 3)–6) please refer to Section 5.3.2 and Algorithm 5.5.

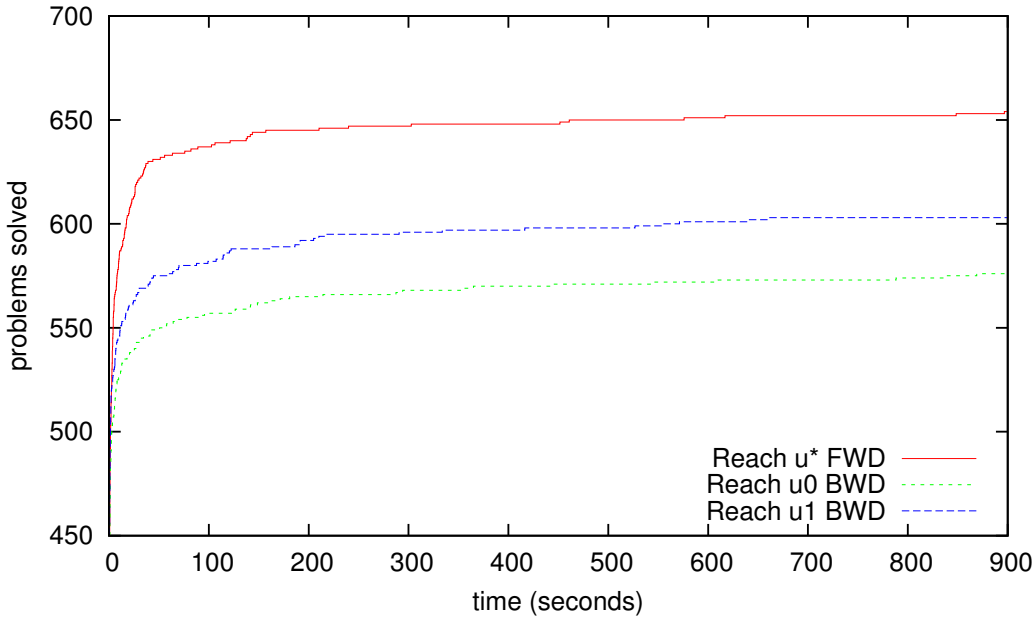


Figure 5.4: Performance of Reach – all instances.

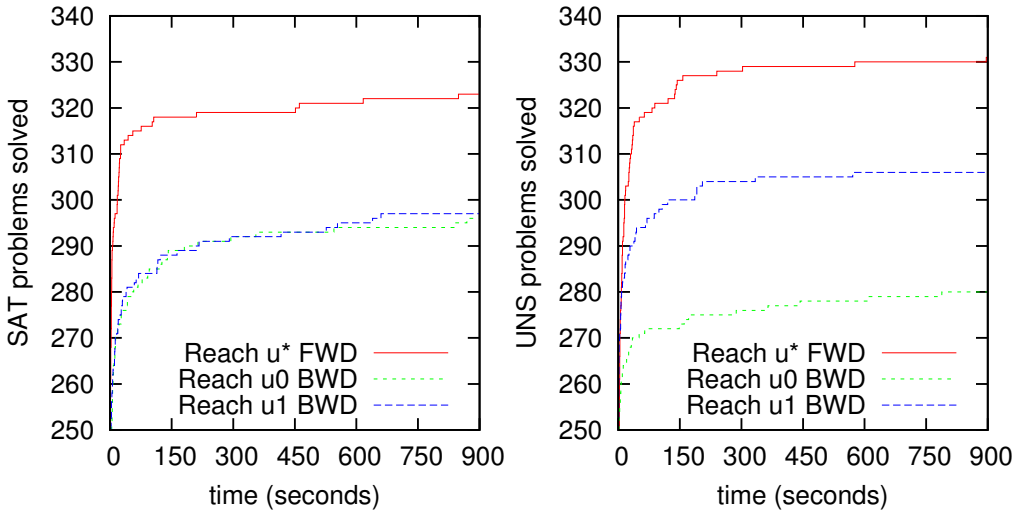


Figure 5.5: Performance of Reach – separately SAT and UNS instances.

We can see that Reach is generally more successful in the forward direction. In the backward direction, reusing universal clauses (`u1`) substantially helps for recognizing unsatisfiable instances, but has no significant effect with satisfiable ones.[21] In the next experiment, we will only show the performance of Reach with reusing universal clauses.

Making the layers monotone

Figures 5.6 and 5.7 on page 157 compare the performance of Reach to Monot, the version of the algorithm with monotone layers. We can see that on the whole, the change makes the performance worse both in the forward direction (where it drops significantly) and in the backward direction (where the deterioration is less severe). With this development, the backward direction becomes preferable to the forward with respect to the total number of problems solved.

The split by satisfiability reveals that the forward direction is mainly impaired on unsatisfiable instances, whereas the the backward direction on the satisfiable ones. The performance in the backward direction even improves on the unsatisfiable instances to approximately match the corresponding performance of Reach in the forward direction. We believe that a theoretical justification of these observations, which currently eludes us, could be helpful in obtaining a deeper understanding of the success of PDR itself.

Adding obligation rescheduling and clause propagation

The effect of adding obligation rescheduling (a switch from `r0` to `r1`) and clause propagation (a switch from `p0` to `p1`) to Monot is visualized in Figures 5.8 and 5.9 on page 158 and Figures 5.10 and 5.11 on page 159, respectively. The global view shows that both techniques are beneficial and improve performance both in the forward and backward direction. We can also notice that clause propagation provides for almost twice as large a gain in the number of additional problems solved than obligation rescheduling. Such an observation, however, depends largely on the used problem set and should not be therefore emphasized too much.

More interesting are, perhaps, the separate views by satisfiability status. They show that obligation rescheduling helps only on satisfiable instances (and more so in the forward direction), whereas on unsatisfiable ones it actually slightly impairs performance. Complementarily, the positive effect of clause propagation is only slight for satisfiable problems (almost negligible in the forward direction), but quite huge for unsatisfiable ones. These observations are in accord with our theoretical understanding of the techniques. Rescheduling helps the algorithm to solve more satisfiable problems, because with the technique the algorithm does not need to focus on providing minimal length witnessing paths. Propagation, on the other hand, mainly speeds up the discovery of repetitions in the unsatisfiable case by reducing the size of the set difference between neighboring layers.

[21] There does not seem to be an obvious explanation for this phenomenon.

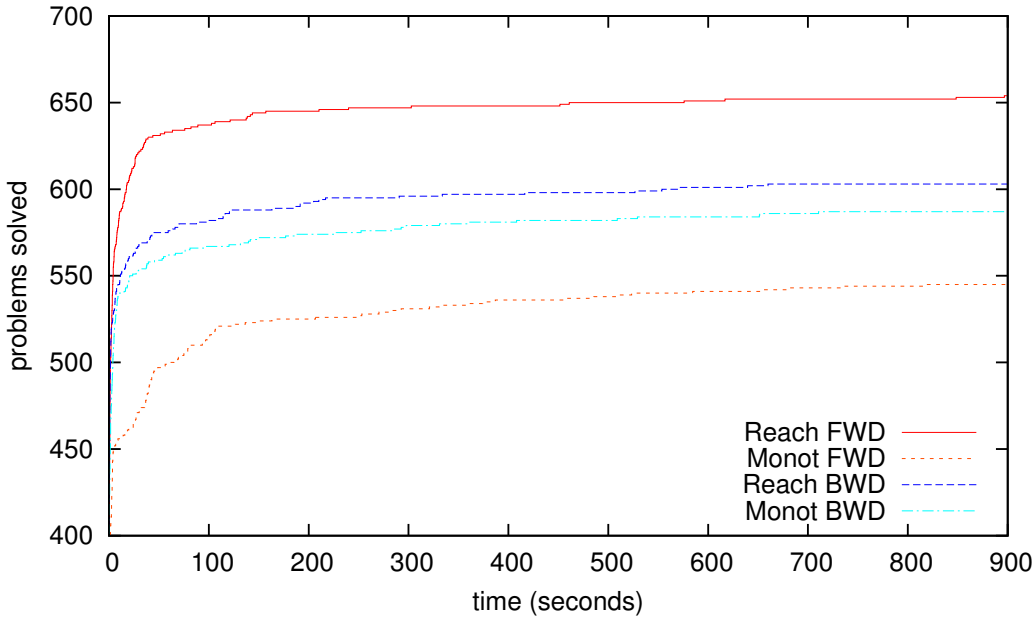


Figure 5.6: Making the layers in Reach monotone – all instances.

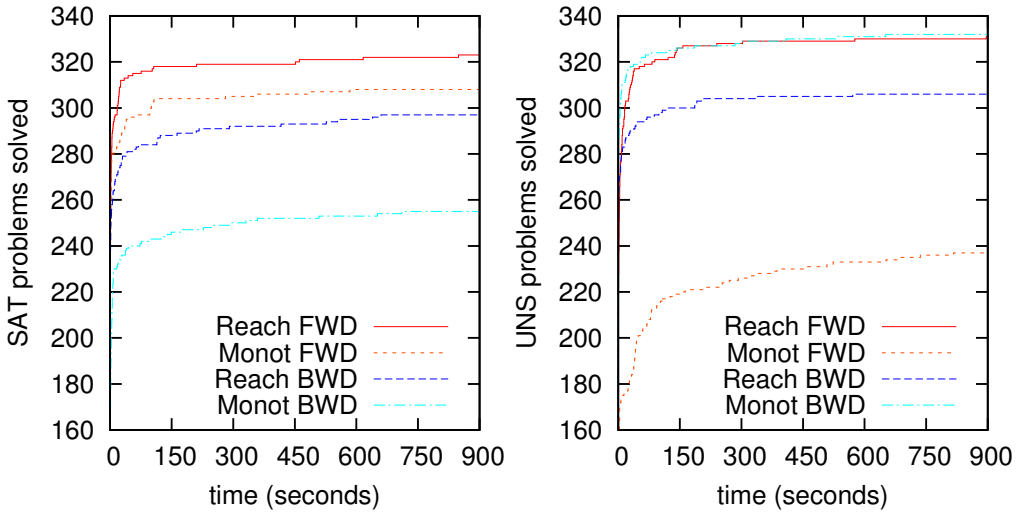


Figure 5.7: Making the layers in Reach monotone – separately SAT and UNS.

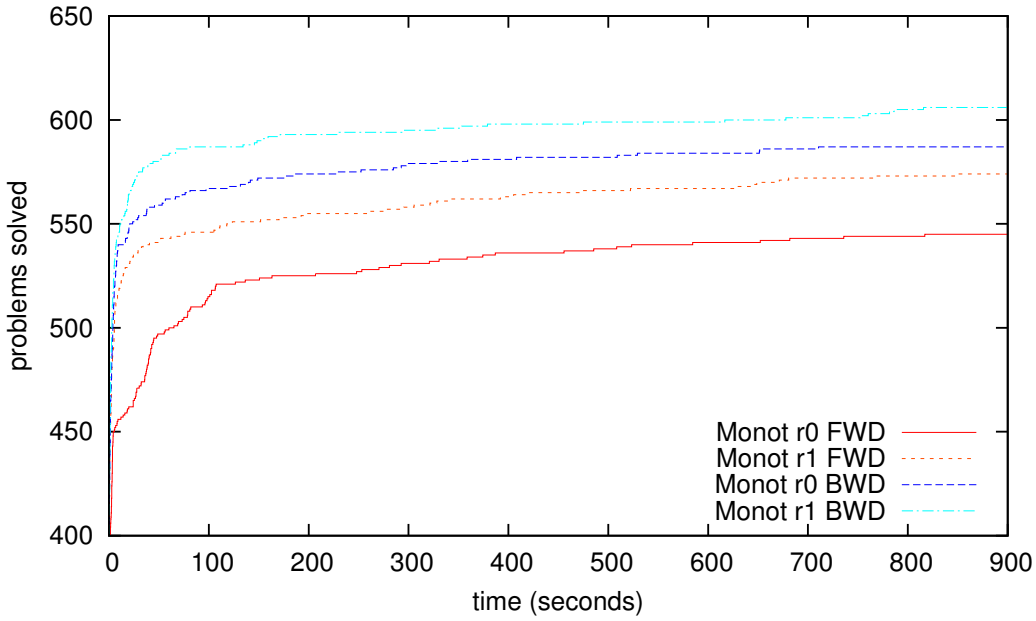


Figure 5.8: Adding obligation rescheduling – all instances.

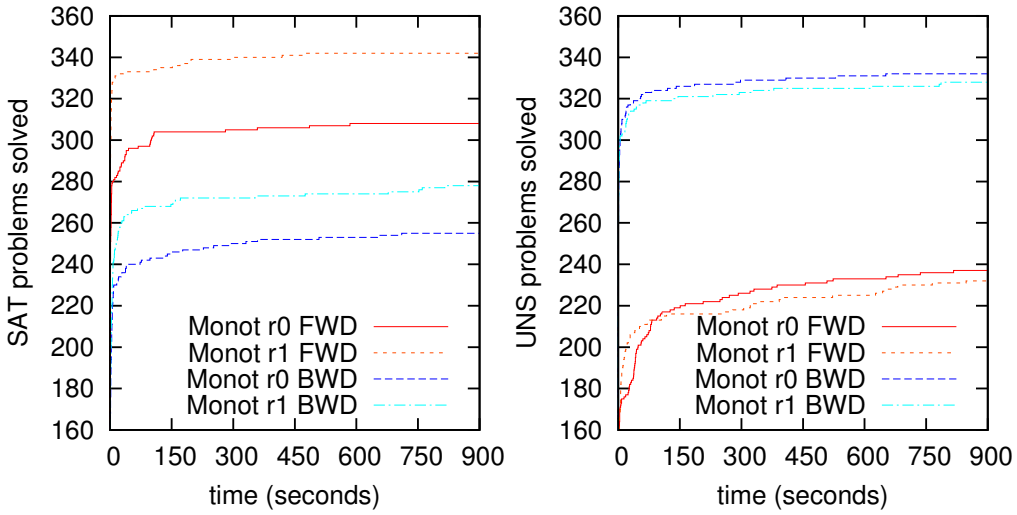


Figure 5.9: Adding obligation rescheduling – separately SAT and UNS.

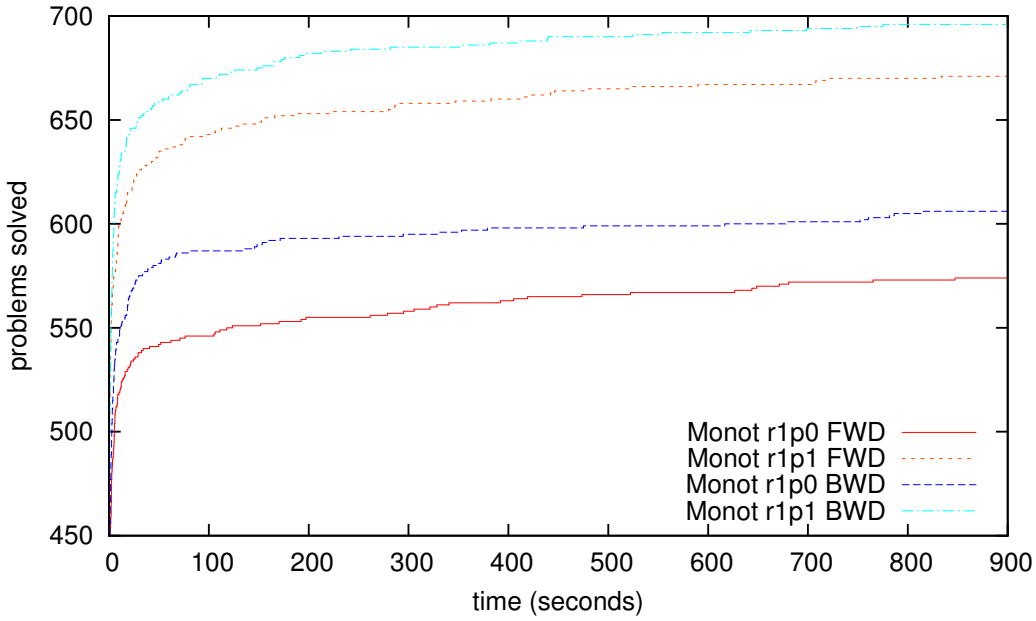


Figure 5.10: Adding clause propagation – all instances.

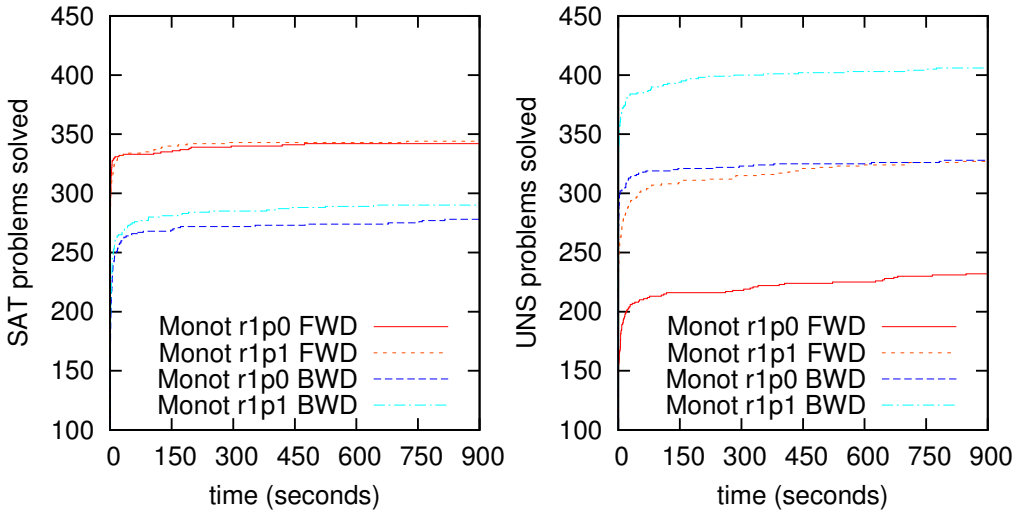


Figure 5.11: Adding clause propagation – separately SAT and UNS.

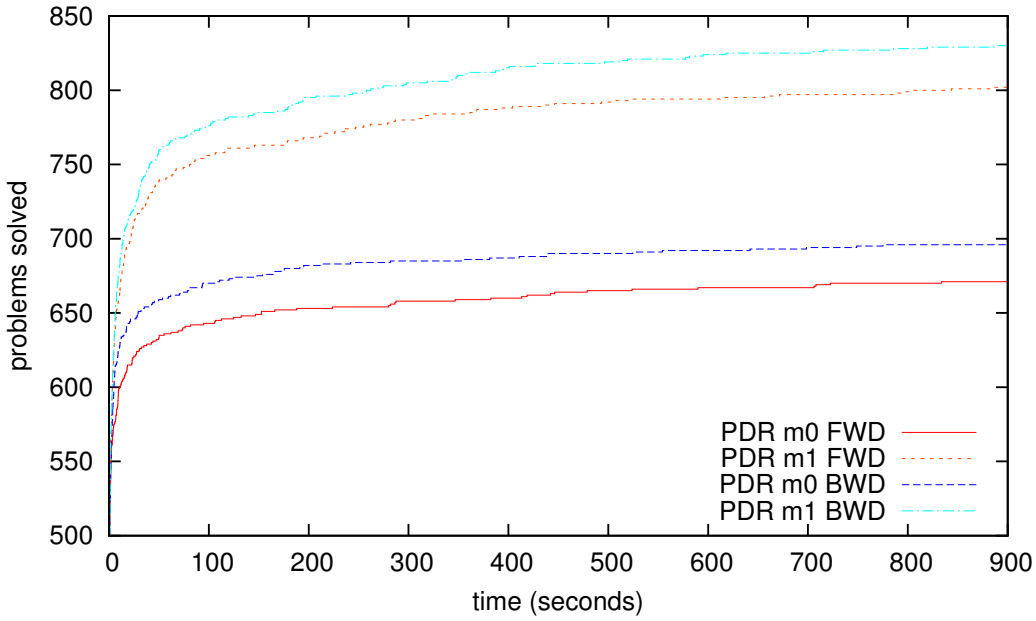


Figure 5.12: Adding explicit minimization – all instances.

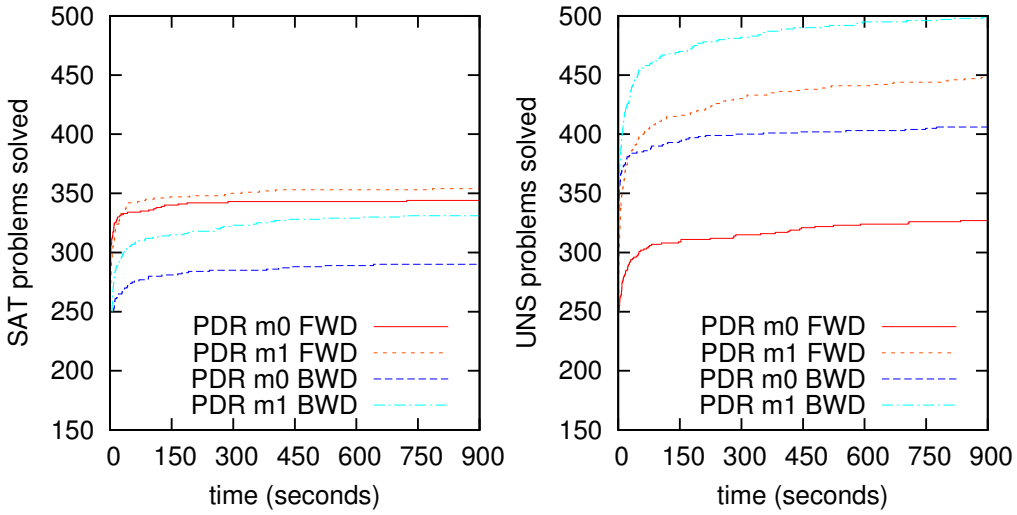


Figure 5.13: Adding explicit minimization – separately SAT and UNS.

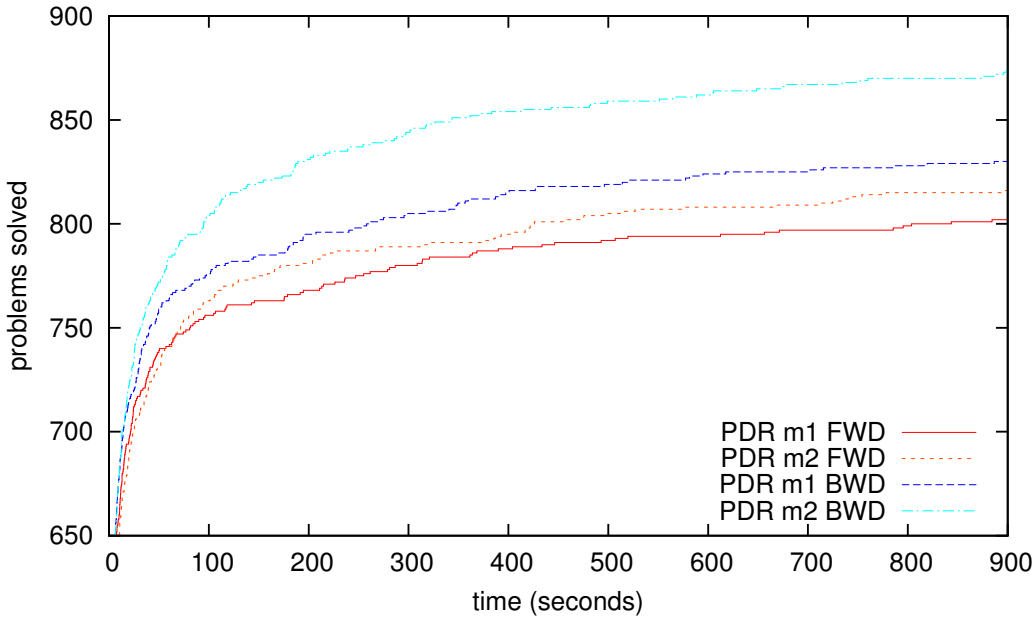


Figure 5.14: Extending to inductive minimization – all instances.

