Labelled Superposition

Arnaud Fietzke

Dissertation zur Erlangung des Grades
des Doktors der Naturwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

vorgelegt im
Oktober 2013

Tag des Kolloquiums:	5. Juni 2014
Dekan:	Prof. Dr. Markus Bläser
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Reinhard Wilhelm
Berichterstatter:	Prof. Dr. Maria Paola Bonacina
	Prof. Dr. Holger Hermanns
	Prof. Dr. Christoph Weidenbach
Akademischer Mitarbeiter:	Dr. Martin Hoefer

Abstract

The work presented in this thesis consists of two parts: The first part presents a formalization of the splitting rule for case analysis in superposition and a proof of its correctness, as well as the integration into splitting of a novel approach to lemma learning, which allows the derivation of non-ground lemmas. The second part deals with the application of hierarchic superposition, and in particular superposition modulo linear arithmetic SUP(LA), to the verification of timed and probabilistic timed systems. It contains the following three main contributions: Firstly, a SUP(LA) decision procedure for reachability in timed automata, which is among the first decision procedures for free first-order logic modulo linear arithmetic; secondly, an automatic induction rule for SUP(LA) based on loop detection, whose application allows a generalization of the decidability result for timed automata to timed automata with unbounded integer variables; and thirdly, a formalism for modelling probabilistic timed systems with first-order background theories, as well as a novel approach for the analysis of such systems based on a reduction to model checking using SUP(LA) proof enumeration.

Zusammenfassung

Diese Arbeit besteht aus zwei Teilen: Im ersten Teil wird die Splitting-Regel zur Fallunterscheidung im Superpositionskalkül formalisiert und die Korrektheit der Formalisierung bewiesen. Ausserdem wird die Splitting-Regel mit einem neuartigen Verfahren zum Lernen von Lemmata erweitert, welches das Lernen von nicht-grund Lemmata erlaubt. Der zweite Teil befasst sich mit der Anwendung des hierarchischen Superpositionskalküls, insbesondere von Superposition modulo linearer Arithmetik SUP(LA), zur Verifikation von Echtzeit- und probabilistischen Echtzeitsystemen. Die drei wichtigsten Beiträge in diesem Teil sind: Erstens, ein SUP(LA)-basiertes Entscheidungsverfahren für Timed Automata, welches zu den ersten Entscheidungsverfahren für die hierarchische Kombination der freien Logik erster Stufe mit linear Arithmetik gehört; zweitens, eine Regel zur automatischen Induktion in SUP(LA), die auf der Erkennung von Schleifen basiert, und dank derer das Entscheidbarkeitsresultat für Timed Automata hin zu Timed Automata mit unbeschränkten Integer-Variablen verallgemeinert wird; und drittens, ein Formalismus zur Modellierung probabilistischer Echtzeitsysteme mit Hintergrundtheorien erster Stufe, sowie ein neuartiges Verfahren zur Analyse ebensolcher Systeme, welches auf einer Aufzählung von Erreichbarkeitsbeweisen in SUP(LA) sowie einer Zurückführung auf das Model Checking-Verfahren basiert.

Acknowledgements

I am grateful to my thesis advisor Christoph Weidenbach for his constant support and insightful guidance, and to Holger Hermanns for accepting to be my co-advisor and for his expertise on concurrency and probability.

Working in the Automation of Logic group has been an inspiring and enriching experience, and I want to thank my current and former colleagues for providing a great research environment. Special thanks go to Willem Hagemann, Matthias Horbach, Carsten Ihlemann, Evgeny Kruglov, Tianxiang Lu, Viorica Sofronie-Stokkermans, Thomas Sturm, Martin Suda, 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, Arnd Hartmanns for answering all my questions about probabilistic timed automata, and Martin Suda and Marek Kosta for proofreading parts of this thesis.

I would also like to thank Maria Paola Bonacina who kindly agreed to review this thesis.

This work has been partly supported by the International Max Planck Research School and by the German Research Foundation (DFG), in particular by the German Transregional Collaborative Research Center SFB/TR 14 AVACS.

Finally, I want to thank my wife and my mother for their unrelenting support, which has made this work possible.

Contents

List of Figures

1 Introduction

This thesis presents novel extensions and applications of the superposition calculus. On the one hand, these concern the splitting rule for case analysis, which we formalize and extend with a new conflict-driven clause learning technique. On the other hand, we explore the use of hierarchic superposition to solve the reachability problem for timed automata, timed automata extended with unbounded integer variables, and probabilistic timed automata extended with first-order background theories. As part of this investigation, we develop a new inference rule for automatic induction in hierarchic superposition, based on loop detection. A recurrent theme across these approaches is the use of clause labels to capture extra-logical information, hence the title *labelled superposition*.

This chapter provides an overview of the central topics underlying this thesis: Superposition in the context of refutational theorem proving (Section 1.1), splitting (Section 1.2), hierarchic superposition (Section 1.3), and the automation of inductive reasoning (Section 1.4). The chapter ends with an outline of the main contributions of this thesis (Section 1.5).

1.1 Superposition

In refutational theorem proving, a formula ϕ is shown to follow logically from a set of formulas N, by proving $N \cup \{\neg\phi\}$ to be contradictory. Saturation is a method to derive a contradiction from a set of formulas by computing its closure under a set of inference rules. A prominent example of this approach is the resolution method [Rob65], which operates on normalized formulas called clauses and is particularly well-suited to automation. Since saturation typically produces large numbers of formulas, a key ingredient in making saturation-based theorem proving efficient in practice is redundancy elimination, whereby only certain formulas and inferences required for the proof are considered, and all others are discarded.

The presence of the equality predicate, ubiquitous in mathematics, poses additional challenges, because equality axioms cause the search space for resolution to become prohibitively large. The most successful answer to this problem is the superposition calculus [KB70, RW69, HR87, Rus91, BG91b, NR01, Wei01], which combines resolution with completion techniques and ordering restrictions. It constitutes a refutationally complete calculus for full first-order logic with equality, meaning that a contradiction,

in the form of the empty clause, can be derived from any unsatisfiable clause set in a finite number of inference steps. As unsatisfiability of first-order formulas is only semi-decidable however, a superposition-based saturation procedure may well diverge when applied to a satisfiable clause set. The identification of decidable fragments of first-order logic and the design of superposition-based decision procedures for such fragments are therefore important research topics, and they often go hand in hand, as evidenced by fragments whose decidability has been first established by means of superposition [Nie96, JMW98, Wei99a, JRV06].

1.2 Splitting

Boolean satisfiability (SAT) solvers are among the most successful systems for automated reasoning available today, a fact witnessed not only by their widespread use in industry, but also by a large and ever growing number of research projects which reduce the reasoning in more expressive formalisms to Boolean satisfiability by an encoding into propositional logic. Most state-of-the-art SAT solvers are based on CDCL [SS96], which itself builds on the DPLL procedure [DP60, DLL62]. The DPLL procedure performs a systematic case analysis, trying to build a satisfying assignment for the variables in the given SAT problem instance by successive decisions and propagations, and backtracking whenever a clause becomes false under the current assumptions. CDCL adds to this the learning of clauses from conflicts, which is a key ingredient that makes SAT solvers fast in practice.

The splitting rule [Wei01] is similar to the DPLL decision step, but suited for non-ground clauses. Instead of making a given propositional variable true or false, splitting instead *splits* a clause C occurring in a clause set N into variable-disjoint components C_1, C_2, and conceptually replaces N by the two sets $N \cup \{C_1\}$ and $N \cup \{C_2\}$. Satisfiability of either of these two sets is equivalent to satisfiability of N, and since C_1 and C_2 both subsume C, splitting transforms a complex problem (the satisfiability of N) into two simpler problems (the satisfiability of $N \cup \{C_1\}$ or $N \cup \{C_2\}$). This makes splitting an important component of superposition-based decision procedures [BGW93, FLHT01, HW07].

In practical implementations, splitting is not carried out by duplicating clause sets, but by a depth-first backtracking search, like in DPLL. Since superposition-based theorem proving relies on the generation of new clauses via inferences and the deletion of redundant clauses via reductions, this backtracking search however requires a much more complex bookkeeping than in DPLL.

Finally, the integration of CDCL-style clause learning into superposition with splitting is a challenging problem as well, because of the universally quantified variables in clauses. So far, only ground clause learning has been available for splitting.

1.3 Hierarchic Superposition

While reasoning in pure first-order logic has many applications, in practice one often deals with formulas where the semantics of certain symbols is given by fixed theories, like the theory of linear integer or real arithmetic, yielding a *hierarchic* combination of pure first-order logic with a base theory. Often, such base theories are not axiomatizable in first-order logic, or using an explicit axiomatization is impractical. This has led to the development of reasoning *modulo theory*.

A prominent example of this is Satisfiability Modulo Theories (SMT, see [dMB11] for an overview), which combines a CDCL-style SAT solver with theory-specific solvers, which are used to decide conjunctions of ground formulas. While effective in practice, SMT procedures are typically incomplete for formulas outside the existential fragment.

Superposition modulo theories, or SUP(T) [BGW92, BGW94], is an alternative approach to the hierarchic reasoning problem, and is based on the superposition calculus. In contrast to SMT, SUP(T) offers the advantage of refutational completeness in the presence of full quantification, under the condition that the clause sets are *sufficiently complete*, which essentially requires the non-base symbols of base sort to be sufficiently defined. Additionally, the powerful redundancy elimination mechanisms of superposition carry over to SUP(T). In particular, the hierarchic combination of first-order logic with linear real arithmetic, FOL(LA), has been in the focus of recent research, which has resulted in the development of the SUP(LA) calculus [AKW09, Kru13].

Obtaining decision procedures for FOL(LA) can be considered even more challenging than for the pure first-order case, because the hierarchic combination of pure first-order logic with linear arithmetic yields a very expressive logic [Hal91]. Even when restricted to a decidable first-order fragment like Bernays-Schönfinkel-Horn (BSH), the resulting FOL(LA) theory is undecidable.[1]

1.4 Automation of Inductive Reasoning

Another active research topic in the area of automated reasoning concerns the mechanization of inductive reasoning. While proof by induction in general requires human guidance in the form of inductive hypotheses, many forms of inductive reasoning can be carried out automatically.

[1] See [FW12] for a discussion of the combination of BSH with linear arithmetic.

There are many automatic and semi-automatic methods for inductive reasoning in first-order logic and extensions or fragments thereof,[2] but they typically rely on predefined induction schemata, or on pre-saturated clause sets.

An interesting approach to the automation of induction in saturation-based theorem proving relies on the detection of cycles in the proof search: A prover may deduce a sequence of formulas of the same shape, like $P(1), P(2), P(3), \ldots$, and this sequence may be infinite. A human observer would eventually notice the regularity and check whether the step that led from $P(k)$ to $P(k+1)$ could also be applied to deduce $P(n+1)$ from $P(n)$, for arbitrary n, in which case he or she could add $\forall n(n \in \mathbb{N} \to P(n))$ as an additional axiom, allowing the prover to terminate. If the prover is able to detect this regularity by itself, then this kind of reasoning can be automated.

In the area of symbolic model checking, *acceleration* is another form of induction which consists in computing the transitive closure of a sequence of transitions of an infinite-state system, to allow the set of reachable states to be computed in finite time. Acceleration and loop detection in proof search can be combined, as we will show.

1.5 Contributions of This Thesis

The contributions of this thesis can be divided into two parts. The first part deals with splitting, and contains the following contributions:

- We present a formalization of splitting with backtracking in terms of a transition system, in a modular way by turning an arbitrary superposition-based calculus `C` without splitting into a calculus `SC` with splitting (Section 3.2). Dependencies between clauses and splits are captured by clause labels. We prove that soundness and completeness of the underlying calculus `C` carry over to `SC`. The classical notion of fairness for derivations without splitting is too weak to ensure completeness in the presence of splitting, and we introduce a stronger notion of fairness and use it to prove the completeness result.
- We extend the splitting calculus `SC` by a rule for conflict-driven clause learning (Section 3.3). This rule relies on the instantiations of variables in split clauses that have been applied in the derivation of the empty clause to allow the "negation" of non-ground clauses without using Skolemization. The instantiation information is tracked by the clause labels, which we generalize for this purpose. Thanks to the labels, conflict-driven lemmas can be computed without the need for additional resolution steps.
- We describe an implementation of splitting with backtracking and clause learning inside the SPASS theorem prover, and we provide experimental evidence showing

[2] See Section 4.10 for a discussion of existing approaches.

that the integration of clause learning improves the performance of superposition with splitting (Section 3.6).

The second part is concerned with superposition modulo linear arithmetic SUP(LA), and its application to the reachability problem for timed automata, probabilistic timed automata and extensions thereof. It contains the following main contributions:

- We extend well-known minimal-model results for pure first-order logic to the hierarchic combination of first-order logic with a base theory $\mathcal{T}$, in order to ensure the existence of unique minimal Herbrand models of reachability theories (Section 4.3). We show that
  - if $\mathcal{T}$ has a unique minimal Herbrand model (possibly among other Herbrand models), then any extension by a sufficiently complete Horn clause set yields again a theory with a unique minimal Herbrand model, and
  - if $\mathcal{T}$ has a unique Herbrand model, then any extension by a sufficiently complete clause set that is Horn modulo $\mathcal{T}$ has a unique minimal Herbrand model.
- We formally define reachability theories and define their semantics in terms of minimal Herbrand models (Section 4.4).
- We prove that SUP(LA) constitutes a decision procedure for reachability in timed automata (Section 4.5.6).
- We introduce a SUP(T) inference rule called constraint induction, which derives clauses by induction (Section 4.6). The rule is based on loop detection and on the computation of the transitive closure of base theory constraints. We present an effective instance of the rule for SUP(LA).
- We provide evidence of the usefulness of constraint induction by extending the superposition-based decision procedure for timed automata to timed automata with unbounded integer variables.
- We define the formalism of first-order probabilistic timed automata (FPTA) which extends probabilistic timed automata (PTA) with first-order background theories. We present a SUP(LA)-based approach for reducing the max-reachability problem in FPTA to the reachability problem in PTA. The approach relies on the enumeration of reachability proofs, again using a clause labelling scheme (Section 4.7).

2 Foundations

In this chapter, we recall basic notions and definitions related to orderings, first-order logic and first-order theorem proving. More detailed introductions can be found in [BN98] and [BG01, Wei01].

2.1 Mathematical Foundations

Definition 2.1 (Properties of Binary Relations)
A binary relation $\triangleright$ on a set S is called

- *transitive*, if $s_1 \triangleright s_2$ and $s_2 \triangleright s_3$ imply $s_1 \triangleright s_3$, for all $s_1, s_2, s_3 \in S$;
- *reflexive*, if $s \triangleright s$ holds for all $s \in S$;
- *symmetric*, if $s_1 \triangleright s_2$ implies $s_2 \triangleright s_1$, for all $s_1, s_2 \in S$;
- *asymmetric*, if for no $s_1, s_2 \in S$, both $s_1 \triangleright s_2$ and $s_2 \triangleright s_1$ hold;
- *antisymmetric*, if $s_1 \triangleright s_2$ and $s_2 \triangleright s_1$ imply $s_1 = s_2$, for all $s_1, s_2 \in S$. ■

Definition 2.2 (Orderings)
A *partial ordering* $\succeq$ on a set S is a binary relation on S that is reflexive, antisymmetric, and transitive. A *strict partial ordering* $\succ$ on a set S is a binary relation on S that is asymmetric and transitive. If $=$ is the identity relation on S and $\succeq$ is a partial ordering on S, then $\succ = (\succeq \setminus =)$ is a strict partial ordering. Conversely, if $\succ$ is a strict partial ordering on S, then $\succeq = (\succ \cup =)$ is a partial ordering. A (strict) partial ordering $\succeq$ is *total* on $S' \subseteq S$ if $s_1 \succeq s_2$ or $s_2 \succeq s_1$ holds for any $s_1, s_2 \in S'$. ■

Definition 2.3 (Multisets)
A *multiset* M over a set S is a function $M : S \to \mathbb{N}$. An element $s \in S$ is an *element* of M if and only if $M(s) \neq 0$. The empty multiset over S, denoted $\emptyset$, satisfies $M(s) = 0$ for all $s \in S$. The standard relations and operations on sets, like $\subseteq, \cup$ and $\cap$, naturally extend to multisets, for instance, $(M_1 \cup M_2)(s) = M_1(s) + M_2(s)$. Multisets are written in set-like notation, for instance, $\{1, 1\}$ denotes a multiset M over $\mathbb{N}$ with $M(1) = 2$ and $M(x) = 0$ for all $x \neq 1$ in $\mathbb{N}$. ■

Definition 2.4 (Lexicographic and Multiset Orderings)
A strict partial ordering $\succ$ on a set S can be extended to a strict partial ordering $\succ^{mul}$ on multisets over S as follows: $M \succ^{mul} M'$ if $M \neq M'$ and whenever there is $s \in S$ such that $M'(s) \succ M(s)$, then $M(s') \succ M'(s')$ for some $s' \succ s$. The ordering $\succ$ can be extended to a strict partial ordering $\succ^{lex}$ on n-tuples over S as follows: $(s_1, \ldots, s_n) \succ^{lex} (s'_1, \ldots, s'_n)$ if there is $i \in [1, n]$ such that $s_j = s'_j$ for all $1 \leq i < i$, and $s_i \succ s'_i$. Analogously, a partial ordering $\succeq$ on S can be extended to a partial ordering $\succeq^{mul}$ on multisets over S and to a partial ordering $\succeq^{lex}$ on n-tuples over S. ■

Definition 2.5 (Minimal and Maximal Elements)
Let $\succ$ be a strict partial ordering on a set S and let S' be a subset of S or a multiset over S. An element $s \in S$ is called

- *maximal* if there is no $s' \in S'$ with $s' \succ s$,
- *minimal* if there is no $s' \in S'$ with $s \succ s'$,
- *strictly maximal* if s is maximal and, if S' is a multiset, then $S'(s) = 1$,
- *strictly minimal* if s is minimal and, if S' is a multiset, then $S'(s) = 1$. ■

Definition 2.6 (Equivalence Relations and Quotients)
An *equivalence relation* $\cong$ on a set S is a binary relation on S that is reflexive, symmetric, and transitive. For every element $s \in S$, the subset $[s]_\cong = \{s' \in S \mid s \cong s'\}$ of S is called the *equivalence class* of s. Note that $s \cong s'$ if and only if $[s]_\cong = [s']_\cong$. The set $S/_\cong = \{[s]_\cong \mid s \in S\}$ is the *quotient of S by $\cong$*. If the considered equivalence relation is clear from the context, $[s]_\cong$ is also written simply as $[s]$. ■

Definition 2.7 (Composition and Closure of Binary Relations)
Given two binary relations $\triangleright_1 \subseteq S_1 \times S_2$ and $\triangleright_2 \subseteq S_2 \times S_3$, their *composition* $(\triangleright_1 \circ \triangleright_2) \subseteq S_1 \times S_3$ is defined by $s_1(\triangleright_1 \circ \triangleright_2)s_3$ if and only if there exists $s_2 \in S_2$ such that $s_1 \triangleright_1 s_2$ and $s_2 \triangleright_2 s_3$. Given any binary relation $\triangleright$ on S, we define

- $\triangleright^0 = \{(s, s) \mid s \in S\}$, the *identity*;
- $\triangleright^{i+1} = \triangleright^i \circ \triangleright$ for $i \geq 0$;
- $\triangleright^+ = \bigcup_{i>0} \triangleright^i$, the *transitive closure* of $\triangleright$;
- $\triangleright^* = \triangleright^+ \cup \triangleright^0$, the *reflexive transitive closure* of $\triangleright$. ■

2.2 Syntax

Definition 2.8 (Signatures)
A *many-sorted signature* is a tuple $\Sigma = (\mathcal{S}, \Omega)$ consisting of a finite, non-empty set $\mathcal{S}$ of *sort symbols*, and a set Ω of *operator symbols*. Every operator symbol $f \in \Omega$ comes with a unique *arity* $n \in \mathbb{N}$ and a unique sort declaration $f : S_1 \ldots S_n \to S$, where

$S, S_1, \ldots, S_n \in \mathcal{S}$. A *constant* is an operator symbol of arity zero. For every sort $S \in \mathcal{S}$, there exists at least one $f \in \Omega$ and $S_1, \ldots, S_n \in \mathcal{S}$, $n \geq 0$, such that $f : S_1 \ldots S_n \to S$.

In addition to any signature Σ, we assume a countably infinite set $\mathcal{X}$ of variables, disjoint from Ω. Every variable $x \in \mathcal{X}$ comes with a unique sort declaration $x : S$ with $S \in \mathcal{S}$, and we assume that for every $S \in \mathcal{S}$, there are infinitely many variables of sort S in $\mathcal{X}$. If X is any set of variables $x_1, x_2, \ldots$, we denote by X' the set of primed copies $x'_1, x'_2, \ldots$ of variables in X. For convenience, we also assume an arbitrary, fixed, total ordering on $\mathcal{X}$, so that any finite set $X \subseteq \mathcal{X}$ can be uniquely written as a vector $\vec{x}$, and we accordingly use set-like notation for vectors of variables ($\vec{x} = X$, $\vec{u} = \vec{x} \cup \vec{z}$, $\vec{x} = \vec{u} \setminus \vec{z}$, etc.) with the obvious meaning. ∎

Given signatures $\Sigma = (\mathcal{S}, \Omega)$ and $\Sigma' = (\mathcal{S}', \Omega')$, we write $\Sigma \subseteq \Sigma'$ if $\mathcal{S} \subseteq \mathcal{S}'$ and $\Omega \subseteq \Omega'$. By $\Sigma' \setminus \Sigma$ we mean $(\mathcal{S}' \setminus \mathcal{S}, \Omega' \setminus \Omega)$.

Definition 2.9 (Terms)
Given a signature $\Sigma = (\mathcal{S}, \Omega)$ and a sort $S \in \mathcal{S}$, the set $T_\Sigma(S, \mathcal{X})$ of *terms* of sort S over $\Sigma, \mathcal{X}$ is the least set containing all $x : S \in \mathcal{X}$, and containing $f(t_1, \ldots, t_n)$ whenever $f : S_1 \ldots S_n \to S \in \Omega$ and $t_i \in T_\Sigma(S_i, \mathcal{X})$. We also write $t : S$ whenever $t \in T_\Sigma(S, \mathcal{X})$. By $T_\Sigma(S)$ we denote the set of *ground* terms in $T_\Sigma(S, \mathcal{X})$, i.e., terms not containing any variables. The set of terms over Σ (or Σ-terms) is defined as $T_\Sigma(\mathcal{X}) = \bigcup_{S \in \mathcal{S}} T_\Sigma(S, \mathcal{X})$ and the set of ground terms over Σ is defined as $T_\Sigma = \bigcup_{S \in \mathcal{S}} T_\Sigma(S)$. The set of all ground instances of a term t is denoted by $\mathrm{gnd}(t)$. ∎

A list $t_1, \ldots, t_n$ of terms is often written as $\vec{t}$, and the n-fold application $f(\ldots(f(t))\ldots)$ of a unary function symbol f to a term t is written as $f^n(t)$.

Definition 2.10 (Equations)
An *equation* over a signature Σ is a multiset of two Σ-terms s, t of the same sort, written as $s \simeq t$. ∎

Definition 2.11 (Atoms, Formulas)
An *atom* (or atomic formula) over Σ is an equation over Σ. The set $F_\Sigma(\mathcal{X})$ of *formulas* over Σ (or Σ-formulas) is the least set containing all atoms over Σ, the two logical constants $\top$ (true) and $\bot$ (false), and for all Σ-formulas ϕ_1, ϕ_2, the *negation* $\neg\phi_1$, the *conjunction* $\phi_1 \wedge \phi_2$, the *disjunction* $\phi_1 \vee \phi_2$, and for any $x \in \mathcal{X}$, the *universal quantification* $\forall x.\phi_1$ and the *existential quantification* $\exists x.\phi_1$. ∎

Formally, equations are the only atomic formulas. To deal with predicate symbols other than equality, we use the following, standard encoding: For each predicate symbol P taking arguments of sorts $S_1, \ldots, S_n$, we add a distinct sort S_P, and we introduce a new constant true_P of sort S_P, and a new operator symbol $f_P : S_1 \ldots S_n \to S_P$ to the signature. The predicative atom $P(t_1, \ldots, t_n)$ is then regarded as an abbreviation[1] for $f_P(t_1, \ldots, t_n) \simeq \mathrm{true}_P$.

[1] A single extra sort S_{bool} would suffice, but adding a "Boolean" sort for each predicate symbol is more suited for the hierarchic superposition, see Section 4.3.4.

We will use the following conventions: For any formulas ϕ_1, ϕ_2, ϕ_3, we write $\phi_1 \vee \phi_2 \vee \phi_3$ for $(\phi_1 \vee \phi_2) \vee \phi_3$ and $\phi_1 \wedge \phi_2 \wedge \phi_3$ for $(\phi_1 \wedge \phi_2) \wedge \phi_3$. The notation $\phi_1 \to \phi_2$ denotes the formula $\neg\phi_1 \vee \phi_2$, and $\phi_1 \leftrightarrow \phi_2$ denotes $(\phi_1 \to \phi_2) \wedge (\phi_2 \to \phi_1)$. For any set $I = \{i_1, \ldots, i_n\}$, we write $\bigvee_{i \in I} \phi_i$ for $\phi_{i_1} \vee \phi_{i_2} \vee \ldots \vee \phi_{i_n}$ and $\bigwedge_{i \in I} \phi_i$ for $\phi_{i_1} \wedge \phi_{i_2} \wedge \ldots \wedge \phi_{i_n}$. For a list of variables $\vec{x} = x_1, \ldots, x_n$, we write $\forall\vec{x}.\phi$ for $\forall x_1 \ldots \forall x_n.\phi$, and $\exists\vec{x}.\phi$ for $\exists x_1 \ldots \exists x_n.\phi$.

Definition 2.12 (Subterms, Subformulas)
An *expression* is any term or formula. A *position* is a finite sequence of natural numbers. The empty position is denoted by ϵ. Given an expression e and a position p, *subexpression* $e|_p$ *of* e *at position* p is defined recursively as

- $e|_\epsilon = e$, and
- $e|_p = e_i|_q$ if $p = iq$ and either
  - $e = f(e_1, \ldots, e_n)$ and $i \leq n$, or
  - $e = e_1 \vee e_2$ or $e = e_1 \wedge e_2$ and $i \leq 2$, or
  - $e = \neg e_1$ or $e = \exists x.e_1$ or $e = \forall x.e_1$ and $i = 1$.

The expression e is said to *contain* e' and e' *occurs* in e if e' is a subexpression of e. If e' is a subexpression of e and $e' \neq e$, then e' is a *proper subexpression* of e. A *subformula* is a subexpression that is a formula and a *subterm* is a subexpression that is a term. The number of positions p for which $e|_p$ is defined is called the *size* of e and denoted by $|e|$. A formula is *quantifier-free* if it does not contain a subformula $\exists x.\phi$ or $\forall x.\phi$, and *ground* if it does not contain any variable. ∎

Definition 2.13 (Variables, Universal and Existential Closure)
Let Σ be a signature and let e be a formula or a term over Σ. The set $\mathrm{var}(e)$ of *variables of* e is defined as the set of all variables occurring in e. If e does not contain any variables, it is *ground*.

A variable x *occurs freely* in a formula ϕ if $\phi|_{i_1 \ldots i_n} = x$ and none of the subexpressions $\phi|_{i_1 \ldots i_m}$, $m \leq n$, is of the form $\phi|_{i_1 \ldots i_m} = \exists x.\phi_1$ or $\phi|_{i_1 \ldots i_m} = \forall x.\phi_1$. A formula that does not contain any free variable occurrences is *closed*. If $x_1, \ldots, x_n$ are the free variables in ϕ, then $\forall x_1. \cdots \forall x_n.\phi$ and $\exists x_1. \cdots \exists x_n.\phi$ are the *universal closure* and *existential closure* of ϕ, respectively. The universal and existential closure of a formula ϕ is often abbreviated as $\forall\phi$ and $\exists\phi$, respectively. A closed formula is also called a *sentence*. ∎

Definition 2.14 (Literals, Clauses)
Let Σ be a signature. A *literal* over Σ is either an atom A over Σ or its negation $\neg A$. It is called *positive* if it is an atom, and *negative* otherwise. A literal $\neg(s \simeq t)$, written as $s \not\simeq t$, is called a *disequation*. A *clause* over Σ is a pair (Γ, Δ) of multisets of atoms over Σ, written $\Gamma \to \Delta$. The multiset Γ is called the *antecedent* and the multiset Δ is called the *succedent* of the clause. Equivalently, a clause can be viewed as a multiset of literals, where the atoms from Γ occur negatively and the atoms from Δ occur positively. Logically, a clause represents the universal closure of the disjunction of its literals. The

clause is *Horn* if Δ contains at most one atom. The *empty clause*, where $\Gamma = \Delta = \emptyset$, is denoted by $\square$. ∎

The notion of ground instance of a term is naturally lifted to atoms, literals, clauses and clause sets.

Definition 2.15 (Selection Function)
A *selection function* Sel assigns to each clause $C = \Gamma \to \Delta$ a multiset of atoms in the antecedent Γ. The atoms (and the corresponding negative literals) in $\mathrm{Sel}(C)$ are said to be *selected* in C. ∎

Definition 2.16 (Substitutions)
For any signature Σ, a (Σ-)*substitution* σ is a mapping from $\mathcal{X}$ to $T_\Sigma(\mathcal{X})$, written in postfix notation, such that $x\sigma$ always has the same sort as x, and $x\sigma \neq x$ holds for only finitely many x. The set $\mathrm{dom}(\sigma) = \{x \in \mathcal{X} \mid x\sigma \neq x\}$ is the *domain* of σ, and the set $\mathrm{im}(\sigma) = \{x\sigma \mid x \in \mathrm{dom}(\sigma)\}$ is the *image* of σ. The set $\mathrm{cdom}(\sigma) = \bigcup_{t \in \mathrm{im}(\sigma)} \mathrm{var}(t)$ is the *codomain* of σ. The *modification* of σ at x is defined as

$$\sigma[x \mapsto t](y) = \begin{cases} t & \text{if } x = y, \\ y\sigma & \text{otherwise.} \end{cases}$$

A substitution σ is identified with its homomorphic extension to terms, quantifier-free formulas and clauses. It is extended to quantified formulas by $(Qx.\phi)\sigma = Qy.(\phi\sigma[x \mapsto y])$, where y is a fresh variable. The composition $\sigma\sigma'$ of two substitutions is the substitution that maps every x to $(x\sigma)\sigma'$. For an expression e, we call $e\sigma$ an *instance* of e. A bijective substitution is called a *renaming*. A substitution σ is *ground* if all terms in $\mathrm{im}(\sigma)$ are ground; it is *grounding* for an expression e if $e\sigma$ is ground. ∎

Definition 2.17 (Variants)
Two clauses $C_1 = \Gamma_1 \to \Delta_1$, $C_2 = \Gamma_2 \to \Delta_2$ are *variants* of each other, if there exists a renaming σ such that $C_1\sigma = C_2$. ∎

Definition 2.18 (Unifiers)
Two terms s, t are *unifiable* if they are of the same sort and there is a substitution σ such that $s\sigma = t\sigma$; similarly, two atoms A, B are *unifiable* if there is a substitution σ such that $A\sigma = B\sigma$. In both cases, σ is called a *unifier*. A substitution σ is called *more general* than a substitution τ, denoted by $\sigma \leq \tau$, if there exists a substitution ρ such that $\sigma\rho = \tau$. A unifier of two terms or atoms e and e' is called a *most general unifier* (or mgu) if it is more general than any other unifier of e and e'. The most general unifier is unique up to variable renaming and (ambiguously) denoted by $\mathrm{mgu}(e, e')$. A substitution σ is called a matcher from e to e' if $e\sigma = e'$. ∎

Definition 2.19 (Relations on Terms)
Let Σ be a signature, and let $\succ$ be a binary relation on $T_\Sigma(\mathcal{X})$. The relation $\succ$ is called

- *well-founded* if there is no infinite chain $t_1, t_2, \ldots$ such that $t_i \succ t_{i+1}$ for all $i \geq 1$;

- *stable under substitutions* if $t_1 \succ t_2$ implies $t_1\sigma \succ t_2\sigma$, for all terms t_1, t_2 and substitutions σ;
- *compatible with contexts* if $t_1 \succ t_2$ implies $s[t_1]_p \succ s[t_2]_p$, for all terms t_1, t_2 and s and all positions p in s.

The relation $\succ$ has the *subterm property* if $t[s] \succ s$, for all terms t and all proper subterms s of t. A *rewrite ordering*, also called *rewrite relation*, is a strict ordering on terms that is stable under substitutions and compatible with contexts. A *reduction ordering* is a well-founded rewrite ordering. A *simplification ordering* is a reduction ordering with the subterm property. A *congruence relation* is an equivalence relation on terms that is compatible with contexts. ■

Definition 2.20 (Clause Orderings)
Let Σ be a signature. Given a partial ordering $\succeq$ on $T_\Sigma(\mathcal{X})$, a partial ordering on clauses over Σ is obtained as follows: The occurrence of an atom $s \simeq t$ in the antecedent is identified with the multiset $\{s, s, t, t\}$; the occurrence of $s \simeq t$ in the succedent is identified with the multiset $\{s, t\}$. The multiset extension of $\succeq$ on terms thereby yields a partial ordering on atom occurrences, which we again denote by $\succeq$. This ordering on atom occurrences is in turn extended to clauses—where both the antecedent and succedent are viewed as as multisets of atom occurrences—by taking its multiset extension, which we again denote by $\succeq$. ■

The ordering on atom occurrences (which may equivalently be viewed as an ordering on literals) has the property that an occurrence of an atom $s \simeq t$ in the antecedent (equivalently: the literal $s \not\simeq t$) is strictly larger than an occurrence of the same atom in the succedent (equivalently: the literal $s \simeq t$), because $\{s, s, t, t\} \succ \{s, t\}$.

There are several ways in which a well-founded strict partial ordering $\succ$, also called a *precedence*, on the operator symbols Ω of a signature Σ can be turned into a reduction ordering. A prominent example, often used for superposition, is the lexicographic path ordering (LPO):

Definition 2.21 (Lexicographic Path Ordering $\succ_{\mathsf{lpo}}$)
Let $\Sigma = (\mathcal{S}, \Omega)$ be a signature and $\succ$ be a strict partial ordering on Ω. The *lexicographic path ordering* $\succ_{lpo}$ on $T_\Sigma(\mathcal{X})$ induced by $\succ$ is defined by $s \succ_{\mathrm{lpo}} t$ if and only if

(i) $t \in \mathrm{var}(s)$ and $t \neq s$, or

(ii) $s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, and

  a) $s_i \succeq_{\mathrm{lpo}} t$ for some i, or

  b) $f \succ g$ and $s \succ_{\mathrm{lpo}} t_j$ for all j, or

  c) $f = g$, $s \succ_{\mathrm{lpo}} t_j$ for all j, and $(s_1, \ldots, s_m) \succ^{lex}_{\mathrm{lpo}} (t_1, \ldots, t_n)$. ■

If the precedence $\succ$ is total on Ω, then $\succ_{\mathrm{lpo}}$ is total on ground terms T_Σ.

Definition 2.22 (Rewrite Systems)
A set R of equations is called a *rewrite system* with respect to a strict partial term ordering $\succ$, if $s \succ t$ or $t \succ s$ holds for each $s \simeq t \in R$. If R is a rewrite system then an equation $s \simeq t \in R$ with $s \succ t$ is called a *rewrite rule*, and is written $s \twoheadrightarrow t$. The *rewrite relation* $\twoheadrightarrow_R$ is defined as $q \twoheadrightarrow_R r$ iff there is $s \twoheadrightarrow t \in R$ such that $q|_p = s\sigma$ and $r = q[t\sigma]_p$ for some position p of q and substitution σ. A term s is *reducible* by R if $s \twoheadrightarrow_R t$ for some term t, and *irreducible* or *in normal form* (with respect to R) otherwise.

A rewrite system R is *ground* if all equations in R are ground. It is *terminating* if $\twoheadrightarrow$ is well-founded, and it is *confluent* if for all terms t, t_1, t_2 satisfying $t \twoheadrightarrow_R^* t_1$ and $t \twoheadrightarrow_R^* t_2$ there is a term t_3 such that $t_1 \twoheadrightarrow_R^* t_3$ and $t_2 \twoheadrightarrow_R^* t_3$. ∎

2.3 Semantics

Definition 2.23 (Interpretations)
An *interpretation* over a signature $\Sigma = (\mathcal{S}, \Omega)$ (also called a Σ-interpretation or Σ-*algebra*) is a mapping $\mathcal{I}$ that assigns to every sort $S \in \mathcal{S}$ a non-empty carrier set $\mathcal{I}(S)$, and to every operator symbol $f : S_1 \dots S_n \to S \in \Omega$ a function $\mathcal{I}(f) : \mathcal{I}(S_1) \times \dots \times \mathcal{I}(S_n) \to \mathcal{I}(S)$. We assume that $\mathcal{I}(S_1), \mathcal{I}(S_2)$ are disjoint for any distinct $S_1, S_2 \in \mathcal{S}$. The set $U_\mathcal{I} = \bigcup_{S \in \mathcal{S}} \mathcal{I}(S)$ is called the *universe* of $\mathcal{I}$. We occasionally write $S_\mathcal{I}$ and $f_\mathcal{I}$ for $\mathcal{I}(S)$ and $\mathcal{I}(f)$, respectively.

An *assignment* for $\mathcal{I}$ over a set of variables $X \subseteq \mathcal{X}$ is a function $\nu : X \to U_\mathcal{I}$ such that $\nu(x) \in \mathcal{I}(S)$ for every $x : S \in X$. We denote the set of all assignments for $\mathcal{I}$ over X by $\mathit{Val}_\mathcal{I}(X)$. Given a sequence $\vec{x} = x_1, \dots, x_n$ containing all variables of X, we write $\nu(\vec{x})$ for the sequence $\nu(x_1), \dots, \nu(x_n)$, and $\mathit{Val}_\mathcal{I}(\vec{x})$ for $\{\nu(\vec{x}) \mid \nu \in \mathit{Val}_\mathcal{I}(X)\} \subseteq U_\mathcal{I}^{|X|}$. In general, we identify $\mathit{Val}_\mathcal{I}(X)$ with $\mathit{Val}_\mathcal{I}(\vec{x})$, where $\vec{x}$ contains the variables of X in the order given by the total ordering on $\mathcal{X}$.

For $\nu \in \mathit{Val}_\mathcal{I}(X)$ and $Y \subseteq X$, the *restriction* of ν to Y is the assignment $\nu|_Y \in \mathit{Val}_\mathcal{I}(Y)$ that agrees with ν on all variables in Y.

The *modification* of ν at x is defined as

$$\nu[x \mapsto a](y) = \begin{cases} a & \text{if } x = y, \\ \nu(y) & \text{otherwise.} \end{cases}$$

The homomorphic extension of ν to terms, denoted by $\mathcal{I}(\nu)$, is defined by

$$\begin{aligned} \mathcal{I}(\nu)(x) &= \nu(x) && \text{for } x \in \mathcal{X}, \\ \mathcal{I}(\nu)(f(t_1, \dots, t_n)) &= f_\mathcal{I}(\mathcal{I}(\nu)(t_1), \dots, \mathcal{I}(\nu)(t_n)) && \text{for } f \in \Omega. \end{aligned}$$

∎

Definition 2.24
An interpretation $\mathcal{I}$ is called *term-generated*, if for every $e \in U_\mathcal{I}$, there is a ground term t such that $e = \mathcal{I}(t)$. ∎

Definition 2.25 (Herbrand Interpretations)
An interpretation $\mathcal{I}$ over Σ is called a *Herbrand interpretation* if there is a congruence relation $\cong$ on T_Σ such that $U_\mathcal{I} = T_\Sigma/_\cong$ is a quotient of T_Σ and every ground term over Σ is interpreted by its equivalence class, i.e., for every function symbol f it holds that $\mathcal{I}(f)([t_1]_\cong, \ldots, [t_n]_\cong) = [f(t_1, \ldots, t_n)]_\cong$, where $[t]_\cong$ is the $\cong$ equivalence class of t in T_Σ. ∎

Historically, the name Herbrand interpretation denotes an interpretation in which every ground term is interpreted by itself. The presence of equality gives rise to *equality Herbrand interpretations*, in which the equality predicate is interpreted as a congruence over ground terms. *Normal interpretations*, on the other hand, are interpretations in which the equality predicate is interpreted as equality on the universe. It is clear that there can be only a single normal Herbrand interpretation for any signature, namely the one in which any two ground terms are different. However, it is well-known that equality Herbrand interpretations and term-generated normal interpretations are equivalent, in the sense that they can always be transformed into one another. In particular, a term-generated normal interpretation is obtained from an equality Herbrand interpretation by interpreting ground terms by their congruence classes. This justifies Definition 2.25.

As a Herbrand interpretation is uniquely characterized by the congruence $\cong$, we will often identify the two. Moreover, we identify any congruence $\cong$ on ground terms with the set of all equations $s \simeq t$ such that $s \cong t$. It is well-known that the intersection of any set of congruences on a given algebra is again a congruence [DW09], hence also the intersection of a set of Herbrand interpretations over Σ yields a Herbrand interpretation over Σ.

Definition 2.26 (Specifications)
A *specification* or *theory* is a pair $\mathsf{Sp} = (\Sigma, \mathscr{C})$ consisting of a signature Σ and a non-empty class $\mathscr{C}$ of Herbrand interpretations over Σ. If $\mathscr{C}$ is the set of all Herbrand models of a clause set N, we also write the specification as (Σ, N). ∎

In [BGW94, Kru13], the model class $\mathscr{C}$ is only required to consist of term-generated Σ-interpretations, and, in the context of hierarchic specifications, assumed to be closed under isomorphisms. As we will only consider hierarchic specifications (in Chapter 4, Section 4.3), the slight change in our definition is justified by the fact that, as we discussed above, term-generated normal interpretations are equivalent to Herbrand interpretations, and by the fact that isomorphic Herbrand interpretations over the same signature are always equal.

Definition 2.27 (Semantics of Formulas)
Let Σ be a signature, and $\mathcal{I}$ be a Σ-interpretation. An assignment $\nu : \mathcal{X} \to U_\mathcal{I}$ *satisfies* a formula ϕ over Σ, written $\mathcal{I}, \nu \models \phi$ if

- $\phi = \top$, or
- $\phi = t_1 \simeq t_2$ and $\mathcal{I}(\nu)(t_1) = \mathcal{I}(\nu)(t_2)$, or
- $\phi = \neg\phi_1$ and $\mathcal{I}, \nu \not\models \phi_1$, or
- $\phi = \phi_1 \vee \phi_2$ and $\mathcal{I}, \nu \models \phi_1$ or $\mathcal{I}, \nu \models \phi_2$, or
- $\phi = \phi_1 \wedge \phi_2$ and $\mathcal{I}, \nu \models \phi_1$ and $\mathcal{I}, \nu \models \phi_2$, or
- $\phi = \exists x.\phi_1$, $x : S$ and $\mathcal{I}, \nu[x \mapsto a] \models \phi_1$ for some $a \in \mathcal{I}(S)$, or
- $\phi = \forall x.\phi_1$, $x : S$ and $\mathcal{I}, \nu[x \mapsto a] \models \phi_1$ for all $a \in \mathcal{I}(S)$.

A formula ϕ is *satisfiable in* $\mathcal{I}$ if $\mathcal{I}, \nu \models \phi$ for some assignment ν. It is called *satisfiable* if there is an interpretation $\mathcal{I}$ such that ϕ is satisfiable in $\mathcal{I}$, and *unsatisfiable* otherwise. The formula ϕ is *valid in* $\mathcal{I}$, written $\mathcal{I} \models \phi$, if $\mathcal{I}, \nu \models \phi$ for all assignments ν; in that case, $\mathcal{I}$ is called a *model* of ϕ. The formula ϕ is *valid* (or a *tautology*), written $\models \phi$, if ϕ is valid in any interpretation. ■

Observe that for a sentence (i.e., a closed formula) ϕ, the notions of satisfiability and validity in an interpretation coincide: $\mathcal{I}, \nu \models \phi$ for some ν if and only if $\mathcal{I} \models \phi$. By definition of validity in an interpretation then, one of $\mathcal{I} \models \phi$ or $\mathcal{I} \models \neg\phi$ always holds.

Definition 2.28 (Solutions)
Let $\mathcal{I}$ be an interpretation and ϕ a formula over Σ, and let $X \subseteq \mathcal{X}$. The set $\mathrm{Sol}_\mathcal{I}(\phi, X)$ of *solutions of* ϕ *in* $\mathcal{I}$ *with respect to* X is defined as

$$\mathrm{Sol}_\mathcal{I}(\phi, X) = \{\nu \in \mathit{Val}_\mathcal{I}(X) \mid \mathcal{I}, \nu \models \phi\}.$$

Two formulas ϕ, ϕ' are *equivalent with respect to* $\mathcal{I}$ if $\mathrm{Sol}_\mathcal{I}(\phi, X) = \mathrm{Sol}_\mathcal{I}(\phi', X)$, where X contains all free variables of ϕ and ϕ'. Two formulas are *equivalent* if they are equivalent with respect to every interpretation. Given a sequence $\vec{x} = x_1, \ldots, x_n$ containing all variables of X, we also write $\mathrm{Sol}_\mathcal{I}(\phi, \vec{x})$ for $\{\nu(\vec{x}) \mid \nu \in \mathrm{Sol}_\mathcal{I}(\phi, X)\}$, and we identify $\mathrm{Sol}_\mathcal{I}(\phi, X)$ with $\mathrm{Sol}_\mathcal{I}(\phi, \vec{x})$, where $\vec{x}$ contains the variables of X in the order given by the total ordering on $\mathcal{X}$. We write $\mathrm{Sol}_\mathcal{I}(\phi)$ for $\mathrm{Sol}_\mathcal{I}(\phi, \mathrm{var}(\phi))$. ■

Definition 2.29 (Semantics of Clauses)
A clause $C = \Gamma \to \Delta$ with $\Gamma = \{A_1, \ldots, A_m\}$, $\Delta = \{B_1, \ldots, B_n\}$ is *valid in* $\mathcal{I}$, written $\mathcal{I} \models C$, if $\mathcal{I} \models \forall(\neg A_1 \vee \ldots \vee \neg A_m \vee B_1 \vee \ldots \vee B_n)$. If $\mathcal{I}$ is a Herbrand interpretation (viewed as a set of ground equations) and C is ground, then $\mathcal{I} \models C$ if and only if $\Gamma \not\subseteq \mathcal{I}$ or $\Delta \cap \mathcal{I} \neq \emptyset$. ■

The following definition introduces the model construction originally presented in [BG94], which is central to the proof of completeness of the superposition calculus.

Definition 2.30 (Candidate Interpretation $\mathcal{I}_N$)
Let $\Sigma = (\mathcal{S}, \Omega)$ be a signature, let $\succ$ be a reduction ordering that is total on T_Σ, and let N be a set of clauses over Σ. By induction on $\succ$, we define, for each $C \in N$, the ground rewrite systems E_C and R_C, and the Herbrand interpretation $\mathcal{I}_C$. If $C = \Gamma \to \Delta, s \simeq t$ is a ground instance of a clause in N such that

(i) $s \simeq t$ is a strictly maximal occurrence of an atom in C and $s \succ t$,

(ii) s is irreducible by R_C,

(iii) $\Gamma \subseteq \mathcal{I}_C$, and

(iv) $\Delta \cap \mathcal{I}_C = \emptyset$

then $E_C = \{s \twoheadrightarrow t\}$, otherwise $E_C = \emptyset$. Furthermore, $R_C = \bigcup_{C \succ D} E_D$ and $\mathcal{I}_C = \{s \simeq t \mid s \twoheadrightarrow t \in R_C^*\}$. Finally, we let $R_N = \bigcup_C E_C$ and define the *candidate interpretation* for N as $\mathcal{I}_N = T_\Sigma / R_N$. ∎

2.4 Calculi

Definition 2.31 (Inferences, Reductions and Calculi)
An *inference rule* is an $n+1$-ary relation on clauses. Its elements are called *inferences* and written as

$$\mathcal{I}\frac{C_1 \;\ldots\; C_n}{C} \quad .$$

The clauses $C_1, \ldots, C_n$ are called the *premises*, and C the *conclusion* of the inference. A *reduction rule* is an $n+m$-ary relation on clauses. Its elements are called *reductions* and written as

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

The clauses $C_1, \ldots, C_n$ are called the *premises*, and $D_1, \ldots, D_m$ the *conclusions* of the reduction. By $\mathrm{concl}(\pi)$ we denote the conclusions of an inference or reduction π. A *calculus* $\mathtt{C}$ is a set of inference and reduction rules. ∎

An inference can be applied to a clause set N containing the premises, resulting in $N \cup \{C\}$. A reduction can be applied to a clause set N containing the premises, resulting in $(N \setminus \{C_1, \ldots, C_n\}) \cup \{D_1, \ldots, D_m\}$. A reduction can be viewed as a combination of an inference, deriving clauses $\mathrm{New} = \{D_1, \ldots, D_m\} \setminus \{C_1, \ldots, C_n\}$, followed by the removal of clauses $\mathrm{Red} = \{C_1, \ldots, C_n\} \setminus \{D_1, \ldots, D_m\}$.

Definition 2.32 (Derivations)
A *derivation* (with respect to a calculus $\mathtt{C}$) is a sequence of clause sets, denoted $N_0 \triangleright N_1 \triangleright \ldots$, such that each N_{i+1} is the result of applying an inference or a reduction (of $\mathtt{C}$) to N_i. ∎

The following notion of redundant clauses and redundant inference is the one normally used for superposition. It corresponds to the *standard redundancy criterion* by Bachmair and Ganzinger [BG01].[2]

Definition 2.33 (Redundancy)
A ground clause C is *redundant* with respect to a set N of ground clauses, if there are clauses $C_1, \ldots, C_n \in N$, $n \geq 0$, such that $C_1, \ldots, C_n \models C$ and $C \succ C_i$ for all $1 \leq i \leq n$. A non-ground clause C is redundant with respect to a set N of clauses, if all ground instances of C are redundant with respect to the set of all ground instances of clauses in N.

A ground inference with conclusion C is *redundant* with respect to a set N of ground clauses, if there are clauses $C_1, \ldots, C_n \in N$, $n \geq 0$, smaller than the maximal premise (with respect to $\succ$), such that $C_1, \ldots, C_n \models C$. A non-ground inference is redundant with respect to a set N of clauses, if all its ground instances are redundant with respect to the set of all ground instances of clauses in N. ■

Definition 2.34 (Saturation)
A clause set N is *saturated* with respect to a calculus $\mathtt{C}$ if all inferences in $\mathtt{C}$ with non-redundant premises from N are redundant in N.[3] ■

Definition 2.35 (Soundness)
An inference is *sound* if its conclusion is a logical consequence of its premises, i.e., $C_1, \ldots, C_n \models C$. A reduction is *sound* if

- $C_1, \ldots, C_n \models \text{New}$, and
- the clauses in Red are redundant with respect to $D_1, \ldots, D_m$.

where $\text{New} = \{D_1, \ldots, D_m\} \setminus \{C_1, \ldots, C_n\}$ and $\text{Red} = \{C_1, \ldots, C_n\} \setminus \{D_1, \ldots, D_m\}$. A calculus $\mathtt{C}$ is *sound* if all its inferences and reductions are sound. ■

Observe that, whenever N' is obtained by applying to N a sound inference or reduction, N and N' are logically equivalent, written as $N \models\!\models N'$.

Definition 2.36 (Refutational Completeness)
A calculus $\mathtt{C}$ is *refutationally complete* if a contradiction can be derived from an unsatisfiable clause set by finitely many applications of inferences and reductions from $\mathtt{C}$. ■

[2] In Section 3.2.1, we will give a completeness proof based on the more general notion of redundancy criterion by Bachmair and Ganzinger, of which the standard one is a special case.
[3] In [BG01], this is called *saturation up to redundancy*.

2.5 Architecture of Saturation-Based Theorem Provers

The pseudocode shown in Algorithm 2.1 is an abstract view of the SPASS [Wei01] main loop when splitting is deactivated, i.e., using only inference and reduction rules. The structure of the main loop goes back to the *Otter* theorem prover [McC03, MW97] and is common to most saturation-based provers. The procedure maintains two sets of clauses: Usable, containing the "active" clauses, which have not yet been used for inferences, and WorkedOff, containing the clauses for which inferences have already been computed. Inside the loop, a clause Given is chosen and removed from Usable (lines 4 and 5), all inferences between Given and the clauses in WorkedOff are computed (line 6), including inferences of Given with itself, yielding a set Derived of newly derived clauses. The Given clause is added to WorkedOff (line 7), and the new Derived clauses are reduced with respect to clauses in Usable and WorkedOff (line 8). This step is called *forward reduction*. Then the clauses in Usable and WorkedOff are reduced with respect to the remaining Derived clauses (line 9). This step is called *backward reduction*. Finally, the remaining Derived clauses are added to the Usable set. At this point, the sets Usable and WorkedOff are completely inter-reduced. Assuming that the inferences and reductions underlying the procedures Inf, FRed and BRed form a refutationally complete calculus, and that the function Choose implements a fair selection criterion, the procedure produces a saturated set WorkedOff, possibly in the limit, and derives the empty clause $\square$ in finitely many steps whenever N is unsatisfiable. In the latter case, SPASS will answer "Proof found" (and produce a proof if the corresponding option was active), while the answer "Completion found" signals that the set WorkedOff contains a finite saturation of the initial clause set N.