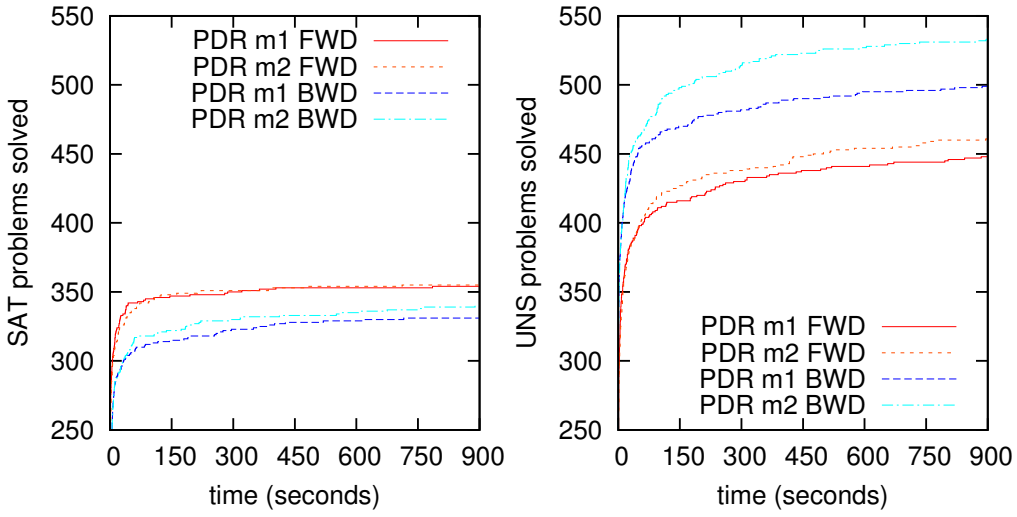


Figure 5.15: Extending to inductive minimization – separately SAT and UNS.

Minimization in PDR

Let us now have a look at pages 160 and 161 (Figures 5.12–5.15), to evaluate the effect of reason minimization. For simplicity, the figures refer to Monot `r1p1`, i.e. the last configuration of the previous comparison, by a shorter name PDR `m0`. Adding explicit minimization is then denoted by a switch to `m1` and a further transition to inductive minimization (in the sense of Algorithm 5.4) by a switch to `m2`.

We can observe that adding explicit minimization has the greatest positive effect on the performance of the algorithm we have seen so far. There are more than 130 additional problems solved within the 900 seconds time limit both in the forward and backward direction. The improvement is by a larger margin coming from the unsatisfiable part of the instances, but is significant also on the satisfiable ones (especially in the backward direction).

On the theoretical basis, we have already argued that minimization helps to generalize the information learned from unsuccessful extensions and that shorter explaining clauses provide for better guidance to the goal. A possible explanation for the improvement on unsatisfiable instances is that by removing literals not truly relevant to the failure of an extension, the learned clauses contain less "noise" and so a repetition check, which in the end relies on precise *syntactic* equality between layers, is likely to occur sooner.

The effect of switching to inductive version of minimization (page 161) is less significant than the introduction of minimization as such, but we can still observe an additional improvement, mostly in the backward direction and on unsatisfiable problems (the improvement in the forward direction on satisfiable instances is, on the other hand, insignificant). This confirms that the importance of deriving short clauses is so large that it pays off to invest a non-trivial amount of time to it. Recall that without minimization the algorithm is spending one SAT-solver call per extension. With explicit minimization up to $|\Sigma|$-many calls may be needed and potentially even more with the inductive version. Yet the last version is the most successful on our benchmark set.

Triggered clause pushing

The last step in our sequence of extensions is the addition of triggered clause pushing to PDR, visualized in Figures 5.16 and 5.17 on page 163. Standard PDR (PDR `m2` from the previous comparison) is here denoted PDRmi to stress that it is the version of the algorithm with inductive minimization. Switching an extra specifier from `t0` to `t1` signifies the addition of the new technique.

We can see from the figures that although triggered clause pushing slightly helps in the forward direction, it actually impairs performance in the backward direction. The improvement in the forward direction happens on the unsatisfiable instances, where the performance in the backward direction stagnates. The deterioration in the backward direction stems, on the other hand, from the satisfiable part, on which, symmetrically, the forward direction does not seem affected.

These observations suggest that the potential benefits of triggered clause pushing in the backward direction are in general outweighed by the computational cost connected

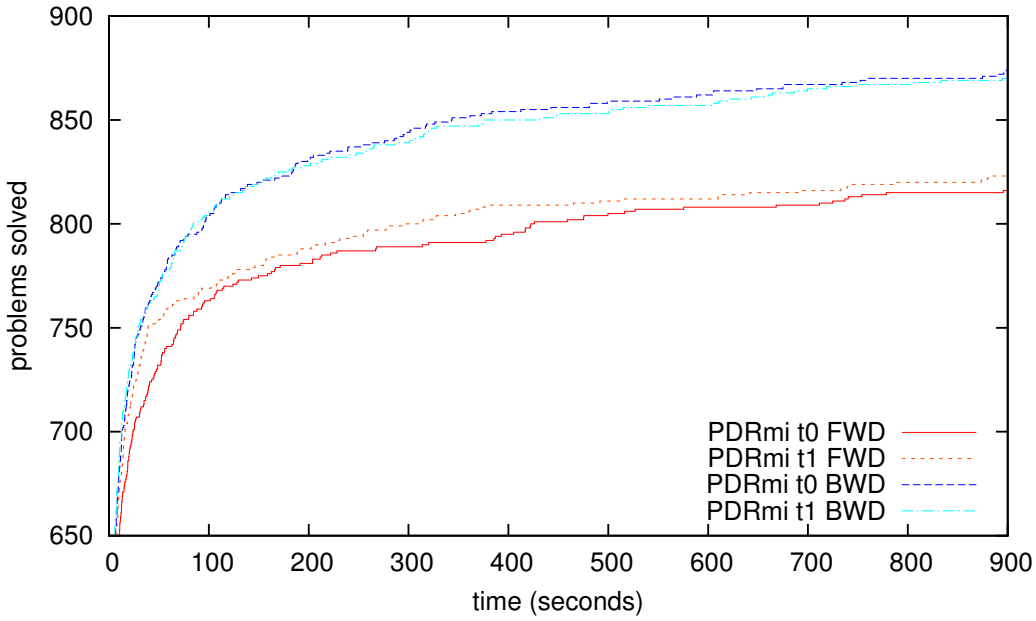


Figure 5.16: Enhancing with triggered clause pushing – all instances.

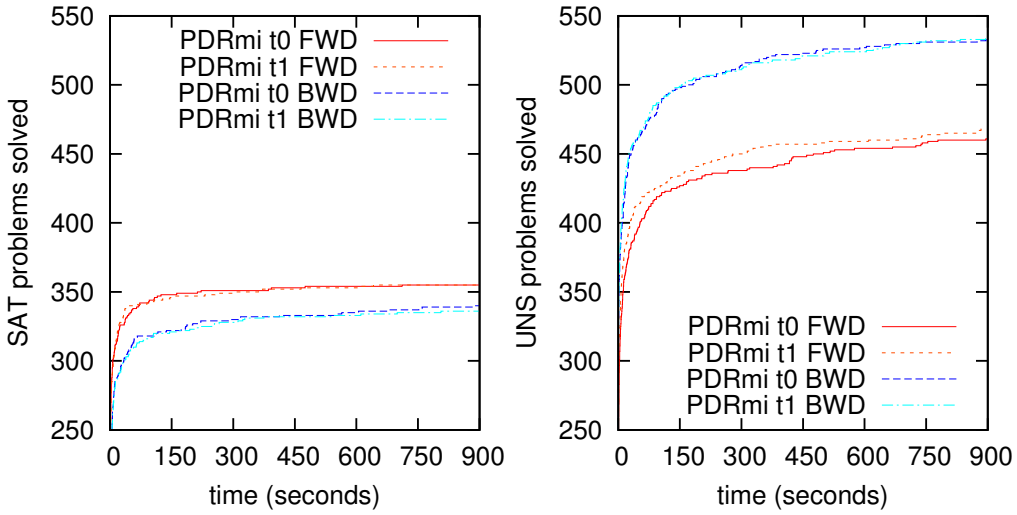


Figure 5.17: Enhancing with triggered clause pushing – separately SAT and UNS.

with maintaining witnesses.[22] Because the deterioration of performance is not dramatic, however, the technique could still be useful in contexts where propagation dominates in the solution times. We explore one such context below.

Interlude: triggered clause pushing for multi-property checking

Because triggered clause pushing did not turn out to bring the expected benefit on standard single-property benchmarks, we decided to test the technique for solving multi-property problems using shared clause layers. A multi-property benchmark contains a description of a circuit together with a collection of properties to be decided about the circuit. If we pick a single property and solve it with PDR in the backward direction, the generated layer clauses depend only on the initial state and the transition relation of the circuit and can be reused when attempting to solve the next property in the collection. Sharing the layers typically helps but can hurt, for a particular property, if there is too much irrelevant information. In particular, propagation times dominate (Bradley, 2014b). Triggered clause pushing should be able to address this problem, because it allows the algorithm to avoid lots of futile push attempts.

We implemented a simple extension of PDR for solving multi-property problems using shared clause layers. In order to avoid spending too much time on a particular difficult property from a collection, our tool switches between all the (yet unsolved) properties in a simple round robin fashion. In each new round, an attempt to solve a particular property is given twice as much resources to continue solving as in the previous round, and the attempt is interrupted when it runs out of the resources. The resource controlled is the number of unsuccessful extensions. When we interrupt an attempt, we remember the index of the last completed iteration to restart from there in the next round, but discard all the generated obligations, which, unlike the layers, cannot be shared. We implemented two versions of this tool, one based on standard PDR and the other based on PDR enhanced with triggered clause pushing.

We tested the tool on the 76 multi-property benchmarks from HWMCC 2012 (Biere et al., 2012). These problems contain 81082 properties in total, which gives us an average of 1066 properties per problem, but the actual median is only 30. When giving the tool 900 second per problem, the version based on standard PDR was able to decide 6286 properties in total, while the version enhanced with triggered clause pushing decided 7981 properties. This constitutes an improvement by almost 27 percent.

5.5.3 Tabular view and the preferable search direction

We complement our incremental evaluation of the transformation from Reach to PDR (and beyond) by presenting a summarizing view on our experimental data in a tabular form. Tables 5.1 and 5.2 show the number of problems solved by each of the tested tools

[22] In a previous version of our experiments, in which our implementation of PDR did not incorporate inductive minimization, a positive effect of triggered clause pushing occurred even in the backward direction on the same set of benchmarks. This indicates that the individual extensions tested in this section are not fully independent.

Table 5.1: Incremental evaluation summary – forward direction.

tool	total	delta	gained	lost
Reach	654 (323, 331)			
Monot	545 (308, 237)	-109 (-15, -94)	13 (2, 11)	122 (17, 105)
Monot `r1`	574 (342, 232)	29 (34, -5)	41 (36, 5)	12 (2, 10)
Monot `r1p1`	671 (344, 327)	97 (2, 95)	98 (3, 95)	1 (1, 0)
PDR `m1`	802 (354, 448)	131 (10, 121)	146 (17, 129)	15 (7, 8)
PDR `m2`	816 (355, 461)	14 (1, 13)	26 (3, 23)	12 (2, 10)
PDRmi `t1`	823 (355, 468)	7 (0, 7)	17 (1, 16)	10 (1, 9)

Table 5.2: Incremental evaluation summary – backward direction.

tool	total	delta	gained	lost
Reach	576 (296, 280)			
Reach `u1`	603 (297, 306)	27 (1, 26)	29 (2, 27)	2 (1, 1)
Monot	587 (255, 332)	-16 (-42, 26)	35 (2, 33)	51 (44, 7)
Monot `r1`	606 (278, 328)	19 (23, -4)	33 (29, 4)	14 (6, 8)
Monot `r1p1`	696 (290, 406)	90 (12, 78)	93 (14, 79)	3 (2, 1)
PDR `m1`	830 (331, 499)	134 (41, 93)	143 (43, 100)	9 (2, 7)
PDR `m2`	874 (340, 534)	44 (9, 35)	49 (11, 38)	5 (2, 3)
PDRmi `t1`	870 (336, 534)	-4 (-4, 0)	9 (1, 8)	13 (5, 8)

within 900 seconds in the forward direction and in the backward direction, respectively. The tables show the overall performance of each tool (total) and the difference between the performance of two successive tools (delta). The difference is also decomposed into additionally solved problems (gained) and problems only solved by the previous tool in the sequence (lost). Moreover, each mentioned field has its value additionally separated into one for satisfiable and one for unsatisfiable instances (in brackets, in this order).

An interesting observation following from the tables is that none of the presented improvements is unambiguous across the whole problem set: with the single exception of Monot `r1p1` in the *forward* direction, which does not lose a single *unsatisfiable* problem with respect to its predecessor Monot `r1`, all the remaining "lost" fields are invariably non-zero. This may teach us that the concept of hard and easy problems is quite elusive and that, in general, there are only hard and easy problems with respect to a given tool.

By comparing the corresponding fields between the two tables, we can also establish which of the search directions is more successful for each tool. We observe that while in Reach the preferable search direction was forward, with the transition to Monot the backward direction became more successful and remained so the whole way to the full-fledged PDR. However, when focusing only on satisfiable problems the forward direction consistently dominates the backward direction throughout the whole development of the algorithm.

So which of the search directions should be preferred in general? The answer is that search in both direction should be combined in order to solve the most problems. For instance, PDR `m2` solves 816 problems in the forward direction and 874 in the backward direction in 900 seconds. Obviously, the number of problems solved in at least one direction in 900 seconds is higher, namely 931 problems.[23] But even with the time limit reduced to 450 seconds, to simulate a setup when the two searches would be run in sequence, the number of problems solved in at least one direction is still 915, i.e. more than in either of the directions alone even with the full 900 second time limit. This observation is in accord with the prevailing trend in the design of modern model checkers – the most successful tools employ portfolios of several algorithms and multiple variations of a single algorithm (Biere et al., 2012).

5.5.4 Comparison with other publicly available implementations

We close the practical part of this chapter by a final set of plots in which we compare our tool to three other publicly available implementations of PDR. This should serve to establish the quality of our tool, but also to provide an estimate of the potential of the more recent improvements of PDR suggested by related work, as mentioned in Section 5.3.4.

In particular, we compare our tool in the configuration PDR `m2` to the following implementations:

- Bradley's first implementation, which participated in HWMCC 2010. The tool, denoted here IC3 2010, is described in the original paper on PDR (Bradley, 2011)

[23] This number was obtained separately and cannot be deduced from the tables.

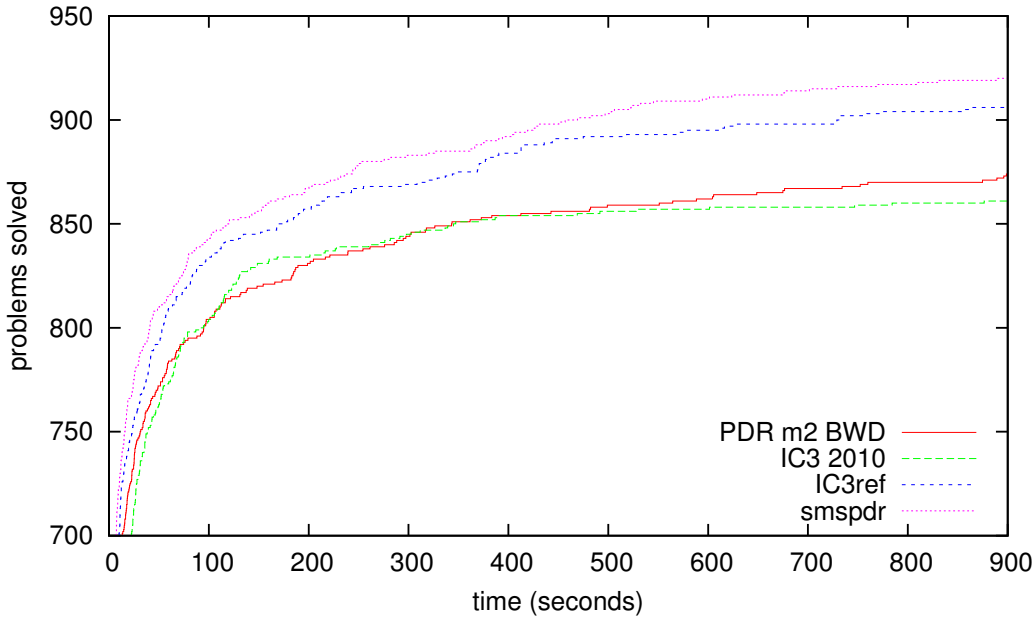


Figure 5.18: Comparing with other implementations – all instances.

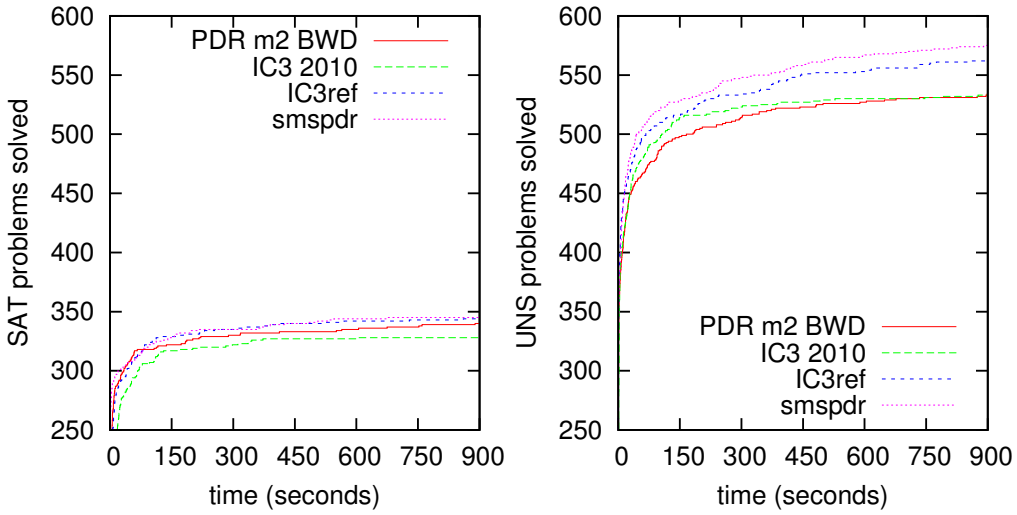


Figure 5.19: Comparing with other implementations – separately SAT and UNS.

and is available from the author's web-page (Bradley, 2013).

- A more recent implementation from the same author, suggested as a reference meant to be used as a starting point for tuning and experimenting (Bradley, 2014a).

  We denote the tool IC3ref. It implements lifting of states using the approach proposed by Chockler et al. (2011) and exploits counter-examples to generalization (Hassan et al., 2012).

- Finally, an implementation accompanying the paper by (Bayless et al., 2013) on the "SAT Modulo SAT" idea. The tool is called smspdr (Bayless, 2013) and additionally implements lifting of states via ternary simulation (Eén et al., 2011).

None of the mentioned tools implement circuit specific preprocessing of the input, which makes our comparison reasonably fair.

The results of the comparison are shown in Figures 5.18 and 5.19 on page 167. Note that unlike PDR `m2`, all the other tools support only one search direction, which corresponds to BWD, the direction shown for our tool. This limitation cannot be easily overcome with IC3ref and smspdr, because the states lifting technique they implement is circuit-specific and only works in this direction.

We can see that the performance of PDR `m2` is comparable to that of IC3 2010, or even slightly better on the SAT instances. However, our tool lags behind IC3ref and smspdr. This is most likely due to the extra techniques mentioned above, which these tools introduce on top of standard version of the algorithm.

5.6 Conclusion

In this chapter, we studied the reachability problem in transition systems that have a symbolical representation based on propositional logic. First, we defined an STS, which serves as a canonical description of such a problem. Then we have shown how to adapt LS4, our algorithm for LTL satisfiability from Chapter 3, to decide reachability for an STS. Algorithm Reach thus obtained is very similar to the famous PDR algorithm (Bradley, 2011), also known as IC3. In fact, we have shown that Reach can be turned into PDR by one simple change in the interpretation of layers, which renders the layers monotone with respect to set inclusion, and three independent enhancements: obligation rescheduling, clause propagation and minimization. This provides a new perspective on PDR, relating it to the model guidance idea. Furthermore, we have proposed triggered clause pushing, an additional improvement of the clause propagation phase of PDR.

An important class of problems that can be formulated as reachability in STS arises in sequential verification of hardware circuits. We used the safety property benchmarks from the Hardware Model Checking Competition (Biere et al., 2012) to experimentally evaluate: 1) our algorithm Reach, 2) the effect of the transformation steps that lead from Reach to PDR, and also 3) triggered clause pushing added on top of PDR. Our experiment has shown that although the step of making layers monotone in Reach slightly impairs performance of the algorithm, the three subsequent enhancements, some of which

essentially rely on monotone layers, make up for the loss and, moreover, improve the performance further. Our own proposed improvement, triggered clause pushing, did not help the algorithm to solve more problems in the standard single-property setting, but substantially improved performance of a PDR-based multi-property solver in which clause layers are shared and faster propagation becomes of higher importance for the overall success rate.

Future work

After having isolated the individual steps needed to transform Reach into PDR and having established their respective positive effect on practical performance, we can ask to what extent could these ideas be adapted to the context of LTL-satisfiability checking and used to improve the performance of LS4. While the adaptation seems straightforward for explicit minimization and obligation rescheduling, clause propagation and inductive minimization substantially rely on monotone layers and as such cannot be directly translated to LS4. It is an interesting research question whether the model construction in LS4 could be guided by, at least partially, monotone layers and whether the algorithm could thus benefit from all the discussed improvements.

Another interesting direction for future work is to try using PDR with triggered clause pushing inside the k-Liveness algorithm by Claessen and Sörensson (2012). k-Liveness relies on sharing layers between its individual reachability queries and could thus benefit from faster clause propagation provided by triggered clause pushing, similarly to our extension of PDR for solving multi-property problems.

6 Property directed reachability in automated planning

6.1 Introduction

The basic task in automated planning is the following. Given an initial state, a goal condition on states, and a set of available actions, which can be used to turn one state into another, decide whether there is a sequence of actions, simply called a plan, which consecutively turns the initial state to a state satisfying the goal condition. It is well known that this task ultimately leads to the reachability problem in a (succinctly represented) transition system. One may therefore ask to what extent one could adapt and successfully apply to this task PDR (Bradley, 2011; Eén et al., 2011), the new and promising algorithm discovered in the related field of model checking (see Chapter 5). The current chapter sets out to answer this question.

A well established approach to automated planning is called *planning as satisfiability* (Kautz and Selman, 1992). It is based on the idea to formulate and encode the existence of a plan of a particular fixed length as a propositional formula and then iteratively check formulas corresponding to increasing plan lengths for satisfiability using a SAT solver. This is essentially the same idea as in Bounded Model Checking (recall Section 5.2.3).

A lot of research in planning as satisfiability focuses on devising and improving encoding schemes, i.e. the concrete translations from a planning problem to the formulas (see, e.g., Kautz et al., 1996; Rintanen et al., 2006; Huang et al., 2012). Similarly to BMC, these formulas tend to have a specific structure consisting of parts describing, respectively, the initial state, the goal condition, and the appropriate number of steps through the transition relation. By observing that these parts naturally correspond to the initial, goal and transition formulas of an STS, we can already give an affirmative preliminary answer to our question: one can solve a planning problem by using an encoding scheme to translate it to an STS and then applying PDR to it.