Algorithm 2.1: Main loop of SPASS without splitting

Input: Clause set N

```
1  WorkedOff := ∅;
2  Usable := N;
3  while Usable ≠ ∅ and □ ∉ Usable do
4  |  Given := Choose(Usable);
5  |  Usable := Usable \{Given};
6  |  Derived := Inf(Given, WorkedOff);
7  |  WorkedOff := WorkedOff ∪{Given};
8  |  Derived := FRed(Derived, Usable, WorkedOff);
9  |  (Usable, WorkedOff) := BRed(Usable, WorkedOff, Derived);
10 |  Usable := Usable ∪ Derived;
11 if Usable = ∅ then
12 |  print "Completion found";
13 else if □ ∈ Usable then
14 |  print "Proof found";
```

3 Labelled Splitting

3.1 Introduction

A common approach to solving combinatorial problems works by successive case distinctions. Such an approach resembles a binary search, and is based on the following idea: If the solution to the problem must assign one of two values to a variable, one assumes the first choice to be the right one, and explores the implications of that choice. This process is repeated until a solution is found[1], or it becomes obvious that there can be no solution under the current set of assumptions. This situation typically manifests itself by the appearance of a *conflict*, i.e., a set of mutually unsatisfiable constraints. In this case, some of the decisions have to be revoked, and their alternatives tried. If all possible choices have been exhausted without finding a solution, then we can conclude that the problem has no solution at all.

In the case of the Boolean satisfiability problem, this approach is the basis of the DPLL procedure [DP60, DLL62], which lies at the heart of today's most powerful SAT solvers: Given a propositional clause set N, and a propositional variable A occurring both positively and negatively in N, one assumes A to be true, making it a *decision literal*, and simplifies the clause set accordingly. Any clause containing A (as a literal) is satisfied and can be removed, while all clauses containing $\neg A$ are simplified by removing that literal. If one of the simplified clauses becomes unit, meaning it has a single literal left, that literal must be made true, and is added to the current set of assumptions. This process, known as *unit propagation*, is repeated, until either a clause becomes false, or there are no more unit clauses, at which point another decision has to be made. In case a clause has become false, the solver *backtracks* to an earlier decision and flips the corresponding decision literal. In modern solvers, this backtracking is *non-chronological*, and relies on *conflict analysis*. The goal of conflict analysis is to gain as much useful information as possible from the conflict, in order to drive the search for a solution into more fruitful directions and avoid repeating bad decisions. It is achieved by analyzing how the various assumptions interacted to create the conflict. In CDCL [SS96], the conflict analysis not only yields a backtracking level, but also a *conflict clause*, or lemma, which follows logically from the clause set and summarizes the conflict.[2] The clause is then added to the

[1] One may also want to find all solutions, or a specific solution, in which case one would continue the search.

[2] See Section 3.8 for a more detailed account of CDCL.

clause set, in a process called *clause learning*. Clause learning is a key ingredient that makes CDCL-based SAT-solvers very efficient in practice.

In first-order logic, the DPLL-style case analysis typically does not make sense, because a given first-order atom A can have infinitely many ground instances $A\sigma$, and it is not known a priori which instances eventually contribute to a proof or a model. In case of models having infinite domains, such a procedure won't terminate on satisfiable clause sets. Furthermore, the clause sets $N \cup \{A\sigma\}$ and $N \cup \{\neg A\sigma\}$ are strict subsets of N (after unit propagation and subsumption) only if $A\sigma$ already occurs in N. Therefore, in the context of clauses with variables, a different style of case analysis is used: Given a clause $C \in N$ that can be decomposed into two non-trivial variable disjoint subclauses C_1, C_2 , we *split* into the clause sets $N \cup \{C_1\}$ and $N \cup \{C_2\}$. As C_1 and C_2 both subsume C, the split clause sets are again strict subsets of N and smaller with respect to the standard superposition ordering for clause sets (see Definition 2.20). Furthermore, both clauses C_1 and C_2 contain strictly less different variables than C. This property is essential in making superposition a decision procedure for certain first-order clause fragments.

The rest of this chapter is organized as follows:

In Section 3.2, we present a formalization of splitting with backtracking for superposition, based on clause labels, and prove it to be sound and complete. The completeness in particular relies on an updated notion of derivation fairness. This work is based on our earlier publications [FW08, FW09], but has been thoroughly revised and improved.

As a new contribution, we extend the splitting calculus with a rule for conflict-driven clause learning, presented in Section 3.3. While clause learning in CDCL relies on the negation of (propositional) literals, negating a non-ground literal A requires Skolemization, which causes the introduction of fresh Skolem constants. This is a problem, because the resulting Skolemized clauses cannot be used for reductions, and also because modifying the problem signature during proof search interferes with the well-foundedness required for completeness. Until now, this problem was avoided by restricting lemma learning to ground clauses [Wei01]. We solve it by using the clause labels to track instantiations of variables in split clauses. When an empty clause has been derived, we use the instantiation information in its label to compute lemmas. These lemmas are similar to CDCL-style conflict clauses, but can be non-ground.

Section 3.4 provides an overview of our implementation of splitting with backtracking and clause learning in SPASS. In Section 3.5, we compare the old splitting calculus and its implementation from [FW09] to the new one. The results of our experimental evaluation show the superiority of the new splitting calculus over the old one. In Section 3.6, we present an experimental evaluation of clause learning, which demonstrates that clause learning, especially learning of non-ground clauses, improves the performance of SPASS on problems where splitting is used. This chapter ends with a discussion of related

work, in Section 3.7, and a more detailed comparison of our clause learning with that of CDCL, in Section 3.8.

3.2 A Calculus for Explicit Splitting

In the following, we assume a calculus `C` consisting of inference and reduction rules, but without a splitting rule, and modularly turn it into a calculus `SC` with splitting. We will assume that all inferences and reductions in `C` are sound.

We will lift the calculus `C` to a calculus `SC` with explicit splitting and backtracking, in such a way that the soundness and refutational completeness of `C` carries over to `SC`. The calculus `SC` is defined in terms of a transition system, that is, a transition relation over a set of states. We extend the notion of *derivation* (Definition 2.32) to include sequences $\mathcal{S}_0 \Longrightarrow \mathcal{S}_1 \Longrightarrow \ldots$ over states. The transition system is shown in Figure 3.1.

Definition 3.1 ($\max^+$)
Let $\alpha \subseteq \mathbb{N}$. We define

$$\max{}^+ \alpha = \begin{cases} 0 & \text{if } \alpha = \emptyset, \\ \max \alpha & \text{otherwise.} \end{cases}$$

■

Definition 3.2 (C_{+L}, C_{-L})
We assume that the literals in a clause have a fixed order and are identified by indices $0, 1, \ldots$. Given a clause C with n literals and a set $L \subseteq \{0, \ldots, n-1\}$, we write C_{+L} for the clause consisting of the literals of C identified by L, and C_{-L} for the clause consisting of the remaining literals of C. ■

Definition 3.3 (Labelled Clauses)
Labelled clauses are of the form $\alpha : C$ or $(\beta, \alpha) : C$, where $\alpha, \beta \subseteq \mathbb{N}$, and C is a clause. If N is a set of clauses, we write $\alpha : N$ for $\{\alpha : C \mid C \in N\}$. ■

Definition 3.4 (Splits, Stacks)
A *split* is a tuple $(n, \alpha : C, L)$, where $n \in \mathbb{N}$ is the *split level* of the split, $\alpha : C$ is a labelled clause, called the *parent clause* of the split, and $L \subseteq \mathbb{N}$ is a set of literal indices, such that $C = C_{+L} \vee C_{-L}$ is a variable-disjoint decomposition of C. A *stack* S is a (possibly empty) sequence $s_1, \ldots, s_n$ of splits. We write levels(S) for the set of split levels occurring in S. By an abuse of set notation, we write $\emptyset$ for the empty stack, and $S \setminus \alpha$ for the result of removing from stack S all splits with levels in $\alpha \subseteq \mathbb{N}$. ■

Definition 3.5 (States)
A *state* $\mathcal{S}$ is a tuple $S \mid M \mid N$, where S is a stack, and M, N are sets of labelled clauses. The set N contains the *active clauses* of the form $\alpha : C$, while the set M contains the *conditionally deleted clauses* of the form $(\alpha, \beta) : C$. We write levels($\mathcal{S}$) for levels(S). ■

Splitting:

$$S \mid M \mid N \quad \Longrightarrow_{\mathtt{sc}} \quad S,\, (n, \alpha : C, L) \mid M \mid N,\ \alpha \cup \{n\} : C_{+L}$$

where

(i) there is some clause $\alpha : C \in N$ such that L splits C, and

(ii) $n = (\max^{+} \text{levels}(S)) + 1$.

Inference:

$$S \mid M \mid N \quad \Longrightarrow_{\mathtt{sc}} \quad S \mid M \mid N,\, \alpha : C$$

where

(i) there are clauses $\alpha_1 : C_1, \ldots, \alpha_n : C_n \in N$, and

(ii) $\mathcal{I}\dfrac{C_1 \ldots C_n}{C}$ is an inference of $\mathtt{C}$, and

(iii) $\alpha = \alpha_1 \cup \ldots \cup \alpha_n$.

Reduction:

$$S \mid M \mid N,\, \alpha_1 : C_1, \ldots, \alpha_n : C_n \quad \Longrightarrow_{\mathtt{sc}} \quad S \mid M \cup M' \mid N,\, \beta_1 : D_1, \ldots, \beta_m : D_m$$

where

(i) $\mathcal{R}\dfrac{C_1 \ldots C_n}{D_1 \ldots D_m}$ is a reduction of $\mathtt{C}$, and

(ii) $\gamma = \alpha_1 \cup \ldots \cup \alpha_n$,

(iii) $\beta_i = \begin{cases} \alpha_j & \text{if } D_i = C_j \text{ for some } j, \\ \gamma & \text{otherwise,} \end{cases}$

(iv) $M' = \{(\gamma, \alpha_i) : C_i \mid C_i \notin \{D_1, \ldots, D_m\} \text{ and } \gamma \not\subseteq \alpha_i\}$.

Backtracking:

$$S \mid M \mid N,\, \alpha : \square \quad \Longrightarrow_{\mathtt{sc}} \quad S' \mid M \setminus M' \mid (N \setminus N') \cup N'',\, \alpha' : C_{-L_{\max \alpha}}$$

if $\alpha \neq \emptyset$, where

(i) $S' = \text{bt}(S, \alpha)$, for a backtracking function bt,

(ii) $\alpha' = \alpha \setminus \{\max \alpha\}$,

(iii) $M' = \{(\gamma, \beta) : D \in M \mid \gamma \not\subseteq \text{levels}(S') \text{ or } \beta \not\subseteq \text{levels}(S')\}$,

(iv) $N' = \{\alpha : C \in N \mid \alpha \not\subseteq \text{levels}(S')\}$,

(v) $N'' = \{\beta : D \mid (\gamma, \beta) : D \in M \text{ and } \gamma \not\subseteq \text{levels}(S') \text{ and } \beta \subseteq \text{levels}(S')\}$.

Figure 3.1: Transition Rules of a Splitting Calculus With Backtracking

The intuition behind labelled clauses, splits and stacks is the following: Each assumption made during the proof search is represented by a split, which is pushed on the stack and uniquely identified by its split level. Splits are removed from the stack upon backtracking, after an empty clause has been derived. The label of a clause $\alpha : C$ contains the levels of the splits the clause depends on, i.e., the splits that contributed to the derivation of the clause. A clause of the form $(\beta, \alpha) : C$ is a conditionally deleted clause, which depends on the split levels in α, and has been reduced (e.g., subsumed) by clauses depending on the split levels in β.

Definition 3.6 (Admissible Stacks, Clauses and States, L_k)
A stack S is called *admissible* if it satisfies the following three conditions:

(i) no two splits in S have the same level;
(ii) $n > \max^+ \alpha$ for every split $(n,\ \alpha{:}\_,\ \_)$ in S;
(iii) if $S = S'(n,\ \alpha{:}\_,\ \_)S''$, then $\alpha \subseteq \text{levels}(S')$.

Given a state $S \mid M \mid N$, a clause $\alpha : C \in N$ is *admissible* if $\alpha \subseteq \text{levels}(S)$, and a clause $(\gamma, \beta) : D \in M$ is *admissible* if $\beta, \gamma \subseteq \text{levels}(S)$. The state $S \mid M \mid N$ is called *admissible* if S is admissible and all clauses in N, M are admissible. In the context of an admissible stack, we denote by L_k the unique set of literal indices associated with the split of level k, i.e., the stack contains a split of the form $(k, \alpha : C, L_k)$. ∎

We will show in Lemma 3.20 that only admissible states are derivable in `SC` from an initial state.

Definition 3.7 (Valid Clauses, $\mathcal{S}_\gamma$, $\gamma^{\#k}$, $\mathcal{S}_{\#k}$)
We say that a clause $\alpha : C$ is *valid* with respect to a label γ if $\alpha \subseteq \gamma$. We say that a clause $(\beta, \alpha) : D$ is *valid* with respect to γ if $\alpha \subseteq \gamma$ and $\beta \not\subseteq \gamma$. For a set N of labelled clauses, we define

$$\text{valid}(N, \gamma) = \{C \mid \alpha : C \in N \text{ and } \alpha : C \text{ is valid wrt. } \gamma\},$$

and analogously for sets of doubly labelled clauses. Given a state $\mathcal{S} = (S \mid N \mid M)$ and a label γ, we define

$$\mathcal{S}_\gamma = \text{valid}(N, \gamma) \cup \text{valid}(M, \gamma).$$

Finally, for $\# \in \{<, \leq, \geq, >\}$, we define $\gamma^{\#k} = \{k' \in \gamma \mid k' \# k\}$, and we write $\mathcal{S}_{\#k}$ as shorthand for $\mathcal{S}_{\text{levels}(\mathcal{S})^{\#k}}$. ∎

For example, if $\text{levels}(\mathcal{S}) = \{1, 2, 3\}$, then $\mathcal{S}_{\leq 2} = \mathcal{S}_{\{1,2\}}$.

While admissibility of a (singly or doubly) labelled clause is a simple well-formedness condition which ensures that the labels only refer to existing splits, validity of a clause with respect to a label can be understood as follows: An active clause $\alpha : C$ is valid with respect to γ, if γ contains the levels of all splits on which $\alpha : C$ depends; a conditionally

deleted clause $(\beta, \alpha) : D$ is valid with respect to γ, if γ contains the levels of all splits on which $\alpha : D$ depends, and additionally, at least one level in β is missing from γ, meaning that the active clause $\beta : C'$ which was used to reduce $\alpha : D$ (thereby turning it into the conditionally deleted clause $(\beta, \alpha) : D$), is itself *not* valid with respect to γ, and D may thus not be redundant anymore.[3]

The set γ can be viewed as representing a context, or a set of assumptions, consisting of the split clauses at the split levels contained in γ. Hence $\mathcal{S}_\gamma$ contains all non-redundant clauses in $\mathcal{S}$ that need to be considered for inferences and reductions, under the context γ.

Definition 3.8 (Open Branches)
Let $\mathcal{S}$ be a state. For any $k \in \text{levels}(\mathcal{S})$, we define

$$\text{open}(\mathcal{S}, k) = \mathcal{S}_{<k} \cup \{C_{-L_k}\}.$$

The set of *open branches* of $\mathcal{S}$ is defined as

$$\text{open}(\mathcal{S}) = \{\mathcal{S}_{\text{levels}(\mathcal{S})}\} \cup \bigcup_{k \in \text{levels}(\mathcal{S})} \{\text{open}(\mathcal{S}, k)\}.$$

For convenience, we will identify the set $\text{open}(\mathcal{S})$ with the disjunction over its elements, i.e., with

$$\bigvee_{N \in \text{open}(\mathcal{S})} N \,. \qquad \blacksquare$$

Definition 3.9 (Satisfiability of $S \mid M \mid N$)
We call a state $\mathcal{S} = (S \mid M \mid N)$ *satisfiable* if $\text{open}(\mathcal{S})$ is satisfiable. $\blacksquare$

Lemma 3.10
Assume $\mathcal{S} \Longrightarrow_{\mathsf{SC}} \mathcal{S}'$ by an application of Inference or Reduction, and let $\delta \subseteq \text{levels}(\mathcal{S})$ be arbitrary.

(i) If ${\mathcal{S}'}_\delta \setminus \mathcal{S}_\delta \neq \emptyset$, then ${\mathcal{S}'}_\delta$ contains all premises that are not redundant with respect to ${\mathcal{S}'}_\delta$.
(ii) If $\mathcal{S}_\delta \setminus {\mathcal{S}'}_\delta \neq \emptyset$, then ${\mathcal{S}'}_\delta$ contains all conclusions.

Proof. We only consider Reduction, as Inference is a special case of it. Let Premises $= \{C_1, \ldots, C_n\}$, Conclusions $= \{D_1, \ldots, D_m\}$, New $=$ Conclusions $\setminus$ Premises and Red $=$ Premises $\setminus$ Conclusions.
(i) Let $D_i \in {\mathcal{S}'}_\delta \setminus \mathcal{S}_\delta$. Then $D_i \in$ New, so the conclusion $\gamma : D_i$ was derived, and $\gamma \subseteq \delta$. Hence $\alpha_j \subseteq \delta$ for all $j \in [1, n]$ (since $\gamma = \bigcup_j \alpha_j$), so all Premises are valid wrt. δ. Now consider an arbitrary premise $\alpha_j : C_j$ that is not redundant wrt. ${\mathcal{S}'}_\delta$. As $C_j \in \mathcal{S}_\delta$ and $C_j \notin$ Red, it follows that $C_j \in {\mathcal{S}'}_\delta$.
(ii) Let $C_i \in \mathcal{S}_\delta \setminus {\mathcal{S}'}_\delta$. Then $C_i \in$ Red, so the premise $\alpha_i : C_i$ was removed. As $\alpha_i : C_i \in \mathcal{S}_\delta$, the clause is valid wrt. δ, hence $\alpha_i \subseteq \delta$. Since $C_i \notin {\mathcal{S}'}_\delta$, it must be the case that either

[3] This will be made explicit in Lemma 3.13.

- $(\gamma, \alpha_i) : C_i \notin M'$, or
- $(\gamma, \alpha_i) : C_i \in M'$, and $(\gamma, \alpha_i) : C_i$ is not valid wrt. δ.

In the first case, it follows by definition of M' that $\gamma \subseteq \alpha_i$. As $\beta_j \subseteq \gamma$ for all $j \in [1, m]$, and $\alpha_i \subseteq \delta$, it follows that also $\beta_j \subseteq \delta$ for all $j \in [1, m]$, i.e., all Conclusions are contained in $\mathcal{S}'_\delta$. In the second case, as $\alpha_i \subseteq \delta$, the only way for $(\gamma, \alpha_i) : C_i$ to be not valid wrt. δ is that $\gamma \subseteq \delta$. As $\beta_j \subseteq \gamma$, we also know that $\beta_j \subseteq \delta$ for all $j \in [1, m]$, hence it again follows that all Conclusions are contained in $\mathcal{S}'_\delta$. □

Lemma 3.11
Assume $\emptyset \mid \emptyset \mid N_0 \implies^*_{\mathsf{SC}} S \mid M \mid N$. Then for every clause $(\beta, \alpha) : D \in M$, there exist clauses $\alpha_1 : C_1, \ldots, \alpha_n : C_n \in N$ and $(\beta_1, \alpha_{n+1}) : C_{n+1}, \ldots, (\beta_m, \alpha_{n+m}) : C_{n+m} \in M$, $n, m \geq 0$, $n+m \geq 1$, such that $\beta \supseteq \alpha_1 \cup \ldots \cup \alpha_{n+m}$ and D is redundant with respect to $C_1, \ldots, C_{n+m}$.

Proof. We proceed by induction on the derivation. The statement obviously holds for the initial state. So let us assume that $\emptyset \mid \emptyset \mid N_0 \implies^*_{\mathsf{SC}} \mathcal{S} \implies_{\mathsf{SC}} \mathcal{S}'$. By induction, the statement holds for $\mathcal{S}$. We distinguish which rule was applied to $\mathcal{S}$. It suffices to consider the rules Reduction and Backtracking, as they are the only ones modifying the set M or removing clauses from the set N.

- Reduction: We have $\mathcal{S} = S \mid M \mid N$, $\alpha_1 : C_1, \ldots, \alpha_n : C_n$ and $\mathcal{S}' = S \mid M \cup M' \mid N$, $\beta_1 : D_1, \ldots, \beta_m : D_m$. We have to establish the existence of suitable reducing clauses for each clause in $M \cup M'$. The clauses in M' are of the form $(\gamma, \alpha_i) : C_i$ and by soundness of the underlying reduction, they are redundant with respect to $D_1, \ldots, D_m$, and $\gamma = \alpha_1 \cup \ldots \cup \alpha_n = \beta_1 \cup \ldots \cup \beta_m$. Now consider a clause $(\beta, \alpha) : D \in M$. By induction, there are clauses $\alpha'_1 : C'_1, \ldots, \alpha'_k : C'_k \in N \cup \{\alpha_1 : C_1, \ldots, \alpha_n : C_n\}$ and $(\beta'_1, \alpha'_{k+1}) : C'_{k+1}, \ldots, (\beta'_l, \alpha'_{k+l}) : C'_{k+l} \in M$ such that $\beta \supseteq \alpha'_1 \cup \ldots \cup \alpha'_{k+l}$ and D is redundant with respect to $C'_1, \ldots, C'_{k+l}$. Assume that some clause $\alpha'_i : C'_i$ is not contained in $N \cup \{\beta_1 : D_1, \ldots, \beta_m : D_m\}$. Then, by soundness of the underlying reduction, C'_i is redundant with respect to $D_1, \ldots, D_m$. Now there are two cases: Either $\gamma \not\subseteq \alpha'_i$, in which case $(\gamma, \alpha'_i) : C'_i \in M'$ and the property is still satisfied for $(\beta, \alpha) : D$, as C'_i "moves" from N to M. Or $\gamma \subseteq \alpha'_i$, in which case $\alpha'_i : C'_i$ disappears and is replaced by $\beta_1 : D_1, \ldots, \beta_m : D_m$. The property is still satisfied for $(\beta, \alpha) : D$, as $\beta_1 \cup \ldots \cup \beta_m = \gamma \subseteq \alpha'_i$.
- Backtracking: We have $\mathcal{S} = S \mid M \mid N$, $\alpha : \square$ and

$$\mathcal{S}' = S \mid M \setminus M' \mid (N \setminus N') \cup N'',\ \alpha' : C_{-L_{\max \alpha}} \ .$$

  We have to establish the existence of suitable reducing clauses for each clause in $M \setminus M'$. Let $(\beta, \alpha') : D \in M \setminus M'$. By induction, there are clauses $\alpha_1 : C_1, \ldots, \alpha_n : C_n \in N \cup \{\alpha : \square\}$ and $(\beta_1, \alpha_{n+1}) : C_{n+1}, \ldots, (\beta_m, \alpha_{n+m}) : C_{n+m} \in M$, such that $\beta \supseteq \alpha_1 \cup \ldots \cup \alpha_{n+m}$ and D is redundant with respect to $C_1, \ldots, C_{n+m}$. Now the key observation is that $\alpha_i \subseteq \text{levels}(\mathcal{S}')$ holds for all $i \in [1, n+m]$, for otherwise, we would have $\beta \not\subseteq \text{levels}(S')$, implying $(\beta, \alpha') : D \in M'$, a contradiction to our

assumption. Hence all the $\alpha_i : C_i$ are still contained in $N \setminus N'$. At worst, one of the $(\beta_i, \alpha_{n+i}) : C_{n+i}$ can end up in M', but since $\alpha_{n+i} \subseteq \text{levels}(S')$, the corresponding $\alpha_{n+i} : C_{n+i}$ is contained in N'', hence the property is satisfied for $(\beta, \alpha') : D$.

□

Remark 3.12
Lemma 3.11 establishes a tree-like relation between the clauses in a state: Each $(\beta, \alpha) : D \in M$ is the root of a tree and has child nodes $\alpha_i : C_i \in N$ (leaves) and $(\beta_j, \alpha_{n+j}) : C_{n+j} \in M$ (internal nodes), and each internal node is redundant with respect to its child nodes, as illustrated in Figure 3.2. Simple circles represent clauses in N, double circles represent clauses in M. C_1 is redundant wrt. C_2, C_3, C_4 and $\beta_1 = \alpha_2 \cup \alpha_3 \cup \alpha_4$. C_2 is redundant wrt. C_5 and $\beta_2 = \alpha_5$. C_5 is redundant wrt. C_6 and $\alpha_6 \subseteq \alpha_5$. ■

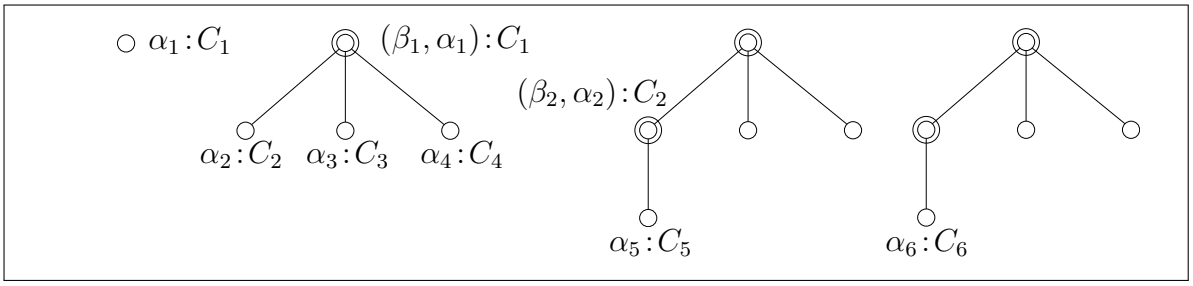


Figure 3.2: Three Reduction steps (left to right) illustrating Lemma 3.11

Lemma 3.13
Assume $\emptyset \mid \emptyset \mid N_0 \implies_{\mathsf{SC}}^* \mathcal{S} = (S \mid M \mid N)$. Any clause $(\beta, \alpha) : D \in M$ is redundant with respect to $\mathcal{S}_\gamma$ whenever $\beta \subseteq \gamma$.