The main contribution of this chapter, however, is a discovery of a less direct but arguably a more rewarding approach. We show that for a specific planning formalism called the STRIPS planning one can replace the SAT solver inside PDR and implement both the extension query and (under some mild assumptions) also the clause pushing query by a planning-specific procedure. Although this does not affect the complexity of the overall algorithm, it is surprising to learn that the procedure itself runs in polynomial time. We show that the procedure is correct by relating it to a certain simple encoding. This suggests that the same complexity guarantee could be obtained for a SAT solver running on the same encoding and with a specific way of driving its search. However, the mere fact that the encoding would need to be explicitly generated would most likely

make a difference in practice. As another reward for devising the procedure we gain additional insights and ideas for further improvements that go beyond what could be achieved with the SAT-based version.

Relating PDR to the main planning approaches

How does PDR relate to the traditional approaches studied in automated planning? As already mentioned, planning as satisfiability closely resembles BMC and thus the connection between BMC and PDR (via Reach), which we have studied previously, manifests itself here in an analogous way.

What should perhaps be stressed is that in the standard planning as satisfiability approach the algorithm always discovers a plan of minimal length (with respect to the number of steps through the encoded transition relation), because the formulas corresponding to increasing plan lengths are tried one by one until the first satisfiable formula is found. In scenarios where such optimality is not required (so called satisficing planning), PDR could have an advantage thanks to its obligation rescheduling technique, which allows the algorithm to discover a plan of a certain length while a potentially expensive proof that no shorter plan exists is still waiting to be completed. A similar effect can be achieved in the planning as satisfiability approach by running several SAT solvers in parallel or interleaved (Rintanen, 2004). Such a modification, however, requires a non-trivial engineering effort and the resulting system contains parameters that need to be tuned for the problem at hand. In contrast, rescheduling in PDR can be realized literally just by adding one line to the code the algorithm. When this line is disabled, PDR resorts to the more expensive search for an optimal length plan.

Another distinguishing feature of PDR in comparison to planning as satisfiability is the inherent ability of the algorithm to detect unsatisfiable inputs, i.e. to show that no plan of any length can exists. We return to the topic of detecting unsatisfiable planning problems in Section 6.4.8.

PDR can also be naturally compared to *heuristic search planning* (Bonet and Geffner, 2001), the currently prevailing and most studied approach to automated planning. Also in heuristic search planning, the algorithm operates with a set of explicit states – typically those reachable from the initial state. The idea is to use a heuristic to estimate the distance of each state to the goal and to always select the state with the lowest estimate, i.e. the one which currently appears to be the closest to the goal, for the next expansion. One can see that a similar role is played in PDR by the layers. It follows from the invariants of the algorithm that when a state s does not satisfy the clauses of a layer L_k it must lie at least k steps from the goal. Thus the layers implicitly represent an *admissible* heuristic, a heuristic which never overestimates the true distance to the goal. But while a heuristic value of a particular state is normally computed only once and it remains constant during the search for a plan, the layers in PDR are refined continually. The refinement happens on demand, driven by the states encountered during the search.

Chapter overview

We begin our exposition in Section 6.2 by introducing the STRIPS planning formalism and showing on an example of two simple encodings how to translate a STRIPS planning problem into STS. This is all one needs to know to start planning with the standard PDR.

The planning specific version of PDR, which does not rely on a SAT solver, is developed in Section 6.3. After explaining how to deal with both the extension query (including inductive minimization) and the clause pushing query, we discuss the possibility of inverting the search direction in STRIPS planning. Last but not least, we propose several improvements of PDR, some of which are only possible in the SAT-solver-free perspective.

We implemented the proposed idea in a new planner PDRplan. In Section 6.4 we experimentally confirm that it is more efficient than the standard PDR combined with encodings. We also evaluate the practical impact of various improvements and compare the most successful configuration of PDRplan to the state-of-the-art planners with encouraging results.

Section 6.5 returns to related work and uncovers a perhaps surprising connection between PDR and the Graphplan algorithm of Blum and Furst (1997). Finally, Section 6.6 uses examples of the behavior of PDR on two classical planning domains to discuss possibilities for future extensions of the algorithm and Section 6.7 concludes.

The material of the chapter has been published in (Suda, 2014a).

6.2 The planning problem and encodings

6.2.1 Propositional STRIPS planning

Our focus in this chapter will be on planning problems described in the STRIPS planning formalism. Similarly to states of symbolic transition systems, states in STRIPS planning are identified with propositional valuations. The propositional variables encoding the state are in this context called *state variables* and we denote their set by X.

An action a is determined by a tuple $a = (\mathit{pre}_a, \mathit{eff}_a)$, where pre_a, called the *precondition set*, and eff_a, the *effect set*, are cubes over X, i.e. consistent conjunctive sets of literals. An action a is *applicable* in a state s if $s \models \mathit{pre}_a$. If this is the case then applying the action a in s results in a *successor* state $t = \mathit{apply}(s, a)$, which is the unique state that satisfies eff_a and for every $p \in X$ not occurring in eff_a it has $t(p) = s(p)$. A degenerate action with empty precondition and effect sets is called the *noop* action. It is applicable in any state s and the corresponding successor is identical to the original state: $\mathit{apply}(s, \mathit{noop}) = s$.

A *STRIPS planning problem* is a tuple $\mathcal{P} = (X, s_I, g, \mathcal{A})$, where X is the set of state variables, s_I the initial state, g the goal condition in the form of a cube over X, and $\mathcal{A}$ a set of actions. A *plan* for $\mathcal{P}$ is a finite sequence $a_1, \ldots, a_k$ of actions from $\mathcal{A}$ such that there are states $s_0, \ldots, s_k$ satisfying the following conditions:

- $s_0 = s_I$,

- a_j is applicable in s_{j-1} for $j = 1, \ldots, k$,
- $s_j = \mathit{apply}(s_{j-1}, a_j)$ for $j = 1, \ldots, k$,
- and $s_k \models g$.

Notice that the empty sequence λ is a plan for $\mathcal{P}$ if and only if $s_I \models g$.

6.2.2 Two simple encodings

The various encoding schemes proposed in the planning literature can be characterized based on how many actions they allow to be applied in one step. If an encoding scheme uses a so called *sequential plan* semantics, one step in the transition relation corresponds to exactly one action application. *Parallel plan* semantics allow multiple actions to be applied in one step. This leads to a more compact representation and potentially faster discovery of plans. Additional conditions on the parallel actions need to be imposed, however, to guarantee that a true sequential plan can be recovered in the end (see Rintanen et al., 2006, for more details).

Here we present two encodings of a STRIPS planning problem $\mathcal{P} = (X, s_I, g, \mathcal{A})$ into an STS. They are perhaps the simplest representatives of encoding schemes with the sequential and parallel plan semantics, respectively. We will later refer to them in our theoretical considerations and in the experiments.

The symbolic transition systems $\mathcal{S}^{seq}_{\mathcal{P}}$ and $\mathcal{S}^{par}_{\mathcal{P}}$ corresponding to the two encodings share several building blocks. Let the signature Σ consist of the state variables X in union with a set of fresh auxiliary variables $A = \{p_a \mid a \in \mathcal{A}\}$ used for encoding applied actions. Further, let us identify the initial formula I with the cube $\mathit{Lits}(s_I)$ and define the goal formula G by reinterpreting the goal condition g, which is formally a cube, as a set of unit clauses $G = \{\{l\} \mid l \in g\}$. The action mechanics is in both encodings captured by the following *action precondition* axioms AP and *action effect* axioms AE:

$$AP = \{\neg p_a \vee l \mid a \in \mathcal{A},\ l \in \mathit{pre}_a\}, \qquad AE = \{\neg p_a \vee l' \mid a \in \mathcal{A},\ l \in \mathit{eff}_a\}.$$

The encodings differ in how they formalize the "preserving" part of actions' semantics.

The *sequential encoding* $\mathcal{S}^{seq}_{\mathcal{P}}$ relies on the so called *classical frame* axioms CF (McCarthy and Hayes, 1969) complemented by the single *at-least-one* axiom $\mathit{alo} = \bigvee_{a \in \mathcal{A}} p_a$:

$$CF = \{\neg p_a \vee l \vee {\sim}l' \mid a \in \mathcal{A},\ l \text{ literal over } X \text{ such that } l \notin \mathit{eff}_a \text{ and } {\sim}l \notin \mathit{eff}_a\}.$$

Putting these together, we obtain $\mathcal{S}^{seq}_{\mathcal{P}} = (\Sigma, I, G, T^{seq})$, where $T^{seq} = AP \wedge AE \wedge CF \wedge \mathit{alo}$. Note that the at-least-one axiom is needed, because without it a transition into an arbitrary state would be possible from a state where no action is applied, i.e. a state in which p_a is false for every $a \in \mathcal{A}$. On the other hand, the classical frame axioms ensure that if two actions are applied together in a state their effects must be identical. Thus when extracting a (sequential) plan from a witnessing path for $\mathcal{S}^{seq}_{\mathcal{P}}$ we can arbitrarily choose in each step any action $a \in \mathcal{A}$ such that p_a is true in the corresponding state.

The *parallel encoding* $\mathcal{S}^{par}_{\mathcal{P}}$ uses the following *explanatory frame* axioms *EF* (Haas, 1987)

$$EF = \{l \vee {\sim}l' \vee \bigvee_{a \in \mathcal{A}\ l \in \mathit{eff}_a} p_a \mid l \text{ literal over } X\},$$

in combination with the so called *conflict exclusion* axioms *CE*

$$CE = \{\neg p_a \vee \neg p_b \mid a, b \in \mathcal{A},\ a \neq b, \text{ and the actions } a \text{ and } b \text{ are conflicting}\},$$

where two actions are considered *conflicting* if one's precondition is inconsistent with the other's effect, i.e. if there is a literal l over X such that

$$\text{either } l \in \mathit{pre}_a \text{ and } {\sim}l \in \mathit{eff}_b, \text{ or } l \in \mathit{pre}_b \text{ and } {\sim}l \in \mathit{eff}_a.$$

In sum, we define $\mathcal{S}^{par}_{\mathcal{P}} = (\Sigma, I, G, T^{par})$ where $T^{par} = AP \wedge AE \wedge EF \wedge CE$. In this encoding two actions can be applied in parallel if they have consistent effects (action effect axioms) and one action does not destroy a precondition of the other (conflict exclusion axioms). When recovering a sequential plan, such parallel actions can be serialized in any order.

Please consult the work of Ghallab et al. (2004, Chapter 7.4) for further details.

6.3 PDR without a SAT solver

Although it is possible to encode a STRIPS planning problem into an STS and use a general implementation of PDR to solve it, a more efficient approach can be adopted. The approach relies on an observation that the work normally delegated in PDR to the SAT solver can in the case of planning with the sequential plan semantics be instead implemented directly by a planning-specific procedure. Not only do we gain with this procedure a polynomial time guarantee for the response of each extension query, but the ensuing perspective also enables us to devise new improvements of the overall algorithm.

The SAT solver is employed in several places within PDR. We will start by focusing on its primary role which lies in extending the current path by one step. In Section 6.3.1 we develop procedure `extend` to replace the SAT solver in path extension queries. A separate Section 6.3.2 is then devoted to discussing inductive reason minimization in the planning context. In Section 6.3.3 we deal with replacing the remaining SAT-solver calls. We show how to efficiently implement clause pushing for positive STRIPS planning problems, a subclass of STRIPS problems that is typically used in practice. We then discuss the possibility of reversing the default search direction of PDR in Section 6.3.4 and, finally, we propose several improvements of the algorithm in Section 6.3.5.

6.3.1 Planning-specific path extensions

Let us recall the interface for path extensions, which is normally implemented in PDR by a call to a SAT solver (also recall Section 5.3). Given a state s and a set of clauses L, decide whether there exists a state t, a successor of s with respect to the transition relation T, such that t satisfies L. In the positive case, which we refer to as a *successful*

extension, return such a t. In the negative case, when no such a successor exists, compute a reason r for the failure in the form of a preferably small subset of the literals defining s, such that no state satisfying r has a successor that would satisfy L. We remark that the set L stands for some of the layers L_i considered during the run of PDR; we drop the index i here as it is irrelevant to the current presentation.

Let us assume a STRIPS planning problem $\mathcal{P} = (X, s_I, g, \mathcal{A})$ is given. We now gradually work towards a planning-specific implementation of the above interface within the procedure `extend`(s, L). Our central idea is to emulate the mechanics of the sequential encoding $\mathcal{S}^{seq}_{\mathcal{P}}$. This makes the implementation particularly straightforward from the perspective of successful extensions. Given a state s, we can simply iterate over all the actions $a \in \mathcal{A}$, generate a successor $t_a = \mathit{apply}(s, a)$ whenever a is applicable in s, and check for each t_a whether it satisfies the clauses of L. If such a successor is found, it is returned and the procedure terminates. Such an iteration is clearly affordable from the complexity point of view. In fact, it is very similar in spirit to what all explicit state planners need to do: they enumerate successor states and evaluate their heuristic value.

The non-trivial part of the `extend` procedure deals with computing a small reason in the case of an unsuccessful extension. We conceptually simplify the problem by first separately collecting a set of reasons R_a for every action $a \in \mathcal{A}$ and then computing the overall reason r as a union

$$r = \bigcup_{a \in \mathcal{A}} r_a \tag{6.1}$$

of reason contributions $r_a \in R_a$ selected in a way that minimizes the size of the union. The idea is that each $r_a \in R_a$ is a distinct reason for why the action a cannot be applied in s to produce a successor state t that would satisfy all the clauses from L. The union (6.1) then justifies why there is no such a successor state via any action $a \in \mathcal{A}$ whatsoever.

In the rest of this section, we first explain how the individual reasons $r_a \in R_a$ for an action $a \in \mathcal{A}$ are derived from the action's failed preconditions and from those clauses of L which the respective successor state fails to satisfy. We then show how this reason collecting process can be in practice sped up by employing certain subsumption concepts. Finally, we present our approach to obtaining a small overall reason r, along with a detailed pseudocode of the `extend` procedure and a proof of its correctness.

Remark 6.1. We know from Section 5.3.1 of the previous chapter that in order to ensure correctness of PDR each explaining clause C derived from an unsuccessful extension must satisfy the weaker-than-goal condition: $G \Rightarrow C$. Here, in the context of STRIPS planning and with our preference to talking about reasons rather than about the corresponding explaining clauses, the condition can be restated as the unsatisfiability of $r \wedge g$.

The condition can be in planning automatically satisfied if we add the noop action into the problem's action set. Such an addition does not affect the existence or the length of the shortest witnessing path, but it has the effect of making the represented transition relation reflexive by adding self-loops to every state. This causes each set of states to be included in its own preimage, and, as a result, we then necessarily have $r \wedge L$ unsatisfiable and our condition follows since $g \Rightarrow L$ always holds in PDR.

Procedure `extend` includes the noop action in the action set to ensure the weaker-than-goal condition.

Reasons for individual actions

We construct the set of reasons R_a for a particular action a as follows. First we check whether the action a is applicable in the given state s. If not then there is a precondition literal $l \in \mathit{pre}_a$ false in s. The complement of each such literal represents a singleton reason $\{\sim l\} \subseteq \mathit{Lits}(s)$ which we add to R_a. Clearly, as long as a state satisfies $\sim l$ there is no way a can be used to produce a successor state, let alone one that would satisfy L.

Next, we compute the successor state $t_a = \mathit{apply}(s, a)$. Strictly speaking, t_a cannot be regarded as a true successor if a is not applicable in s. Nevertheless, t_a is useful even then for computing further reasons, namely reasons corresponding to clauses of L that are false in t_a. These are either clauses that were already false in s and a failed to make them true or clauses that became false due to an effect of a. For each such clause C we add to R_a a reason r_C consisting of negations of literals $l \in C$. As an optimization, we only include those negated literals which were not made false by an effect of a. Since the other literals will always be false after a is applied due to its effects, as long as s satisfies r_C, the successor t_a cannot satisfy C. Summarizing formally, this is the final set of reasons we obtain:

$$R_a = \{\{\sim l\} \mid l \in \mathit{pre}_a \text{ and } s \not\models l\} \cup \{r_C \mid C \in L \text{ and } t_a \not\models C\},$$

where $r_C = \{\sim l \mid l \in C \text{ and } \sim l \notin \mathit{eff}_a\}$. It is easy to check that $r_C \subseteq \mathit{Lits}(s)$ as required. Notice that the set R_a is empty if an only if the action a is applicable in s and the successor t_a satisfies all the clauses from L.

Example 6.1. Starting from a state $s = \{o \mapsto \mathbf{0}, p \mapsto \mathbf{0}, q \mapsto \mathbf{0}, r \mapsto \mathbf{0}\}$, let us compute the reasons for an action $a = (\mathit{pre}_a, \mathit{eff}_a)$ with $\mathit{pre}_a = \{\neg p, q\}$ and $\mathit{eff}_a = \{o, \neg r\}$ with respect to the clause set $L = \{o \vee q, p \vee r\}$. Because the precondition q is not satisfied in s, one reason is $\{\neg q\}$. Next we compute $t_a = \mathit{apply}(s, a) = \{o \mapsto \mathbf{1}, p \mapsto \mathbf{0}, q \mapsto \mathbf{0}, r \mapsto \mathbf{0}\}$. The first clause, $o \vee q$ is satisfied in t_a and so does not give rise to a reason. The second clause, $p \vee r$, however, is false in t_a. The reason corresponding to the second clause is $\{\neg p\}$. The other negated literal, $\neg r$, is not part of the reason, because it was explicitly set to false by an effect of a. The final reason set R_a we obtain is thus $\{\{\neg p\}, \{\neg q\}\}$. Notice that both the computed reasons are subsets of $\mathit{Lits}(s) = \{\neg o, \neg p, \neg q, \neg r\}$.

Correctness of the reason set construction is captured by the following lemma.

Lemma 6.1. *Let $r_a \in R_a$ be any reason for an action $a \in \mathcal{A} \cup \{\mathit{noop}\}$ as defined above. Then*

$$r_a \wedge AP \wedge AE \wedge CF \wedge (L)' \models \neg p_a,$$

where AP, AE and CF are, respectively, the action precondition, the action effect and the classical frame axioms used in the transition formula T^{seq} of the sequential encoding $\mathcal{S}^{seq}_{\mathcal{P}}$.

Proof. Let us first assume that $r_a = \{\sim l\}$ is a reason derived from a failed precondition literal $l \in pre_a$. There is an action precondition axiom $\neg p_a \vee l \in AP$ from which the conclusion $\neg p_a$ follows by a single resolution inference with the unit assumption $\sim l$.

The other possibility is that $r_a = \{\sim l \mid l \in C \text{ and } \sim l \notin \mathit{eff}_a\}$ for some clause $C \in L$ false in the successor state t_a. There must be an action effect axiom $\neg p_a \vee \sim l' \in AE$ for every literal $l \in C$ such that $\sim l \in \mathit{eff}_a$ and also a classical frame axiom $\neg p_a \vee l \vee \sim l' \in CF$ for every literal $l \in C$ such that $\sim l \notin \mathit{eff}_a$ (if the literal l was in eff_a the clause C would be satisfied in t_a). By resolving these axioms on the respective primed literals $\sim l'$ with the primed version $C' \in (L)'$ of the clause C we obtain a clause $\neg p_a \vee \bigvee_{l \in r_a} \sim l$ from which the final unit clause $\neg p_a$ can be derived by resolution with the available assumptions from r_a. □

Reason Subsumption

Before we describe how to compute the overall reason r from the actions' contributions R_a, let us note that there are two useful notions of subsumption both between individual reasons and between reason sets, which can be used to simplify the reason sets before the computation is started. The subsumption between individual reasons inside one particular R_a is simply the subset relation. It does not make sense to keep both r_1 and r_2 inside R_a if $r_1 \subseteq r_2$. Keeping the smaller r_1 is sufficient, because whenever we would decide to pick r_2 as the reason for a inside the union (6.1), switching to r_1 instead could only make the result smaller. In practice, we only check for this kind of subsumption between the unit reasons of failed preconditions and the reasons from the false clauses.[1] This can be implemented by simply ignoring those false clauses that would have been true if the action was applicable.

Dually to the above, we can discard the whole reason set R_a of an action a if there is another action b with reason set R_b such that

$$\forall r_b \in R_b \; \exists r_a \in R_a \; r_a \subseteq r_b.$$

Here we remove the reason set R_a, which is in some sense more lean, because for any contribution $r_b \in R_b$ there is a choice for $r_a \in R_a$ which would be dominated by r_b in the final union r. For efficiency reasons, we only exploit this trick in our implementation with respect to one particular action in the role of the "subsuming" action b, namely the noop action. As mentioned before, we include the noop action to the action set to ensure PDR's correctness. Its reason set R_{noop} consists of reasons corresponding to those clauses from L which are false in s.[2] If an action a does not make any of these clauses true in its corresponding successor state, its reason set R_a will be subsumed in the described sense by R_{noop} and can be skipped.

[1] This is sufficient, because PDR keeps layers L subsumption reduced and so the reasons for false clauses are subsumption reduced automatically.

[2] PDR only calls `extend`(s, L) when $s \not\models L$, so there is always at least one such clause.

Computing the overall reason

Computing the overall reason r amounts to selecting for every a a particular reason $r_a \in R_a$ such that union (6.1) is as small as possible. Stated in this general form we are facing an optimization version of an NP-complete problem. In fact, it is easily seen to be a dual of the Maximum Subset Intersection problem shown NP-complete by Clifford and Popa (2011). We therefore do not attempt to find an optimal solution for it and contend ourselves with a reasonable approximation instead.

We order the reason sets R_a according to their size $|R_a|$ and traverse them from smaller to larger ones. The idea is to deal with the more constrained cases first before moving on to those where we have more freedom. During the traversal, we maintain an unfinished union r_0 which is initialized as the empty set $\emptyset$. Then each reason set R_a is considered in turn and we pick from each a reason $r_a \in R_a$ that minimizes the size of $r_0 \cup r_a$ and update the set r_0 accordingly to describe the union of those reasons selected so far. Although this greedy pass through the action sets does not guarantee that the final value of r_0 is minimal, it already gives satisfactory results.

To improve the quality of the reason set even further, we then minimize r_0 with respect to the subset relation by explicitly trying to remove individual literals and checking whether the result is still a valid overall reason. This is a direct adaptation of the explicit reason minimization procedure employed in the original PDR. In detail, we iteratively pick a literal $l \in r_0$ and check for every action a whether there is a reason $r_a \in R_a$ such that $r_a \subseteq (r_0 \setminus \{l\})$. If this is indeed the case, r_0 can be shrunk to $(r_0 \setminus \{l\})$, otherwise we continue with the old r_0 and try another literal instead. When all the literals have been tried out, we obtain the final result r.

Pseudocode and correctness

The code of the procedure `extend`(s, L) is detailed in Algorithm 6.1. The corresponding reason construction proceeds in three stages. In the first stage we collect reasons from the individual actions, constructing the sets R_a. This is performed during the same iteration through the action set which establishes whether a successor state t satisfying L exists. It either terminates by discovering such a t or computes a non-empty set R_a for every action a. The first stage also includes the subsumption-based filtering of reasons, both within a particular action's reason set and between the reason sets of the noop action and one other action. In the second stage, the above described simple greedy pass through the sets R_a computes an initial overall reason, which is then explicitly minimized in stage three.

Correctness of the `extend` procedure in the positive case as well as the fact that for any returned reason r we have $r \subseteq \mathit{Lits}(s)$ are easy to establish. The remaining argument is captured by the following lemma.