Proof. We proceed by induction on M, ordering the clauses in M by increasing height of their associated tree.[4] So let $(\beta, \alpha) : D$ be an arbitrary clause in M with $\beta \subseteq \gamma$, and let $\alpha_i : C_i \in N$, $i \in [1, n]$, and $(\beta_j, \alpha_{n+j}) : C_{n+j} \in M$, $j \in [1, m]$ be its child clauses, as per Lemma 3.11. In the base case, $(\beta, \alpha) : D$ is the root of a tree of height 1, and $n > 0, m = 0$. For the inductive step, we have $n \geq 0, m > 0$ and the inductive hypothesis holds for the $(\beta_j, \alpha_{n+j}) : C_{n+j}$. As $\beta \subseteq \gamma$, it follows from Lemma 3.11 that $\alpha_1 \cup \ldots \cup \alpha_n \subseteq \gamma$, hence $C_i \in \text{valid}(N, \gamma) \subseteq \mathcal{S}_\gamma$, for all $i \in [1, n]$. This covers the base case. Now for the inductive step, consider an arbitrary child clause $(\beta_j, \alpha_{n+j}) : C_{n+j}$, $j \in [1, m]$. As $\beta \subseteq \gamma$, it follows from Lemma 3.11 that $\alpha_{n+j} \subseteq \gamma$. If $\beta_j \in \gamma$, then by induction, C_{n+j} is redundant with respect to $\mathcal{S}_\gamma$. Otherwise, $\beta_j \notin \gamma$, hence $C_{n+j} \in \text{valid}(M, \gamma) \subseteq \mathcal{S}_\gamma$. In sum, as D is redundant with respect to $C_1, \ldots, C_{n+m}$ and each C_i, $i \in [1, n+m]$ is either contained in $\mathcal{S}_\gamma$ or redundant with respect to it, it follows that D itself is redundant with respect to $\mathcal{S}_\gamma$. □

Lemma 3.14
Assume $\emptyset \mid \emptyset \mid N_0 \implies_{\mathsf{SC}}^* \mathcal{S}$. If $\alpha \subseteq \beta$, then $\mathcal{S}_\alpha \subseteq \mathcal{S}_\beta \cup \text{Red}_\beta$, where all clauses in Red_β are redundant with respect to $\mathcal{S}_\beta$.

[4] As defined in Remark 3.12

Proof. Let $\mathcal{S} = (S \mid M \mid N)$ and $\alpha \subseteq \beta$. Clearly, $\text{valid}(N, \alpha) \subseteq \text{valid}(N, \beta)$, so it remains to show that $\text{valid}(M, \alpha) \subseteq \text{valid}(M, \beta) \cup \text{Red}_\beta$. Let $D \in \text{valid}(M, \alpha)$. Then there is $(\beta', \alpha') : D \in M$ such that $\alpha' \subseteq \alpha$ (implying $\alpha' \subseteq \beta$) and $\beta' \not\subseteq \alpha$. If also $\beta' \not\subseteq \beta$, then $D \in \text{valid}(M, \beta)$. Otherwise, $\beta' \subseteq \beta$, thus by Lemma 3.13, D is redundant with respect to $\mathcal{S}_\beta$ and hence $D \in \text{Red}_\beta$. □

As an immediate consequence of Lemma 3.14, we get that $\alpha \subseteq \beta$ implies $\mathcal{S}_\beta \models \mathcal{S}_\alpha$. Thus for any k, we get $\mathcal{S}_{\leq k} \models \mathcal{S}_{<k}$, and for $k \leq k'$, we get $\mathcal{S}_{\alpha^{\leq k'}} \models \mathcal{S}_{\alpha^{\leq k}}$ and $\mathcal{S}_{\leq k'} \models \mathcal{S}_{\leq k}$.

Definition 3.15 (Dependency Graph)
Given a stack S, the *dependency graph* of S is $\text{DG}(S) = (V, E)$, where V consists of $\emptyset$ and $\alpha \cup \{n\}$ for every split $(n,\ \alpha : \_,\ \_)$ in S, and $E \subseteq V \times V$ is the smallest set containing $(\alpha_i, \alpha \cup \{n\})$ for all $i \in [1, m]$, whenever $\alpha = \alpha_1 \cup \ldots \cup \alpha_m$. The set of *descendants* of a split level k wrt. S is defined[5] as

$$\text{desc}_S(k) = \{k' \mid \text{there is } \gamma \in \text{DG}(S) \text{ such that } k \in \gamma \text{ and } k' = \max \gamma\},$$

and extended to labels by

$$\text{desc}_S(\alpha) = \bigcup_{k \in \alpha} \text{desc}_S(k)\ .$$ ■

Lemma 3.16
Let S be an admissible stack. For any union α of nodes in $\text{DG}(S)$, there exist unique nodes $\alpha_1, \ldots, \alpha_n \in \text{DG}(S)$, such that $\alpha = \alpha_1 \cup \ldots \cup \alpha_n$, and no two α_i, α_j are subsets of each other.

Proof. Let $\alpha_1, \ldots, \alpha_n \in \text{DG}(S)$ be arbitrary such that $\alpha = \alpha_1 \cup \ldots \cup \alpha_n$ and $\alpha_i \not\subseteq \alpha_j$ whenever $i \neq j$. Observe that, by definition of $\text{DG}(S)$, for every $k \in \text{levels}(S)$, there is a unique γ with $\max \gamma = k$, and $k \in \beta$ implies $\gamma \subseteq \beta$. Now assume for contradiction that $\alpha = \beta_1 \cup \ldots \cup \beta_m \cup \beta$, where β is not a subset of any β_j, and β is different from all α_i. Consider any α_i with $\max \beta \in \alpha_i$: By assumption, $\beta \neq \alpha_i$, so it follows that $\beta \subset \alpha_i$. Now consider any β_j with $\max \alpha_i \in \beta_j$: We have $\alpha_i \subseteq \beta_j$ and hence $\beta \subset \beta_j$, a contradiction. □

Lemma 3.17
Let S be an admissible stack. Let α be a union of $\text{DG}(S)$-nodes, and let $k \in \alpha$ be arbitrary. Then $\alpha^{<k}$ is also a union of $\text{DG}(S)$-nodes.

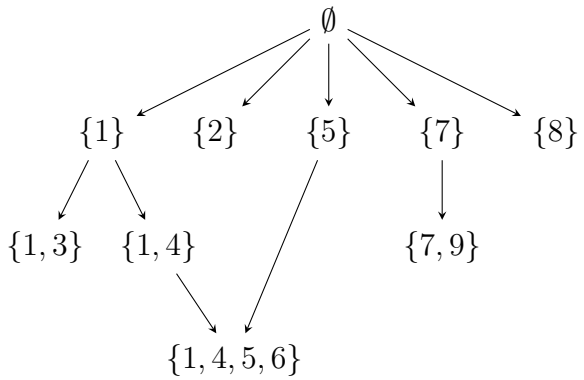
Proof. Observe that $\alpha^{<k} = ((\ldots(\alpha^{<k_m})\ldots)^{<k_1})^{<k}$ if there are $k_m > \ldots k_1 > k \in \alpha$, hence we can assume without loss of generality that $k = \max \alpha$. So let $\alpha = \alpha_1 \cup \ldots \cup \alpha_n$ and α_i be the unique node with $\max \alpha_i = k$. Then we obtain $\alpha^{<k}$ from α, by replacing α_i with the union of its ancestors in $\text{DG}(S)$. □

[5] We write $\gamma \in \text{DG}(S)$ for $\gamma \in V$.

Example 3.18
Consider the stack

$$\begin{aligned} S = (1,\, \emptyset{:}\_,\, \_), (2,\, \emptyset{:}\_,\, \_), (3,\, \{1\}{:}\_,\, \_), \\ (4,\, \{1\}{:}\_,\, \_), (5,\, \emptyset{:}\_,\, \_), (6,\, \{1,4,5\}{:}\_,\, \_), \\ (7,\, \emptyset{:}\_,\, \_), (8,\, \emptyset{:}\_,\, \_), (9,\, \{7\}{:}\_,\, \_). \end{aligned}$$

S is admissible, and $\mathrm{DG}(S)$ looks as follows:



Observe that $\mathrm{desc}_S(1) = \{1, 3, 4, 6\}$. ∎

Definition 3.19 (Backtracking Function)
Given an admissible stack S and non-empty $\alpha \subseteq \mathrm{levels}(S)$, a *backtracking function* is any function defined by

$$(S, \alpha) \mapsto S \setminus \mathrm{desc}_S(\beta)$$

where β is any set with $\beta \cap \alpha = \{\max \alpha\}$. We distinguish three backtracking functions:

- *regular backtracking* (or *backjumping*), with[6] $\beta = \mathrm{levels}(S)^{\geq \max \alpha}$,
- *eager backtracking* (or *branch condensing*), with $\beta = (\mathrm{levels}(S) \setminus \alpha) \cup \{\max \alpha\}$,
- *lazy backtracking*, with $\beta = \{\max \alpha\}$. ∎

Regular backtracking consists in removing the levels below the last level that contributed to the conflict, while eager backtracking removes all levels that did not contribute to the conflict. Lazy backtracking only removes the last level that contributed to the conflict. The three backtracking modes are sound, and one could expect eager backtracking to perform better on unsatisfiable problems, while lazy backtracking might do better on satisfiable problems. We leave these questions as future work however, and focus on regular backtracking for the rest of this chapter.

Lemma 3.20
Assume $\emptyset \mid \emptyset \mid N_0 \implies_{\mathsf{SC}}^{*} \mathcal{S}$. Then $\mathcal{S}$ is admissible.

[6] See Definition 3.7

Proof. We proceed by induction on the derivation. The initial state is obviously admissible. So let $\mathcal{S} = (S \mid M \mid N)$, $\mathcal{S}' = (S' \mid M' \mid N')$ and assume $\emptyset \mid \emptyset \mid N_0 \implies_{\mathsf{sc}}^* \mathcal{S} \implies_{\mathsf{sc}} \mathcal{S}'$. By induction, $\mathcal{S}$ is admissible. We distinguish which rule was applied:

- Splitting: S' obviously satisfies conditions (i) and (ii) of Definition 3.6 by induction; furthermore, $\alpha : C$ is admissible by induction hence S' also satisfies condition (iii), and the clause $\alpha \cup \{n\} : C_{+L}$ is admissible.
- Inference, Reduction: $S' = S$, and admissibility of all newly derived clauses follows immediately from the definition.
- Backtracking: S' obviously satisfies conditions (i) and (ii) of Definition 3.6 by induction; furthermore, $(\alpha \setminus \{\max \alpha\}) : C_{-L_{\max \alpha}}$ is admissible, as $(\alpha \setminus \{\max \alpha\}) \subseteq \text{levels}(S')$. Admissibility of the clauses in N'' follows immediately from the definition. It remains to show that S' satisfies condition (iii) of Definition 3.6: Assume not. Then there must be two splits $(n_i,\ \alpha_i : \_,\ \_)$ and $(n_j,\ \alpha_i : \_,\ \_)$ in S, such that $n_j \in \text{levels}(S')$, $n_i \notin \text{levels}(S')$ and $n_i \in \alpha_j$. From $n_i \notin \text{levels}(S')$, it follows, by definition of the backtracking function, that $n_i \in \text{desc}_S(\beta)$. Since $n_i \in \alpha_j$, and thus $n_i \in \alpha_j \cup \{n_j\}$, it follows from the definition of $\text{DG}(S)$ that also $n_j \in \text{desc}_S(\beta)$. But then $n_j \notin \text{levels}(S')$, a contradiction.

We conclude that $\mathcal{S}'$ is admissible. □

Justified by Lemma 3.20, we will assume that all considered states in the following are admissible. In particular, this justifies the notation C_{+L_k}, C_{-L_k} for any $k \in \text{levels}\,\mathcal{S}$.

Invariant 3.21
Let $\mathcal{S} = (S \mid M \mid N)$ be a state. Let $\gamma \neq \emptyset$ be a union of nodes in $\text{DG}(S)$, and let $k = \max \gamma$. Then[7]

$$\mathcal{S}_{\gamma^{<k}} \models \mathcal{S}_\gamma \vee \mathcal{S}_{\gamma^{<k}}, C_{-L_k}.$$ ■

Lemma 3.22
Let $\mathcal{S} = (S \mid M \mid N)$ be a state satisfying Invariant 3.21. Let $\alpha = \{k_1, \ldots, k_m\}$ be a union of nodes in $\text{DG}(S)$, such that $k_1 < \ldots < k_m$. Then

$$\mathcal{S}_\emptyset \models\!\models \mathcal{S}_\alpha \vee \mathcal{S}_{\alpha^{<k_m}}, C_{-L_{k_m}} \vee \ldots \vee \mathcal{S}_{\alpha^{<k_2}}, C_{-L_{k_2}} \vee \mathcal{S}_\emptyset, C_{-L_{k_1}}.$$

Proof. We have $\mathcal{S}_\emptyset = \mathcal{S}_{\alpha^{<k_1}}$. By Lemma 3.17, since α is a union of nodes, so is every $\alpha^{<k_i}$. Hence, by Invariant 3.21, we have

$$\begin{aligned}
\mathcal{S}_{\alpha^{<k_1}} &\models \mathcal{S}_{\alpha^{<k_2}} \vee \mathcal{S}_{\alpha^{<k_1}}, C_{-L_{k_1}}, \\
\mathcal{S}_{\alpha^{<k_2}} &\models \mathcal{S}_{\alpha^{<k_3}} \vee \mathcal{S}_{\alpha^{<k_2}}, C_{-L_{k_2}}, \\
&\vdots \\
\mathcal{S}_{\alpha^{<k_m}} &\models \mathcal{S}_\alpha \vee \mathcal{S}_{\alpha^{<k_m}}, C_{-L_{k_m}}.
\end{aligned}$$

[7] We treat the clause sets $\mathcal{S}_\gamma$, $\mathcal{S}_{\gamma^{<k}}$ as conjunctions of their clauses.

By induction on m, it follows that

$$\mathcal{S}_\emptyset \models \mathcal{S}_\alpha \vee \mathcal{S}_{\alpha^{<k_m}}, C_{-L_{k_m}} \vee \ldots \vee \mathcal{S}_{\alpha^{<k_2}}, C_{-L_{k_2}} \vee \mathcal{S}_\emptyset, C_{-L_{k_1}}.$$

We can strengthen the entailment to an equivalence, because $\mathcal{S}_\emptyset$ is a subset of each of the sets $\mathcal{S}_\alpha, \mathcal{S}_{\alpha^{<k_m}}, \ldots, \mathcal{S}_\emptyset$. □

Observe that by taking $\alpha = \mathrm{levels}(\mathcal{S})$, Lemma 3.22 yields $\mathcal{S}_\emptyset \models\!\models \mathrm{open}(\mathcal{S})$.

Lemma 3.23
Let $\mathcal{S} = (S \mid M \mid N)$ be a state satisfying Invariant 3.21, such that $\alpha : \square \in N$, and let $\alpha' = \alpha \setminus \{\max \alpha\}$. Then $\mathcal{S}_\delta \models C_{-L_{\max \alpha}}$ whenever $\alpha' \subseteq \delta$.

Proof. By Invariant 3.21, $\mathcal{S}_{\alpha'} \models \mathcal{S}_\alpha \vee \mathcal{S}_{\alpha'}, C_{-L_{\max \alpha}}$. As $\square \in \mathcal{S}_\alpha$, it follows that $\mathcal{S}_{\alpha'} \models \mathcal{S}_{\alpha'}, C_{-L_{\max \alpha}}$ and thus $\mathcal{S}_{\alpha'} \models C_{-L_{\max \alpha}}$. If $\alpha' \subseteq \delta$, then $\mathcal{S}_\delta \models \mathcal{S}_{\alpha'}$ (by Lemma 3.14), and hence also $\mathcal{S}_\delta \models C_{-L_{\max \alpha}}$. □

Lemma 3.24
Assume $\mathcal{S} \Longrightarrow_{\mathsf{SC}} \mathcal{S}'$ by an application of Backtracking. Then, for all $\delta \subseteq \mathrm{levels}(\mathcal{S}')$, it holds that

$$\mathcal{S}'_\delta = \begin{cases} \mathcal{S}_\delta \cup \{C_{-L_{\max \alpha}}\} & \text{if } \alpha' \subseteq \delta, \\ \mathcal{S}_\delta & \text{otherwise.} \end{cases}$$

Proof. Let $\delta' = \mathrm{levels}(\mathcal{S}')$. First observe that

$$\begin{aligned} \mathrm{valid}((N \setminus N') \cup N'', \delta) &= \{C \mid \beta : C \in (N \setminus N') \cup N'' \wedge \beta \subseteq \delta\} \\ &= \{C \mid \beta : C \in N \wedge \beta \subseteq \delta\} \setminus \\ &\quad \{C \mid \beta : C \in N \wedge \beta \not\subseteq \delta' \wedge \beta \subseteq \delta\} \cup \\ &\quad \{C \mid (\gamma, \beta) : D \in M \wedge \gamma \not\subseteq \delta' \wedge \beta \subseteq \delta' \wedge \beta \subseteq \delta\} \\ &= \{C \mid \beta : C \in N \wedge \beta \subseteq \delta\} \cup \\ &\quad \{C \mid (\gamma, \beta) : D \in M \wedge \gamma \not\subseteq \delta \wedge \beta \subseteq \delta\} \\ &= \mathrm{valid}(N, \delta) \cup \mathrm{valid}(M, \delta) \end{aligned}$$

and

$$\begin{aligned} \mathrm{valid}(M \setminus M', \delta) &= \{D \mid (\gamma, \beta) : D \in M \wedge \beta \subseteq \delta \wedge \gamma \not\subseteq \delta\} \setminus \\ &\quad \{D \mid (\gamma, \beta) : D \in M \wedge (\gamma \not\subseteq \delta' \vee \beta \not\subseteq \delta') \wedge \gamma \not\subseteq \delta \wedge \beta \subseteq \delta\} \\ &= \mathrm{valid}(M, \delta) \setminus \emptyset \\ &= \mathrm{valid}(M, \delta). \end{aligned}$$

Therefore

$$\begin{aligned}
\mathcal{S}'_\delta &= \mathrm{valid}((N \setminus N') \cup N'' \cup \{\alpha' : C_{-L_{\max\alpha}}\}, \delta) \cup \mathrm{valid}(M \setminus M', \delta) \\
&= \mathrm{valid}((N \setminus N') \cup N'', \delta) \cup \mathrm{valid}(\{\alpha' : C_{-L_{\max\alpha}}\}, \delta) \cup \mathrm{valid}(M \setminus M', \delta) \\
&= \mathrm{valid}(N, \delta) \cup \mathrm{valid}(M, \delta) \cup \mathrm{valid}(\{\alpha' : C_{-L_{\max\alpha}}\}, \delta) \\
&= \mathcal{S}_\delta \cup \mathrm{valid}(\{\alpha' : C_{-L_{\max\alpha}}\}, \delta) \\
&= \begin{cases} \mathcal{S}_\delta \cup \{C_{-L_{\max\alpha}}\} & \text{if } \alpha' \subseteq \delta, \\ \mathcal{S}_\delta & \text{otherwise.} \end{cases}
\end{aligned}$$

□

Lemma 3.25
Assume $\mathcal{S} \implies_{\mathsf{SC}} \mathcal{S}'$ by an application of Backtracking, such that $\mathcal{S}$ satisfies Invariant 3.21. Then $\mathcal{S}_\delta \models\!\!\models \mathcal{S}'_\delta$ for all $\delta \subseteq \mathrm{levels}(\mathcal{S}')$.

Proof. Follows immediately from Lemmas 3.24 and 3.23. □

Theorem 3.26 (Soundness of $\implies_{\mathsf{SC}}$)
Assume $\emptyset \mid \emptyset \mid N_0 \implies^*_{\mathsf{SC}} \mathcal{S}$. Then $N_0 \models\!\!\models \mathrm{open}(\mathcal{S})$.

Proof. We prove the stronger statement

(i) $\mathcal{S}$ satisfies Invariant 3.21, and
(ii) $N_0 \models\!\!\models \mathrm{open}(\mathcal{S})$,

by induction on the derivation: For $\mathcal{S} = (\emptyset \mid \emptyset \mid N_0)$, $\mathrm{DG}(\emptyset)$ has $\emptyset$ as only node, and (i) and (ii) hold trivially. So let $\mathcal{S} = (S \mid M \mid N)$, $\mathcal{S}' = (S' \mid M' \mid N')$ and assume $\emptyset \mid \emptyset \mid N_0 \implies^*_{\mathsf{SC}} \mathcal{S} \implies_{\mathsf{SC}} \mathcal{S}'$. By induction, $\mathcal{S}$ satisfies (i) and (ii). We distinguish which rule was applied:

- Splitting: For (i), observe that $\mathrm{DG}(S')$ has one additional node, namely $\alpha \cup \{n\}$, and that $\mathcal{S}'_\beta = \mathcal{S}_\beta$ for all nodes β in $\mathrm{DG}(S)$. Furthermore, $n > \max(\gamma \cup \alpha)$ for any union γ of nodes in $\mathrm{DG}(S)$. Therefore it suffices to show that

$$\mathcal{S}_{\gamma\cup\alpha} \models \mathcal{S}_{\gamma\cup\alpha\cup\{n\}} \vee \mathcal{S}_{\gamma\cup\alpha}, C_{-L}.$$

  This is straightforward, since $C \in \mathcal{S}_\alpha$ and $\mathcal{S}_{\alpha\cup\gamma} \models \mathcal{S}_\alpha$ (by Lemma 3.14), and $\mathcal{S}_{\gamma\cup\alpha\cup\{n\}} = \mathcal{S}_{\gamma\cup\alpha} \cup \{C_{+L}\}$.
  For (ii), let $\mathrm{levels}(\mathcal{S}) = \{l_1, \ldots, l_m\}$, $n = l_m + 1$, and observe that

$$\begin{aligned}
\mathrm{open}(\mathcal{S}) &= \{\mathrm{open}(\mathcal{S}, l_1),\ \ldots,\ \mathrm{open}(\mathcal{S}, l_m),\ \mathcal{S}_{<n}\},\ \text{and} \\
\mathrm{open}(\mathcal{S}') &= \{\mathrm{open}(\mathcal{S}', l_1),\ \ldots,\ \mathrm{open}(\mathcal{S}', l_m),\ \mathrm{open}(\mathcal{S}', n),\ \mathcal{S}'_{\leq n}\} \\
&= \{\mathrm{open}(\mathcal{S}, l_1),\ \ldots,\ \mathrm{open}(\mathcal{S}, l_m),\ \mathcal{S}_{<n} \cup \{C_{+L}\},\ \mathcal{S}_{<n} \cup \{C_{-L}\}\}.
\end{aligned}$$

  As $C \in \mathcal{S}_{<n}$, it follows that $\mathrm{open}(\mathcal{S}) \models\!\!\models \mathrm{open}(\mathcal{S}')$, and thus (ii) holds by induction.

- Inference, Reduction: For (i), observe that $S' = S$. For (i) and (ii), let $\mathcal{S}^*$ be one of $\mathcal{S}_\gamma$, $\mathcal{S}_{\gamma^{<k}}$, $\mathrm{open}(\mathcal{S}, k)$, or $\mathcal{S}_{\leq n}$ for $n = \max \mathrm{levels}(\mathcal{S})$ and arbitrary $k \in \mathrm{levels}(\mathcal{S})$, $\gamma \subseteq \mathrm{levels}(\mathcal{S})$, and let $\mathcal{S}'^*$ be the analogous set for $\mathcal{S}'$. Assume that $\mathcal{S}'^* \neq \mathcal{S}^*$. If $\mathcal{S}'^* \setminus \mathcal{S}^* \neq \emptyset$, then $\mathcal{S}'^*$ contains a newly derived conclusion, and, by Lemma 3.10, $\mathcal{S}'^*$ also contains all non-redundant premises. If $\mathcal{S}^* \setminus \mathcal{S}'^* \neq \emptyset$, then some redundant clause was removed, and, by Lemma 3.10, $\mathcal{S}'^*$ also contains all conclusions. Hence, by soundness of the underlying reduction, $\mathcal{S}^* \models\!\models \mathcal{S}'^*$, and thus (i) and (ii) hold by induction.
- Backtracking: For (i), note that every node of $\mathrm{DG}(S')$ is a node of $\mathrm{DG}(S)$. By induction and Lemma 3.25, we can deduce that $\mathcal{S}'$ also satisfies Invariant 3.21. For (ii), it follows from Lemma 3.25 that $\mathrm{open}(\mathcal{S}) \models\!\models \mathrm{open}(\mathcal{S}')$, and thus (ii) holds by induction.

□

More generally, we can observe that, given a state $\mathcal{S} = S \mid M \mid N$, adding labelled clauses N' to N preserves soundness as long as $\mathcal{S}_\delta \models \mathrm{valid}(N', \delta)$ holds for all $\delta \subseteq \mathrm{levels}(\mathcal{S})$. This motivates the following definition:

Definition 3.27 (Labelled Inference Rule)
A *labelled inference rule* is any function mapping a state $\mathcal{S} = S \mid M \mid N$ to a set N' of labelled clauses. The rule is *sound* if $\mathcal{S}_\delta \models \mathrm{valid}(N', \delta)$ holds for all $\delta \subseteq \mathrm{levels}(\mathcal{S})$. ■

We have already established (in Lemma 3.10 and Theorem 3.26) that inference rules of `C` yield sound labelled inference rules of `SC` via the Inference rule. It is easy to see that `SC` can be extended by sound labelled inference rules without affecting soundness of `SC`, by adding transitions

$$S \mid M \mid N \implies_{\mathtt{SC}} S \mid M \mid N \cup N'$$

where N' is the result of applying the labelled inference rule to N. In Section 3.3, we will consider clause learning as an example of a labelled inference rule which is not lifted from an underlying inference rule of `C`.

3.2.1 Completeness

Refutational completeness of a superposition-based (or more generally, saturation-based) theorem proving system relies on static as well as dynamic conditions.

Static completeness refers to the refutational completeness of the calculus, and requires that there exist a derivation of the empty clause from any unsatisfiable clause set, or, equivalently, that any unsatisfiable set saturated by the calculus contains the empty clause. Refutational completeness of superposition relies on the definition of a model

functor, which maps any clause set N to the candidate interpretation $\mathcal{I}_N$ (see Definition 2.30), which is a model of N whenever N is saturated and does not contain the empty clause.

Dynamic completeness concerns the strategy with which inferences and reductions are applied, and requires that any derivation following the strategy eventually produces a saturated set (possibly in the limit). Dynamic completeness relies on *fairness*, which essentially requires that no inference be postponed forever if it is required for saturation. Formally, given a derivation $N_0, N_1, \ldots$ in a calculus without splitting, one defines the set $N_\infty = \bigcup_i \bigcap_{j \geq i} N_j$ of *persistent clauses*. The derivation is fair if every inference with premises in N_∞ is redundant with respect to $\bigcup_j N_j$. This captures the intuitive notion of fairness, because inferences become redundant when their conclusions are added.

When splitting with explicit backtracking is added to the calculus however, this notion of fairness is no longer sufficient to guarantee dynamic completeness. This is because the proof of dynamic completeness assumes the derivation to be "monotonic", in the sense that each step $N_i \triangleright N_{i+1}$ consists of adding derived clauses and/or removing redundant clauses, and redundant clauses stay redundant forever. This property does not hold for derivations in SC: The Backtracking rule removes clauses that depend on removed splits, but are not necessarily redundant, and the Reduction rule removes clauses that are redundant with respect to the current set N_i, but may become non-redundant later.

The following example shows a derivation in SC that is fair in the classical sense, yet fails to produce a saturated set.

Example 3.28
Consider the clause set N_0 containing the following clauses:

$$\begin{array}{lrl}
(1) & \emptyset: & \to S(a) \\
(2) & \emptyset: & S(a) \to \\
(3) & \emptyset: & \to P(a) \\
(4) & \emptyset: & P(x) \to Q(y), P(f(x)) \\
(5) & \emptyset: & Q(x) \to S(y)
\end{array}$$

Because of (1), (2), the set is unsatisfiable. Now we derive

$$\begin{array}{rrl}
\text{Res(3,4)=(6)} & \emptyset: & \to Q(x), P(f(a)) \\
\text{Split(6)=(7)} & \{1\}: & \to Q(x) \\
\text{Res(7,5)=(8)} & \{1\}: & \to S(x)
\end{array}$$

and use (8) to subsume (1), removing it from the active clause set. Finally, we derive

$$\begin{array}{rrl}
\text{Res(8,2)=(9)} & \{1\}: & \square \\
\text{Backtrack(9)=(10)} & \emptyset: & \to P(f(a))
\end{array}$$

at which point (1) is reinserted. We now repeat the same steps with (10) in place of (3), and so on, producing an infinite derivation. Because of the repeated subsumption, (1) is not persistent, and the derivation is fair in the sense discussed above. But the "real" contradiction $\emptyset : \square$ is never derived. ■

In the light of the above example, we have to modify the notions of persistent clause and fair derivation in order to obtain dynamic completeness for SC.

Definition 3.29 (Persistent Levels, Persistent Clauses)
Consider a derivation $\mathcal{S}_0 \Longrightarrow_{\mathtt{SC}} \mathcal{S}_1 \Longrightarrow_{\mathtt{SC}} \ldots$ We define the set of *persistent levels* at step $i \geq 0$ as

$$\text{levels}_i^\infty = \bigcap_{j \geq i} \text{levels}(\mathcal{S}_j),$$

and define

$$\mathcal{S}_i^\infty = (\mathcal{S}_i)_{\text{levels}_i^\infty}.$$

The set of *persistent clauses* is defined as

$$\mathcal{S}_\infty = \bigcup_i \bigcap_{j \geq i} \mathcal{S}_j^\infty.$$

■

In Example 3.28 above, the stack alternates between being empty and consisting of a single split with level 1. Hence $\text{levels}_i^\infty = \emptyset$, for all $i \geq 0$. The subsumption of (1) by (8) causes (1) to be removed by N, and added to M as $(\{1\}, \emptyset) : \rightarrow S(a)$. This clause however is valid with respect to $\emptyset$, hence both clauses $\rightarrow S(a)$ and $S(a) \rightarrow$ are contained in $\mathcal{S}_i^\infty$, for all $i \geq 0$.

The following lemma states that $(\mathcal{S}_i^\infty)_{i \geq 0}$ forms a "monotonic" sequence, as opposed to $(N_i)_{i \geq 0}$:

Lemma 3.30
For every $i \geq 0$, $\mathcal{S}_{i+1}^\infty = (\mathcal{S}_i^\infty \cup \text{New}) \setminus \text{Red}$, where the clauses in Red are redundant with respect to $\mathcal{S}_i^\infty \cup \text{New}$.

Proof. Inference is trivial. For Reduction, observe that as in the soundness proof, the conclusions are in $\mathcal{S}_{i+1}^\infty$ whenever $\text{Red} \neq \emptyset$, and hence all clauses in Red are redundant in $\mathcal{S}_{i+1}^\infty$. For Backtracking, observe that $\text{levels}_{i+1}^\infty = \text{levels}_i^\infty$, as no new splits are added, and any removed split level cannot have been in levels_i^∞. By Lemma 3.24, it follows that $\text{Red} = \emptyset$. □

Note that in Lemma 3.30, we don't care about the set New, because for completeness, we only need to ensure that the successive sets $\mathcal{S}_i^\infty$ don't become too weak.

A *redundancy criterion* [BG01] is a pair $\mathcal{R} = (\mathcal{R}_\mathcal{F}, \mathcal{R}_\mathcal{I})$ of mappings which associate with each clause set a set of clauses and a set of inferences, respectively, that are deemed to

be redundant with respect to that clause set. An *effective* redundancy criterion, such as the standard redundancy criterion employed in superposition, satisfies the following four requirements, for all clause sets N and N':

(R1) if $N \subseteq N'$, then $\mathcal{R}_\mathcal{F}(N) \subseteq \mathcal{R}_\mathcal{F}(N')$ and $\mathcal{R}_\mathcal{I}(N) \subseteq \mathcal{R}_\mathcal{I}(N')$;

(R2) if $N' \subseteq \mathcal{R}_\mathcal{F}(N)$, then $\mathcal{R}_\mathcal{F}(N) \subseteq \mathcal{R}_\mathcal{F}(N \setminus N')$ and $\mathcal{R}_\mathcal{I}(N) \subseteq \mathcal{R}_\mathcal{I}(N \setminus N')$;

(R3) if N is inconsistent, then $N \setminus \mathcal{R}_\mathcal{F}(N)$ is also inconsistent;

(R4) an inference is in $\mathcal{R}_\mathcal{I}(N)$ whenever its conclusion is in $N \cup \mathcal{R}_\mathcal{F}(N)$.

In the following, we assume an effective redundancy criterion $\mathcal{R} = (\mathcal{R}_\mathcal{F}, \mathcal{R}_\mathcal{I})$. The following is a slightly modified version of Lemma 4.2 from [BG01]:

Lemma 3.31
Let $\mathcal{S}_0 \Longrightarrow_{\mathtt{SC}} \mathcal{S}_1 \Longrightarrow_{\mathtt{SC}} \ldots$ be a derivation. Then

$$\mathcal{R}(\bigcup_j \mathcal{S}_j^\infty) \subseteq \mathcal{R}(\mathcal{S}_\infty)$$

(where $\mathcal{R}$ stands for $\mathcal{R}_\mathcal{F}$ and $\mathcal{R}_\mathcal{I}$) and $\mathcal{S}_\infty \models N_0$.

Proof. By Lemma 3.30, any clause in $\bigcup_j \mathcal{S}_j^\infty$ but not in $\mathcal{S}_\infty$ must be in some $\mathcal{R}_\mathcal{F}(\mathcal{S}_i^\infty)$. Hence $(\bigcup_j \mathcal{S}_j^\infty) \setminus \mathcal{S}_\infty \subseteq \bigcup_j \mathcal{R}_\mathcal{F}(\mathcal{S}_j^\infty)$. By (R1), we have $\bigcup_j \mathcal{R}_\mathcal{F}(\mathcal{S}_j^\infty) \subseteq \mathcal{R}_\mathcal{F}(\bigcup_j \mathcal{S}_j^\infty)$ and hence

$$(\bigcup_j \mathcal{S}_j^\infty) \setminus \mathcal{R}_\mathcal{F}(\bigcup_j \mathcal{S}_j^\infty) \subseteq \mathcal{S}_\infty. \tag{3.1}$$

By (R1) and (R2), we get $\mathcal{R}(\bigcup_j \mathcal{S}_j^\infty) \subseteq \mathcal{R}(\mathcal{S}_\infty)$ (see [BG01] for details). Furthermore,

$$\begin{aligned}
\mathcal{S}_\infty &\models (\bigcup_j \mathcal{S}_j^\infty) \setminus \mathcal{R}_\mathcal{F}(\bigcup_j \mathcal{S}_j^\infty) && \text{by 3.1} \\
&\models \bigcup_j \mathcal{S}_j^\infty && \text{by definition of redundancy} \\
&\models \mathcal{S}_0^\infty,
\end{aligned}$$

and $\mathcal{S}_0^\infty \not\models N_0$, hence $\mathcal{S}_\infty \models N_0$. □

Definition 3.32 (Fairness)
We call a derivation $\mathcal{S}_0 \Longrightarrow_{\mathtt{SC}} \mathcal{S}_1 \Longrightarrow_{\mathtt{SC}} \ldots$ *fair* if the conclusion of every non-redundancy inference from non-redundant clauses in $\mathcal{S}_\infty$ is contained in $\bigcup_j \mathcal{S}_j^\infty$, or redundant with respect to it, i.e.,

$$\mathrm{concl}(\mathrm{Inf}(N') \setminus \mathcal{R}_\mathcal{I}(N')) \subseteq \bigcup_j \mathcal{S}_j^\infty \cup \mathcal{R}_\mathcal{F}\left(\mathcal{S}_j^\infty\right)$$

where $N' = \mathcal{S}_\infty \setminus \mathcal{R}_\mathcal{F}(\mathcal{S}_\infty)$. ■

The following lemma is equivalent to Theorem 4.3 from [BG01] and proved in the same way, using Lemma 3.31 in place of Lemma 4.2 [BG01].

Lemma 3.33
Let $\mathcal{S}_0 \Longrightarrow_{\mathtt{SC}} \mathcal{S}_1 \Longrightarrow_{\mathtt{SC}} \ldots$ be a fair derivation. Then $\mathcal{S}_\infty$ is saturated up to redundancy .

Theorem 3.34 (Completeness)
Assume the underlying calculus $\mathtt{C}$ is refutationally complete, and let $\mathcal{S}_0 \Longrightarrow_{\mathtt{SC}} \mathcal{S}_1 \Longrightarrow_{\mathtt{SC}} \ldots$ be a fair derivation, such that, whenever there is $\alpha : \square \in N_i$ with $\alpha \neq \emptyset$, then there exists $j \geq i$ such that Backtracking is applied to $\mathcal{S}_j$ and $\alpha : \square$. Then $\emptyset : \square \in \mathcal{S}_\infty$ whenever N_0 is unsatisfiable.

Proof. If the derivation is fair, then $\mathcal{S}_\infty$ is saturated up to redundancy (by Lemma 3.33), and $\mathcal{S}_\infty \models N_0$ (by Lemma 3.31). Hence, if N_0 is unsatisfiable, then $\mathcal{S}_\infty$ is unsatisfiable as well, and by saturation and refutational completeness of $\mathtt{C}$, $\mathcal{S}_\infty$ contains the empty clause. Hence there are $i \geq 0$ and α such that $\alpha : \square \in \bigcup_{j \geq i} \mathcal{S}_j^\infty$. Therefore Backtracking was not applied to $\alpha : \square$, so it must be the case that $\alpha = \emptyset$. □

In Section 3.4, we will discuss how to obtain fair derivations in practice.

3.2.2 Consistency-Preserving Rules

In the context of refutational theorem proving, one sometimes considers a weaker notion of soundness, namely *consistency preservation*. Whenever N' is obtained by applying to N a consistency-preserving inference or reduction, N and N' are equisatisfiable, but not necessarily logically equivalent. It is straightforward to adapt the results of this section to consistency-preserving rules:

- Invariant 3.21 is weakened to

$$[\mathcal{S}_{\gamma<k}] \Leftrightarrow [\mathcal{S}_\gamma] \vee [\mathcal{S}_{\gamma<k}, C_{-L_k}];$$

- Theorem 3.26 is weakened to stating that N_0 and open($\mathcal{S}$) are equisatisfiable (instead of equivalent), and is proven analogously;

- Lemma 3.31 is weakened to stating that N_0 is satisfiable if $\mathcal{S}_\infty$ is, i.e., one loses the property that saturated branches yield models of the input clause set.

Theorem 3.34 is then proven analogously, using the weakened version of Lemma 3.31.

3.3 Splitting With Clause Learning

One of the major breakthroughs in DPLL-based SAT-solving was the introduction of conflict-driven clause learning (CDCL) [SS96]. In CDCL, whenever a conflict occurs, it is analyzed and a *conflict clause* is derived and added to the clause set. This conflict clause can be thought of as summarizing the reason for the conflict. Adding the clause to the clause set has two benefits: First, the conflict clause is used to guide the backtracking process of the solver. The solver undoes all decisions up to the second-highest decision level of the conflict clause, after which the presence of the conflict clause causes a unit propagation step that drives the model search away from the conflict. For this reason a conflict clause used in this fashion is often called an *asserting clause*. The second benefit of learning the conflict clause is that its presence in the clause sets prevents the solver from running into the same or a similar conflict again in the future.

In our setting, we don't use conflict clauses for backtracking. The intuitive reason is that backtracking based on conflict clauses relies on the possibility of negating literals. Splitting generalizes the decision process of DPLL/CDCL in two dimensions: First, instead of decision *literals*, we have split *clauses*, which in general contain more than one literal. But more importantly, as we are in the first-order setting, split clauses may contain variables. While negating a propositional literal is no problem, negating a non-ground (and universally quantified) clause requires Skolemization. For instance, assume that we have a split clause $A(x)$. The usefulness of splitting is based in large parts on the possibility to use split clauses to reduce other clauses. If, during backtracking, we negate this clause, we get $\neg A(c)$, where c must be a fresh Skolem constant. As c does not occur in any other clauses, the clause $\neg A(c)$ cannot be used for reductions.

Nevertheless, adding the negation of the "left" split clause when backtracking into the corresponding "right" branch is a powerful technique. In the case where the split clause is ground, negating it and adding the resulting unit clauses can be done without problems, and this has been implemented in SPASS for a long time, significantly improving performance. Moreover, the addition of ground lemmas allows the splitting rule to simulate DPLL on propositional problems.

Even though we cannot negate non-ground split clauses without Skolemization, we can use the terms instantiating the split clauses' variables in the refutation of the left branch instead of Skolem constants. For example, assume again a split clause $A(x)$, and suppose that the refutation of the branch involved substituting x by both a and $f(b)$ in different subtrees (of the refutation). Then $\neg A(a) \vee \neg A(f(b))$ could be used instead of $\neg A(c)$ for the fresh Skolem constant c. More formally, assume the refutation is based on the split clause $A(x)$ and some other clauses $C_1, \ldots, C_m$, that is, there is a derivation $A(x), C_1, \ldots, C_m \vdash \square$. From this it follows that $C_1, \ldots, C_m \models \neg\forall x A(x)$. But if a and $f(b)$ were the only instances of x used in the refutation, we actually have $A(a), A(f(b)), C_1, \ldots, C_m \vdash \square$ from which it follows that $C_1, \ldots, C_m \models \neg A(a) \vee \neg A(f(b))$. More interestingly, variables occurring in such instantiations and

that never get instantiated by ground terms, can be viewed as "don't care" or existentially quantified. For instance, suppose that x is only instantiated by $f(y)$. Then we have $\exists y A(f(y)), C_1, \ldots, C_m \vdash \square$ from which it follows that $C_1, \ldots, C_m \models \forall y\,(\neg A(f(y)))$. This opens up the possibility of learning non-ground clauses, which would not be possible with a Skolemization approach.

It is clear that in the context of superposition with splitting, no form of conflict-driven clause learning should be expected to deliver the same boost in performance as it does in the SAT setting [BKS04]. The reason is simply that in a SAT solver, clause learning is the only way to generate new clauses, whereas clauses are generated all the time in a saturation-based prover. In a sense, superposition already performs clause learning, albeit not necessarily a "conflict-driven" one. Nevertheless, the addition of negated ground split clauses during backtracking has proven to be very useful for superposition-based proof search, and it is a natural question to ask whether the addition of more powerful conflict-driven clause learning may also benefit superposition with splitting. In particular, it could be hoped that clauses learned from conflicts would prevent the prover from repeating past mistakes, as they do in the SAT setting.

In order to achieve clause learning in the first-order case, our first step will be to extend the calculus such that instantiations of split clauses can be tracked in derivations. For this we will generalize the clause labels of the previous section, so that they carry instantiation information.

Definition 3.35 (Simple and Complex Labels)
We call *complex labels* sets of tuples consisting of a natural number l and $m \geq 0$ terms $t_1, \ldots, t_m$. We write such an element as $l(t_1, \ldots, t_m)$ and treat it as a term with function symbol l. For clarity we refer to labels as used until now (i.e., subsets of $\mathbb{N}$) as *simple labels*. ∎

The splitting calculus is extended to handle complex labels. We call the resulting calculus `SCL`. Its transition rules are shown in Figure 3.3.

As usual, we assume that the premises of an inference or a reduction are pairwise variable-disjoint, and this also applies to variables which occur only in the labels. More concretely, we can assume that prior to computing a unifier, a renaming of all variables in the clause and in the label has been applied to each premise.

Definition 3.36
Let α, β be a complex and a simple label, respectively. We define

$$\begin{aligned} \langle \alpha \rangle &= \{l \mid l(\vec{t}) \in \alpha \text{ for some } \vec{t}\}, \\ \alpha^{\cap\beta} &= \{l(\vec{t}) \in \alpha \mid l \in \beta\}, \text{ and} \\ \alpha^{\setminus\beta} &= \{l(\vec{t}) \in \alpha \mid l \notin \beta\}. \end{aligned}$$ ∎

For example, if $\alpha = \{1(x), 2(f(x))\}$ and $\beta = \{1\}$, then $\langle \alpha \rangle = \{1, 2\}$, $\alpha^{\cap\beta} = \{1(x)\}$ and $\alpha^{\setminus\beta} = \{2(f(x))\}$.

Splitting:

$$S \mid M \mid N \quad \Longrightarrow_{\mathtt{SCL}} \quad S,\, (n, \alpha : C, L) \mid M \mid N,\; \alpha \cup \{n(\vec{x})\} : C_{+L}$$

where

(i) there is some clause $\alpha : C \in N$ such that L splits C, and

(ii) $n = (\max^+ \text{levels}(S)) + 1$, and

(iii) $\vec{x} = \text{var}(C_{+L})$.

Inference:

$$S \mid M \mid N \quad \Longrightarrow_{\mathtt{SCL}} \quad S \mid M \mid N,\, \alpha\sigma : C$$

where

(i) there are clauses $\alpha_1 : C_1, \ldots, \alpha_n : C_n \in N$, and

(ii) $\mathcal{I}\dfrac{C_1 \ldots C_n}{C}$ is an inference of $\mathtt{C}$ with unifier σ, and

(iii) $\alpha = \alpha_1 \cup \ldots \cup \alpha_n$.

Reduction:

$$S \mid M \mid N,\, \alpha_1 : C_1, \ldots, \alpha_n : C_n \quad \Longrightarrow_{\mathtt{SCL}} \quad S \mid M \cup M' \mid N,\, \beta_1 : D_1, \ldots, \beta_m : D_m$$

where

(i) $\mathcal{R}\dfrac{C_1 \ldots C_n}{D_1 \ldots D_m}$ is a reduction of $\mathtt{C}$ with unifier σ, and

(ii) $\gamma = \alpha_1 \cup \ldots \cup \alpha_n$,

(iii) $\beta_i = \begin{cases} \alpha_j & \text{if } D_i = C_j \text{ for some } j, \\ \gamma\sigma & \text{otherwise,} \end{cases}$

(iv) $M' = \{(\langle\gamma\rangle, \alpha_i) : C_i \mid C_i \notin \{D_1, \ldots, D_m\} \text{ and } \langle\gamma\rangle \not\subseteq \langle\alpha_i\rangle\}$.

Backtracking:

$$S \mid M \mid N,\, \alpha : \square \quad \Longrightarrow_{\mathtt{SCL}} \quad S' \mid M \setminus M' \mid (N \setminus N') \cup N'',\, \alpha' : C_{-L_{\max\alpha}}$$

if $\alpha \neq \emptyset$, where

(i) $S' = \text{bt}(S, \alpha)$, for a backtracking function bt,

(ii) $\alpha' = \alpha \setminus \{\max \alpha\}$,

(iii) $M' = \{(\gamma, \beta) : D \in M \mid \gamma \not\subseteq \text{levels}(S') \text{ or } \langle\beta\rangle \not\subseteq \text{levels}(S')\}$,

(iv) $N' = \{\alpha : C \in N \mid \langle\alpha\rangle \not\subseteq \text{levels}(S')\}$,

(v) $N'' = \{\beta : D \mid (\gamma, \beta) : D \in M \text{ and } \gamma \not\subseteq \text{levels}(S') \text{ and } \langle\beta\rangle \subseteq \text{levels}(S')\}$.

Figure 3.3: Transition Rules of a Splitting Calculus With Complex Labels

Definition 3.37
Given a stack S and respective complex and simple labels α, β with $\langle\alpha\rangle, \beta \subseteq \text{levels}(S)$, we define the operators

$$\begin{aligned} \text{sc}^+(S,\alpha) &= \{\, (C_{+L_k})[\vec{t}] \mid k(\vec{t}) \in \alpha\}, \\ \text{sc}^+(S,\beta) &= \{\, C_{+L_k} \mid k \in \beta\}, \\ \text{sc}^-(S,\beta) &= \{\, C_{-L_k} \mid k \in \beta\}. \end{aligned}$$ ■

The set $\text{sc}^+(S,\alpha)$ contains the instances of left split clauses of S represented by α, while $\text{sc}^-(S,\beta)$ contains the right split clauses of S represented by β. Note that the clauses in $\text{sc}^+(S,\alpha)$ may share variables, because the terms in α may share variables. On the other hand, we assume the clauses in $\text{sc}^+(S,\beta)$ and $\text{sc}^-(S,\beta)$ to be pairwise variable-disjoint.

Example 3.38
Consider the stack $S = (1,\ (\alpha_1 : P(x,y) \to Q(z)),\ \{0\}),\ (2,\ (\alpha_2 : \to R(u), S(v)),\ \{0\})$. We have[8]

$$\begin{aligned} \text{sc}^+(S,\, \{1(a,b),\, 1(x,f(y)),\, 2(y)\}) &= \{\, \{\neg P(a,b)\},\, \{\neg P(x,f(y))\},\, \{Q(y)\}\,\} \\ \text{sc}^+(S,\, \{1,2\}) &= \{\, \{\neg P(x,y)\},\, \{R(u)\}\,\} \\ \text{sc}^-(S,\, \{1,2\}) &= \{\, \{Q(z)\},\, \{S(v)\}\,\}\ . \end{aligned}$$ ■

In the following, in order to simplify notation, we treat the (implicitly universally quantified) variables in clauses like free variables. For instance, if C is the clause $P(x,y) \to Q(x)$, then $\neg C$ is the (implicitly universally quantified) formula $\neg\neg P(x,y) \wedge \neg Q(x)$, or, equivalently, $P(x,y) \wedge \neg Q(x)$. Furthermore, we extend the clause notation $\Gamma \to \Delta$ (see Definition 2.14) to include the case where Γ and Δ are sets of (not necessarily variable-disjoint) clauses, with the usual semantics: $\Gamma \to \Delta$ stands for $\forall\vec{x}.\neg(\bigwedge\Gamma) \vee \bigvee\Delta$, where $\vec{x}$ are all variables occurring in Γ, Δ.

Definition 3.39 (Clause Learning Function)
Let S be a stack, and α be a complex label. A (conflict-driven) *clause learning function* is any function defined by

$$(S,\alpha) \mapsto \gamma : \text{cnf}(\text{sc}^+(S,\, \alpha \setminus \gamma) \to).$$

where $\gamma \subseteq \alpha$ such that $\alpha \setminus \gamma = \alpha^{\setminus\langle\gamma\rangle}$. ■

Different styles of clause learning can be modelled using the above definition. With $\gamma = \emptyset$, we get a global learning function. With $\gamma = \alpha^{\setminus\{\max\langle\alpha\rangle\}}$ and the additional requirement of $\text{sc}^+(S, \alpha \setminus \gamma)$ being ground, we get local ground lemma learning, as described at the beginning of this section. We will consider more examples of concrete clause learning functions in Section 3.6.

Learning:

$$S \mid M \mid N \implies_{\mathtt{SCL}} S \mid M \mid N \cup \mathrm{learn}(S, \alpha)$$

if there is $\alpha : \square \in N$, and learn is a clause learning function.

Figure 3.4: Clause Learning Rule

Clause learning is integrated into SCL via the rule Learn, shown in Figure 3.4.

It is straightforward to adapt Theorem 3.26 and all lemmas it depends on to the rules shown in Figure 3.3, simply by replacing α by $\langle\alpha\rangle$ in the appropriate places. However, to obtain soundness of SCL, it still remains to show that the clause learning rule is actually a sound labelled inference rule. For this, we first establish the following simple lemma:

Lemma 3.40
Assume $\emptyset \mid \emptyset \mid N_0 \implies^*_{\mathtt{SCL}} \mathcal{S}$ with $\mathcal{S} = S \mid M \mid N$. Then $\mathcal{S}_\delta \models \mathrm{sc}^+(S, \delta)$ holds for any $\delta \subseteq \mathrm{levels}(S)$.