Lemma 6.2. *Let r be a cube returned by the procedure* `extend`(s, L)*. Then the formula*

$$r \wedge T^{seq} \wedge (L)' \tag{6.2}$$

is unsatisfiable, where T^{seq} is the transition formula of the sequential encoding $\mathcal{S}^{seq}_{\mathcal{P}}$.

Algorithm 6.1 Procedure `extend`(s, L):

Input:
State s; a set of clauses L such that $s \not\models L$

Output:
Either state t, a successor of s such that $t \models L$
or a reason $r \subseteq \mathit{Lits}(s)$ such that no state satisfying r has a successor satisfying L

1: /* Stage one: look for the successor state and prepare the reason sets */
2: $L^s \leftarrow \{C \in L \mid s \not\models C\}$ /* Clauses false in s */
3: $R_{noop} \leftarrow \{\sim C \mid C \in L^s\}$ /* The reasons for the noop action */
4: **assert** $R_{noop} \neq \emptyset$ /* Follows from the contract with the caller */
5: $\mathcal{R} \leftarrow \{R_{noop}\}$ /* The set of reason sets collected so far */
6:
7: **foreach** $a \in \mathcal{A}$ **do**
8: $\quad \mathit{pre}_a^s \leftarrow \{l \in \mathit{pre}_a \mid s \not\models l\}$ /* Preconditions false in s */
9: $\quad t \leftarrow \mathit{apply}(s, (\emptyset, \mathit{eff}_a))$ /* Ignore the preconditions and apply the effects of a */
10: $\quad L^t \leftarrow \{C \in L \mid t \not\models C\}$ /* Clauses false in t */
11: $\quad$ **if** $\mathit{pre}_a^s = \emptyset$ and $L^t = \emptyset$ **then**
12: $\quad\quad$ **return** t /* Successful extension: returning a successor */
13: $\quad$ **else if** $L^s \subseteq L^t$ **then**
14: $\quad\quad$ **pass** /* Do nothing: the reason set would be subsumed by R_{noop} */
15: $\quad$ **else**
16: $\quad\quad L_0^t \leftarrow \{C \in L^t \mid C \cap \mathit{pre}_a^s = \emptyset\}$ /* False clauses with a non-subsumed reason */
17: $\quad\quad R_a \leftarrow \{\{\sim l\} \mid l \in \mathit{pre}_a^s\} \cup \{\{\sim l \mid l \in C \text{ and } \sim l \notin \mathit{eff}_a\} \mid C \in L_0^t\}$
18: $\quad\quad \mathcal{R} \leftarrow \mathcal{R} \cup \{R_a\}$ /* Record the reason set */

19: /* Stage two: compute an overall reason */
20: $r \leftarrow \emptyset$
21: **foreach** $R_a \in \mathcal{R}$ ordered by $|R_a|$ from small to large **do**
22: $\quad$ pick $r_a \in R_a$ such that $|r \cup r_a|$ is minimal
23: $\quad r \leftarrow r \cup r_a$

24: /* Stage three (optional): minimize the reason */
25: **foreach** $l \in r$ **do**
26: $\quad$ **if** for every $R_a \in \mathcal{R}$ there is $r_a \in R_a$ such that $r_a \subseteq (r \setminus \{l\})$ **then**
27: $\quad\quad r \leftarrow (r \setminus \{l\})$

28: **return** r /* Unsuccessful extension: returning a (subset minimal) reason cube */

Proof. We first observe that for every action $a \in \mathcal{A} \cup \{noop\}$ there is a reason r_a such that $r = \bigcup_{a \in \mathcal{A} \cup \{noop\}} r_a$. For those actions a for which $R_a \in \mathcal{R}$ this reason is initially picked during stage two (line 22) and possibly later changed to the reason r_a for which $r_a \subseteq (r \setminus \{l\})$ during stage three (line 26). For those actions whose reason set is subsumed by R_{noop} (line 14) we can formally pick the same reason as for noop.

Since $r_a \subseteq r$ for every action $a \in \mathcal{A} \cup \{noop\}$, we can use Lemma 6.1 to infer that formula (6.2) entails the unit clause $\neg p_a$ for every $a \in \mathcal{A} \cup \{noop\}$. But because formula (6.2) also trivially entails the at-least-one axiom $alo = \bigvee_{a \in \mathcal{A} \cup \{noop\}} p_a$, it must be unsatisfiable. □

It is easy to see that the procedure `extend`(s, L) runs in time polynomial in $|X|$, the number of state variables, $|\mathcal{A}|$, the number of actions of the planning problem, and $|L|$, the size of the given clause set. This is mainly enabled by the fact that `extend` emulates the sequential encoding $\mathcal{S}^{seq}_{\mathcal{P}}$ and the individual actions can be in the first stage considered independently.[3]

A similar complexity guarantee seems to be achievable within a general-purpose SAT solver when supplied with the same encoding and configured to prefer branching on the action variables A and setting them first to true. However, the inherent overhead connected with explicitly generating all the corresponding axioms and storing them in memory will be probably noticeable in practice. Moreover, the reason set subsumption optimization does not have a counterpart in a general-purpose solver.

6.3.2 Inductive reason minimization in procedure `extend`

Inductive minimization is based on the idea that when checking whether a particular literal l can be removed from the final reason r we can assume that the clause $C = \sim r_0$ corresponding to the reduced reason $r_0 = (r \setminus \{l\})$ is already present in the set of clauses L (recall the general description in Section 5.3.2). We can perform inductive minimization within the `extend` procedure by speculating for each action a whether we would be able to satisfy the additional clause C by applying a. Only if the answer is positive do we need to look for a "proper" reason $r_a \in R_a$.

The idea is demonstrated in Algorithm 6.2, which should be regarded as a replacement for stage three of the original `extend` procedure. Notice that we no longer consider the noop action to be part of the action set[4] and thus we need to explicitly check that there remains at least one literal incompatible with the goal condition g (line 4). There can still be actions, however, whose reason set has been subsumed by R_{noop} and for these we look for a reason in R_{noop} (line 11) whenever they fail to pass the inductiveness check (line 7). To avoid confusion we remark that the **continue** and **break** commands refer to the inner-most cycle, which iterates over actions (line 6). Finally, we note that the

[3] When devising an analogous procedure for a parallel plan semantics, one would in general need to consider every subset of actions that can be applied together. This seems to make a polynomial time solution much more difficult, if not hopeless. However, see Section 6.5 for an interesting connection.

[4] The noop action trivially passes the inductiveness check, because it can never make any clause true.

Algorithm 6.2 Stage three of `extend`(s, L); inductive version:

```
1:  r_0 ← r
2:  loop
3:    foreach l ∈ r do /* Check each literal of r once */
4:      if there is l_0 ∈ (r_0 \ {l}) such that ∼l_0 ∈ g then /* Can try removing l */
5:        r_0 ← (r_0 \ {l})
6:        foreach a ∈ A do
7:          if for every l_0 ∈ eff_a : ∼l_0 ∉ r_0 then
8:            continue /* a passed by the inductive argument */
9:          if R_a ∈ R and there is r_a ∈ R such that r_a ⊆ r_0 then
10:           continue /* a passed; it has its own small reason */
11:         if R_a ∉ R and there is r_a ∈ R_noop such that r_a ⊆ r_0 then
12:           continue /* R_a was subsumed by R_noop which has a small reason */
13:
14:         /* Action a says: "Literal l cannot be removed" */
15:         r_0 ← (r_0 ∪ {l}) /* Put the literal back */
16:         break
17:   if r = r_0 then /* No removal in the last iteration */
18:     return r
19:   r ← r_0
```

presence of a small reason in R_{noop} depends only on the current value of r_0 and so the corresponding check could be precomputed outside the inner cycle.

Example 6.2. Recall Example 6.1, in which the action $a = (\{\neg p, q\}, \{o, \neg r\})$ failed in the state $s = \{o \mapsto \mathbf{0}, p \mapsto \mathbf{0}, q \mapsto \mathbf{0}, r \mapsto \mathbf{0}\}$ to provide a successor state that would satisfy the clauses from $L = \{o \vee q, p \vee r\}$ and so we computed a reason set $R_a = \{\{\neg p\}, \{\neg q\}\}$. Assume that apart from a the action set $\mathcal{A}$ contains just one other action, namely $b = (\{\neg r\}, \{p\})$, for which we obtain a reason set $R_b = \{\{\neg o, \neg q\}\}$. The overall reason after stage two is thus necessarily $r = \{\neg o, \neg q\}$. Assuming that the goal condition of the given problem is $g = \{o, p, q, r\}$, inductive minimization of the reason r could proceed as follows.

First we try the reason $r_0 = \{\neg o\}$. Since $o \in \mathit{eff}_a$ we cannot use the inductive argument for the action a and also no proper reason $r_a \in R_a$ has the property that $r_a \subseteq r_0$. Thus the literal $\neg q$ cannot be removed from r. Next we try the reason $r_0 = \{\neg q\}$. Since neither the action a nor b contain the literal q in their effect sets, the smaller reason is justified inductively for both actions and the overall reason r is reduced to $\{\neg q\}$. We cannot minimize r further, because there has to remain at least one literal $l_0 \in r$ such that $\sim l \in g$.

Looking from the perspective of the final learned clause $C = \sim r$ we observe that inductive minimization allows us (as in the example above) to remove from C every literal that cannot be made true by any action of the action set $\mathcal{A}$. Although this may seem like a powerful (global) criterion, it is effectively made redundant in practice by

the so called *relaxed reachability analysis* (see Hoffmann and Nebel, 2001, Section 4.3), a standard preprocessing step which, before the actual search is started, removes from the problem all such unattainable variables as well as all actions that mention them in their precondition sets. Non-trivial invocations of inductive minimization were actually quite rare in our experiments.

6.3.3 Replacing the remaining SAT-solver calls

Beside the query for extending states, there are two other points in the formulation of PDR (recall Algorithm 5.5 on page 145) where a SAT-solver call is employed. It is used to pick initial states at the beginning of the path construction phase (line 4) and is also central to verifying the condition for pushing clauses during the clause propagation phase (line 26). In planning, we can easily do without a SAT solver in the first case, because there is only one initial state to be picked, namely the state s_I, and we just need to verify that s_I satisfies the clauses of L_k before the path construction phase of iteration k can be started.

We have basically two options how to deal with the second case. Since clause pushing is not needed for ensuring correctness of PDR, we can simply leave the operation out. As we later show in our experiments, this does not significantly affect the performance on planning benchmarks, which are typically satisfiable. As a second option, we propose to restrict the planning formalism such that the query corresponding to a push check of a clause C, i.e.,

$$SAT?[\sim C \wedge T \wedge (L)'], \tag{6.3}$$

can be decided in polynomial time.[5]

We say that a STRIPS planning problem is *positive* if the precondition set of every action and the goal condition of the problem consist of positive literals only.[6] It is easy to see that when running on a positive STRIPS problem, PDR only deals with positive clauses. The unit clauses of layer L_0, which describe the goal, are positive by assumption and all the learned clauses are transitively built only from the goal literals and from the action precondition literals. This observation allows us to reduce query (6.3) to the evaluation of "the positive part of the interface for path extensions":

Lemma 6.3. *Let $\mathcal{P} = (X, s_I, g, \mathcal{A})$ be a positive STRIPS planning problem and T^{seq} the transition formula of the sequential encoding $\mathcal{S}^{seq}_{\mathcal{P}}$. Further, let L be a set of positive clauses over X, C a positive clause over X, and $s_C : X \to \{\mathbf{0}, \mathbf{1}\}$ a state defined for every $p \in X$ by*

$$s_C(p) = \begin{cases} \mathbf{0} & \text{if } p \in C, \\ \mathbf{1} & \text{otherwise.} \end{cases}$$

[5] In our current setting, there does not seem to be a general polynomial solution. In fact, even in the degenerate case of T encoding a transition by the single noop action and C being the empty clause, the query (6.3) boils down to satisfiability of L and its evaluation is thus an NP-complete problem.

[6] Most of the standard planning benchmarks are positive STRIPS. Moreover, there is a well-known reduction (Gazen and Knoblock, 1997) that turns a general STRIPS problem into a positive one. The reduction introduces a new variable p^* for every variable p that occurs negatively in a precondition or in the goal and updates the actions to always force p^* to have the opposite value to that of p.

Then the formula

$$F_C = {\sim}C \wedge T^{seq} \wedge (L)'$$

is satisfiable if and only if there is an action $a \in \mathcal{A}$ such that

$$s_C \models \mathit{pre}_a \text{ and } \mathit{apply}(s_C, a) \models L.$$

Proof. Let us first assume that there is an action $a \in \mathcal{A}$ applicable in s_C such that the successor state $t = \mathit{apply}(s_C, a)$ satisfies the clauses from L. Notice that $\mathit{Vars}(F_C) = X \cup A \cup X'$, where $A = \{p_a \mid a \in \mathcal{A}\}$ is the set of variables used for encoding applied actions. We define the following valuation $\alpha_a : A \to \{\mathbf{0}, \mathbf{1}\}$:

$$\alpha_a = \{p_a \mapsto \mathbf{1}\} \cup \{p_b \mapsto \mathbf{0} \mid b \in \mathcal{A}, b \neq a\}.$$

It is easy to verify that the joint valuation $(s_C \cup \alpha_a \cup t')$ satisfies F_C.

For the opposite direction, let us assume that a valuation $V : X \cup A \cup X' \to \{\mathbf{0}, \mathbf{1}\}$ satisfies the formula F_C. We fix an action $a \in \mathcal{A}$ such that $V(p_a) = \mathbf{1}$. Such an action must exist, because V satisfies the at-least-one axiom $\mathit{alo} = \bigvee_{a \in \mathcal{A}} p_a$, which is part of T^{seq}. By restricting V, first, to the state variables X, and, second, to the primed variables X', we extract, respectively, a state $s = V \restriction X$ and a state t such that $t' = V \restriction X'$. The axioms of T^{seq} ensure that the action a is applicable in s and that $t = \mathit{apply}(s, a)$.

We now notice that $s \models {\sim}C$, which means that $s(p) = \mathbf{0}$ for every $p \in C$. Thus if there is a difference between the states s and s_C it is only because of variables $p \notin C$ for which $s(p) = \mathbf{0}$ and $s_C(p) = \mathbf{1}$. But this means, for one thing, that since the action a is applicable in s, it must also be applicable in s_C (preconditions are positive) and, for the other, since $t \models L$, the successor state $t_C = \mathit{apply}(s_C, a)$ corresponding to s_C must also satisfy the clauses from L (the implication $\forall p \in X : s(p) = \mathbf{1} \Rightarrow s_C(p) = \mathbf{1}$ is preserved by the transition and becomes $\forall p \in X : t(p) = \mathbf{1} \Rightarrow t_C(p) = \mathbf{1}$, and the clauses from L are positive by assumption). □

A version of PDR specialized to positive STRIPS planning is shown in Algorithm 6.3. The calls to a SAT solver of the original formulation (Algorithm 5.5 on page 145) were replaced, respectively, by a simple entailment check (line 5), a call to the `extend` procedure (line 11), and by an enumeration of the successor states of the state s_C as defined in Lemma 6.3 (line 26).

6.3.4 Reversing the search direction

It has been mentioned that the original formulation of PDR is based on the opposite search direction than the one adopted in this thesis and that it extends the paths from a goal state backwards towards the initial state. We would like to test the algorithm with both directions to see which one is more favorable in practice.

One possibility to achieve this is to provide PDR with an inverted version of the input, where the initial and goal states have been swapped and the transition relation "turned around". This is straightforward to do when the input is an STS (recall Remark 5.1), as is the case with the general version of PDR. The situation is more complicated with the

Algorithm 6.3 `PDRplan`$(X, s_I, g, \mathcal{A})$:

Input:
A positive STRIPS planning problem $\mathcal{P} = (X, s_I, g, \mathcal{A})$

Output:
A plan for $\mathcal{P}$ or a guarantee that no plan exists

```
1:  L_0 ← {{p} | p ∈ g} /* The goal cube treated as a set of unit clauses */
2:  foreach j > 0 : L_j ← ∅
3:  for k = 0, 1, ... do
4:    /* Path construction: */
5:    if s_I ⊨ L_k then
6:      Q ← {(s_I, k)}
7:      while Q not empty do
8:        pop some (s, i) from Q with minimal i
9:        if i = 0 then
10:         return PLAN FOUND
11:       if extend(s, L_{i−1}) returns a successor state t then
12:         Q ← Q ∪ {(t, i − 1), (s, i)}
13:       else
14:         extend returned a reason r ⊆ Lits(s)
15:         foreach 0 ≤ j ≤ i : L_j ← L_j ∪ {∼r}
16:
17:         /* Obligation rescheduling: */
18:         if i < k then
19:           Q ← Q ∪ {(s, i + 1)}
20:
21:   /* Clause propagation: */
22:   for i = 1, ..., k + 1 do
23:     foreach C ∈ L_{i−1} \ L_i do
24:       /* Clause push check */
25:       s_C ← {p ↦ 0 | p ∈ C} ∪ {p ↦ 1 | p ∈ (X \ C)}
26:       if for every a ∈ A : s_C ⊭ pre_a or apply(s_C, a) ⊭ L_{i−1} then
27:         L_i ← L_i ∪ {C}
28:     /* Convergence check */
29:     if L_{i−1} = L_i then
30:       return NO PLAN POSSIBLE
```

SAT-solver-free version, which directly takes a STRIPS planning problem as an input. Indeed, it seems the `extend` procedure substantially relies on the forward direction.

Interestingly, there exists a transformation for inverting STRIPS planning problems. It was first described by Massey (1999) in his dissertation. We present here a more streamlined version due to Pettersson (2005) which relies on the problem being positive. Let us start by introducing an alternative representation of positive STRIPS planning problems, which makes the description of the transformation particularly straightforward. A positive STRIPS planning problem in the *subset representation* is given by a tuple $\mathcal{P} = (X, i, g, \mathcal{A})$, where $i, g \subseteq X$ are the initial and goal conditions, respectively, and every action $a \in \mathcal{A}$ is encoded by a triple $a = (pre_a, add_a, del_a)$, consisting of a precondition set, an add set and a delete set, which are subsets of X such that $pre_a \cap add_a = \emptyset$ and $add_a \cap del_a = \emptyset$. The subset representation differs from the one presented in Section 6.2 by encoding the initial state by the set of those variables that are true in it:

$$s_I(p) = 1 \text{ if and only if } p \in i,$$

and by splitting action's effects into positive and negative ones:

$$\mathit{eff}_a = add_a \cup \{\neg p \mid p \in del_a\}.$$

The goal condition g and precondition sets pre_a remain intact, but now may be understood as subsets of X since the problem is positive. It should be clear that the subset representation and the one of Section 6.2 are equivalent.

Now, for an action $a = (pre_a, add_a, del_a)$ an inverted action a^{-1} is formed by exchanging the precondition and delete set: $a^{-1} = (del_a, add_a, pre_a)$. For a set of actions $\mathcal{A}$ the set of inverted actions is $\mathcal{A}^{-1} = \{a^{-1} \mid a \in \mathcal{A}\}$. Given a planning problem $\mathcal{P} = (X, i, g, \mathcal{A})$ in the subset representation, the *inverted* problem $\mathcal{P}^{-1}$ is obtained by exchanging the initial and goal conditions while taking their complements with respect to X and using the inverted action set:

$$\mathcal{P}^{-1} = (X, (X \setminus g), (X \setminus i), \mathcal{A}^{-1}).$$

The original problem and its inverted version are related in the following sense:

Theorem 6.1. *The sequence of actions $a_0, a_1, \ldots, a_k$ is a plan for the planning problem $\mathcal{P}$ if and only if the sequence $a_k^{-1}, a_{k-1}^{-1}, \ldots, a_0^{-1}$ is a plan for $\mathcal{P}^{-1}$.*

This means that performing forward search (also called progression) in $\mathcal{P}$ is equivalent to performing backward search (regression) in $\mathcal{P}^{-1}$ and vice versa. Notice that, a priori, there is no computational overhead incurred by the transformation: the inverted problem has the same number of actions as well as the same set of state variables X and so the representation of states is of the same size. A proof of Theorem 6.1 along with further intuition behind the transformation and its theoretical and practical implications are described by Suda (2013b).

6.3.5 Further improvements

We describe three additional modifications of PDR that aim to make the algorithm more efficient in practice. While the first is a planning-specific improvement of the `extend` procedure, the other two focus on how obligations are handled by the overall algorithm. In Section 6.4 we experimentally evaluate the effect of these modifications on solving planning problems. We present the pseudocode of all these three improvements together, at the end of this section.

Lazy False Clause Computation

One way to speed up the `extend` procedure in practice is a technique we named *lazy false clause computation*. It is based on the following two observations:

- L^s, the set of clauses false in the state s, is typically only a small subset of L, the set of all the clauses the successor state should satisfy,
- only a small fraction of the available actions makes any of the clauses of L^s true in their respective successor.

The idea is to avoid the relatively expensive computation of the set of clauses false in the successor t, i.e. the computation of the set L^t on line 10 (Algorithm 6.1), and instead first only look at the truth value of the clauses from L^s. (Notice that L^s is precomputed before we start iterating over the actions.) Only if we find an action a such that all its preconditions are satisfied in s and it makes all the clauses from L^s true in the respective successor t, we classify the action as *promising* and go back to computing the full L^t. Thus, with non-promising actions we save computational time. We may pay for it a little on the side of the quality of the reason set, because for them we only use $L^{s,t} = \{C \in L^s \mid t \not\models C\}$ instead of the full L^t for computing the reasons. On the other hand, with promising actions a complete test is necessary to distinguish a true successor t satisfying all of L from an action that "repairs" everything which was false in s, but "breaks" something else instead.

Sidestepping

Sidestepping is a technique we propose to make PDR more active in early exploration of promising paths. It partially circumvents the limitation stemming from the fact the `extend` procedure emulates the sequential encoding $\mathcal{S}^{seq}_{\mathcal{P}}$.

Imagine we want to extend an obligation (s, i), i.e. to find a successor of s that would satisfy L_{i-1}, and there are two clauses $C_1, C_2 \in L_{i-1}$ false in s. Let us think of the two clauses as of two independent subgoals to be achieved. There are two actions a_1 and a_2 applicable in s. Action a_1 makes C_1 true in the successor state and a_2 makes C_2 true, but no action can make both the clauses true in one step. This means the extension cannot be successful and PDR will learn a new clause $C = C_1 \vee C_2$ (or a superset thereof). The clause C expresses the fact that in order to reach a state satisfying L_{i-1} in one step, at least one of the two clauses C_1, C_2 must be satisfied beforehand. This

could be an important ingredient to showing that no path of length k can reach the goal, helping the algorithm eventually advance to the next iteration. However, because in planning we are usually more interested in actually finding plans than showing their non-existence, deriving the clause C could represent unnecessary extra work. The idea behind sidestepping is to make the `extend` procedure succeed more often, even if that does not mean directly advancing into the next layer. In our example, we return the successor state $t = \mathit{apply}(s, a_1)$ with an additional flag informing the caller that the new obligation should have index i and not the usual $i - 1$. We are effectively sidestepping from (s, i) to (t, i). In the next round the obligation (t, i) will be picked and successfully (provided the actions a_1 and a_2 do not interfere) extended into $(u, i - 1)$ via the action a_2. This way we end up with a state satisfying L_{i-1} almost as if we executed the two actions a_1 and a_2 in parallel.

Let us now present the sidestepping technique in more detail. In order for an action a and its respective successor $t_a = \mathit{apply}(s, a)$ to qualify for a sidestep during extension of an obligation (s, i) the following conditions must to be met:

1) a is applicable in s,

2) t_a improves over s with respect to the set of satisfied L_{i-1} clauses:

$$L_{i-1}^{t_a} = \{C \in L_{i-1} \mid t_a \not\models C\} \subset L_{i-1}^{s} = \{C \in L_{i-1} \mid s \not\models C\},$$

3) t_a satisfies all the L_i clauses.