Proof. By induction on the derivation. Use Lemma 3.25 for Backtracking. □

As we will show in Lemma 3.42, the above Lemma 3.40 suffices to prove soundness of Learn, if additionally, the invariant

$$\mathcal{S}_\emptyset \models \mathrm{sc}^+(S, \alpha) \to C$$

can be shown to hold for every clause $\alpha : C$ in any state $\mathcal{S}$ reachable by SCL. Because of the Backtracking rule, the above invariant is not inductive with respect to $\Longrightarrow_{\mathtt{SCL}}$, and we therefore prove a stronger property in the following lemma:

Lemma 3.41
Assume $\emptyset \mid \emptyset \mid N_0 \implies^*_{\mathtt{SCL}} \mathcal{S} = (S \mid M \mid N)$. Let $\alpha : C \in N$, and let $\beta \subseteq \langle\alpha\rangle$. Then

$$\mathcal{S}_\emptyset \;\models\; \mathrm{sc}^+(S, \alpha^{\setminus\beta}) \;\to\; C, \mathrm{sc}^-(S, \beta).$$

Proof. By induction on the derivation. For the initial state, the statement reduces to

$$\mathcal{S}_\emptyset \;\models\; \top \;\to\; C$$

which holds as $C \in \mathcal{S}_\emptyset$. So let $\mathcal{S} = (S \mid M \mid N)$, $\mathcal{S}' = (S' \mid M' \mid N')$ and assume $\emptyset \mid \emptyset \mid N_0 \implies^*_{\mathtt{SCL}} \mathcal{S} \Longrightarrow_{\mathtt{SCL}} \mathcal{S}'$. We distinguish which rule was applied:

- Inference: Assume the clause $\alpha\sigma : C \in N'$ was derived from premises $\alpha_1 : C_1, \ldots, \alpha_n : C_n \in N$ by an inference with unifier σ, and let $\beta \subseteq \langle\alpha\rangle$. Let $i \in [1, n]$, and $\beta_i = \beta \cap \alpha_i$. By induction, we have

$$\mathcal{S}_\emptyset \;\models\; \mathrm{sc}^+(S, \alpha_i^{\setminus\beta_i}) \to C_i, \mathrm{sc}^-(S, \beta_i)$$

[8] We write the clauses as multisets of literals to improve readability.

and, since $\mathrm{sc}^-(S, \beta_i)$ shares no variables with $\mathrm{sc}^+(S, \alpha_i^{\setminus\beta_i})$ or C_i,

$$\mathcal{S}_\emptyset \models \mathrm{sc}^+(S, \alpha_i^{\setminus\beta_i})\sigma \to C_i\sigma, \mathrm{sc}^-(S, \beta_i).$$

It follows that

$$\mathcal{S}_\emptyset \models \bigcup_i \mathrm{sc}^+(S, \alpha_i^{\setminus\beta_i})\sigma \to (C_1\sigma \wedge \ldots \wedge C_n\sigma), \bigcup_j \mathrm{sc}^-(S, \beta_j)$$
$$\Leftrightarrow \mathcal{S}_\emptyset \models \mathrm{sc}^+(S, \alpha^{\setminus\beta})\sigma \to (C_1\sigma \wedge \ldots \wedge C_n\sigma), \mathrm{sc}^-(S, \beta).$$

By soundness of the underlying inference (or reduction), we obtain

$$\mathcal{S}_\emptyset \models \mathrm{sc}^+(S, \alpha^{\setminus\beta})\sigma \to C, \mathrm{sc}^-(S, \beta)$$

or, equivalently (by $S' = S$):

$$\mathcal{S}_\emptyset \models \mathrm{sc}^+(S', \alpha^{\setminus\beta})\sigma \to C, \mathrm{sc}^-(S', \beta).$$

Finally, as $\mathcal{S}_\emptyset \subseteq \mathcal{S}'_\emptyset$, we get

$$\mathcal{S}'_\emptyset \models \mathrm{sc}^+(S', \alpha^{\setminus\beta})\sigma \to C, \mathrm{sc}^-(S', \beta).$$

- Reduction: Analogous to Inference. In the last step, $\mathcal{S}_\emptyset \subseteq \mathcal{S}'_\emptyset$ does not necessarily hold, but it follows by Lemma 3.10 that $\mathcal{S}_\emptyset \models\!\models \mathcal{S}'_\emptyset$.
- Splitting: Let $\beta \subseteq \langle\alpha\rangle$. If $n \notin \beta$, then $n(\vec{x}) \in (\alpha \cup \{n(\vec{x})\})^{\setminus\beta}$ and $C_{+L} \in \mathrm{sc}^+(S, (\alpha \cup \{n(\vec{x})\})^{\setminus\beta})$, so $\mathrm{sc}^+(S, (\alpha \cup \{n(\vec{x})\})^{\setminus\beta}) \to C_{+L}$ is valid. If $n \in \beta$, then $C_{-L} \in \mathrm{sc}^-(S, \beta)$. Let $\beta' = \beta \setminus \{n\}$. By induction,

  $$\mathcal{S}'_\emptyset \models \mathrm{sc}^+(S, \alpha^{\setminus\beta'}) \to C_{+L}, C_{-L}, \mathrm{sc}^-(S, \beta')$$

  or, equivalently, as $\mathrm{levels}(S) \subseteq \mathrm{levels}(S')$:

  $$\mathcal{S}'_\emptyset \models \mathrm{sc}^+(S', \alpha^{\setminus\beta'}) \to C_{+L}, C_{-L}, \mathrm{sc}^-(S', \beta').$$

  As $\mathrm{sc}^-(S', \beta) = \mathrm{sc}^-(S', \beta') \cup \{C_{-L}\}$, it follows that

  $$\mathcal{S}'_\emptyset \models\!\models \mathrm{sc}^+(S', (\alpha \cup \{n(\vec{x})\})^{\setminus\beta}) \to C_{+L}, \mathrm{sc}^-(S', \beta).$$

- Backtracking: Let $m = \max\alpha$, and let $\beta \subseteq \alpha'$ (so $m \notin \beta$). By induction applied to $\alpha : \Box \in N$,

  $$\mathcal{S}'_\emptyset \models\!\models \mathrm{sc}^+(S, \alpha^{\setminus(\beta\cup\{m\})}) \to \mathrm{sc}^-(S, \beta \cup \{m\})$$

  or, equivalently, as $\alpha' = \alpha \setminus \{m\}$ and $\mathrm{sc}^-(S, \beta \cup \{m\}) = \mathrm{sc}^-(S, \beta) \cup \{C_{-L_m}\}$:

  $$\mathcal{S}'_\emptyset \models\!\models \mathrm{sc}^+(S, \alpha'^{\setminus\beta}) \to C_{-L_m}, \mathrm{sc}^-(S, \beta)$$

  which is equivalent to

  $$\mathcal{S}'_\emptyset \models\!\models \mathrm{sc}^+(S', \alpha'^{\setminus\beta}) \to C_{-L_m}, \mathrm{sc}^-(S', \beta).$$

- Learn: Let $\beta \subseteq \langle\gamma\rangle$. By induction applied to $\alpha{:}\square \in N$,

$$\mathcal{S}_\emptyset \models \mathrm{sc}^+(S, \alpha^{\setminus\beta}) \to \mathrm{sc}^-(S, \beta).$$

Because of $\beta \subseteq \langle\gamma\rangle$ and $\gamma \subseteq \alpha$, it follows that $(\alpha \setminus \gamma) \cup \gamma^{\setminus\beta} = \alpha^{\setminus\beta}$, and hence

$$\mathcal{S}_\emptyset \models \mathrm{sc}^+(S, \gamma^{\setminus\beta}) \cup \mathrm{sc}^+(S, \alpha \setminus \gamma) \to \mathrm{sc}^-(S, \beta)$$

or, equivalently,

$$\mathcal{S}_\emptyset \models \mathrm{sc}^+(S, \gamma^{\setminus\beta}) \to \neg\mathrm{sc}^+(S, \alpha \setminus \gamma), \mathrm{sc}^-(S, \beta)$$

Now assume that $\mathrm{cnf}(\neg\mathrm{sc}^+(S, \alpha \setminus \gamma)) = \{C_1, \ldots, C_m\}$. Then

$$\mathcal{S}_\emptyset \models \mathrm{sc}^+(S, \gamma^{\setminus\beta}) \to (C_1 \wedge \ldots \wedge C_m), \mathrm{sc}^-(S, \beta)$$

which can be weakened to

$$\mathcal{S}_\emptyset \models \mathrm{sc}^+(S, \gamma^{\setminus\beta}) \to C_i, \mathrm{sc}^-(S, \beta)$$

for any $i \in [1, m]$. Thus the the invariant holds for any learned clause $\gamma : C_i$ in $\gamma : \mathrm{cnf}(\mathrm{sc}^+(S, \alpha \setminus \gamma) \to)$.

□

Lemma 3.42 (Soundness of Learn)
Learn is a sound labelled inference rule, i.e., for any $\mathcal{S} = (S \mid M \mid N)$ with $\emptyset \mid \emptyset \mid N_0 \implies^*_{\mathtt{SCL}} \mathcal{S}$, and $\alpha : \square \in N$, it holds that $\mathcal{S}_\delta \models C_i$, for any $C_i \in \mathrm{cnf}(\mathrm{sc}^+(S, \alpha \setminus \gamma) \to)$ returned by a clause learning function, for any δ with $\langle\gamma\rangle \subseteq \delta$.

Proof. Assume $\mathcal{S}$, α, γ and δ as described above. By Lemma 3.41 applied to $\alpha : \square$,

$$\mathcal{S}_\emptyset \models \mathrm{sc}^+(S, \alpha) \to$$

which implies

$$\mathcal{S}_\delta \models \mathrm{sc}^+(S, \alpha) \to$$

as $\mathcal{S}_\emptyset \subseteq \mathcal{S}_\delta$. By Lemma 3.40,

$$\mathcal{S}_\delta \models \mathrm{sc}^+(S, \delta).$$

Hence,

$$\mathcal{S}_\delta \models \mathrm{sc}^+(S, \delta) \wedge (\mathrm{sc}^+(S, \alpha) \to).$$

As $\langle\gamma\rangle \subseteq (\langle\alpha\rangle \cap \delta)$, this implies

$$\mathcal{S}_\delta \models \mathrm{sc}^+(S, \delta) \wedge (\mathrm{sc}^+(S, \alpha^{\setminus\langle\gamma\rangle}) \rightarrow)$$

which, by $\alpha^{\setminus\langle\gamma\rangle} = \alpha \setminus \gamma$, implies

$$\mathcal{S}_\delta \models \mathrm{sc}^+(S, \alpha \setminus \gamma) \rightarrow$$

Now let $\{C_1, \ldots, C_m\} = \mathrm{cnf}(\mathrm{sc}^+(S, \alpha \setminus \gamma) \rightarrow)$. Then

$$\mathcal{S}_\delta \models C_1 \wedge \ldots \wedge C_m$$

and thus

$$\mathcal{S}_\delta \models C_i$$

for any $i \in [1, m]$. □

Theorem 3.43 (Soundness of $\Longrightarrow_{\mathtt{SCL}}$)
Assume $\emptyset \mid \emptyset \mid N_0 \Longrightarrow^*_{\mathtt{SCL}} \mathcal{S}$. Then $N_0 \models\!\models \mathrm{open}(\mathcal{S})$.

Proof. The proof is analogous to the proof of Theorem 3.26, with α replaced by $\langle\alpha\rangle$ in the appropriate places in the proof, and in all required lemmas. The additional rule Learn is sound by Lemma 3.42. □

Example 3.44
Consider a stack S with $\mathrm{levels}(S) = \{1, 2, 3\}$ and $\mathrm{sc}^+(S, \{1\}) = \{P(x)\}$, $\mathrm{sc}^+(S, \{2\}) = \{Q(y)\}$ and $\mathrm{sc}^+(S, \{3\}) = \{R(z)\}$, and let $\alpha = \{1(x), 2(a), 2(b), 3(f(x))\}$. Then, for $\gamma = \alpha^{\setminus\{\max\langle\alpha\rangle\}} = \{1(x), 2(a), 2(b)\}$ we get

$$\begin{aligned} \gamma : \mathrm{cnf}(\mathrm{sc}^+(S,\, \alpha \setminus \gamma) \rightarrow) &= \gamma : \mathrm{cnf}(\mathrm{sc}^+(S,\, \{3(f(x))\}) \rightarrow) \\ &= \{\, \{1(x),\, 2(a),\, 2(b)\} : R(f(x)) \rightarrow \,\} \end{aligned}$$

which corresponds to a "local" learning scheme. On the other hand, for $\gamma = \emptyset$, we get

$$\begin{aligned} \gamma : \mathrm{cnf}(\mathrm{sc}^+(S,\, \alpha \setminus \gamma) \rightarrow) &= \emptyset : \mathrm{cnf}(\mathrm{sc}^+(S,\, \alpha) \rightarrow) \\ &= \{\, \emptyset : P(x), Q(a), Q(b), R(f(x)) \rightarrow \,\} \end{aligned}$$

which corresponds to a "global" learning scheme. ■

3.4 Implementation

Algorithm 3.1 shows the main loop of SPASS with splitting based on the calculus SC. For easier comparison with Algorithm 2.1, states (as they appear in the calculus SC) do not appear explicitly in the pseudocode: The sets Usable and WorkedOff

together make up the clause set N in the corresponding state $S \mid M \mid N$, while the set M of conditionally deleted clauses is modified implicitly by the procedures FRed, BRed and Backtrack. Clause learning is implemented inside the procedure Backtrack.

The loop is quite similar to the loop without splitting (Algorithm 2.1), with the following exceptions:

- The occurrence of an empty clause $\alpha : \square \in$ Usable causes the loop to terminate only if the stack is empty (line 4), in which case α must be empty as well;
- If Usable contains an empty clause $\alpha : \square$ with $\alpha \neq \emptyset$, backtracking is performed (line 6);
- The Given clause is split if certain conditions are satisfied (lines 11 and 12, where n is $|S| + 1$ and L is the chosen set of literals to split on), otherwise it is used for inferences as usual (lines 14 and 15).

The function IsSplittable has three purposes: (i) To check whether the clause is splittable at all, i.e., whether it has non-empty variable-disjoint components, (ii) to implement a splitting heuristic, and (iii) to ensure fairness in the sense of Definition 3.32.

A splitting heuristic could consist in splitting only if the resulting split clauses both contain at least one positive literal, in order to get "closer" to Horn [FW09].

In order to ensure fairness, different approaches can be used. In SPASS, the "reductive potential" of a split clause, i.e., the number of clauses in the Usable and WorkedOff sets that would be subsumed by the split clause, is computed. Consecutive splitting steps are only allowed if their corresponding reductive potential is monotonically increasing. Thus in SPASS, IsSplittable(Given) returns true if and only if Given is splittable into variable-disjoint components having both at least one positive literal, and the reductive potential of consecutive splits is monotonically increasing.

A less sophisticated approach simply consists in performing at least one "fair" inference step between any two splitting steps, for instance by maintaining the Usable clauses in a priority queue according to some fair measure, like clause weight, size or derivation depth.

3.4.1 Representation of Complex Labels

We represent complex labels as additional special literals in clauses (in a way similar to answer literals [Gre68]). The predicate symbols for these literals don't occur in the original problem signature, and the calculus rules are modified so that they completely ignore these extra literals, except when applying substitutions to a clause. This has the advantage that the existing data structures and operations for terms and substitutions can be directly used on the labels. For efficiency, clauses with a complex label α still keep their simple label $\langle\alpha\rangle$ in bit vector representation, even though this information

Algorithm 3.1: Main loop of SPASS with splitting

Input: Clause set N

```
1  WorkedOff := ∅;
2  Usable := N;
3  S := ∅;
4  while Usable ≠ ∅ and (α : □ ∉ Usable or S ≠ ∅) do
5      if α : □ ∈ Usable with α ≠ ∅ then
6          (S, Usable, WorkedOff) := Backtrack(S, Usable, WorkedOff);
7      else
8          Given := Choose(Usable);
9          Usable := Usable \ {Given};
10         if IsSplittable(Given) then
11             Derived := {α ∪ {n} : C_{+L}};
12             S := S, (n, Given, L);
13         else
14             Derived := Inf(Given, WorkedOff);
15             WorkedOff := WorkedOff ∪ {Given};
16         Derived := FRed(Derived, Usable, WorkedOff);
17         (Usable, WorkedOff) := BRed(Usable, WorkedOff, Derived);
18         Usable := Usable ∪ Derived;
19 if Usable = ∅ then
20     print "Completion found";
21 else if ∅ : □ ∈ Usable then
22     print "Proof found";
```

is redundant. Keeping it allows to efficiently handle conditional clause deletion, as explained below.

3.4.2 Handling of Conditionally Deleted Clauses

We store conditionally deleted clauses in an array deleted, where deleted$[k]$ contains the list of all conditionally deleted clauses $(\gamma, \alpha) : C$ with $k = \max\{\max^+ \gamma,\ \max^+ \alpha\}$. The rationale for this indexing scheme is that the validity and/or redundancy of a conditionally deleted clause in deleted$[k]$ can only be affected by the removal of a split with level less or equal to k. Therefore, after a Backtracking step, only the indices greater or equal to the minimum of the removed levels need to be checked for clauses that may have become invalid, or that have to be reinserted. Algorithm 3.2 takes as input the set $\beta = \text{levels}(S) \setminus \text{levels}(S')$ of levels which have been removed, and the level n of the topmost split of S, and returns the set N'' of clauses that have to be reinserted, deleting them and all invalid clauses from deleted. Labels are represented as bit vectors, so that checking for an empty intersection between two labels is done by computing the bitwise AND and checking if the result is zero.

Algorithm 3.2: ReinsertDeletedClauses

Input: $\beta = \text{levels}(S) \setminus \text{levels}(S')$, $n = \max \text{levels}(S)$
Output: Set N'' of clauses to reinsert

```
1 N'' := ∅;
2 for l := min β to n do
3     foreach (γ, α) : C ∈ deleted[l] do
4         if β ∩ α ≠ ∅ then                    /* α : C is no longer valid */
5             remove (γ, α) : C from deleted[l];
6         else if β ∩ γ ≠ ∅ then               /* α : C is no longer redundant */
7             remove (γ, α) : C from deleted[l];
8             N'' := N'' ∪ {α : C};
9 return N''
```

3.5 Comparison With Previous Work

The splitting calculus SC presented in Section 3.2 is an improved version of the calculus presented in [FW08] and [FW09], which we refer to as the "old" splitting procedure in the following. The main improvements are the following:

- The calculus SC is much simpler than its predecessor: There is no more distinction between "left" and "right" splits, and no more need for "leaf markers". Backtracking is defined as a single rule in SC, instead of four (Backjump, Branch-condense, Right-collapse, Enter-right). Hence the correctness proofs are also simpler and easier to understand.

- The calculus SC can model different backtracking styles, via the bactracking function, and clause learning can be straightforwardly integrated into SC, as we did in Section 3.3. Adding clause learning to the earlier calculus and proving correctness of the resulting system would have been much more tedious. This was one of the motivations for developing the improved calculus.

- The handling of conditionally deleted clauses in SC is more precise and efficient than in the previous calculus: On the theoretical side, conditionally deleted clauses are reinserted only when the level of the subsuming clause is deleted—in the previous calculus, conditionally deleted clauses were stored on the stack, at the level of the reducing clause, which required a complicated traversal of the whole stack whenever a split was to be deleted, and also had the consequence that clauses were being reinserted even though they were still redundant. On the practical side, the simplified handling of conditionally deleted clauses makes backtracking in the SC-based implementation twice as fast as in the earlier implementation, on average.

- Finally, in the previous calculus ground lemmas unnecessarily depend on the current right split. In SC, this dependency no longer exists and the lemmas thus have a greater scope.

Figure 3.5 shows a comparison between the old implementation of splitting with backtracking and the new implementation based on SC, on a random sample of 1946 problems from the TPTP problem library, version 5.4.0 [Sut09], where splitting was used. In both procedures, regular backtracking (see Definition 3.19) and the default ground lemma generation (learning scheme 1 in Section 3.6) are used. The plot shows the number of problems solved (vertical axis) within a given time limit of up to five minutes (horizontal axis). It can be seen that the new splitting procedure performs better, solving about 20 more problems within the five minute time limit.

Figure 3.6 compares the old and the new splitting procedures in terms of the number of derived clauses per problem (upper plot) and the number of splits performed per problem (lower plot). It can be seen that the number of derived clauses and splits tends to be lower under the new backtracking scheme, especially for problems with large search spaces.

The reason why the performance degrades for some problems is that first-order theorem provers—just like SAT solvers—can be very sensitive to the order in which clauses are processed. In the new splitting procedure, conditionally deleted clauses may be reinserted earlier than in the old procedure, and such a reinserted clause may then be selected

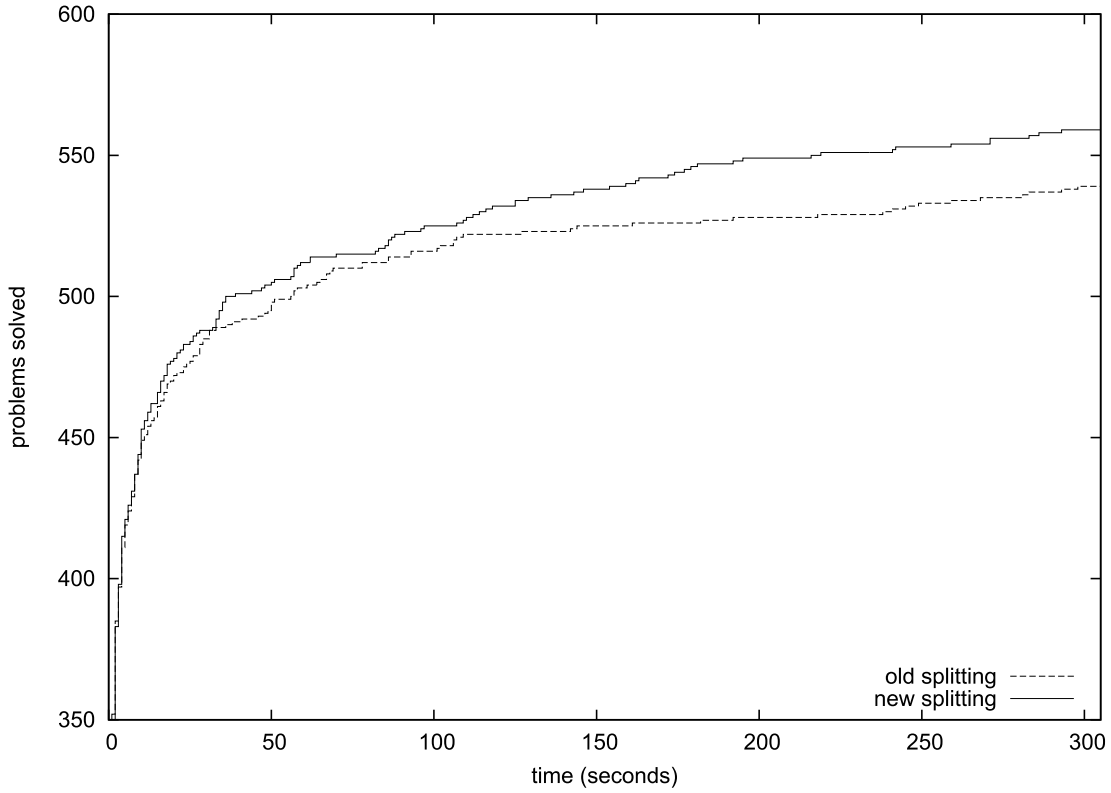


Figure 3.5: Comparison of old and new splitting: Number of problems solved

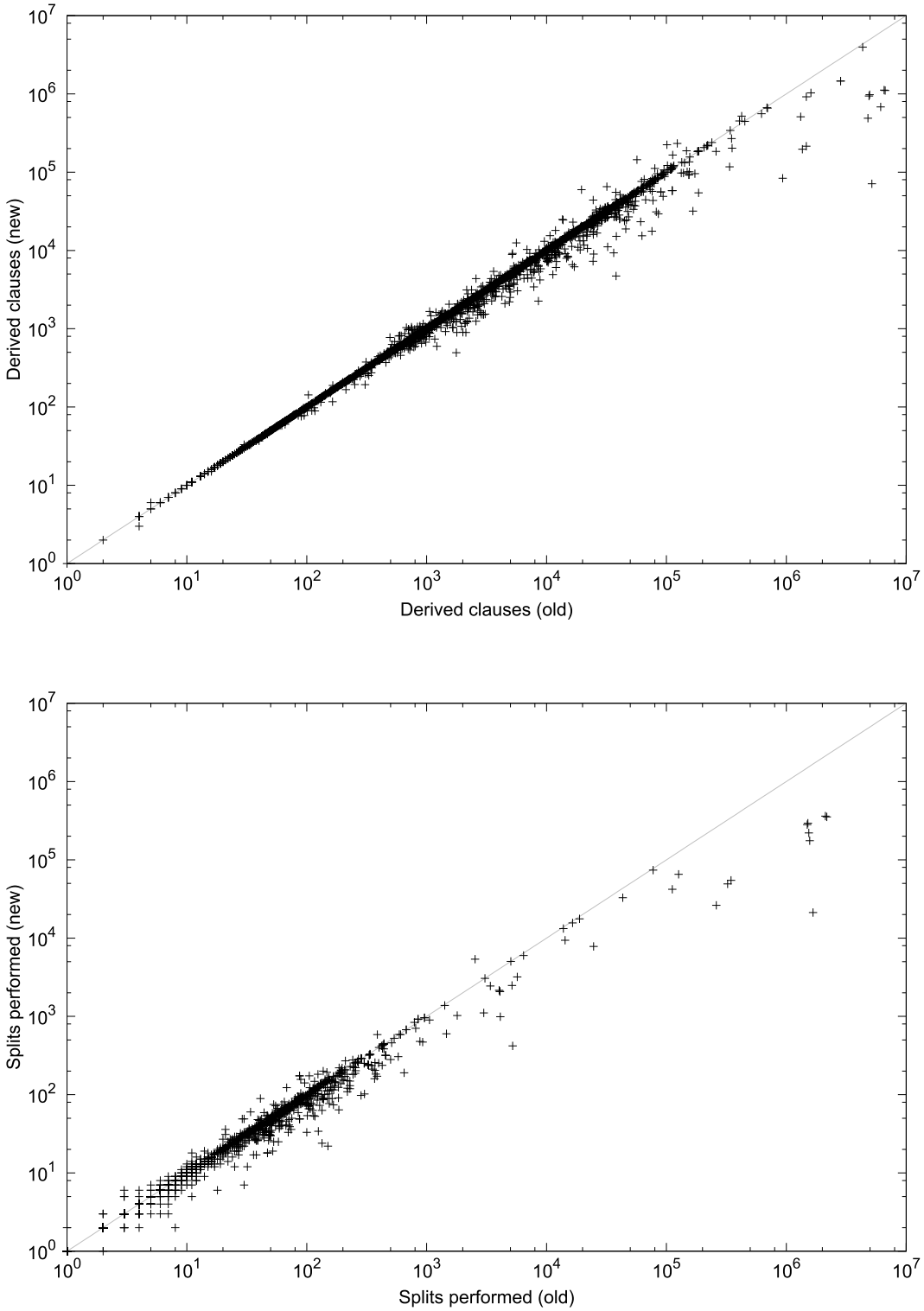


Figure 3.6: Comparison of old and new splitting: Derived clauses and splits performed per problem

for inferences leading to a completely different part of the search space being explored, and causing divergence of the two prover runs from that point on.

3.6 Experimental Evaluation of Clause Learning

We have implemented five different clause learning functions, shown in Table 3.1.

Scheme	Lemmas based on
1	the last used split clause if it is ground: $\gamma = \alpha^{\setminus\{m\}}$, if C_{+L_m} is ground
2	the single used ground instance of the last used split clause, if there is such a single ground instance: $\gamma = \alpha^{\setminus\{m\}}$, if $\mathrm{sc}^+(S, \alpha \setminus \gamma) = \{C\}$ and C is ground;
3	the single used (possibly non-ground) instance of the last used split clause, if there is such a single instance: $\gamma = \alpha^{\setminus\{m\}}$, if $\mathrm{sc}^+(S, \alpha \setminus \gamma) = \{C\}$;
4	all used instances of the last used split clause: $\gamma = \alpha^{\setminus\{m\}}$;
5	all used split clause instances: $\gamma = \emptyset$.

Table 3.1: Clause learning schemes, with $m = \max\langle\alpha\rangle$.

The learning functions 1 to 4 all produce *local* lemmas, usually having non-empty labels and hence being valid only in a certain part of the split tree, in particular in the right branch entered upon backtracking. The learning functions 1 to 4 become increasingly more powerful, as any lemma produced by function 1 is also produced by functions 2, 3, and 4, and so on. Function 5 produces *global* lemmas only, i.e., clauses with empty labels that are globally valid. The motivation for global lemma learning is that global lemmas can be reused across the split tree and are not discarded upon backtracking.

Table 3.2 shows the effect of the clause learning schemes 2 to 5 on the number of derived clauses and splits per problem, measured as the average over all problems of the ratio between the number of derived clauses (or splits performed) under the respective learning scheme and the number of derived clauses (or splits performed) under scheme 1 (the default scheme), expressed in percent (ignore the last row for now). The numbers are separately aggregated for problems yielding a proof and for problems yielding a completion. For instance, the first number in the top left corner means that on average over all problems where both splitting without learning and learning scheme 2 yield a proof, 1.5% fewer clauses are derived per problem under learning scheme 2 than under learning scheme 1.

	Clauses derived			Splits performed		
Scheme	Proofs	Completions	Overall	Proofs	Completions	Overall
2	-1.5	-0.2	-1.2	-1.6	-0.2	-1.3
3	-1.9	0.0	-1.5	-2.0	-0.4	-1.6
4	-3.7	-0.5	-2.9	-5.1	-0.7	-4.1
5	+11.1	+1.6	+9.0	+23.2	+0.6	+18.1
6	-4.7	-0.5	-3.8	-6.5	-0.7	-5.2

Table 3.2: Effect of the different clause learning schemes on the number of derived clauses and splits per problem, compared to the default scheme 1.