Notice that we require the improvement to be strict (condition 2). This ensures that sidestepping does not compromise termination. We also make sure that the new state stays within L_i (condition 3) – an improvement in one respect should not be payed for by an overall deterioration.

If there is no action that qualifies for a sidestep, we compute and return a reason set as usual. Otherwise, we choose among them an action a for which the size of $L_{i-1}^{t_a}$ is the smallest. The case when $|L_{i-1}^{t_a}| = 0$ corresponds to a regular successful extension and a new obligation $(t_a, i - 1)$ will be stored in the set Q. If $|L_{i-1}^{t_a}| > 0$, we store (t_a, i) instead, which means that we perform a sidestep.

After sidestepping both the old obligation (s, i) and the new (t_a, i) occupy the same index in Q. It is important that we prioritize the latter over the former for picking (e.g., even if we otherwise want to use queue tie-breaking strategy; see Remark 5.2), by which we prevent the algorithm from sidestepping in the same way more than once.[7]

Notice that sidestepping is an extension of PDR that relies on a modification of the planning-specific `extend` procedure. As such it does not have an immediate counterpart in the original algorithm, where path extensions are delegated to a SAT solver.

[7] By the time (s, i) is reconsidered we must have had an unsuccessful extension of (t_a, i), which means L_i got in the meantime strengthened and t_a no longer satisfies it.

Keeping obligations between iterations

Let us return to obligation rescheduling (lines 18 and 19 of Algorithm 6.3) to discuss one additional aspect of this feature. Notice that we reschedule an obligation (s, i) only if $i < k$ so that the new obligation $(s, i + 1)$ is never positioned further from the goal than k steps during iteration k. Obligations of the form (s, k) are simply forgotten which ensures that the path construction phase eventually terminates. A viable alternative to this strategy is to reschedule these obligations on the queue Q with index $k + 1$, but set them into a "dormant state" and return to them only during the next iteration. This can be understood as effectively enlarging the set of initial states for the next iteration so that it includes all the states reached so far.

This modification is quite simple to implement and seems to go well with the spirit of obligation rescheduling itself. It has recently been also described by Bayless et al. (2013). A potential disadvantage of the modification could be the increased memory consumption, since all the states ever encountered during the run must be stored by the algorithm. Its utility may, therefore, depend on the application domain.

Pseudocode of the improvements

Algorithm 6.4 displays stage one of procedure `extend`$^+$, an enhancement of the `extend` procedure by the lazy false clause computation technique and with a support for sidestepping. Stage two of `extend`$^+$ is meant to be supplemented from the same stage of the original `extend` procedure (Algorithm 6.1) and stage three may employ inductive minimization (Algorithm 6.2).

Algorithm 6.5 realizes sidestepping with the help of the `extend`$^+$ procedure. Moreover, it incorporates the technique for keeping obligations between iterations. The clause propagation phase is identical to the one already presented in Algorithm 6.3.

6.4 Experiments

In this section we report on a series of experiments aimed to establish the practical relevance of PDR for automated planning. We first compare the standard version of the algorithm combined with encodings to the SAT-solver-free variant of PDR proposed in this chapter. The latter is implemented in our new planner PDRplan. Next, we measure the influence of the individual "enhancements" which make up full-fledged PDR (recall Section 5.3.2) as well as of the various improvements proposed in Section 6.3.5 on the performance of PDRplan. The most successful configuration of PDRplan is then compared to other planners, including state-of-the-art representatives of the heuristic search planning and planning as satisfiability paradigms. Finally, we also assess PDRplan from the perspective of plan quality, finding optimal length plans and detecting unsatisfiable planning problems.

Algorithm 6.4 Stage one of $\texttt{extend}^+(s, i)$:

Input:
Obligation (s, i), i.e. a state s and an index i, such that $s \not\models L_{i-1}$

Output:
Either an obligation $(t, i-1)$ where t is a successor of s and $t \models L_{i-1}$, or
an obligation (t, i) where t is a successor of s, $t \models L_i$ and
t satisfies strictly more clauses from L_{i-1} than s, or
an inductive reason $r \subseteq \mathit{Lits}(s)$

1: $L^s \leftarrow \{C \in L_{i-1} \mid s \not\models C\}$ /* Clauses false in s */
2: $R_{noop} \leftarrow \{\sim C \mid C \in L^s\}$ /* The reasons for the noop action */
3: **assert** $R_{noop} \neq \emptyset$ /* Follows from the contract with the caller */
4: $\mathcal{R} \leftarrow \{R_{noop}\}$ /* The set of reason sets collected so far */
5: $a_{side} \leftarrow noop$ /* Current best candidate for sidestepping (*noop* as a dummy) */
6: $x_{side} \leftarrow |L^s|$ /* Score of the current best candidate */
7:
8: **foreach** $a \in \mathcal{A}$ **do**
9: $\quad pre^s_a \leftarrow \{l \in pre_a \mid s \not\models l\}$ /* Preconditions false in s */
10: $\quad t \leftarrow \mathit{apply}(s, (\emptyset, \mathit{eff}_a))$ /* Ignore the preconditions and apply the effects of a */
11: $\quad L^{s,t} \leftarrow \{C \in L^s \mid t \not\models C\}$ /* The lazy approach: clauses false both in s and t */
12: $\quad$ **if** $L^{s,t} = L^s$ **then** /* No improvement over s */
13: $\quad\quad$ **continue** /* Do nothing: the reason set would be subsumed by R_{noop} */
14:
15: $\quad$ **if** $pre^s_a = \emptyset$ and $|L^{s,t}| < x_{side}$ **then** /* The action is promising ... */
16: $\quad\quad L^t \leftarrow \{C \in L_{i-1} \mid t \not\models C\}$ /* ... we must compute the full L^t */
17: $\quad\quad$ **if** $L^t = \emptyset$ **then**
18: $\quad\quad\quad$ **return** $(t, i-1)$ /* Successful extension: returning a true successor */
19: $\quad\quad$ **if** $L^t = L^{s,t}$ and $t \models L_i$ **then** /* No false clauses besides those from $L^{s,t}$ */
20: $\quad\quad\quad a_{side} \leftarrow a$
21: $\quad\quad\quad x_{side} \leftarrow |L^{s,t}|$
22: $\quad$ **else**
23: $\quad\quad L^t \leftarrow L^{s,t}$ /* Save time by using $L^{s,t}$ instead of the full L^t below */
24:
25: $\quad L^t_0 \leftarrow \{C \in L^t \mid C \cap pre^s_a = \emptyset\}$ /* False clauses with a non-subsumed reason */
26: $\quad R_a \leftarrow \{\{\sim l\} \mid l \in pre^s_a\} \cup \{\{\sim l \mid l \in C \text{ and } \sim l \notin \mathit{eff}_a\} \mid C \in L^t_0\}$
27: $\quad \mathcal{R} \leftarrow \mathcal{R} \cup \{R_a\}$ /* Record the reason set */
28:
29: **if** $x_{side} < |L^s|$ **then**
30: $\quad$ **assert** $a_{side} \neq noop$
31: $\quad$ **return** $(\mathit{apply}(s, a_{side}), i)$ /* Successfully sidestepping with the best candidate */

32: /* Continue with stage two of Algorithm 6.1 and stage three of Algorithm 6.2 */

Algorithm 6.5 `PDRplan1.1`$(X, s_I, G, \mathcal{A})$:

Input:
A positive STRIPS planning problem $\mathcal{P} = (X, s_I, g, \mathcal{A})$

Output:
A plan for $\mathcal{P}$ or a guarantee that no plan exists

$L_0 \leftarrow \{\{p\} \mid p \in g\}$ /* The goal cube treated as a set of unit clauses */
foreach $j > 0 : L_j \leftarrow \emptyset$
$Q \leftarrow \{(s_I, 0)\}$
for $k = 0, 1, \ldots$ **do**
  /* Path construction: */
  while there is (s, i) in Q with $i \leq k$ **do**
    pop some (s, i) from Q with minimal i
    if $s \not\models L_i$ **then**
      $Q \leftarrow Q \cup (s, i + 1)$
    else if $i = 0$ **then**
      return PLAN FOUND
    else if `extend`$^+(s, i)$ returns an obligation (t, j) **then**
      assert $j = i - 1$ or $j = i$ /* Either a regular extension or a sidestep */
      $Q \leftarrow Q \cup \{(s, i), (t, j)\}$
    else
      `extend`$^+$ returned a reason $r \subseteq \mathit{Lits}(s)$
      foreach $0 \leq j \leq i : L_j \leftarrow L_j \cup \{\sim r\}$

      /* Obligation rescheduling: */
      $Q \leftarrow Q \cup \{(s, i + 1)\}$ /* Keep obligations with $i + 1 > k$ till next iteration */

  /* Clause propagation: */
  for $i = 1, \ldots, k + 1$ **do**
    foreach $C \in L_{i-1} \setminus L_i$ **do**
      /* Clause push check */
      $s_C \leftarrow \{p \mapsto \mathbf{0} \mid p \in C\} \cup \{p \mapsto \mathbf{1} \mid p \in (X \setminus C)\}$
      if for every $a \in \mathcal{A} : s_C \not\models \mathit{pre}_a$ or $\mathit{apply}(s_C, a) \not\models L_{i-1}$ **then**
        $L_i \leftarrow L_i \cup \{C\}$
    /* Convergence check */
    if $L_{i-1} = L_i$ **then**
      return NO PLAN POSSIBLE

6.4.1 The setup

As before, we performed the experiments on machines with 3.16 GHz Intel Xeon CPU, 16 GB RAM, running Debian 6.0. Let us stress that although multiple cores are available on each machine, all the planners used only one core and we made sure that there was no other busy process running concurrently that would compete with a planner for memory, etc. The main measured resource was computation time. We used a time limit of 180 seconds per problem instance for most of the runs, but increased it to 1800 seconds for the main comparison.

To increase the level of confidence towards the correctness of our implementation all the generated plans were subsequently checked by the latest version of plan validator `VAL` (Howey et al., 2004). No discrepancies were found during the experiments reported in this chapter.

We tested the planners on the STRIPS[8] benchmarks of the International Planning Competition (IPC) of years 1998–2011 (IPC, 2013). The benchmarks are grouped together into benchmark domains of various planning scenarios. We used all the available STRIPS domains except the following:

- 1998-MOVIE, where it turned out to be technically difficult to validate the plans.[9] Note that the domain is, in fact, trivial to solve.
- 2000-SCHEDULE, which is originally an ADL domain. The competition archive contains also a STRIPS version, but accompanied by a note saying that this version later proved to be problematic and was dropped from the competition.
- 2002-ROVERS, the problems of which are included in the set 2006-ROVERS.
- 2002-SATELLITE and 2011-TIDYBOT, which make use of actions with negative preconditions, a feature not supported by our parser.

Altogether, we collected 1561 problems in 49 domains (see Table 6.2 on page 202 for a detailed list). The 2008 and 2011 competition benchmarks specify action costs. We modified the respective files to remove this feature, which is not supported by PDRplan.

We implemented PDR with the `extend` procedure as described in Section 6.3 in the PDRplan system. The code of PDRplan (approximately 2K lines of C++) is built on top of a PDDL parser and a grounder adopted from SatPlan 2006 (Kautz et al., 2006). We modified the parser to successfully process the large problems of the more recent IPC domains. The source code of PDRplan is publicly available on our web page (Suda, 2014b), which also contains all the other material relevant for reproducing the experiments.

[8] The richer ADL formalism is currently not supported by PDRplan.

[9] The parser we adopted for PDRplan removes vacuous arguments of operators from the resulting actions' names. The validator `VAL` then complains about the resulting plans.

6.4.2 PDRplan v.s. standard PDR plus encodings

The main purpose of the first experiment was to compare PDRplan and its planning-specific implementation of the `extend` procedure to a composition of the general PDR, which uses a SAT solver to answer the one-step reachability queries, with various encodings of planning into an STS. We also wanted to establish which of the two possible search directions in PDR is more favorable for discovering plans.

We took our implementation of PDR previously developed as a model checking tool for hardware circuits as described in Chapter 5. We will here refer to the tool as `minireachIC3`.[10] We extended `minireachIC3` such that it is able to read a description of an STS. We designed a new input format for that purpose, which we call DIMSPEC (Suda, 2013c). It is a simple modification of the well-known DIMACS CNF format used by most SAT solvers extended to define the individual clause sets of an STS.

We coupled `minireachIC3` with four encoders of planning into an STS. The first two encoders, `seq` and `par`, are our implementations of the two simple encoding $\mathcal{S}_{\mathcal{P}}^{seq}$ and $\mathcal{S}_{\mathcal{P}}^{par}$, respectively (recall Section 6.2.2). The third encoder is a version of the planner `Mp` (Rintanen, 2012) modified to output the encoded instance in the form of an STS and quit before starting the actual solving process. `Mp` uses the $\exists$-step parallel encoding scheme of Rintanen et al. (2006). Finally, the fourth encoder implements the SASE encoding scheme introduced by Huang et al. (2012). The particular implementation we used derives from the `FreeLunch` planning library (Balyo et al., 2012).

In order to obtain a fair comparison we used a basic version of PDRplan configured in a way that most resembles the workings of `minireachIC3`. The configuration follows the planning-specific version of the overall algorithm (Algorithm 6.3) and relies on the `extend` procedure (Algorithm 6.1) with the minimization phase of the reason computation (stage three) enhanced by induction (Algorithm 6.2). The additional improvements of Section 6.3.5 were disabled for this experiment.

We compared the systems in both search directions. In accord with the terminology of previous chapter, the *forward* direction here means that PDR constructs the path from the initial state towards the goal. The opposite, *backward* direction is the one preferred by the original exposition of PDR used in model-checking. To start `minireachIC3` in the backward direction we inverted the encoded STS, to reverse the search direction of PDRplan we inverted the planning problem (as explained in Section 6.3.4).[11]

Adding invariants

An invariant of a transition system is a property of the initial state preserved by all transitions. In planning, one typically considers invariants in the form of binary clauses (Rintanen, 1998), which can be computed by a simple fixpoint algorithm (Rintanen, 2008a). Adding the invariant clauses into an encoding is known to speed up plan search in the planning as satisfiability paradigm.

[10] In detail, `minireachIC3` refers to the final version of the tool, with triggered clause pushing.
[11] One could also experiment with encodings of the inverted problems. We leave this for future work.

We noticed that the performance of PDRplan in the backward direction can also be enhanced with the help of invariants. When PDR is run in the backward direction, it is sound to strengthen every layer by the binary clauses of a precomputed invariant. These clauses then help to guide the path construction towards the initial state. Adding invariants in the forward direction does not make sense for PDRplan, because all the generated states are reachable from the initial state and, therefore, satisfy the invariant automatically.[12]

We used the same invariant generation algorithm as in PDRplan to also enhance the encodings `seq` and `par` for the run of `minireachIC3`. It turned out that in case `minireachIC3` invariants slightly help even in the forward direction.[13] We note that binary clause invariants are also explicitly included in the Mp encoding and implicitly present in the SASE encoding, which relies on the SAS^+ planning formalism (Bäckström and Nebel, 1995) to which a STRIPS problem is converted with the help of invariants (Helmert, 2009).

Detecting auxiliary transition variables

It is essential for a good performance of `minireachIC3` combined with encodings that the algorithm does not make decisions prematurely.

Example 6.3. Consider a run of the algorithm in the forward direction with the encoding $\mathcal{S}^{seq}_{\mathcal{P}}$. Because in this encoding the action variables A occur in the unprimed part of the transition clauses T^{seq}, any given state s, being a valuation over $\Sigma = X \cup A$, already stores the information about the action that will be applied next and, therefore, fully determines the value of the state variables X' of its successor. As a result, contrary to the intuition, the evaluation of the extension query

$$SAT?[\, Lits(s) \wedge T^{seq} \wedge (L)' \,]$$

does *not* boil down to choosing an action applicable in the state $(s \restriction X)$ of the original planning task, such that the successor state would satisfy the clauses of L, but instead involves choosing an action to be applied in the already determined successor such that the successor (as a valuation over X') and the chosen action (as a valuation over A') together satisfy the clauses from $(L)'$, which, in general, span the whole signature Σ'. We can see that, in some sense, all the decisions are made one step too early.

We observed a marked improvement in the performance of `minireachIC3` combined with encodings when we extended the tool with a preprocessing step that detects *auxiliary transition variables* in the unprimed part of the transition clauses and re-encodes

[12] In theory, there is a corresponding notion of a backward invariant, a property of the goal states preserved by traversing the transitions backwards. Symmetrically, backward invariants could be used to enhance the performance of forward PDR. In practice, however, while standard invariants are typically very useful, there is rarely a non-trivial binary clause backward invariant in the planning benchmarks.

[13] This can be explained by observing that the SAT solver does not necessarily construct the successor state by first choosing an action (or a set of actions, in the case of `par`), which would then fully determine the successor. When it starts by deciding on the state variables of the successor, invariants become useful.

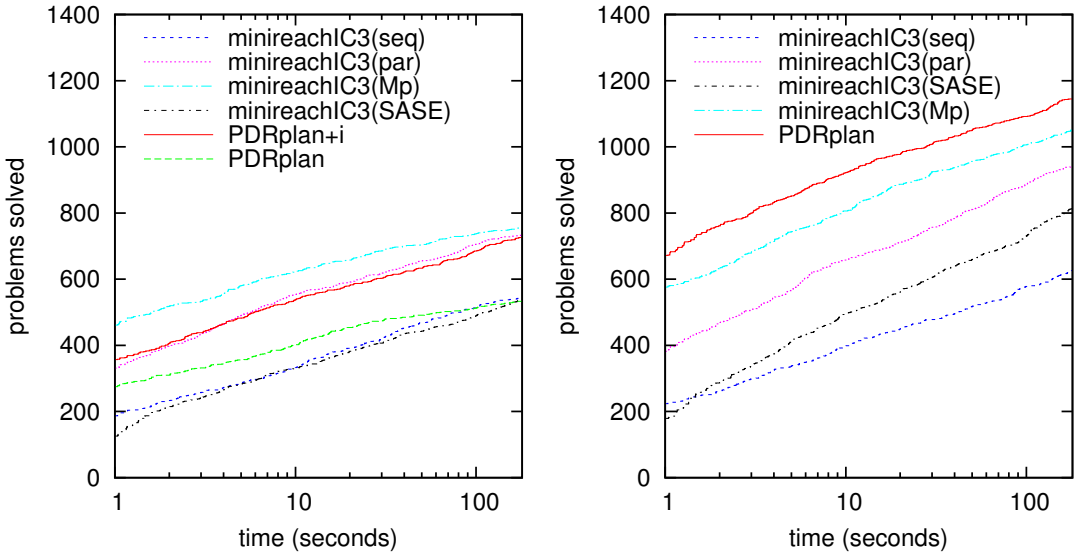


Figure 6.1: Comparing PDRplan to `minireachIC3` combined with encodings. Number of problems solved within the given time limit is shown, separately for the backward direction (left) and the forward direction (right).

them into the primed part in order to avoid committing to decisions prematurely as demonstrated in the example above. Formally, given an STS $\mathcal{S} = (\Sigma, I, G, T)$, auxiliary transition variables Aux are those variables of Σ that do not appear in I or G and, when primed, are not shared by T and T'. This means that

$$Aux' = \Sigma' \setminus (\mathit{Vars}(I') \cup \mathit{Vars}(G') \cup (\mathit{Vars}(T) \cap \mathit{Vars}(T'))).$$

The action variables A of the $\mathcal{S}^{seq}_{\mathcal{P}}$ encoding are an example of auxiliary transition variables. For every transition clause $C \in T$, the preprocessing step identifies literals $l \in C$ such that $\mathit{Vars}(l) \in Aux$ and turns each such l into l'. The soundness of the transformation is easy to establish.

Results of the experiment

The results of the first experiment can be found in Figure 6.1. There are several observations to be made. The forward direction is generally more successful than the backward. Within the time limit of 180 seconds, each of the five systems solves more problems in the forward direction than in the backward direction. We see that in the backward direction, invariants help to improve the performance of PDRplan. Nevertheless, within that direction `minireachIC3` combined with the Mp encoding is more successful. The most successful system is PDRplan in the forward direction. It solves 8.6 percent more problems than the second best system, `minireachIC3` combined with the Mp encoding in the forward direction. Although we do not consider these results as a definitive answer

to the "PDRplan vs. encodings" question,[14] we decided to only focus on PDRplan in the forward direction for (most of) the subsequent experiments.

The overall trends captured by Figure 6.1 are most of the time respected when the comparison is performed on level of individual problem domains (comparing the number of problems solved in 180 seconds), nevertheless there are some notable exceptions worth mentioning.

- On the LOGISTICS domain PDRplan behaves better in the backward direction and without invariants. The best system on this domain, however, is `minireachIC3` with Mp encoding in the forward direction.
- The relatively difficult 2011-BARMAN domain is almost fully solved (19 out of 20 problems) by PDRplan in the backward direction with invariants. The second best system on this domain is `minireachIC3` with `par` encoding in the backward direction with only 7 problems solved.
- The following are among the domains where PDRplan is not the best system: 1998-MYSTERY (`minireachIC3` with SASE and Mp encodings in the forward direction both solve 5 problems more), 2004-PHILOSOPHERS (18 more problems solved by `minireachIC3` both with `par` and Mp encodings in the forward direction), and 2011-VISITALL (`minireachIC3` with Mp encoding solves 4 more problems).
- There are several domains where `minireachIC3` with Mp encoding is better in the backward direction than in the forward direction. The difference is most pronounced on 2006-OPENSTACKS, and 2011-FLOORTILE.

The observation of the last point is in accord with how the Mp encoding is used with the Mp planner itself. What Rintanen (2012) describes is effectively a depth-first backward chaining planning algorithm inside the SAT-solving framework. This can be seen to be very close to backward PDR when coupled with the same encoding. We hypothesize that the suitability of the Mp encoding for the backward direction of search emerges also with PDR.

6.4.3 Tuning PDRplan

In the second experiment (see Figure 6.2) we focused on several features of the standard PDR and tried to established their importance for solving planning problems. We used PDRplan in the forward direction and 180 seconds time limit. We measured the effect of each feature separately with the reference configuration denoted as default. This is the same configuration as the one used in the previous experiment.

Explicit (inductive) reason minimization

By explicit minimization we mean the optional stage three of the reason computation in the `extend` procedure, which can be enhanced by induction as described in Section 6.3.2.

[14] For instance, replacing Minisat in `minireachIC3` by a more recent and more efficient SAT solver could change the picture to a certain degree.

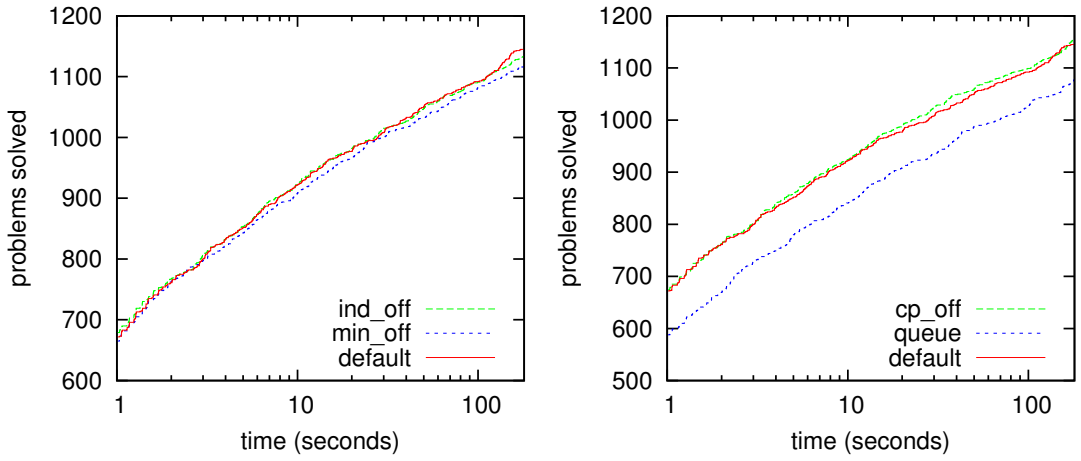


Figure 6.2: Tuning PDRplan. The effect of explicit (inductive) reason minimization (left), and clause propagation and the queue tie-breaking strategy (right).