As expected, scheme 4 gives the best result among the local learning schemes. Surprisingly, the global learning scheme 5 performs even worse than the default local learning scheme 1. The reason turned out to be the following: It often happens that the instances of split clauses used in the derivation of the empty clause together already form an unsatisfiable conjunction, i.e., the clause set $\mathrm{sc}^+(S, \alpha)$ is unsatisfiable. In that case, the lemma $\mathrm{cnf}(\neg\mathrm{sc}^+(S, \alpha))$ is a tautology, hence nothing is learned at all. However, in the same situation, local lemmas could be learned, which in turn could be used to speed up the refutation of the right branch.[9] While this absence of local lemmas causes worse performance for many problems, the learned global lemmas significantly improve performance for other problems. We therefore decided to combine schemes 4 and 5 into an new learning scheme (number 6), producing both local and global lemmas.

Figure 3.7 compares the number of problems solved within a given time limit of up to five minutes under the six different learning schemes, on the same random sample of 1797 TPTP problems used for Figure 3.5. It can be seen that learning scheme 6 indeed performs best.

Figure 3.8 compares the number of derived clauses and splits performed per problem under learning schemes 4, 5 and 6 compared to 1. It can be seen that scheme 6 combines the improvements of schemes 4 and 5 without the degradation incurred by scheme 5.

All experiments in this and the preceding section were run on machines with a 3.16 GHz Xeon CPU and 16 GB RAM, running Debian 6.0.

[9] It is worth noting that an analogous problem cannot occur in CDCL: If the conjunction of the current decision literals were unsatisfiable, the last such literal would have been produced by unit propagation, and not by a decision. See Section 3.8 for a comparison of our clause learning with CDCL.

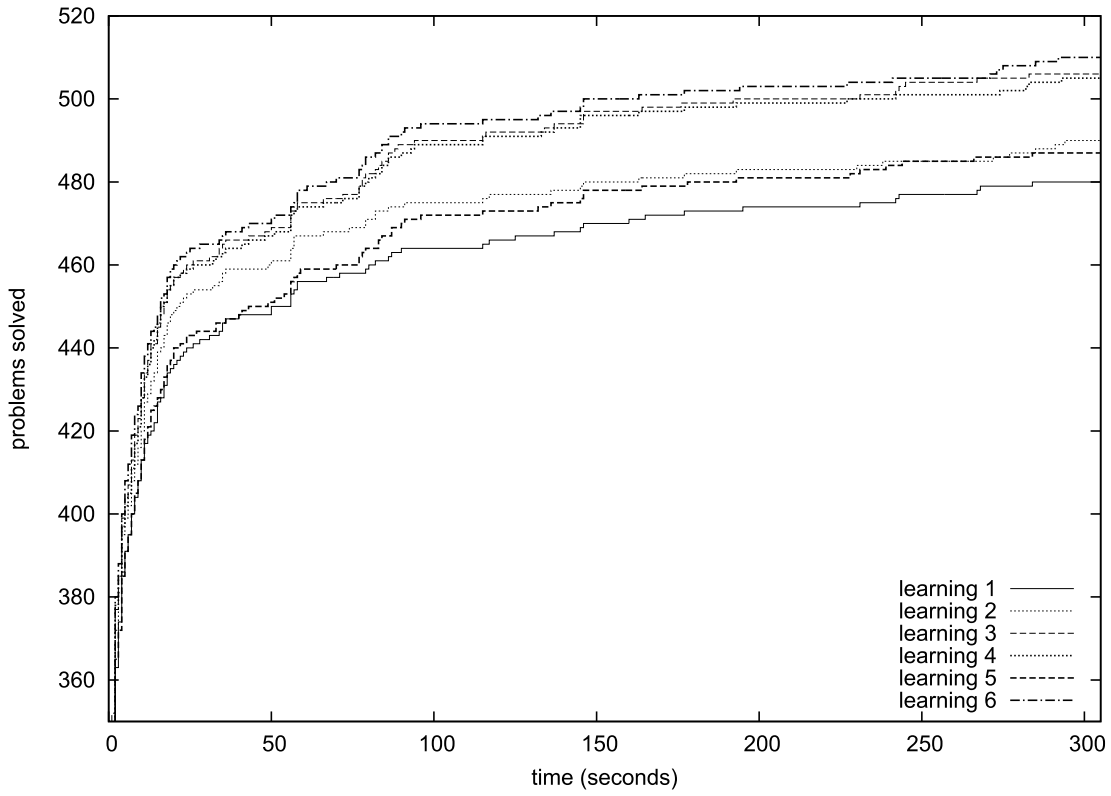


Figure 3.7: Comparison of different clause learning schemes: Number of problems solved within time limit

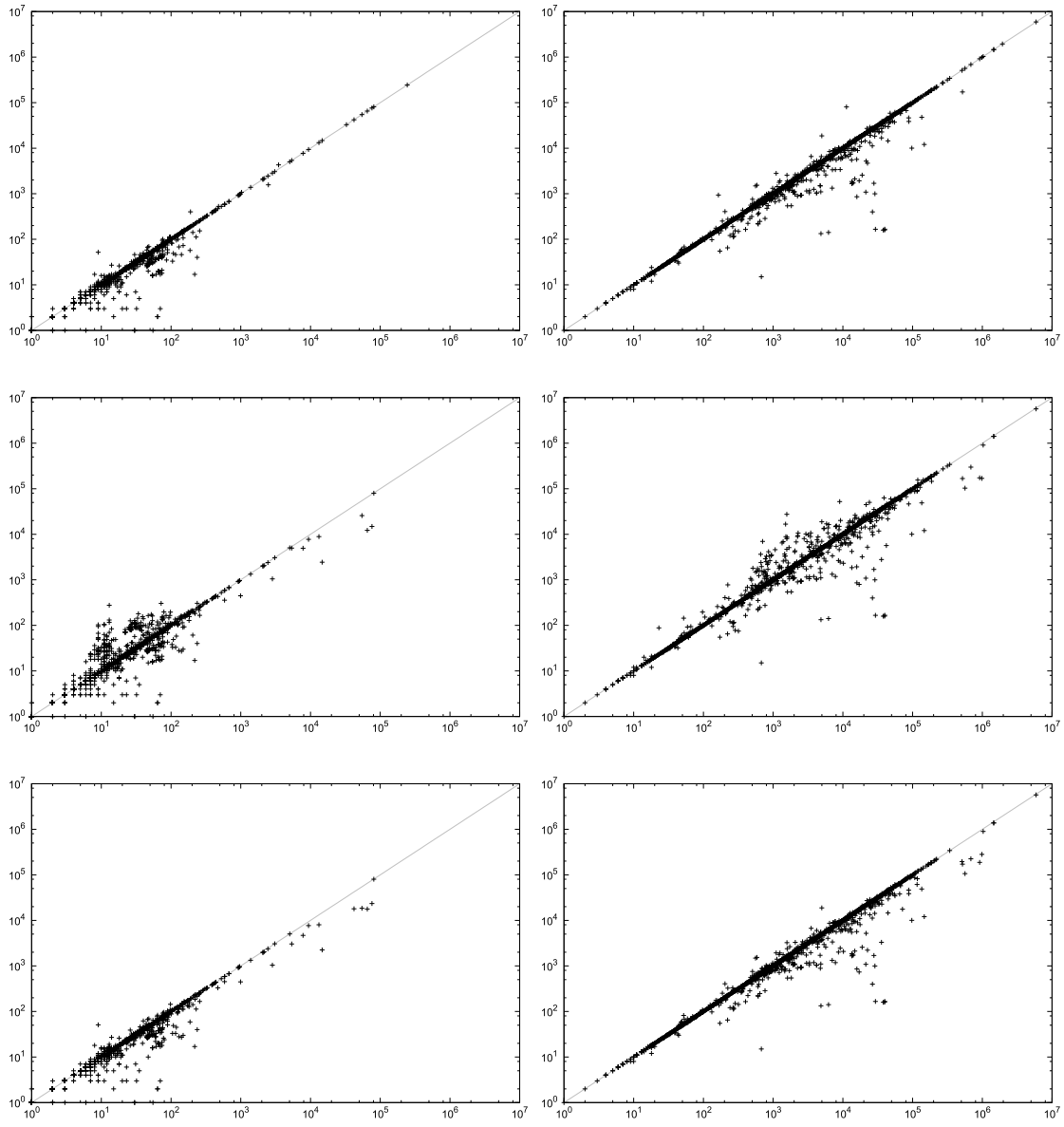


Figure 3.8: Number of splits (left) and derived clauses (right) for learning schemes 4 (top), 5 (middle) and 6 (bottom), compared to 1

3.7 Related Work

The splitting rule goes back to the DPLL procedure [DP60, DLL62] for propositional logic, where a propositional clause set N is split into the clause sets $N \cup \{A\}$ and $N \cup \{\neg A\}$ for some propositional variable A occurring positively and negatively in N, and the resulting tree is traversed in a depth-first way, by backtracking each time a contradiction has been found.

In the context of first-order logic, splitting is an indispensable ingredient of superposition-based decision procedures [BGW93, FLHT01, HW07]. Splitting with backtracking was first implemented in SPASS [Wei01, WGR96, WDF+09]. The formalization of splitting in this chapter (and in our previous work [FW09], out of which this chapter developed) can be viewed as an extension of the formalization of DPLL as an abstract calculus in [NOT06] to the full first-order case, with the additional formalization of clause learning. Clause learning in the SAT setting was first introduced in [SS96]. The use of clause labels to model explicit case analysis was first suggested in [LAWRS07].

A different lifting of DPLL/CDCL to the first-order level is the Model Evolution calculus [BT08, BPT12]. The calculus essentially splits a clause set N into the clause sets $N \cup \{A\}$ and $N \cup \{\text{skolem}(\neg A)\}$, where A is a first-order atom potentially including variables that are replaced by fresh Skolem constants in $\text{skolem}(\neg A)$. The new Skolem constants are not treated naively, instead special refinements of the calculus ensure that they can be reused to a certain extent. The clause set $N \cup \{\text{skolem}(\neg A)\}$ is not a strict subset of N (after redundancy elimination), and it is therefore not clear whether the Model Evolution calculus can be turned into a decision procedure for all the fragments for which superposition with splitting is known to be a decision procedure. Also the fairness issues we discussed in Section 3.2.1 are not addressed by the work, which instead suggests an iterative deepening approach, where the number of splits for any round is finitely bounded. The Model Evolution calculus has also been extended with rules for lemma learning [BFT06], which can also produce non-ground lemmas. Lemmas are synthesized by a guided resolution process, closely resembling the way conflict clauses are computed in CDCL. This is in contrast to the way lemmas are computed in our approach: Since we store instantiation information in the clause labels, no additional inferences need to be performed in order to produce a lemma. Indeed the authors acknowledge the high computational overhead of their approach, which is reflected in the experimental results, where the use of lemma learning does not significantly increase the number of problems solved within a given time bound.

An alternative to splitting with backtracking is based on the introduction of new propositional symbols [dN01, RV01]. In this approach, a clause C that can be decomposed into two variable disjoint components C_1 and C_2, is replaced by two clauses $q \vee C_1$ and $\neg q \vee C_2$, where q is a new propositional symbol. If $\neg q$ is selected in $\neg q \vee C_2$ and q is made the smallest atom in the ordering, no inference step between q and $\neg q$ is done as long as C_1 or C_2 have not been "resolved away", giving a flavor of explicit splitting. While

the transformation preserves satisfiability, the introduction of new propositional symbols prevents the effective use of the generated split clauses for reductions, and cannot replace the explicit splitting needed for decidability results.[10]

Recently, splitting with backtracking has also been integrated into other provers, and its performance has been compared to that of splitting through new propositional symbols [HV13, BP13]. Splitting with backtracking has been shown to perform better, as predicted by the theory.

3.8 Comparison With CDCL

In order to contrast our clause learning approach with CDCL, let us briefly recall how clause learning works in CDCL (see [SLM09] for a detailed account).

A CDCL solver tries to construct a truth assignment for all propositional variables in a given clause set, alternating between an assignment phase and a backtracking phase. During the assignment phase, the solver assigns truth values to yet unassigned variables. If a variable is unconstrained, the solver decides on a value, analogously to splitting, and associates a unique *decision level* with the literal. As soon as a clause becomes unit, meaning that all but one of its literals are assigned to false and the remaining literal is unassigned, this remaining literal is assigned to true. This is done repeatedly, in a process called *unit propagation*, until either there are no more unit clauses, or else a clause has all its literals assigned to false, in which case a conflict has been reached, and the backtracking phase begins.

The unit clause that caused a literal to become true by unit propagation is called the literal's *antecedent*, and the solver keeps track of the relation between literals and their antecedents in the *implication graph*. When a conflict occurs, the implication graph is analyzed and a conflict clause is computed, whose literals are negated *unique implication points* of the implication graph with respect to the current conflict. In the simplest case, these are just negated decision literals. The solver then backtracks to the next-smallest decision level occurring in the conflict clause, and adds the conflict clause to the clause set. Thus the choice of conflict clause affects both the backtracking as well as subsequent propagation steps. In most state-of-the-art CDCL solvers, the conflict clause is computed according to the first unique implication point (or 1-UIP) scheme [ZMMM01], whereby the conflict graph is traversed backwards in a breadth-first fashion until a conflict clause with a single literal at the current decision level is obtained. The 1-UIP scheme is optimal in terms of the obtained backtracking level [ABH+08].

From the point of view of superposition with splitting, the decision steps correspond to splitting steps, and the unit propagation steps correspond to inferences (or rather, reductions). When an empty clause is derived by superposition, its derivation (i.e., a

[10] See [FW09] for a more detailed discussion of the issues related to splitting through new propositional symbols.

DAG over clauses with the empty clause at the root) can be viewed as the counterpart to the implication graph, as illustrated by Figure 3.9. The left part shows an implication graph. The notation $l_i@d$ stands for a literal l_i assigned at decision level d. l_1 and l_2 are decision literals, l_3 is a propagated literal. The conflict occurs because of the clause $\overline{l_5} \vee \overline{l_6}$. The literal $l_4@2$ is the first unique implication point. Possible conflict clauses are $\overline{l_1} \vee \overline{l_2}$ (the negated decision literals) or $\overline{l_4}$ (the 1-UIP). The right part of Figure 3.9 shows the derivation of an empty clause from split clauses C_1 and C_2 and an input clause C_3, not necessarily unit. Assuming all clauses are ground, the global learning function returns $\mathrm{cnf}(\neg(C_1 \wedge C_2))$, which corresponds to the naive CDCL conflict clause consisting entirely of negated decision literals.

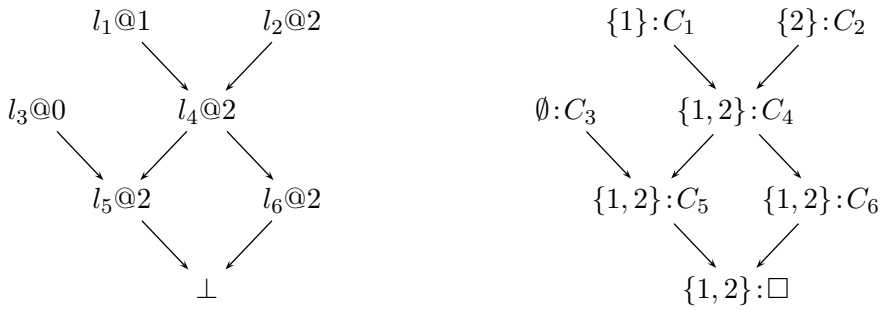


Figure 3.9: Implication graph (left) and derivation of an empty clause (right)

A natural question to ask is whether a learning scheme like 1-UIP can also be used in the context of splitting. In principle, the derivation of the empty clause could also be traversed backwards from the root in a breadth-first way, until a single clause with the same split level as the empty clause remains. In the example above, this would result in learning $\mathrm{cnf}(\neg C_4)$. If the clauses are non-ground however, they cannot be negated without introducing fresh Skolem constants, as discussed in Section 3.3, and the clauses corresponding to the first UIP may contain variables that do not occur in the clause label. Therefore, tracking the instantiation of variables originating from split clauses is not enough, instead *all* variable instantiations need to be recorded. In CDCL, the 1-UIP scheme is useful mainly because it produces better backjump levels. But in our setting, the learned clauses do not affect the backjump level at all, hence the only benefit of the 1-UIP scheme would be to produce shorter lemmas. Whether this benefit justifies the overhead of full instantiation tracking is an open question.

Finally, observe that in CDCL, as the learned clause consists of negated literals corresponding to unique implication points, it forces unit propagation under any partial assignment that makes all but one of its literals false. In this way, the learned clause functions as an explanation of the conflict which allows the solver to avoid repeating the conflict. In the context of splitting, an analogous property would be that a learned clause enables the prover to derive the empty clause using only reductions (and hence no proof search), whenever the split stack contains the same assumptions that caused the conflict. However, since split clauses are not unit in general, such a property does not hold. For example, assume that the stack contains the split clauses $A \vee B$, $C \vee D$,

and E, and that they all contributed to the derivation of the empty clause. Under the global learning scheme, we would learn

$$\begin{aligned}&\mathrm{cnf}(\neg((A \vee B) \wedge (C \vee D) \wedge E)) = \\ &(\neg A \vee \neg C \vee \neg E) \wedge (\neg B \vee \neg C \vee \neg E) \wedge (\neg A \vee \neg D \vee \neg E) \vee (\neg B \vee \neg D \vee \neg E) \qquad (1)\end{aligned}$$

Now assuming we again get a split stack containing $A \vee B$, $C \vee D$, and E, the empty clause cannot be derived from the split clauses and (1) using only reductions (like matching replacement resolution), but inferences have to be performed.

3.9 Future Work

While it would also be interesting to compare the behavior of different backtracking functions, in particular branch condensing and lazy backtracking (see Definition 3.19), our preliminary evaluation has so far been inconclusive, and we leave a thorough investigation of this as future work.

The dependency graph (Definition 3.15) of the split stack, which captures the dependencies between splits, could be exploited for other purposes as well, like for instance parallelization, as sets of clauses depending on mutually independent splits could be saturated independently of each other.

Finally, clause learning needs to be studied in more detail. So far, we have shown empirical evidence suggesting that clause learning improves the overall performance of superposition with splitting, but many open questions remain on the theoretical side: How exactly do the learned clauses contribute to shorter proofs or saturations? What types of problems benefit most from clause learning? Can we learn clauses that provide a "proof by reduction", in the sense discussed in the previous section? What theoretical results about clause learning in CDCL can be carried over or adapted to clause learning for splitting?

4 SUP(T) for Reachability

4.1 Introduction

Formal methods such as verification aim at providing correctness guarantees for hard- and software systems. Among the well-established approaches to verification, model checking, which checks whether a given model of a system satisfies a required property by exhaustively exploring its state space, has been very successful in practice [CGP01, BK08]. While explicit model checking suffers from the state explosion problem and is limited to finite-state systems, symbolic methods usually represent sets of reachable states as a logical formula.

To model practically useful systems, background theories describing the data domain are also needed. Reasoning on logical formulas together with background theories is then done using satisfiability modulo theories (SMT) methods. In a nutshell, applying transitions corresponds to syntactic transformation of the formula, and the SMT procedure is used to detect whether the set of states represented by the current formula either intersects a set of goal states, or whether it is implied by the previous formula, meaning that a fixpoint has been reached. These two checks are sufficient to obtain a procedure for checking reachability, which is a fundamental property to which other system properties can be reduced.

This approach has two drawbacks: First, the formula representing the set of reachable states can become very big. In order to simplify it, theory knowledge has to be used that goes beyond simple satisfiability. Secondly, SMT methods are typically not complete on quantified formulas, meaning that they return "unknown" even for unsatisfiable instances, thus often preventing termination.

A different approach consists in representing the system in (some form of) higher-order logic (see for example [Pau98]). While this approach has been successfully applied in practice, it has the drawback that the high expressivity of higher-order logic in general means that reasoning cannot be carried out automatically, but needs user guidance. In particular, the user has to supply suitable inductive invariants which entail the correctness of the system.

We follow a third approach, relying on first-order logic and automatic theorem proving. In our approach, the set of reachable states is implicitly described by a first-order theory, and reachable states are characterized as logical consequences. Reachability of a set of states is established by a reachability proof, or refutation of a corresponding

set of clauses. Our approach is based on superposition, and in particular on hierarchic superposition modulo theory, SUP(T), which offers completeness guarantees for reasoning in the combination of full first-order logic with a background theory, FOL(T), under suitable conditions. When dealing with infinite-state systems, one of the main challenges of saturation-based reachability analysis is termination, since the reachability problem for arbitrary reachability theories is undecidable, and decidability becomes even more elusive in the FOL(T) setting.

Our first contribution, in Section 4.3, is an extension of well-known minimal-model results to FOL(T), as a prerequisite to the following contributions.

In Section 4.4, we formally define reachability theories and establish the connection between reachability analysis and superposition-based theorem proving.

Then, in Section 4.5 we focus on FOL(LA), the combination of first-order logic with linear arithmetic, and give a first example of the power of the approach by showing that SUP(LA) is a decision procedure for reachability in timed automata. This result has previously been published in [FW12].

As reachability analysis benefits from the addition of loop invariants, we devise a rule for automatic invariant computation in SUP(LA), in Section 4.6, called constraint induction, which strictly increases the power of the calculus. We again provide an example of the usefulness of the rule, by showing that SUP(LA) with constraint induction can be turned into a decision procedure for reachability in timed automata extended with unbounded integer variables. This result has been published in [FKW12], but the treatment here fills some gaps in the presentation.

Finally, we extend the approach in two directions: First by adding background theories beyond LA to model messages and data structures, and secondly, by generalizing it to probabilistic systems, thus moving from qualitative to quantitative reachability problems. This yields the formalism of first-order probabilistic timed automata (FPTA), which we present in Section 4.7. We present a saturation-based approach for reducing max-reachability problems in FPTA to corresponding reachability problems in standard probabilistic timed automata (PTA), which can be dealt with using well-established model checking techniques. The approach relies on a labelling scheme similar to the one in Chapter 3 to enumerate reachability proofs, based on which the FPTA is instantiated into a PTA. Section 4.7 ends with a discussion of our implementation of the FPTA analysis framework and experimental evaluation on a model of a DHCP protocol over an unreliable network. This work has been previously published in [FHW10], but Section 4.7 presents a thoroughly revised version of it.

In Section 4.8, we briefly sketch an alternative approach to path enumeration, also based on labelled superposition, which may be better suited for combination with inductive reasoning.

In Section 4.9, we discuss some ideas on how to exploit the structure of systems consisting of interacting components, to avoid building the exponentially large product automaton.

The chapter ends with a discussion of related work, in Section 4.10.

4.2 Preliminaries

4.2.1 Operations on Relations

We generalize the notions about binary relations from Section 2.1 to $2n$-ary relations, for $n \geq 1$.

Definition 4.1 (Composition and Closure of Relations)
Given two relations $R_1, R_2 \subseteq S^{2n}$, their composition $(R_1 \circ R_2) \subseteq S^{2n}$ is defined by $(R_1 \circ R_2)(\vec{s}_1, \vec{s}_3)$ if and only if there exists $\vec{s}_2 \in S^n$ such that $R_1(\vec{s}_1, \vec{s}_2)$ and $R_2(\vec{s}_2, \vec{s}_3)$. Given any relation $R \subseteq S^{2n}$, we define

- $R^0 = \{(\vec{x}, \vec{x}) \mid \vec{s} \in S^n\}$,
- $R^{i+1} = R^i \circ R$ for $i \geq 0$;
- $R^+ = \bigcup_{i>0} R^i$, the *transitive closure* of R;
- $R^* = R^+ \cup R^0$, the *reflexive transitive closure* of R;
- $R^{>k} = \bigcup_{i>k} R^i$. ∎

Definition 4.2 (Product of Relations)
Given two relations $R_1, R_2 \subseteq S^{2n}$, their *product* $R_1 \cdot R_2 \subseteq S^{4n}$ is defined by $(R_1 \cdot R_2)(\vec{s}_1, \vec{s}_2, \vec{s}_1', \vec{s}_2',)$ if and only if $R_1(\vec{s}_1, \vec{s}_1')$ and $R_2(\vec{s}_2, \vec{s}_2')$. ∎

It is easy to see that $(R_1 \cdot R_2)^i = R_1^i \cdot R_2^i$ for all $i \geq 0$, and hence also $(R_1 \cdot R_2)^+ = R_1^+ \cdot R_2^+$ and $(R_1 \cdot R_2)^* = R_1^* \cdot R_2^*$

To streamline notation, we occasionally[1] take real-valued (meta-)variables to range over $\mathbb{R} \cup \{-\infty, \infty\}$, with the obvious meaning: For any arithmetic expression e, $e + \infty$ is equivalent to ∞, $e - \infty$ is equivalent to $-\infty$, and $e \geq -\infty$, $e \leq \infty$ are equivalent to true.

[1] For instance in Proposition 4.90, page 102.

4.2.2 Theory of Fixpoints

We review a few standard results of the theory of monotonic mappings and their fixpoints. Proofs of these propositions, as well as further details can be found e.g., in [Llo93].

Definition 4.3 (Least Upper Bound, Greatest Lower Bound)
Let S be a set with a partial order $\preceq$, and let $X \subseteq S$. An element $a \in S$ is an *upper bound* of X if $x \preceq a$ for all $x \in X$; it is the *least upper bound* of X if additionally $a \preceq a'$ holds for any upper bound a' of X. The least upper bound of X is unique if it exists, and is denoted by $\mathrm{lub}(X)$. Similarly, $b \in S$ is a *lower bound* of X if $b \preceq x$ for all $x \in X$; it is the *greatest lower bound* of X if additionally $b' \preceq b$ holds for any lower bound b' of X. The greatest lower bound of X is unique if it exists, and is denoted by $\mathrm{glb}(X)$. ■

Definition 4.4 (Complete Lattice)
A partially ordered set L is a *complete lattice* if $\mathrm{lub}(X)$ and $\mathrm{glb}(X)$ exist for every subset $X \subseteq L$. The top element $\mathrm{lub}(L)$ is denoted by $\top$, and the bottom element $\mathrm{glb}(L)$ is denoted by $\bot$. ■

Definition 4.5 (Directed Subset)
Let L be a complete lattice. A subset $X \subseteq L$ is called *directed* if every finite subset of X has an upper bound in X. ■

Definition 4.6 (Monotonic and Continuous Mappings)
Let L be a complete lattice. A mapping $T : L \to L$ is called *monotonic* if $x \preceq y$ implies $T(x) \preceq T(y)$, for all $x, y \in L$; it is called *continuous* if $T(\mathrm{lub}(X)) = \mathrm{lub}(T(X))$ for every directed subset[2] $X \subseteq L$. Every continuous mapping is monotonic. ■

Definition 4.7 (Least and Greatest Fixpoint)
Let L be a complete lattice and $T : L \to L$ be a mapping. An element $a \in L$ is a *fixpoint* of T if $T(a) = a$; it is the *least fixpoint* of T if additionally $a \preceq a'$ holds for any fixpoint a' of T; it is the *greatest fixpoint* of T if $a' \preceq a$ holds for any fixpoint a' of T. ■

Proposition 4.8 (Fixpoints of a Monotonic Mapping)
Let L be a complete lattice. Any monotonic mapping $T : L \to L$ has a least fixpoint, denoted by $\mathrm{lfp}(T)$, and a greatest fixpoint, denoted by $\mathrm{gfp}(T)$. Furthermore,

$$\mathrm{lfp}(T) = \mathrm{glb}\{x \mid T(x) = x\} = \mathrm{glb}\{x \mid T(x) \preceq x\}, \text{ and}$$
$$\mathrm{gfp}(T) = \mathrm{lub}\{x \mid T(x) = x\} = \mathrm{lub}\{x \mid x \preceq T(x)\}.$$ ■

Definition 4.9 (Ordinal Powers of T up to ω)
Let L be a complete lattice and $T : L \to L$ be monotonic. Then the ordinal powers of T up to ω are

$$\begin{aligned} T^0 &= \bot, \\ T^{n+1} &= T(T^n), \\ T^\omega &= \mathrm{lub}\{T^n \mid n \in \mathbb{N}\}. \end{aligned}$$

■

[2] T is naturally extended to subsets of L.