In Figure 6.2 (left) we compare the performance of the default configuration, which relies on the inductive version of reason minimization (Algorithm 6.2), to configuration ind_off, which does not use induction and implements minimization as described in Algorithm 6.1, and to configuration min_off, which skips the optional stage three altogether.

We can see that while the positive effect of explicit minimization is slight but consistent along the time axis, induction only starts to pay off on the global scale when the time limit exceeds 100 seconds. At the 180 seconds mark the ind_off configuration solved 1.0 percent fewer and the min_off configuration 2.4 percent fewer problems than default.

Per domain view reveals that induction is especially important for the success on 2000-BLOCKS, 2004-PHILOSOPHERS, and 2008-CYBER-SECURITY. On domains such as 2002-ZENOTRAVEL or 2008-TRANSPORT it is better to turn minimization off completely, and there are also domains, such as 1998-MYSTERY or 2006-TRUCKS, where it pays off to minimize, but not inductively. In the last two categories, however, the difference is never by more than a problem or two per domain and thus it could be potentially equalized within a higher time limit.

Interestingly, out of the total of 1561 problems, the execution of default and ind_off diverged only on 145 problems.[15] This means that on most of the problems induction does not help to minimize reasons beyond what can be achieved with non-inductive minimization. To give another statistics, we note that over the whole problem set during a call to the `extend` procedure inductive minimization removes 1.49 literals and computes a reason with 50.60 literals on average. The non-inductive minimization in min_off removes 1.44 literals and generates a reason with 51.22 literals on average.

[15] By either generating a different number of obligations before a successful termination or differing in whether they successfully terminated at all before the 180 seconds mark.

Clause propagation

In Figure 6.2 (right) we can compare the default configuration to a configuration in which clause propagation has been turned off (cp_off). We see that clause propagation slows PDRplan slightly down without any clear benefit before the 180 seconds mark. Although a later independent experiment with a 1800 second time limit showed that clause propagation can be useful on planning problems, it is questionable whether the effect justifies the relatively high effort connected with implementing the technique.[16]

A closer look reveals that only on 28 percent of the tested problems a clause was successfully pushed forward during the 180 seconds bounded runs. This may seem to be in contrast with the experience from hardware model checking where clause propagation plays a key role. Its main effect there, however, lies in speeding up the occurrence of layer repetition on the unsatisfiable problems. Since more than 99 percent of our planning benchmarks are satisfiable, this role of clause propagation cannot be demonstrated. In fact, these results are in accord with those obtained with PDR on satisfiable hardware benchmarks in the forward direction (recall Figure 5.11 on page 159).

Stack vs. queue tie-breaking

Here we evaluate the effect of the strategy for breaking ties during popping obligations from the set Q (recall Remark 5.2). The stack strategy used in the default configuration is compared to the curve of the queue strategy in Figure 6.2 (right). The queue strategy solves about 5.9 percent fewer problems in total. However, there are 18 problems solved by the queue strategy only (and 85 problems solved only by the stack strategy). The most interesting observations on the per domain scope are probably

- 59 problems (out of 60) from the 2000-BLOCKS domain solved by the stack strategy compared to only 36 solved by the queue strategy, and
- 2 problems (out of 20) from the 2011-BARMAN domain solved by the queue strategy compared to 0 problems solved by the stack strategy.

Preferring to explore longer paths before short ones has the unpleasant side effect that also the plans discovered by the stack strategy tend to be longer. Measured over the 1055 problems solved by both strategies, the plans generated by the stack strategy are on average 24 percent longer.

A more detailed discussion on the topic of plan quality is postponed till Section 6.4.6.

6.4.4 Improving PDRplan

The purpose of the third experiment was to evaluate the three improvements proposed in Section 6.3.5. These were successively: 1) lazy false clause computation (lfcc), 2) the sidestepping technique (side), and 3) keeping obligations between iterations (keep). Figure 6.3 displays the effect of progressively enabling the three techniques in the presented

[16] In the final comparison to other planners (see Section 6.4.5) clause propagation is responsible for 6 additional problems scored by PDRplan.

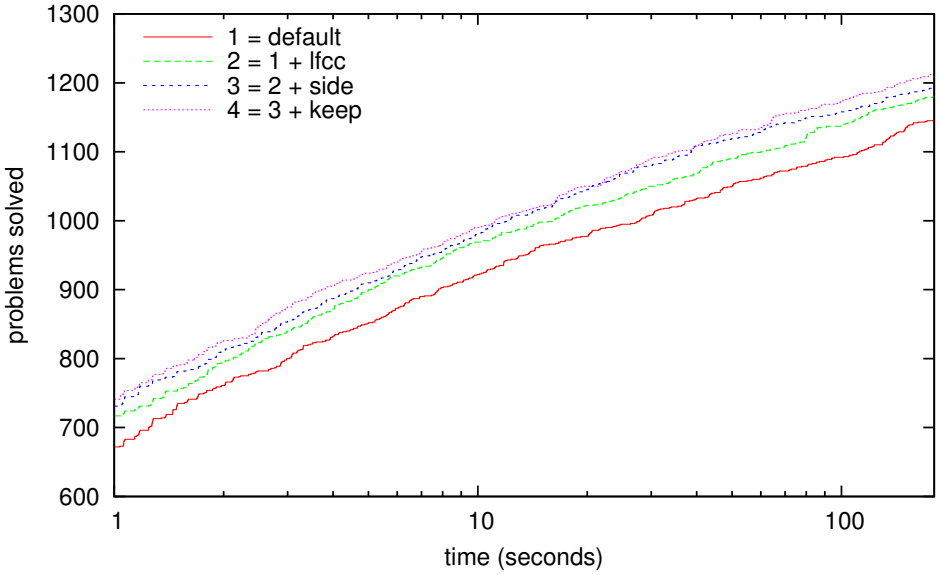


Figure 6.3: Improving PDRplan. The default configuration is progressively extended by turning on three different techniques.

Table 6.1: Number of problems solved within 180 seconds (total). The difference (delta) between two successive configurations decomposed into additionally solved problems (gained) and problems only solved without an improvement (lost).

configuration	total	delta	gained	lost
1 = default	1145	–	–	–
2 = 1 + lfcc	1180	35	55	20
3 = 2 + side	1195	15	67	52
4 = 3 + keep	1212	17	42	25

order. We see that to varying degrees each technique represents an improvement and each successive configuration solves more problems.

A different perspective is provided by Table 6.1 which also reveals how many problems were uniquely solved by only one of the two successive configurations. It shows that none of the improvements are unambiguous across the whole problem set and that there are exceptions to the prevailing trends.

These can be best highlighted on the level of individual domains. For instance, the number of solved problems drops on 2000-BLOCKS and 2008-CYBER-SECURITY with lazy false clause computation (configuration 2), but it is improved again by the subsequently enabled techniques. Sidestepping (configuration 3) makes the performance worse on 2002-DRIVERLOG, 2004-SATELLITE, or 2008-CYBER-SECURITY. On the other hand, the technique represents a huge improvement on 2004-OPTICAL-TELEGRAPH domain (from 2 to all 14 problems solved) and on 2004-PHILOSOPHERS (from 11

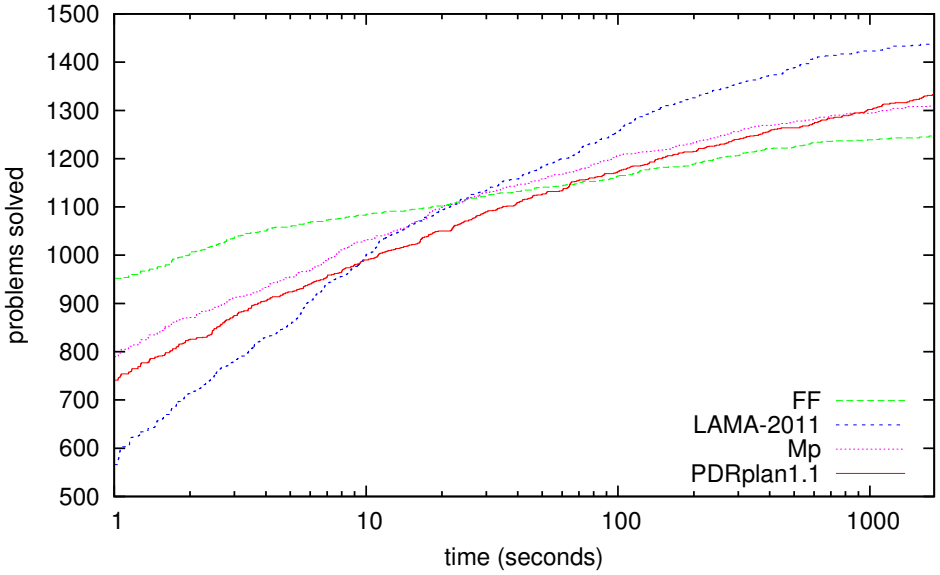


Figure 6.4: Comparing the final version of PDRplan to other planners. Showing the number of problems solved within the given time limit.

to all 29 problems solved). Finally, keeping the obligations (configuration 4) is detrimental to the performance on the 2011-FLOORTILE domain (the number of solved problems drops from 19 to 13), but the technique, for example, helps to "recover" the 2008-CYBER-SECURITY problems that were "lost" due to sidestepping.

6.4.5 Comparing to other planners

We compared the improved PDRplan – the configuration 4 from the previous experiment denoted here PDRplan1.1 – to the following planners:

- The planner FF (Hoffmann and Nebel, 2001) as a baseline representative of heuristic search (Bonet and Geffner, 2001) planners. We used version 2.3, but enhanced its input module to make it cope with the large problems of the more recent IPC domains. The default parameters for FF have been used.

- The planner Fast Downward (Helmert, 2006), the current state-of-the-art heuristic search planner. We used the configuration LAMA-2011 (Richter and Westphal, 2010), the winner of the satisficing track of IPC 2011.[17]

- The Mp planner (Rintanen, 2012), probably the current best representative of the planning as satisfiability approach (Kautz and Selman, 1996). We used version 0.99999 with default parameters.

[17]The winner of the latest IPC 2014, a followup event to IPC 2011, was a portfolio system IBaCoP relying on LAMA-2011 as one of its underlying planners (IPC, 2014).

For this experiment the time limit was increased to 1800 seconds.

The overall performance of the planners can be seen in Figure 6.4. The planner FF has a very fast startup and solves the most problems (952) within one second. However, FF is the worst of the planners to make use of the additional time and solves the fewest problems (1247) in total. On the opposite side stands LAMA-2011 with the slowest startup (566 problems within one second), but with the best total (1437). PDRplan1.1 and Mp are close to each other in performance both at the beginning – PDRplan1.1 solves 741 and Mp 790 problems within one second – and at the end – in total PDRplan1.1 solves 1333 problems gaining a slight edge over Mp with 1310 problems solved.

Table 6.2 shows a domain-by-domain decomposition of the results. We can see that there are several domains where PDRplan1.1 solved the most problems of the four planners: the 2000-BLOCKS, 2002-FREECELL, 2004-PIPESWORLD-NOTANKAGE, and 2006-TRUCKS domains. The domains 2004-PHILOSOPHERS, 2006-PATHWAYS, and 2006-STORAGE were completely solved by only PDRplan1.1 and Mp. On the other hand, a comparatively poor performance of PDRplan1.1 can be observed on the 1998-LOGISTICS and 1998-MPRIME domains, and also on the 2011-PARKING (shared with FF) and 2008+2011-SOKOBAN (shared with Mp) domains.

6.4.6 Plan quality

IPC 2008 (Helmert et al., 2008) introduced a criterion for measuring planner performance which takes into account the quality of the obtained plans. For every problem solved, a planner aggregates a *score* computed as the ratio c^*/c, where c is the cost[18] of the returned plan and c^* the cost of the best known plan (either a plan computed beforehand by the competition organizers or the best plan found by any of the participating systems). When viewing the results of the previous experiment through the lenses of this criterion, one discovers that PDRplan1.1 drops from the second place to the last.

We reviewed all the previously discussed features and improvements and discovered that the configuration of PDRplan1.1 is not the best possible with respect to plan quality. In particular, by switching to the queue tie-breaking strategy (we denote the respective configuration PDRplan1.1+queue) the aggregated score of the planner improves. A slight improvement can also be observed when the lazy false clause computation is turned off in PDRplan1.1. Interestingly, doing both changes at once does not bring a combined benefit.[19]

Figure 6.5 shows the aggregated scores for the runs of the previous experiment together with a run of PDRplan1.1+queue.[20] Although PDRplan1.1+queue solves only 1263 problems in 1800 seconds (compared to 1333 solved by PDRplan1.1), it aggregates a score of 1141.1 points while PDRplan1.1 only reaches 1041.4. This means the former

[18] As mentioned before, we did not consider action costs in our experiments, so a cost of a plan is simply equal to its length.

[19] It seems that the already "carefully advancing" PDRplan1.1+queue benefits from the speed provided by lazy false clause computation, whereas with the stack strategy it helps to wait for the more precise reasons (having lfcc turned off) that will not allow the planner to search too deep too often.

[20] The reference values for the best known cost c^* were collected just from the runs in the figure.

Table 6.2: Number of problems solved within 1800 seconds, grouped by domain. We highlighted those entries where PDRplan1.1 solves the most problems or shares the first place with one other planner. To save space the entries of IPC 2008 domains recurring later in IPC 2011 are merged with the respective entries of IPC 2011.

	size	PDRplan1.1	FF	LAMA-2011	Mp
1998-GRID	5	5	5	5	5
1998-GRIPPER	20	20	20	20	20
1998-LOGISTICS	35	18	35	35	32
1998-MPRIME	35	25	34	35	34
1998-MYSTERY	30	19	18	23	19
2000-BLOCKS	60	**60**	48	55	46
2000-ELEVATOR	150	150	150	150	150
2000-LOGISTICS	36	36	36	36	36
2000-FREECELL	60	57	60	60	44
2002-DEPOTS	22	21	21	22	22
2002-DRIVERLOG	20	18	18	20	20
2002-ZENOTRAVEL	19	19	19	19	19
2002-FREECELL	20	**20**	19	19	15
2004-AIRPORT	50	40	38	33	49
2004-PIPESWORLD-NOTANKAGE	50	**45**	32	44	42
2004-PIPESWORLD-TANKAGE	50	37	17	42	38
2004-OPTICAL-TELEGRAPH	14	14	14	14	14
2004-PHILOSOPHERS	29	**29**	14	13	**29**
2004-PSR	50	50	42	50	50
2004-SATELLITE	36	28	34	36	35
2006-OPENSTACKS	30	30	30	30	19
2006-PATHWAYS	30	**30**	20	29	**30**
2006-PIPESWORLD	50	32	21	40	25
2006-ROVERS	40	39	40	40	40
2006-STORAGE	30	**30**	18	19	**30**
2006-TPP	30	30	28	30	30
2006-TRUCKS	30	**27**	12	15	19
2008-CYBER-SECURITY	30	30	4	30	30
2011-BARMAN	20	6	0	20	8
2008+2011-ELEVATORS	50	40	50	50	50
2011-FLOORTILE	20	14	10	6	20
2011-NOMYSTERY	20	14	7	10	19
2008+2011-OPENSTACKS	50	49	50	50	18
2008+2011-PARCPRINTER	50	50	50	50	50
2011-PARKING	20	8	7	20	20
2008+2011-PEGSOL	50	50	50	50	50
2008+2011-SCANALYZER	50	46	44	50	48
2008+2011-SOKOBAN	50	11	40	48	9
2008+2011-TRANSPORT	50	27	38	49	26
2011-VISITALL	20	9	4	20	0
2008+2011-WOODWORKING	50	50	50	50	50
TOTAL	1561	1333	1247	1437	1310

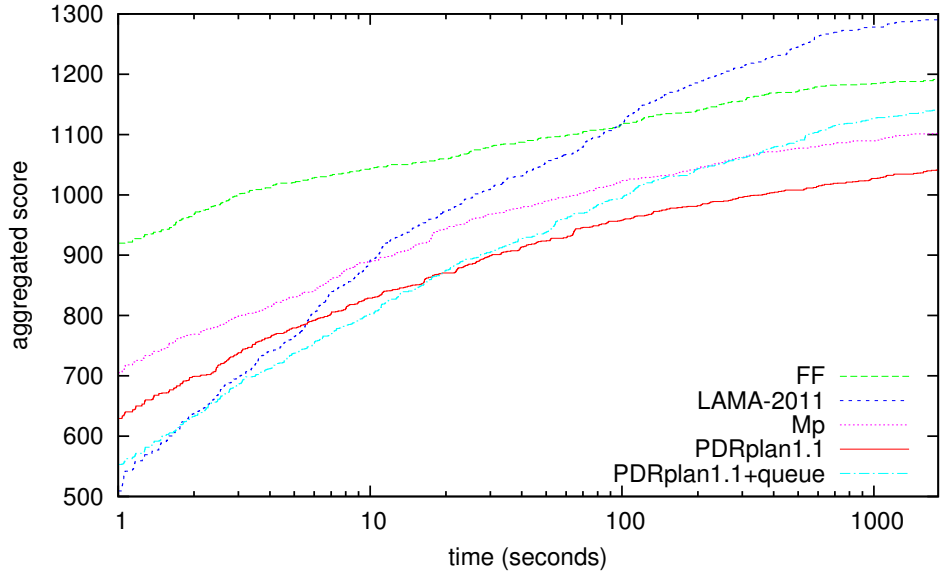


Figure 6.5: Comparing the planners with respect to plan quality. Showing the score aggregated by each planner within the given time limit.

configuration catches up with Mp, which aggregates 1102.7 points.

We note that these statistics should be taken with a grain of salt, because they only provide a "plan quality view" on the satisficing runs of the planners. None of the systems was explicitly attempting to find short plans or making use of the fact that the time limit is 1800 seconds. Moreover, even in such a setting the plan quality can typically be improved afterwards by a post-processing of the discovered plans (Balyo and Chrpa, 2014). We later incorporated the polynomial Action Elimination algorithm (Nakhost and Müller, 2010) as a plan post-processor into PDRplan1.1 and we were able to improve its aggregated score by 7.0 percent. A more thorough investigation of the quality of plans produced by PDRplan, as well as by PDR in general, is left for future work.

6.4.7 Anytime PDR and optimal planning

Recall that PDR can be adjusted to perform optimal planning by turning off the obligation rescheduling technique (and sidestepping).[21] Alternatively, we can modify PDR to continue the computation after a first plan is found, but afterwards only reschedule obligations that can be part of an improving plan.[22] This "anytime version" of PDR progressively reports on better and better solutions until finally terminating with a guarantee that the last reported plan is an optimal one. This happens when it reaches an

[21] PDR then looks for minimal length witnessing paths with respect to the encoded transition relation T. Using an encoding with sequential plan semantics (as implicitly done by PDRplan) ensures optimizing the number of actions of the resulting plan.

[22] Formally, we keep an obligation (s, i) if the length of the path from the initial state s_I to s plus the value of the index i does not exceed the length of the best plan found so far.

iteration i equaling the length of the best discovered plan.

In this experiment, we focused on optimal planning with respect to the sequential plan semantics.[23] We compared the performance of the anytime version of PDRplan1.1 (counting only solutions provably shown to be optimal) to BJOLP (Domshlak et al., 2011), an optimizing version of Fast Downward, and to an optimizing configuration of Mp.[24] Ordering the planners by the number of problems optimally solved within 1800 seconds we obtain:

1. BJOLP with 668 problems solved,
2. PDRplan1.1-anytime with 360 problems solved, and
3. optimizing Mp with 325 problems solved.

This order is preserved on the level of individual domains, except for several domains where Mp does not end up last. Mp solves optimally the most problems from 1998-MYSTERY, 2000-BLOCKS, 2008-CYBER-SECURITY, and also from 1998-MPRIME. The margin is exceptionally pronounced on the last domain, where Mp solves 32 out of 35 problems, while BJOLP solves 21 and PDRplan1.1-anytime only 20 problems.

6.4.8 Detecting unsatisfiable problems

Although the main focus of the planning community, as reflected by the International Planning Competition, has traditionally been on satisfiable problems only, more recently, the importance of detecting unsatisfiable instances is getting recognized and addressed (Bäckström et al., 2013; Hoffmann et al., 2014). According to the experience from hardware model checking, PDR should be particularly strong at detecting unsatisfiable instances. In our last experiment, we tried to established whether this also holds in planning.

As the test problems, we used a collection by Hoffmann et al. (2014) consisting of 8 domains and a total of 183 unsatisfiable benchmarks. Table 6.3 shows domain-by-domain coverage results (for a time limit of 1800 seconds) of the following configurations of PDR:

- PDRplan, in the same configuration as in the first experiment (Section 6.4.2), both in the forward (`fwd`) and backward (`bwd`) direction.
- `minireachIC3` combined with the `par` encoding (with invariants), also in both directions (`fwd`, `bwd`), and, in the backward direction, also with the inductive minimization replaced by the non-inductive version (`noind`), and, independently, with the clause propagation turned off (`nocp`).

[23] This choice ruled out systems like SatPlan (Kautz et al., 2006) or SASE (Huang et al., 2012) from the comparison, because they only optimize with respect to the parallel plan semantics.

[24] It uses a sequential encoding (option `-P 0`), does not skip any horizon length (option `-S 1`), and evaluates a single horizon length at a time (option `-A 1`).

Table 6.3: Unsatisfiable benchmarks. Number of problems solved within 1800 seconds, grouped by domain. Best scores per domain are typeset in bold.

	size	PDRplan		`minireachIC3` with `par`					M&S	
		`fwd`	`bwd`	`fwd`	`bwd`	`noind`	`nocp`	`blind`	`cf1`	`cf2`
3UNSAT	25	10	10	11	**15**	11	5	**15**	**15**	**15**
Bottleneck	30	19	24	**25**	23	20	22	10	10	21
Mystery	9	4	**9**	**9**	**9**	**9**	**9**	2	**9**	6
UnsNoMystery	25	12	11	3	13	13	6	0	**25**	**25**
UnsPegsol	24	14	8	14	8	8	4	**24**	**24**	**24**
UnsRovers	25	11	11	11	**20**	15	12	0	17	9
UnsTiles	20	0	0	0	0	0	0	**10**	**10**	**10**
UnsTPP	25	5	6	4	6	3	3	5	**9**	**9**
Total	183	75	79	66	94	79	61	66	119	119

In addition, Table 6.3 also contains entries adopted from the work of Hoffmann et al. (2014). These belong to the Fast Downward planner equipped with three different heuristics:

- Configuration `blind` uses a heuristic which returns 0 on goal states and 1 elsewhere – it essentially proves unsatisfiability by enumerating all the reachable states.
- Configuration `cf1` and `cf2` each use a version of a *merge-and-shrink* (M&S) heuristic (Helmert et al., 2007), specifically adapted for detecting unsatisfiable problems (Hoffmann et al., 2014). These were the two best performing configurations in the experiment of Hoffmann et al. (2014).

We note that Hoffmann et al. (2014) also used a time limit of 1800 seconds, but ran their experiment on 2.20 GHz Intel E5-2660 machines with a 4 GB memory limit. This means that the last three configurations could be potentially solve more problems in our setup.

Results of the experiment

When comparing the various configurations of PDR, we can see that the backward direction is generally more successful than the forward, although not consistently across all the domains. Interestingly, `minireachIC3` with the `par` encoding in the backward direction solves more problems than PDRplan. In fact, a preliminary test with a lower time limit showed that on these benchmarks this configuration is the strongest of all those considered in our first experiment (Section 6.4.2). Finally, we can also see that both induction and clause propagation are consistently helpful for solving unsatisfiable problems.

PDR does not come out as a winner from the comparison to the heuristic approach of Hoffmann et al. (2014), although it is able to solve the most problems on four domains. On two other domains, however, PDR is even dominated by blind search, i.e. by a

simple state space enumeration. It seems that more benchmarks will be needed to establish which of the two approaches is generally more successful at detecting unsatisfiable planning problems.

Performance on UnsTiles

PDR is particularly bad at "enumerating states" when there is little possibility to generalize from the encountered ones. This is manifested most clearly on the UnsTiles domain, from which PDR could not solve a single problem within the given time limit. The domain represents the well known sliding puzzle and contains 10 problems with 8 tiles in a 3×3 grid and 10 problems with 11 tiles in a 3×4 rectangular grid.[25] We ran PDRplan in the forward direction to the end on the one of the smaller, 3×3 instances. It took about a day to complete, processed 701704 obligations and terminated when all the clauses from layer 11, in total 181440 clauses, where pushed to layer 12 during the clause propagation phase of iteration 11.

Notice that $181440 = 9!/2$ is half the size of the state space. By the classical result of Johnson and Story (1879), the state space of the sliding puzzle decomposes into exactly two connected components depending on the value of a certain *parity* function defined on the states. Unsatisfiable instances are those where the parity of the initial state and the goal state are different. Because the state space consists of just of two components, on a unsatisfiable instance PDR must converge (with the repeating layer) to a CNF description of the component containing the goal state. As we can see, this description is as large (in the number of clauses) as the component itself (in the number of states), and thus on the sliding puzzle PDR does not benefit at all from the symbolic representation via CNF.

6.4.9 Summary

Let us summarize the empirical findings obtained in this section. We state them as general claims while keeping in mind that they are, in fact, derived from the performance on two particular benchmark sets: the main set of 1561 mostly satisfiable IPC problems and the set of 183 unsatisfiable problems used in the last experiment.

- When planning with PDR it pays off to look for a plan from the initial state towards the goal and not vice versa. In other words, progression is preferable to regression in PDR. This holds even when invariants are employed, which help to improve the performance of regression considerably.

  Unsatisfiable instances, however, are typically better detected via regression.

- On satisfiable problems the SAT-solver-free variant of PDR with planning-specific `extend` procedure (as described in Section 6.3) is generally more successful than the standard version of the algorithm combined with various encodings.

[25] Most famous is the 15 tiles puzzle on a 4×4 grid (Wikipedia, 2014).

- Neither clause propagation nor inductive minimization, two techniques which are normally deemed essential for the performance of PDR, are very helpful on satisfiable planning problems. The techniques are, however, useful for detecting unsatisfiability.

- There are various ways of tuning PDR and improving its performance for planning. We tried to identify a configuration of the algorithm that would be most successful in our setup and later used it in PDRplan for a comparison with other planners. For all the techniques that turned out to be an improvement on average, however, there were exceptions in the form of individual problems or domains were performance degraded. These represent an interesting opportunity for future investigations (see Section 6.6 for more details).[26]

- When compared to other planners PDRplan shows respectable performance. In fact, its performance is comparable to or even slightly better than that of the planner Mp, a state-of-the-art representative of the planning as satisfiability approach. It also solves the most problems of all the tested planners on several domains. Although PDRplan does not reach the score of LAMA-2011, the presented results are quite encouraging, especially given that PDR is a relatively young algorithm with a potential for further improvements (again, see also Section 6.6).

- When plan quality is more important than just the number of problems solved, it pays off to switch from the stack to the queue tie-breaking strategy in PDR. Such a configuration is then able to keep up with and improve upon the performance of Mp with respect to the plan quality metric based on aggregated score.

  Another option for improving plan quality is to employ a post-processing step which attempts to remove redundant actions from the generated plan (Balyo and Chrpa, 2014).

- PDR can be easily modified to look for increasingly better solutions when given sufficient time and to eventually terminate with an optimality guarantee (with respect to plan length). Although LAMA-2011 is much more successful in finding optimal plans, the fact the PDRplan's "natural" encoding follows the sequential plan semantics could be the reason why PDRplan scores higher than Mp in this respect.

6.5 Related work: Graphplan

We have argued in the introductory section of this chapter that PDR is an algorithm closely related to the planning as satisfiability approach, although with the planning-specific implementation of the `extend` procedure no explicit encoding is present. We also

[26] On the one hand, by looking at the problems where a particular technique leads to a poor performance, we can identify its weak points and attempt to improve the technique. On the other hand, instead of relying on an overall best configuration, we can also try to decide, prior to running the algorithm itself, on a promising set of enabled features for a given problem based on the problem's characteristics.

highlighted the connection to heuristic search planning, with the direct correspondence on the side of explicitly explored reachable states and a little more subtle one on the side of the guiding layers, which can be seen as a continually refined admissible heuristic estimator. What we would like to discuss here is a perhaps surprising relation of PDR to the well-known Graphplan planning algorithm by Blum and Furst (1997).

The main data structure of Graphplan is a *planning graph*, a layered structure for compressed representation of reachability information about the given problem. The individual layers of the graph over-approximate the set of states reachable by a given number of sets of parallel actions and are computed incrementally by propagation of so called *exclusion relations* between actions and state variables. The planning graph is searched for a plan by a backward-chaining strategy, starting from the goal set and regressing it, in the sense of the parallel plan semantics, to subgoals that do not violate the exclusions of the respective layer. Candidate (sub)goal sets shown not to lead to a plan within a specific number of steps are *memoized* to avoid repeating the same work in the future.

As already shown by Rintanen (2008b) the exclusion relations of the planning graph are equivalent to binary clause representation of k-step reachability information. This means they could be represented inside PDR as binary clauses in the respective layers. We claim additionally that also the memoized goal sets could be stored as layer clauses at the respective position: the clause being simply the negation of the conjunctive description of the goal set. With these two observations in mind, we can state that

> Graphplan is essentially a version of PDR with a specific implementation of the `extend` procedure based on the parallel plan semantics.

This correspondence allows us to highlight some other differences between the two algorithms beyond the preferred semantics of the emulated encoding.

- While the planning graph is built systematically by Graphplan and search for a plan is only started (resumed) when a full new layer has been computed, in PDR the layer construction is lazy, being triggered by unsuccessful path extensions.

  Goal set memoization in Graphplan, however, follows the same lazy pattern.

- Graphplan does not attempt to reduce the size of a memoized goal set, so, apart from the binary clauses, it only deals with long clauses representing the negation of the goal set. Notice that this would in PDR correspond to returning the full reason set $Lits(s)$ after an unsuccessful extension of the state s.

  A *subset* memoization has later been proposed by Long and Fox (1999), which corresponds to finding smaller reason sets.

- Graphplan searches for a plan in the backward direction. In PDR, the direction can be changed, but forward is more successful.[27]

[27] Changing the search direction in Graphplan by running in on an inverted problem (see Section 6.3.4) is possible, but would likely lead to fewer problems solved. This is related to the already mentioned observation that there are very few problems with non-trivial backward invariants in the benchmark set.

- There is no equivalent to obligation rescheduling in Graphplan and so the algorithm always searches for optimal plans (with respect to the parallel plan semantics).

  The wavefront heuristic described by Long and Fox (1999) in their enhancement of Graphplan, however, seems to overcome this limitation, similarly to rescheduling.

The realization that PDR is related to Graphplan made us curious about the differences of the two algorithms in practice. We set up a small experiment where we compared PDR to a mature implementation of Graphplan within the planner IPP (Koehler, 1999). In order to bring PDR as close as possible to what Graphplan does, we represented it by `minireachIC3` combined with the simple parallel encoding $\mathcal{S}^{par}_{\mathcal{P}}$ (see Section 6.2.2) enhanced by the binary clause invariant (as explained in Section 6.4.2). We ran `minireachIC3` in the backward direction and with obligation rescheduling turned off, so, similarly to IPP, it was looking for optimal plans. When measuring the number of problems solved (out of the main problem set described in Section 6.4) within 180 seconds, we obtained 466 solved by IPP and 484 by our configuration of `minireachIC3`. It should be noted that IPP erroneously reports UNSAT for most of the problems from the PARCPRINTER and WOODWORKING domains and we counted this as failures. Because `minireachIC3`, on the other hand, solves most of the problems from these domains, its score should be lowered by 94 problems to obtain a "fair" comparison on the problem set which excludes these two domains.

Notice that the performance of IPP with 466 solved problems is quite low compared to the best configuration of PDRplan1.1, which solves 1212 problems within 180 seconds. This raises the question whether Graphplan could be improved by enhancing it with the obligation rescheduling trick. We were able to confirm this experimentally. A relatively straightforward modification of IPP which retries a candidate goal set at time $t+1$ after it has failed at time t was able to solve 676 problems.[28] Thus obligation rescheduling can be seen as an answer to the long standing question posed in the last remark of the original Graphplan paper by Blum and Furst (1997), i.e., as a way to trade plan quality for speed.

6.6 Discussion: A closer look at two domains

The fact that PDR maintains its reachability information organized in layers and uses the simple language of propositional clauses (CNF) to express the corresponding constraints often allows us to obtain additional insights on how the algorithm traverses the search space by inspecting the layers generated for concrete problems. This is especially rewarding in cases where PDR seems to be struggling with a relatively simple problem, as it often leads to a discovery of ideas for future improvements. In this section we take a closer look at the behavior of PDR on two simple domains. We conjecture that the algorithm could be improved by employing a more expressive constraint formalism than CNF.

[28] Also the performance of the corresponding configuration of `minireachIC3` goes up from the mentioned 484 to 733 solved problems within 180 seconds when obligation rescheduling is turned on.

1998-LOGISTICS

The task in the LOGISTICS domain is to transport packages between locations. Locations belong to cities and within a city trucks may be used to move packages with the help of the load-truck, drive-truck, and unload-truck actions. Additionally, some of the locations are designated as airports and airplanes may be used to transport packages between airports possibly across cities via the load-airplane, fly-airplane, and unload-airplane actions.

Although the LOGISTICS domain is generally considered to be a simple one, Table 6.2 (page 202) reveals relatively poor performance of PDRplan on LOGISTICS problems. Here are two of our initial findings from the inspection of the layer clauses generated by PDR, which shed some light on what is going on "under the hood".

- PDR often generates very long clauses.

  Because there are typically many distinct (although similar) ways to achieve a subgoal and all of them need to be taken into account, large reason sets are computed and subsequently long explaining clauses derived. For example, if a package needs to be transported from one city to another, any of the available airplanes can potentially be used for that purpose. We often encounter derived clauses like

  $$subg \vee at(apn_1, apt) \vee at(apn_2, apt) \vee \ldots \vee at(apn_n, apt) \tag{6.4}$$

  expressing that if the subgoal *subg* has not been reached yet, at least one of the available airplanes apn_i need to be present at the airport apt.

- PDR generates many similar clauses.

  Even if an action has more than one precondition false in the current state, at most one of these preconditions is reflected in the computed reason of an unsuccessful extension. Thus with many actions available for achieving a subgoal, sometimes many clauses are needed as PDR tries to find the right achieving action and satisfy all its preconditions.

  In addition to the above clause (6.4) we could see PDR subsequently derive the following clauses in the same layer:

  $$\begin{aligned} &subg \vee in(obj, apn_1) \vee at(apn_2, apt) \vee \ldots \vee at(apn_n, apt), \\ &subg \vee at(apn_1, apt) \vee in(obj, apn_2) \vee \ldots \vee at(apn_n, apt), \\ &\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\ &subg \vee at(apn_1, apt) \vee at(apn_2, apt) \vee \ldots \vee in(obj, apn_n). \end{aligned} \tag{6.5}$$

  Note that although the pattern indicates n different clauses, there are in the worst case 2^n clauses potentially derivable with the "arbitrary" choice between $at(apn_i, apt)$ and $in(obj, apn_i)$ for every i.

Although we have so far described PDR as an algorithm based on propositional logic, we believe it could be generalized to take advantage of first-order constraints. Consider

the clause (6.4) above. An equivalent first-order version (aware of the type *airplane*) would read

$$subg \ \vee \ \exists Apn \in airplane \,.\, at(Apn, apt),$$

which is much more succinct.[29] Moreover, it could potentially be derived by just analyzing the action schemes *unload-airplane*(Obj, Apn, Loc), ..., etc., instead of iterating through the much larger set of instantiated actions. Working out the missing details is an interesting direction for future research. An inspiration could be found in the work of Ranise (2013), whose setting of security policy analysis is very close to automated planning.

Another independent direction for enhancing the expressive power of the used constraints could be the introduction of "conjunctive literals". Notice that the set of clauses (6.5) is, in fact, subsumed by a single generalized clause

$$subg \vee \bigvee_{i=1}^{n} in(obj, apn_i) \wedge at(apn_i, apt),$$

where we allow conjunctions in place of single literals. In this envisioned generalization of PDR, such conjunctions would naturally come from the precondition sets of actions, and their use could help with solving, e.g., the LOGISTICS problems more efficiently. Of course, there are again details that would need to be worked out.

1998-GRIPPER

GRIPPER is a very simple domain which models a robot with two grippers trying to move balls from one room to another. This domain is fully solved by PDRplan in the default configuration. In fact, although the individual problems differ in size, PDRplan is able (thanks to obligation rescheduling) to solve all of them during iteration 3 of the main loop.[30] The reason for this seems to be the virtual independence of the individual goals, which can be considered one be one by PDR. We conjecture that the algorithm solves problems from the GRIPPER domain in polynomial time.

Despite the simplicity, GRIPPER is known to be difficult to solve optimally by heuristic search planners (see Helmert and Röger, 2008). This also holds for PDR, which exhibits exponential behavior when attempting to find a minimal length plan, i.e., when run with obligation rescheduling (and sidestepping) turned off. To demonstrate the reason, let us abstract and simplify GRIPPER a bit more and consider a domain in which the task is to achieve n independent goals from the set $\{g_1, \ldots, g_n\}$, such that achieving a particular goal is trivial, but the individual goals can only be achieved one by one.

On such a domain, PDR will eventually need to express via layer L_i that *at least* $(n - i)$ goals should already be achieved. Such a "counting" constraint has inherently large clausal description. Namely, the set L_i takes the form

$$\bigwedge g_{j_0} \vee \ldots \vee g_{j_i},$$

[29] Symbols starting with an uppercase letter, like Apn, stand for first-order variables.

[30] Other domains fully solved by PDRplan during a particular fixed iteration are 2002-ZENOTRAVEL (iteration 3), 2004-PHILOSOPHERS (iteration 6), and 2006,2008,2011-OPENSTACKS (iteration 4).

where the conjunction ranges over all $(i+1)$-element subsets $\{j_0, \ldots, j_i\}$ of $\{1, \ldots, n\}$. The size of the layer L_i is, therefore, proportional to the binomial coefficient $\binom{n}{i+1}$, which, in particular, means that the size of the layer $L_{\lfloor n/2 \rfloor}$ grows exponentially with n.

As already suggested by Helmert and Röger (2008) this phenomenon could be overcome by exploiting symmetries (Fox and Long, 1999) inherently present in the problem. This could be particularly rewarding in PDR, where the layer clauses (although derived as a response to unsuccessful extensions of arbitrary reachable states) logically depend only on the goal condition G, where the symmetries typically reside. Thus unlike Fox and Long (1999), who define symmetric objects to be those which are indistinguishable from one another in terms of their initial and goal configuration, one could with PDR use a stronger notion of symmetry derived from the goal condition only.

6.7 Conclusion

In this chapter we have examined PDR, a novel algorithm for analyzing reachability in symbolic transition systems, from the perspective of automated planning. Our main contribution lies in recognizing that a part of the algorithm's work normally delegated to a SAT solver can, in the context of planning, be implemented directly by a polynomial time procedure. We have experimentally confirmed that this modification, as well as several other proposed improvements, boost the performance of PDR on planning benchmarks. Our implementation of the algorithm called PDRplan was able to compete respectably with state-of-the-art planners, solving most problems on several domains.

Despite the already promising results, there is still room for further development. One direction is work on extending PDRplan towards richer planning formalisms. For example, we believe the `extend` procedure can be enhanced to cope with conditional effects of actions in a straightforward way. Efficiently dealing with action costs or domain axioms could turn out to be more difficult. Another promising direction is the idea to generalize PDR to a more expressive constraint language than CNF. While it is clear that stronger constraints imply better guidance towards the goal, devising an efficient method for combining new constraints from old ones is obviously a challenging task. It seems, however, that this "departure beyond the propositional clausal level" could have a simpler solution inside the planning-specific framework of the `extend` procedure than it, perhaps, has within the context of general purpose constraint solvers.

7 Conclusion

In this thesis, we have studied the following three main problems: 1) proving theorems in linear temporal logic, 2) verification of invariance (and safety) properties of hardware circuits, and 3) the problem of classical STRIPS planning. Although the problems come from different research fields, they are closely related. As we have shown, when properly encoded and normalized, these problems can be given a common simple representation based on *clausal propositional logic*. We initially referred to the representation as the temporal satisfiability task (TST) and later adopted the term symbolic transition system (STS) for the same syntactic object. This was to stress the change of the intended task from a one based on the Büchi condition to single time reachability. On the semantic side, the representation describes a graph-like structure – called the *semantic graph* or the state transition system – where traversing an edge corresponds to a single time step and each of the problems can be restated as a search for a certain finite or infinite path through this graph. In this sense, the problems can be jointly characterized as falling into the category of linear temporal reasoning.

When made explicit, the semantic graph may become exponentially larger than its symbolic description. Our strategy for avoiding this immediate blow-up was to employ logic, namely the resolution rule, to only manipulate the symbolic description and learn about the properties of the graph indirectly. Resolution emerged in several forms in this work. *Ordered* resolution constitutes the main inference rule of our calculus LPSup for LTL theorem proving. Resolution *guided by a partial model*, as formalized within the conflict driven clause learning paradigm, lies beyond the success of modern SAT solvers and we employed the idea in the design of our efficient LTL-proving algorithm LS4. *Exhaustive* resolution served us as a means for eliminating variables and clauses in LTL normal forms and thus provided a basis for a useful simplification procedure. We continued to rely on the SAT-solving technology and thus on resolution when we adapted LS4 to deciding single time reachability and obtained the algorithm Reach. Finally, although the main result of the previous chapter showed that in the context of STRIPS planning the SAT solver within the PDR algorithm can be replaced by a more efficient planning-specific procedure for computing the single step reachability queries, resolution was still indirectly employed to justify the replacement.

There is a close connection between the resolution rule on the syntax level and the *elimination of an existential quantifier* on the level of semantics. We could see this most clearly in Chapter 4 when dealing with variable elimination. From the perspective of satisfiability checking all variables of a formula are implicitly existentially quantified and to exhaustively resolve over a variable p is a way to obtain an equisatisfiable form in

which p does not occur any more:

$$\exists p\,(N_p \cup N_{\neg p} \cup N_0) \equiv (N_p \otimes_p N_{\neg p}) \cup N_0,$$

thus eliminating the existential quantifier $\exists p$. In the context of the semantic graph, we relied on the same principle and used resolution to compute the *pre-image operation* on a set of represented vertexes. To set up ordered resolution this way, the trick is to make the primed symbols Σ' corresponding to the target vertexes larger than the basic symbols Σ, which correspond to the source vertexes. Then the set of clauses from which the primed symbols have been resolved away represents those vertexes for which there exists a successor in the graph. We achieved a similar effect in the model guidance setting. By passing to the SAT solver a concrete source vertex encoded as a set of assumptions, one derives a new property of the pre-image in the form of a learned clause whenever no corresponding successor vertex can be found. The main difference between the saturation approach with ordered resolution and the model guidance setting is that the pre-image obtained in the first case is *precise*, whereas in the second case we only work with a continually refined *over-approximation*.

The reasoning power of the resolution rule in the semantic graph is inherently *local*, allowing us to directly derive only the relation between neighboring vertexes. One needs an additional *global* principle to decide a reachability problem in full, in particular to be able to infer from a series of bounded results of the form "there exists no path of length l" for concrete values $l = 0, \ldots, n$ that there is no path of any length. In this thesis, we found this additional principle in *the repetition detection and derivation replaying argument*. We showed that upon the detection of a specific repeating part in a bounded resolution proof, one can establish the ultimate non-reachability by induction. The key observation was that the piece of the proof between the repeating parts can be replicated arbitrary many times, giving us the potential to generate non-existence proofs for paths of any length. It seems that the need for an inductive argument is inherent, as similar global rules are present in all the related complete approaches we have studied.

Along with the theoretical work, we presented in this thesis a series of experiments whose aim was to evaluate the practical performance of the developed algorithms and to compare them to related approaches. Interpreting the results of such experiments can be tricky, because the individual algorithms and their implementations often have complementary strengths and weaknesses which, for example, makes picking a single winner difficult. We may, however, also use the results to identify interesting trends in the behavior of a particular algorithm which can trigger further development of the corresponding theory. An example can be found by looking at some of the common patterns in the behavior of PDR on the hardware benchmarks and in planning. If we compare the results of Chapters 5 and 6, we can, for instance, see that in both cases the forward direction of the search is on average more successful for discovering satisfiable benchmarks, whereas the unsatisfiable ones are better detected in the backward direction. Properly understanding this phenomenon and explaining it theoretically could represent an interesting direction for future research.

Another program for extending the work presented in this thesis would be to change the symbolic representation of the semantic graph and replace the propositional logic

foundation by a more expressive formalism. The most obvious candidate for such a formalism is first-order logic, possibly with equality, or its fragments. The corresponding problems that could be encoded into such generalized representation include proving theorems in first-order linear temporal logic or software model checking and have already been studied; to a certain degree in the first case (Degtyarev et al., 2006) and quite extensively in the second (see Jhala and Majumdar, 2009, for a survey). One needs to be aware that the greater expressibility is payed for by higher computational complexity of the corresponding problems or even their undecidability (Szalas and Holenderski, 1988). Nevertheless, the fact that our methods rely on resolution and indeed superposition, for which the lifting from the propositional basis to the first-order level is well understood (Bachmair and Ganzinger, 2001; Nieuwenhuis and Rubio, 2001), suggests that their generalization to the more expressive setting is possible, at least in some restricted form. Particularly promising for the extension appears to be, for instance, the decidable Bernays-Schönfinkel fragment of first-order logic.

Bibliography

Martín Abadi and Zohar Manna. Nonclausal temporal deduction. In Rohit Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1985. ISBN 3-540-15648-8.

Roberto Javier Asín Achá, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Practical algorithms for unsatisfiability proof and core generation in SAT solvers. *AI Commun.*, 23(2-3):145–157, 2010.

Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998. ISBN 978-0-521-45520-6.

Leo Bachmair and Harald Ganzinger. On restrictions of ordered paramodulation with simplification. In Mark E. Stickel, editor, *CADE*, volume 449 of *Lecture Notes in Computer Science*, pages 427–441. Springer, 1990. ISBN 3-540-52885-7.

Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994.

Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 19–99. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8.

Christer Bäckström and Bernhard Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11:625–656, 1995.

Christer Bäckström, Peter Jonsson, and Simon Ståhlberg. Fast detection of unsolvable planning instances using local consistency. In Malte Helmert and Gabriele Röger, editors, *SOCS*. AAAI Press, 2013. ISBN 978-1-57735-584-7.

Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008. ISBN 978-0-262-02649-9.

Tomas Balyo, Vojtech Bardiovsky, Filip Dvorak, and Dan Toropila. Freelunch planning library, 2012. Available at `http://ktiml.mff.cuni.cz/freelunch/`.

Tomáš Balyo and Lukáš Chrpa. Eliminating all redundant actions from plans using SAT and MaxSAT. In *ICAPS 2014 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, 2014.

Roberto J. Bayardo and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In Benjamin Kuipers and Bonnie L. Webber, editors, *AAAI/IAAI*, pages 203–208. AAAI Press / The MIT Press, 1997. ISBN 0-262-51095-2.

Sam Bayless. Implementation of PDR using "SAT modulo SAT". Web site, `https://bitbucket.org/sambayless/smspdr`, accessed 9/11/2013, 2013.

Sam Bayless, Celina G. Val, Thomas Ball, Holger H. Hoos, and Alan J. Hu. Efficient modular SAT solving for IC3. In *FMCAD*, pages 149–156. IEEE, 2013.

Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, 22:319–351, 2004.

Evert W. Beth. Semantic entailment and formal derivability. *Koninklijke Nederlandse Akademie van Wentenschappen, Proceedings of the Section of Sciences*, 18:309–342, 1955.

Armin Biere. Bounded model checking. In Biere et al. (2009), pages 457–481. ISBN 978-1-58603-929-5.

Armin Biere. AIGER. Web site, `http://fmv.jku.at/aiger/`, accessed January, 2012.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999. ISBN 3-540-65703-7.

Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electr. Notes Theor. Comput. Sci.*, 66(2):160–177, 2002.

Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, 2009. IOS Press. ISBN 978-1-58603-929-5.

Armin Biere, Keijo Heljanko, Martina Seidl, and Siert Wieringa. Hardware model checking competition 2012. Web site, `http://fmv.jku.at/hwmcc12/`, accessed 1/12/2013, 2012.

Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artif. Intell.*, 90(1-2):281–300, 1997.

Alexander Bolotov, Michael Fisher, and Clare Dixon. On the relationship between ω-automata and temporal logic normal forms. *J. Log. Comput.*, 12(4):561–581, 2002.

Blai Bonet and Hector Geffner. Planning as heuristic search. *Artif. Intell.*, 129(1-2): 5–33, 2001.

Aaron Bradley. Presonal communication, 2012.

Aaron Bradley. Implementation of IC3. Web site, `http://ecee.colorado.edu/~bradleya/ic3/`, accessed 9/11/2013, 2013.

Aaron Bradley. Reference implementation of IC3. Web site, `https://github.com/arbrad/IC3ref`, accessed 11/5/2014, 2014a.

Aaron Bradley. Presonal communication, 2014b.

Aaron R. Bradley. k-step relative inductive generalization. *CoRR*, abs/1003.3649, 2010.

Aaron R. Bradley. SAT-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011. ISBN 978-3-642-18274-7.

Aaron R. Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *FMCAD*, pages 173–180. IEEE Computer Society, 2007.

Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In Per Bjesse and Anna Slobodová, editors, *FMCAD*, pages 144–153. FMCAD Inc., 2011. ISBN 978-0-9835678-1-3.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2): 142–170, 1992.

Ana R. Cavalli and Luis Fariñas del Cerro. A decision method for linear temporal logic. In Robert E. Shostak, editor, *CADE*, volume 170 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 1984. ISBN 3-540-96022-8.

Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. Incremental formal verification of hardware. In Per Bjesse and Anna Slobodová, editors, *FMCAD*, pages 135–143. FMCAD Inc., 2011. ISBN 978-0-9835678-1-3.

Hana Chockler, Alexander Ivrii, and Arie Matsliah. Computing interpolants without proofs. In Armin Biere, Amir Nahir, and Tanja E. J. Vos, editors, *Haifa Verification Conference*, volume 7857 of *Lecture Notes in Computer Science*, pages 72–85. Springer, 2012. ISBN 978-3-642-39610-6.

Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2012. ISBN 978-3-642-31423-0.

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 Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002. ISBN 3-540-43997-8. Software available at `http://nusmv.fbk.eu/`.

Koen Claessen and Niklas Sörensson. A liveness checking algorithm that counts. In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 52–59. IEEE, 2012.

Koen Claessen, Niklas Eén, and Baruch Sterin. A circuit approach to LTL model checking. In *FMCAD*, pages 53–60. IEEE, 2013.

Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001. ISBN 978-0-262-03270-4.

Raphaël Clifford and Alexandru Popa. Maximum subset intersection. *Inf. Process. Lett.*, 111(7):323–325, 2011.

Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

Willam Criag. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.

Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

Anatoli Degtyarev, Michael Fisher, and Boris Konev. A simplified clausal resolution procedure for propositional linear-time temporal logic. In Uwe Egly and Christian G. Fermüller, editors, *TABLEAUX*, volume 2381 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 2002. ISBN 3-540-43929-3.

Anatoli Degtyarev, Michael Fisher, and Boris Konev. Monodic temporal resolution. *ACM Trans. Comput. Log.*, 7(1):108–150, 2006.

Clare Dixon. Search strategies for resolution in temporal logics. In Michael A. McRobbie and John K. Slaney, editors, *CADE*, volume 1104 of *Lecture Notes in Computer Science*, pages 673–687. Springer, 1996. ISBN 3-540-61511-3.

Clare Dixon. Temporal resolution using a breadth-first search algorithm. *Ann. Math. Artif. Intell.*, 22(1-2):87–115, 1998.

Carmel Domshlak, Malte Helmert, Erez Karpas, Emil Keyder, Silvia Richter, Gabriele Röger, Jendrik Seipp, and Matthias Westphal. BJOLP: The big joint optimal landmarks planner. In *Seventh International Planning Competition (IPC 2011), Deterministic Part*, pages 91–95, 2011.

Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005. ISBN 3-540-26276-8.

Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003a. ISBN 3-540-20851-8. Software available at `http://minisat.se/`.

Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003b.

Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In Per Bjesse and Anna Slobodová, editors, *FMCAD*, pages 125–134. FMCAD Inc., 2011. ISBN 978-0-9835678-1-3.

E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional μ-calculus (extended abstract). In *LICS*, pages 267–278. IEEE Computer Society, 1986.

Michael Fisher. A resolution method for temporal logic. In John Mylopoulos and Raymond Reiter, editors, *IJCAI*, pages 99–104. Morgan Kaufmann, 1991. ISBN 1-55860-160-0.

Michael Fisher, Clare Dixon, and Martin Peim. Clausal temporal resolution. *ACM Trans. Comput. Log.*, 2(1):12–56, 2001.

Maria Fox and Derek Long. The detection and exploitation of symmetry in planning problems. In Thomas Dean, editor, *IJCAI*, pages 956–961. Morgan Kaufmann, 1999. ISBN 1-55860-613-0.

Oded Fuhrmann and Shlomo Hoory. On extending bounded proofs to inductive proofs. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 278–290. Springer, 2009. ISBN 978-3-642-02657-7.

M. Carmen Fernández Gago, Michael Fisher, and Clare Dixon. Algorithms for guiding clausal temporal resolution. In Matthias Jarke, Jana Koehler, and Gerhard Lakemeyer, editors, *KI*, volume 2479 of *Lecture Notes in Computer Science*, pages 235–252. Springer, 2002. ISBN 3-540-44185-9.

Joxe Gaintzarain, Montserrat Hermo, Paqui Lucio, and Marisa Navarro. Systematic semantic tableaux for PLTL. *Electr. Notes Theor. Comput. Sci.*, 206:59–73, 2008.

B. Cenk Gazen and Craig A. Knoblock. Combining the expressivity of UCPOP with the efficiency of Graphplan. In Sam Steel and Rachid Alami, editors, *ECP*, volume 1348 of *Lecture Notes in Computer Science*, pages 221–233. Springer, 1997. ISBN 3-540-63912-8.

Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Piotr Dembinski and Marek Sredniawa, editors, *PSTV*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1995. ISBN 0-412-71620-8.

Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning – theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6.

Rajeev Goré. Various theorem provers for PLTL-satisfiability. Web site, `http://users.cecs.anu.edu.au/~rpg/PLTLProvers/`, accessed January, 2012.

Rajeev Goré and Florian Widmann. An optimal on-the-fly tableau-based decision procedure for PDL-satisfiability. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 437–452. Springer, 2009. ISBN 978-3-642-02958-5.

Graham D. Gough. Decision procedures for temporal logic. Master's thesis, Department of Computer Science, University of Manchester, October 1984. Also: University of Manchester, Department of Computer Science, Technical Report UMCS-89-10-1.

C. Cordell Green. Application of theorem proving to problem solving. In Donald E. Walker and Lewis M. Norton, editors, *IJCAI*, pages 219–240. William Kaufmann, 1969. ISBN 0-934613-21-4.

Andrew R. Haas. The case for domain-specific frame axioms. In *The Frame Problem in Artificial Intelligence, Proceedings of the 1987 Workshop on Reasoning about Action*. Morgan Kaufmann, 1987.

Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. Incremental, inductive CTL model checking. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 532–547. Springer, 2012. ISBN 978-3-642-31423-0.

Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. Better generalization in IC3. In *FMCAD*, pages 157–164. IEEE, 2013.

Keijo Heljanko, Tommi A. Junttila, and Timo Latvala. Incremental and complete bounded model checking for full PLTL. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2005. ISBN 3-540-27231-3.

Malte Helmert. The Fast Downward planning system. *J. Artif. Intell. Res. (JAIR)*, 26: 191–246, 2006.

Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artif. Intell.*, 173(5–6):503–535, 2009.

Malte Helmert and Gabriele Röger. How good is almost perfect? In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 944–949. AAAI Press, 2008. ISBN 978-1-57735-368-3.

Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In Mark S. Boddy, Maria Fox, and Sylvie Thiébaux, editors, *ICAPS*, pages 176–183. AAAI, 2007. ISBN 978-1-57735-344-7.

Malte Helmert, Minh Do, and Ioannis Refanidis. IPC 2008, deterministic part, 2008. Web site, `http://ipc.informatik.uni-freiburg.de`.

Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In Alessandro Cimatti and Roberto Sebastiani, editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012. ISBN 978-3-642-31611-1.

Jörg Hoffmann. Everything you always wanted to know about planning - (but were afraid to ask). In Joscha Bach and Stefan Edelkamp, editors, *KI 2011: Advances in Artificial Intelligence, 34th Annual German Conference on AI, Berlin, Germany, October 4-7,2011. Proceedings*, volume 7006 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2011. ISBN 978-3-642-24454-4.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res. (JAIR)*, 14:253–302, 2001.

Jörg Hoffmann, Peter Kissmann, and Álvaro Torralba. "Distance"? Who cares? Tailoring Merge-and-Shrink heuristics to detect unsolvability. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *ECAI 2014*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 441–446. IOS Press, 2014. ISBN 978-1-61499-418-3.

Matthias Horbach and Christoph Weidenbach. Superposition for fixed domains. In Michael Kaminski and Simone Martini, editors, *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings*, volume 5213 of *Lecture Notes in Computer Science*, pages 293–307. Springer, 2008. ISBN 978-3-540-87530-7.

Matthias Horbach and Christoph Weidenbach. Deciding the inductive validity of $\forall\exists^*$ queries. In Erich Grädel and Reinhard Kahle, editors, *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, volume 5771 of *Lecture Notes in Computer Science*, pages 332–347. Springer, 2009. ISBN 978-3-642-04026-9.

Richard Howey, Derek Long, and Maria Fox. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *ICTAI*, pages 294–301. IEEE Computer Society, 2004. ISBN 0-7695-2236-X. Software available at `http://www.plg.inf.uc3m.es/ipc2011-deterministic/Resources`.

Ruoyun Huang, Yixin Chen, and Weixiong Zhang. SAS+ planning as satisfiability. *J. Artif. Intell. Res. (JAIR)*, 43:293–328, 2012.

Ullrich Hustadt and Boris Konev. TRP++2.0: A temporal resolution prover. In Franz Baader, editor, *CADE*, volume 2741 of *Lecture Notes in Computer Science*, pages 274–278. Springer, 2003. ISBN 3-540-40559-3.

Ullrich Hustadt and Renate A. Schmidt. Scientific benchmarking with temporal logic decision procedures. In Dieter Fensel, Fausto Giunchiglia, Deborah L. McGuinness, and Mary-Anne Williams, editors, *KR*, pages 533–546. Morgan Kaufmann, 2002. ISBN 1-55860-554-1.

Ullrich Hustadt, Boris Konev, and Renate A. Schmidt. Deciding monodic fragments by temporal resolution. In Robert Nieuwenhuis, editor, *CADE*, volume 3632 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2005. ISBN 3-540-28005-7.

IPC. International planning competition. Web site, `http://ipc.icaps-conference.org/`, accessed 1/5/2013, 2013.

IPC. International planning competition 2014 results page. Web site, `http://helios.hud.ac.uk/scommv/IPC-14/resDoc.html`, accessed 06/12/2014, 2014.

Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010. ISBN 978-3-642-12001-5.

Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012. ISBN 978-3-642-31364-6.

Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41 (4), 2009.

Barbara Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *Formal Methods in Computer-Aided Design, 6th International Conference, FMCAD 2006, San Jose, California, USA, November 12-16, 2006, Proceedings*, pages 117–124. IEEE Computer Society, 2006. ISBN 0-7695-2707-8.

Wm. Woolsey Johnson and William E. Story. Notes on the "15" puzzle. *American Journal of Mathematics*, 2(4):397–404, 1879. ISSN 00029327.

Johan Anthony Willem Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, Calif., 1968.

Henry Kautz, Bart Selman, and Jörg Hoffmann. SatPlan: Planning as satisfiability. In *Working Notes of the 5th International Planning Competition, Cumbria, UK*, 2006. Software available at `http://www.cs.rochester.edu/~kautz/satplan/`.

Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In William J. Clancey and Daniel S. Weld, editors, *AAAI/IAAI, Vol. 2*, pages 1194–1201. AAAI Press / The MIT Press, 1996. ISBN ISBN 0-262-51091-X.

Henry A. Kautz, David A. McAllester, and Bart Selman. Encoding plans in propositional logic. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *KR*, pages 374–384. Morgan Kaufmann, 1996. ISBN 1-55860-421-9.

Jana Koehler. *IPP - A Planning System for ADL and Resource-Constrained Planning Problems*. Habiliation thesis, University of Freiburg, 1999.

Boris Konev. TRP++ : Temporal resolution prover. Web site, `http://cgi.csc.liv.ac.uk/~konev/software/trp++/`, accessed January, 2012.

Boris Konev, Anatoli Degtyarev, Clare Dixon, Michael Fisher, and Ullrich Hustadt. Mechanising first-order temporal resolution. *Inf. Comput.*, 199(1-2):55–86, 2005.

Jan Krajícek. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *J. Symb. Log.*, 62(2):457–486, 1997.

Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *VMCAI*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2003. ISBN 3-540-00348-7.

Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

Stefan Kupferschmid, Matthew D. T. Lewis, Tobias Schubert, and Bernd Becker. Incremental preprocessing methods for use in BMC. *Formal Methods in System Design*, 39 (2):185–204, 2011.

Tal Lev-Ami, Christoph Weidenbach, Thomas W. Reps, and Mooly Sagiv. Labelled clauses. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 311–327. Springer, 2007. ISBN 978-3-540-73594-6.

Derek Long and Maria Fox. Efficient implementation of the plan graph in STAN. *J. Artif. Intell. Res. (JAIR)*, 10:87–115, 1999.

Feng Lu, Madhu K. Iyer, Ganapathy Parthasarathy, Li-C. Wang, Kwang-Ting Cheng, and Kuang-Chien Chen. An efficient sequential SAT solver with improved search strategies. In *DATE*, pages 1102–1107. IEEE Computer Society, 2005. ISBN 0-7695-2288-2.

Michel Ludwig and Ullrich Hustadt. Fair derivations in monodic temporal reasoning. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 261–276. Springer, 2009a. ISBN 978-3-642-02958-5.

Michel Ludwig and Ullrich Hustadt. Resolution-based model construction for PLTL. In Carsten Lutz and Jean-François Raskin, editors, *TIME*, pages 73–80. IEEE Computer Society, 2009b. ISBN 978-0-7695-3727-6.

João Marques-Silva. Computing minimally unsatisfiable subformulas: State of the art and future directions. *Multiple-Valued Logic and Soft Computing*, 19(1-3):163–183, 2012.

João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

João P. Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Biere et al. (2009), pages 131–153. ISBN 978-1-58603-929-5.

Bart Massey. *Directions In Planning: Understanding The Flow Of Time In Planning.* PhD thesis, University of Oregon, 1999.

John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in McC90.

Kenneth L. McMillan. *Symbolic model checking.* Kluwer, 1993. ISBN 978-0-7923-9380-1.

Kenneth L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003. ISBN 3-540-40524-0.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. ISBN 1-58113-297-2.

Hootan Nakhost and Martin Müller. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *ICAPS*, pages 121–128. AAAI, 2010.

M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 371–443. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

Mats Petter Pettersson. Reversed planning graphs for relevance heuristics in AI planning. In *Planning, Scheduling and Constraint Satisfaction: From Theory to Practice*, volume 117 of *Frontiers in Artificial Intelligence and Applications*, pages 29–38. IOS Press, 2005.

Ingo Pill, Simone Semprini, Roberto Cavada, Marco Roveri, Roderick Bloem, and Alessandro Cimatti. Formal analysis of hardware requirements. In Ellen Sentovich, editor, *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, pages 821–826. ACM, 2006. ISBN 1-59593-381-6.

Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers with restarts. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*, pages 654–668. Springer, 2009. ISBN 978-3-642-04243-0.

David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.

Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symb. Log.*, 62(3):981–998, 1997.

Silvio Ranise. Symbolic backward reachability with effectively propositional logic - applications to security policy analysis. *Formal Methods in System Design*, 42(1):24–45, 2013.

Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res. (JAIR)*, 39:127–177, 2010.

Jussi Rintanen. A planning algorithm not based on directional search. In Anthony G. Cohn, Lenhart K. Schubert, and Stuart C. Shapiro, editors, *KR*, pages 617–625. Morgan Kaufmann, 1998.

Jussi Rintanen. Evaluation strategies for planning as satisfiability. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI*, pages 682–687. IOS Press, 2004. ISBN 1-58603-452-9.

Jussi Rintanen. Regression for classical and nondeterministic planning. In Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, editors, *ECAI*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 568–572. IOS Press, 2008a. ISBN 978-1-58603-891-5.

Jussi Rintanen. Planning graphs and propositional clause-learning. In Gerhard Brewka and Jérôme Lang, editors, *KR*, pages 535–543. AAAI Press, 2008b. ISBN 978-1-57735-384-3.

Jussi Rintanen. Planning as satisfiability: Heuristics. *Artif. Intell.*, 193:45–86, 2012.

Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.*, 170(12-13):1031–1080, 2006.

John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

Kristin Y. Rozier and Moshe Y. Vardi. LTL satisfiability checking. *STTT*, 12(2):123–137, 2010.

Kristin Y. Rozier and Moshe Y. Vardi. A multi-encoding approach for LTL symbolic satisfiability checking. In Michael Butler and Wolfram Schulte, editors, *FM*, volume 6664 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 2011. ISBN 978-3-642-21436-3.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010. ISBN 978-0-13-207148-2.

Viktor Schuppan and Luthfi Darmawan. Evaluating LTL satisfiability solvers. In Tevfik Bultan and Pao-Ann Hsiung, editors, *ATVA*, volume 6996 of *Lecture Notes in Computer Science*, pages 397–413. Springer, 2011. ISBN 978-3-642-24371-4.

Stefan Schwendimann. A new one-pass tableau calculus for PLTL. In Harrie C. M. de Swart, editor, *TABLEAUX*, volume 1397 of *Lecture Notes in Computer Science*, pages 277–292. Springer, 1998. ISBN 3-540-64406-7.

Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000. ISBN 3-540-41219-0.

A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

Dominik Stoffel and Wolfgang Kunz. Record & play: a structural fixed point iteration for sequential circuit verification. In *ICCAD*, pages 394–399, 1997.

Ofer Strichman. Pruning techniques for the SAT-based bounded model checking problem. In Tiziana Margaria and Thomas F. Melham, editors, *CHARME*, volume 2144 of *Lecture Notes in Computer Science*, pages 58–70. Springer, 2001. ISBN 3-540-42541-1.

Martin Suda. LS4: A PLTL-prover based on labelled superposition with partial model guidance. Web site, `http://www.mpi-inf.mpg.de/~suda/ls4.html`, 2012a.

Martin Suda. Labelled superposition for PLTL. Web site, `http://people.mpi-inf.mpg.de/~suda/supLTL.html`, 2012b.

Martin Suda. Variable and clause elimination for LTL satisfiability checking. *CoRR*, abs/1306.5539, 2013a.

Martin Suda. Duality in STRIPS planning. *CoRR*, abs/1304.0897, 2013b.

Martin Suda. DIMSPEC, a format for specifying symbolic transition systems. Web site, `http://www.mpi-inf.mpg.de/~suda/DIMSPEC.html`, 2013c.

Martin Suda. Variable and clause elimination for LTL satisfiability checking. In Marek Košta and Thomas Sturm, editors, *MACIS 2013 Nanning, China*, pages 60–74, 2013d.

Martin Suda. `minireachIC3`, a minisat-based implementation of PDR. Web site, `https://github.com/quickbeam123/minireachIC3`, 2013e.

Martin Suda. Property directed reachability for automated planning. *J. Artif. Intell. Res. (JAIR)*, 50:265–319, 2014a.

Martin Suda. Property directed reachability for automated planning. Web site, `http://www.mpi-inf.mpg.de/~suda/PDRplan.html`, 2014b.

Martin Suda and Christoph Weidenbach. A PLTL-prover based on labelled superposition with partial model guidance. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 537–543. Springer, 2012a. ISBN 978-3-642-31364-6.

Martin Suda and Christoph Weidenbach. Labelled superposition for PLTL. In Nikolaj Bjørner and Andrei Voronkov, editors, *LPAR*, volume 7180 of *Lecture Notes in Computer Science*, pages 391–405. Springer, 2012b. ISBN 978-3-642-28716-9.

Martin Suda and Christoph Weidenbach. Labelled superposition for PLTL. Research Report MPI-I-2012-RG1-001, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, January 2012c.

Andrzej Szalas and Leszek Holenderski. Incompleteness of first-order temporal logic with until. *Theor. Comput. Sci.*, 57:317–325, 1988.

G.S. Tseitin. On the complexity of derivation in propositional calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer Berlin Heidelberg, 1983. ISBN 978-3-642-81957-5.

Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344. IEEE Computer Society, 1986.

Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

G. Venkatesh. A decision method for temporal logic based on resolution. In S. N. Maheshwari, editor, *FSTTCS*, volume 206 of *Lecture Notes in Computer Science*, pages 272–289. Springer, 1985. ISBN 3-540-16042-6.

Christoph Weidenbach. *Automated Reasoning – The Art of Generic Problem Solving*. Unpublished.

Christoph Weidenbach. Combining superposition, sorts and splitting. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1965–2013. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8.

Eric W. Weisstein. Least common multiple. From MathWorld—A Wolfram Web Resource, 2013. URL `http://mathworld.wolfram.com/LeastCommonMultiple.html`. Last visited on 19/11/2013.

Wikipedia. 15 puzzle — wikipedia, the free encyclopedia. Web site, `http://en.wikipedia.org/wiki/15_puzzle`, accessed 19/05/2014, 2014.

Richard Williams and Boris Konev. Propositional temporal proving with reductions to a SAT problem. In Maria Paola Bonacina, editor, *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 421–435. Springer, 2013. ISBN 978-3-642-38573-5. Software available at `http://cgi.csc.liv.ac.uk/~rmw/STRP.html`.

Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56 (1/2):72–99, 1983.

Pierre Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, 28:119–136, 1985.

Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 10880–10885. IEEE Computer Society, 2003. ISBN 0-7695-1870-2.

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W