Proposition 4.10 (Least Fixpoint of a Continuous Mapping)
Let L be a complete lattice and $T : L \to L$ be continuous. Then $\mathrm{lfp}(T) = T^\omega$. ∎

The set of all Herbrand interpretations over a given signature forms a complete lattice under the partial order of set inclusion, whose bottom element is the empty set, and whose top element is the set of all ground atoms over the signature.

4.2.3 Transition Systems

The following treatment of transition systems and parallel composition is based on [BK08].

Definition 4.11 (Transition Systems)
A *transition system* is a tuple $(S, Act, \to, S_0)$ where S is a set of *states*, Act is a set of *actions*, $\to \subseteq S \times Act \times S$ is a *transition relation* and $S_0 \subseteq S$ is a set of *initial states*. We write $(S, Act, \to, s_0)$ for $(S, Act, \to, \{s_0\})$. ∎

For convenience, we write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \to$.

Definition 4.12 (Paths in a Transition System)
Let $T = (S, Act, \to, S_0)$ be a transition system. A *finite path* in T is an alternating sequence $\pi = s_0\alpha_1 s_1 \alpha_2 \ldots \alpha_n s_n$ of states and actions ending with a state, such that $s_0 \in S_0$ and $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i < n$, with $n \geq 0$. The *length* of π is n. An *infinite path* in T is an infinite alternating sequence $\pi = s_0\alpha_1 s_1 \alpha_2 \ldots$ of states and actions, such that $s_0 \in S_0$ and $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $i \geq 0$. ∎

Definition 4.13 (Parallel Composition)
Let $T_i = (S_i, Act_i, \to_i, S_{0,i})$, $i = 1, 2$ be two transition systems and $H \subseteq Act_1 \cap Act_2$ be a set of *handshake actions*. The *parallel composition* of T_1 and T_2 is defined as

$$T_1 \parallel_H T_2 = (S_1 \times S_2, Act_1 \cup Act_2, \to, S_{0,1} \times S_{0,2})$$

where $(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')$ if and only if

(i) $\alpha \in H$ and $s_1 \xrightarrow{\alpha}_1 s_1'$ and $s_2 \xrightarrow{\alpha}_2 s_2'$, or
(ii) $\alpha \notin H$ and $s_1 \xrightarrow{\alpha}_1 s_1'$ and $s_2' = s_2$, or
(iii) $\alpha \notin H$ and $s_1' = s_1$ and $s_2 \xrightarrow{\alpha}_2 s_2'$.

We write $T_1 \parallel T_2$ for $T_1 \parallel_H T_2$ with $H = Act_1 \cap Act_2$. ∎

The operator $\parallel_H$ is commutative, but not associative in general, i.e.,

$$T_1 \parallel_H (T_2 \parallel_{H'} T_3) \neq (T_1 \parallel_H T_2) \parallel_{H'} T_3$$

for $H \neq H'$. However, $\|_H$ is associative for a fixed set H.

In the sequel, we will only consider composition with synchronization over the intersection of all actions, i.e., $T_1 \,\|\, T_2$. This is sufficient, as any $T_1 \,\|_H\, T_2$ can be represented as $T_1' \,\|\, T_2'$ by renaming the actions in $(Act_1 \cap Act_2) \setminus H$.

Definition 4.14 (Post, Pre, Reachable States)
Let $T = (S, Act, \to, S_0)$ be a transition system. For $s \in S$ and $\alpha \in Act$, the sets of (α-)*successors* of s are defined as

$$\mathrm{Post}_T(s, \alpha) = \{s' \mid s \xrightarrow{\alpha} s'\}, \quad \text{and} \quad \mathrm{Post}_T(s) = \bigcup_{\alpha \in Act} \mathrm{Post}_T(s, \alpha).$$

Similarly, the sets of (α-)*predecessors* of s are defined as

$$\mathrm{Pre}_T(s, \alpha) = \{s' \mid s' \xrightarrow{\alpha} s\}, \quad \text{and} \quad \mathrm{Pre}_T(s) = \bigcup_{\alpha \in Act} \mathrm{Pre}_T(s, \alpha).$$

These definitions are naturally extended to $Q \subseteq S$:

$$\mathrm{Post}_T(Q, \alpha) = \bigcup_{q \in Q} \mathrm{Post}_T(q, \alpha), \quad \text{and} \quad \mathrm{Post}_T(Q) = \bigcup_{q \in Q} \mathrm{Post}_T(q),$$
$$\mathrm{Pre}_T(Q, \alpha) = \bigcup_{q \in Q} \mathrm{Pre}_T(q, \alpha), \quad \text{and} \quad \mathrm{Pre}_T(Q) = \bigcup_{q \in Q} \mathrm{Pre}_T(q).$$

A state $s \in S$ is *reachable* in T if $s \in \mathrm{Post}_T^*(S_0)$. ∎

An easy consequence of the above definition is that

$$\mathrm{Post}_T(Q_1, \alpha) \cup \mathrm{Post}_T(Q_2, \alpha) = \mathrm{Post}_T(Q_1 \cup Q_2, \alpha)$$
$$\mathrm{Pre}_T(Q_1, \alpha) \cup \mathrm{Pre}_T(Q_2, \alpha) = \mathrm{Pre}_T(Q_1 \cup Q_2, \alpha)$$

holds for any transition system T and sets of states Q_1, Q_2.

Sometimes, we will not be interested in the actions of a transition system. In those cases, we omit them from the definition and consider transition systems to be tuples $(S, \to, S_0)$. The notions of successors, predecessors and reachability carry over in the obvious way.

4.3 Hierarchic Superposition and Minimal Models

Since we want to model a transition system by a reachability theory, a particular model of the theory has to be singled out to represent the transition system. A natural choice is the minimal Herbrand model. For this reason, we will require reachability theories to have unique minimal Herbrand models.

It is well-known that Horn theories have at most one minimal Herbrand model, and our reachability theories will essentially be Horn theories.

However, to be able to model practically useful transition systems, we will require *background theories*. For example, in order to represent the transition system of a timed automaton, one has to be able to express arithmetic operations and relations. The transition system would then be encoded as a transition system *modulo* the theory of linear arithmetic, a notion we will formally introduce in Definition 4.32.

Modular integration of theory reasoning into superposition is an active research topic, which has yielded the concept of *hierarchic superposition*, also called SUP(T) [BGW94, Kru13]. We will base our treatment of background theories on the hierarchic superposition framework. We here recall its main notions and results.

4.3.1 Hierarchic Specifications

Definition 4.15 (Hierarchic Specifications)
A *hierarchic specification* is a pair $(\mathsf{Sp}_b, \mathsf{Sp}_e)$ of specifications, where $\mathsf{Sp}_b = (\Sigma_b, \mathscr{C}_b)$ is the *base specification*, and $\mathsf{Sp}_e = (\Sigma_e, N_e)$ is the *extension*, and $\Sigma_b \subseteq \Sigma_e$. The interpretations in $\mathscr{C}_b$ are called *base models*, and Sp_b is called the *base theory*. The sorts and operator symbols in Σ_b are called *base sorts* and *base operator symbols*, respectively. The sorts and operator symbols in $\Sigma_e \setminus \Sigma_b$ are called *free sorts* and *free operator symbols*, respectively. Together with N_e they are also referred to as the *enrichment*. A free operator symbol is called an *extension symbol* if it ranges into a base sort. We define the sets of *base variables* and *non-base variables*, $X_b, X_e \subseteq \mathcal{X}$, to consist of all variables whose sort is in $\mathcal{S}_b$ or $\mathcal{S}_e$, respectively. ∎

We use the word *theory* to refer both to specifications and to hierarchic specifications.

4.3.2 Syntax

Definition 4.16 (Pure, (Non-)Base and Free Expressions)
A Σ_e-term t is called

- *pure* if the operator symbols occurring in it are either all base or all free;
- *base* if all operator symbols and variables occurring in it are base;
- *non-base* if it is not base;
- *free* if all operator symbols occurring in it are free.

These notions are naturally extended to atoms, literals and clauses. ∎

Definition 4.17 (Abstracted Clauses)
A clause is said to be *abstracted* if all its literals are pure. An abstracted clause C is usually written as

$$C = \Lambda \parallel \Gamma \to \Delta$$

where Λ consists of base literals, and Γ, Δ consist of free atoms. Λ, Γ and Δ are called the (clause) *constraint*, the *antecedent* and the *succedent* of C, respectively. The constraint is also called the *base part* of C, while the antecedent and succedent form the *free part* of C. Two abstracted clauses $\Lambda_1 \parallel C_1$ and $\Lambda_2 \parallel C_2$ are *variants* of each other, if C_1, C_2 are variants of each other. ■

Semantically, an abstracted clause $\Lambda \parallel \Gamma \to \Delta$ represents the implication

$$\bigwedge \Lambda \wedge \bigwedge \Gamma \to \bigvee \Delta.$$

Any disjunction of Σ_e-literals can be transformed into an abstracted clause by the following syntactic transformation called *abstraction* or *purification*: Whenever a subterm t, whose top symbol is base, occurs immediately below a free operator symbol, it is replaced by a new base variable x ("abstracted out") and the equation $x \simeq t$ is added to Γ. The same transformation is applied if the top symbol of t is an extension symbol, and t occurs immediately below a base operator symbol. This transformation is repeated until all terms in the clause are pure; then all base atoms are moved to Λ and all free ones to Γ or Δ, depending on their polarity.

Definition 4.18 (Simple Substitutions and Instances)
A substitution σ is called *simple* if it maps base variables to base terms. If t is a term and σ a simple substitution, then $t\sigma$ is called a *simple instance* of t; if $t\sigma$ is ground, it is called a *simple ground instance* of t. The set of all simple ground instances of t is denoted by $\mathrm{sgi}(t)$. The notion of simple (ground) instance is naturally lifted to atoms, literals, clauses and clause sets. ■

4.3.3 Semantics

Definition 4.19 (Restriction $\mathcal{I}'|_\Sigma$)
Let $\Sigma = (\mathcal{S}, \Omega)$, $\Sigma' = (\mathcal{S}', \Omega')$, $\Sigma \subseteq \Sigma'$ be signatures, and let $\mathcal{I}'$ be a Σ'-interpretation. The *restriction* of $\mathcal{I}'$ to Σ, written $\mathcal{I}'|_\Sigma$, is the Σ-interpretation $\mathcal{I}$ such that $S_\mathcal{I} = S_{\mathcal{I}'}$ for all $S \in \mathcal{S}$, and $f_\mathcal{I} = f_{\mathcal{I}'}$ for all $f \in \Omega$. ■

That is, $\mathcal{I}'|_\Sigma$ is obtained from $\mathcal{I}$ by removing all carrier sets $S_{\mathcal{I}'}$ with $S \in \mathcal{S}' \setminus \mathcal{S}$ and all functions $f_{\mathcal{I}'}$ with $f \in \Omega' \setminus \Omega$.

Definition 4.20 (Hierarchic Interpretation)
A *hierarchic interpretation* for a hierarchic specification $(\mathsf{Sp}_b, \mathsf{Sp}_e)$ is a Σ_e-interpretation $\mathcal{I}$ whose restriction $\mathcal{I}|_{\Sigma_b}$ to Σ_b is isomorphic to some base model $\mathcal{B} \in \mathscr{C}_b$. A hierarchic interpretation that satisfies a set N of clauses over Σ_e is a Sp_b-*hierarchic model* of N, or simply a hierarchic model. ■

The above definition of hierarchic interpretations differs slightly from the one given in [BGW94, Kru13], where $\mathcal{I}|_{\Sigma_b}$ is required to be *contained* in $\mathscr{C}_b$, while we only require $\mathcal{I}|_{\Sigma_b}$ to be isomorphic to some $\mathcal{B} \in \mathscr{C}_b$. This change is in line with our definition of hierarchic specifications (Definition 2.26) and is equivalent to the original definition.

Definition 4.21
Let $\mathcal{T}$ be a base theory (i.e., a base specification), and N an extension over Σ. We write

- $\mathrm{Mod}(N)$ for the set of all models of N,
- $\mathrm{Mod}_\Sigma(N)$ for the set of all Herbrand models of N,
- $\mathrm{Mod}_\mathcal{T}(N)$ for the set of all $\mathcal{T}$-hierarchic models of N,
- $\mathrm{Mod}_{\mathcal{T},\Sigma}(N)$ for the set of all $\mathcal{T}$-hierarchic Herbrand models of N.

If ϕ is a Σ-formula, we write

- $N \models_\Sigma \phi$ if $\mathcal{I} \models \phi$ for all $\mathcal{I} \in \mathrm{Mod}_\Sigma(N)$,
- $N \models_\mathcal{T} \phi$ if $\mathcal{I} \models \phi$ for all $\mathcal{I} \in \mathrm{Mod}_\mathcal{T}(N)$,
- $N \models_{\mathcal{T},\Sigma} \phi$ if $\mathcal{I} \models \phi$ for all $\mathcal{I} \in \mathrm{Mod}_{\mathcal{T},\Sigma}(N)$. ■

It follows from the Definition 4.21 that

$$\mathrm{Mod}_{\mathcal{T},\Sigma}(N) \subseteq \mathrm{Mod}_\mathcal{T}(N) \subseteq \mathrm{Mod}(N)$$

and

$$\mathrm{Mod}_{\mathcal{T},\Sigma}(N) \subseteq \mathrm{Mod}_\Sigma(N) \subseteq \mathrm{Mod}(N).$$

Another easy consequence of the above definitions is that whenever $\mathcal{T}$ is the hierarchic combination of two signature-disjoint theories $\mathcal{T}_1$ and $\mathcal{T}_2$ (which in particular have no sorts in common), then $\models_\mathcal{T} \phi$ is equivalent to $\models_{\mathcal{T}_i} \phi$, for any formula ϕ containing only symbols from $\mathcal{T}_i$.

Definition 4.22 ($E_\mathcal{I}$, $D_\mathcal{I}$, $\overline{E_\mathcal{I}}$, $\overline{D_\mathcal{I}}$)
Let $\mathcal{I}$ be a Herbrand-interpretation over Σ. We define the sets

$$E_\mathcal{I} = \{s \simeq t \mid s,t \in T(\Sigma) \text{ and } s \neq t \text{ and } s \simeq t \in \mathcal{I}\} \text{ and}$$
$$D_\mathcal{I} = \{s \not\simeq t \mid s,t \in T(\Sigma) \text{ and } s \neq t \text{ and } s \simeq t \notin \mathcal{I}\}$$

of equations (disequations) between distinct ground base terms that do (not) hold in $\mathcal{I}$. By $\overline{E_\mathcal{I}}$ ($\overline{D_\mathcal{I}}$) we denote the complement of $E_\mathcal{I}$ ($D_\mathcal{I}$) with respect to the set of all equations (disequations) between distinct ground base terms. ■

Definition 4.23 (Sufficient Completeness)
Given a hierarchic specification $(\mathsf{Sp}_b, \mathsf{Sp}_e)$, a set N of clauses over Σ_b is called *sufficiently complete* (with respect to simple instances), if for every non-base ground term of base sort t, and for any base model $\mathcal{I} \in \mathscr{C}_b$, there exists a base ground term t' of the same sort as t, such that $\mathrm{sgi}(N) \cup E_{\mathcal{I}} \models t \simeq t'$. ■

The above definition is the one given in [AKW09]. A weaker sufficient completeness criterion (i.e., one that is satisfied by more clause sets) is given in [Kru13], involving the notion of *weak algebras*. We stick with the stronger definition for the sake of simplicity. All results presented in the following can be straightforwardly adapted to the refined sufficient completeness criterion.

In the following, we will always view the base specification as $(\Sigma_b, \mathscr{C}_b)$, by taking $\mathscr{C}_b = \mathrm{Mod}_{\Sigma_b}(N_b)$ if the base models are axiomatized by some clause set N_b. That is, we consider the Herbrand models of N_b as base models.[3]

4.3.4 Superposition Modulo Theory

In this section, we briefly introduce the hierarchic superposition calculus SUP(T). More details can be found in [BGW92, BGW94, AKW09, Kru13]. We write SUP(T) to refer to the general hierarchic superposition calculus, while we use SUP($\mathcal{T}$) to denote a particular instance of the calculus with a base theory $\mathcal{T}$, like in the case of SUP(LA). Similarly, we write FOL(T) and FOL($\mathcal{T}$), respectively, to refer to the hierarchic combination of free first-order logic with some unspecified base theory, and to the combination with a particular base theory $\mathcal{T}$. Figures 4.1 and 4.2 show the inference rules constituting the superposition modulo $\mathcal{T}$, or SUP($\mathcal{T}$) calculus.

Besides inference rules, reduction rules can also be lifted from the standard superposition calculus to SUP($\mathcal{T}$), and are indeed essential for efficient reasoning. We will only consider subsumption deletion and tautology deletion, as these are the most important reduction rules, and also the only ones we will need in this chapter. They are shown in Figure 4.3. The function smgu returns the most general *simple* unifier of two terms.

Variable Elimination in Constraints In practical implementations of SUP($\mathcal{T}$), like SUP(LA), the constraints of derived clauses are often simplified (in an equivalence-preserving way) before the clause is added to the set of derived clauses. Such simplification may in particular include the elimination of variables occurring only in the constraint, but not in the free part of the clause. In the case of SUP(LA), real-sorted variables not occurring in the free part are eliminated by default. Formally, a clause

[3] See the discussion after Definition 2.26 on page 14.

Hierarchic Superposition Left:

$$\mathcal{I}\frac{\Lambda_1 \parallel \Gamma_1 \to \Delta_1, l \simeq r \quad \Lambda_2 \parallel s[l'] \simeq t, \Gamma_2 \to \Delta_2}{(\Lambda_1, \Lambda_2 \parallel s[r] \simeq t, \Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2)\sigma}$$

where

(i) $\sigma = \mathrm{smgu}(l, l')$,
(ii) l' is not a variable,
(iii) $l\sigma \not\preceq r\sigma$,
(iv) $s\sigma \not\preceq t\sigma$,
(v) $(l \simeq r)\sigma$ is strictly maximal in $(\Gamma_1 \to \Delta_1, l \simeq r)\sigma$ and no literal is selected in Γ_1, and
(vi) $(s \simeq t)\sigma$ is maximal in $(s \simeq t, \Gamma_2 \to \Delta_2)\sigma$ and either no literal in Γ_2 is selected, or $s \simeq t$ is selected

Hierarchic Superposition Right:

$$\mathcal{I}\frac{\Lambda_1 \parallel \Gamma_1 \to \Delta_1, l \simeq r \quad \Lambda_2 \parallel \Gamma_2 \to \Delta_2, s[l'] \simeq t}{(\Lambda_1, \Lambda_2 \parallel \Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2, s[r] \simeq t)\sigma}$$

where

(i) $\sigma = \mathrm{smgu}(l, l')$,
(ii) l' is not a variable,
(iii) $l\sigma \not\preceq r\sigma$,
(iv) $s\sigma \not\preceq t\sigma$,
(v) $(l \simeq r)\sigma$ is strictly maximal in $(\Gamma_1 \to \Delta_1, l \simeq r)\sigma$ and no literal is selected in Γ_1, and
(vi) $(s \simeq t)\sigma$ is strictly maximal in $(\Gamma_2 \to \Delta_2, s \simeq t)\sigma$ and no literal is selected in Γ_2

Hierarchic Equality Resolution:

$$\mathcal{I}\frac{\Lambda \parallel \Gamma, s \simeq t \to \Delta}{(\Lambda \parallel \Gamma \to \Delta)\sigma}$$

where

(i) $\sigma = \mathrm{smgu}(s, t)$ and
(ii) either $(s \simeq t)\sigma$ is maximal in $(\Lambda \parallel \Gamma, s \simeq t \to \Delta)\sigma$ and no literal is selected in Γ, or $s \simeq t$ is selected in the premise.

Figure 4.1: Inference rules of the Calculus SUP($\mathcal{T}$) (1)

Hierarchic Equality Factoring:

$$\mathcal{I}\frac{\Lambda \parallel \Gamma \to \Delta, s \simeq t, s' \simeq t'}{(\Lambda \parallel \Gamma, t \simeq t' \to \Delta, s' \simeq t')\sigma}$$

where

(i) $\sigma = \mathrm{smgu}(s, s')$,
(ii) $s\sigma \not\preceq t\sigma$, and
(iii) $(s \simeq t)\sigma$ is maximal in $(\Gamma \to \Delta, s \simeq t, s' \simeq t')\sigma$ and no literal is selected in the premise.

Constraint Refutation:

$$\mathcal{I}\frac{\Lambda_1 \parallel \to \quad \ldots \quad \Lambda_n \parallel \to}{\square}$$

where $(\Lambda_1 \parallel \to), \ldots, (\Lambda_n \parallel \to) \models_{\mathcal{T}} \bot$

Figure 4.2: Inference rules of the Calculus SUP($\mathcal{T}$) (2)

$\Lambda[\vec{x}, \vec{y}] \parallel C[\vec{x}]$ is simplified to $\Lambda'[\vec{x}] \parallel C[\vec{x}]$ where Λ' is computed by real quantifier elimination from $\exists\vec{y}.\Lambda$. This also works if Λ contains integer variables, as long as all eliminated variables $\vec{y}$ are real-sorted. Integer-sorted variables cannot be eliminated in general.[4] The original and the simplified clause are obviously equivalent (wrt. $\mathcal{T}$), and it is also easy to see that a clause C_1 with constraint Λ_1 subsumes a clause C_2 with constraint Λ_2 if and only if the simplified version of C_1 subsumes the simplified version of C_2, since

$$\forall\vec{x}.\exists\vec{y}.((\exists\vec{u}.\Lambda_2) \to (\exists v.\Lambda_1\sigma))$$

is logically equivalent to

$$\forall\vec{x}\vec{u}.\exists\vec{y}\vec{v}.(\Lambda_2 \to \Lambda_1\sigma).$$

Therefore we can (and will) assume without loss of generality that all real-sorted variables occurring in the LA constraint of a FOL(LA) clause also occur in the free part of the clause.

Effect of Simple Unifiers on Constraints Another important observation is that, since all unifiers are simple, and base terms are always abstracted, any unifier in SUP(LA) maps base variables to base variables. So whenever a simple unifier σ is applied to an LA constraint Λ, there exists an "inverse renaming" substitution ρ, such that $\Lambda\sigma\rho$ implies $\exists\vec{x}.\Lambda$ for some $\vec{x} \subseteq \mathrm{var}(\Lambda)$. For example, consider $\Lambda = 1 \leq x, y \leq 2$ and $\sigma = \{x \mapsto u, y \mapsto u\}$. Then $\Lambda\sigma = 1 \leq u \leq 2$ and $(1 \leq u \leq 2)\{u \mapsto x\} = 1 \leq x \leq 2$ and $\models_{\mathrm{LA}} \forall x.(1 \leq x \leq 2 \to \exists y.\Lambda)$.

[4] The language of clause constraints is not powerful enough to allow elimination of integer-sorted variables. It could be suitably strengthened by adding the floor function $\lfloor\cdot\rfloor$, see [Wei99b].

Subsumption Deletion:

$$\mathcal{I}\frac{\Lambda_1 \parallel \Gamma_1 \to \Delta_1 \quad \Lambda_2 \parallel \Gamma_2 \to \Delta_2}{\Lambda_1 \parallel \Gamma_1 \to \Delta_1}$$

where σ is a simple matcher such that

(i) $\Gamma_1\sigma \subseteq \Gamma_2$, $\Delta_1\sigma \subseteq \Delta_2$,
(ii) $\models_{\mathcal{T}} \forall\vec{x}.\exists\vec{y}.(\Lambda_2 \to \Lambda_1\sigma)$, where $\vec{x}$ are all variables *of base sort* in $\Lambda_2 \parallel \Gamma_2 \to \Delta_2$, and $\vec{y} = \mathrm{var}(\Lambda_1\sigma) \setminus \vec{x}$, and
(iii) $\Lambda_2 \parallel \Gamma_2 \to \Delta_2 \neq \square$.

Tautology Deletion:

$$\mathcal{I}\frac{\Lambda \parallel \Gamma \to \Delta}{}$$

where $\models \Gamma \to \Delta$ or $\exists\vec{x}.\Lambda \models_{\mathcal{T}} \bot$, for $\vec{x} = \mathrm{var}(\Lambda)$.

[

Figure 4.3: SUP($\mathcal{T}$) reduction rules

Remark 4.24
Consider a SUP($\mathcal{T}$) derivation

$$C_1, \ldots, C_m \vdash C_{m+1}$$

where each C_i is an abstracted clause with constraint Λ_i. Then

$$\Lambda_{m+1} = \bigcup_{i\in[1,m]} \hat{\Lambda}_i \quad \text{with} \quad \hat{\Lambda}_i = \bigcup_{j\in I_i} \Lambda_i\sigma_j$$

where the I_i are non-empty sets of indices and the σ_j are substitutions.

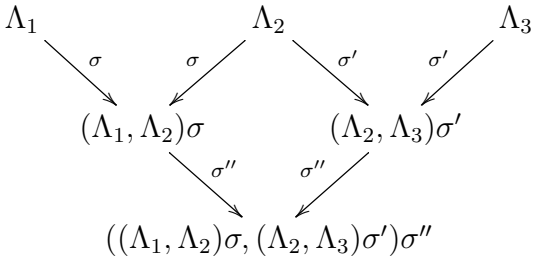
Furthermore, soundness of the inference rules guarantees that

$$\models_{\mathcal{T}} \left(\bigwedge_{i\in[1,m]} \bigwedge_{j\in I_i} C_i\sigma_j \right) \to C_{m+1}$$

■

Example 4.25
Consider the following SUP($\mathcal{T}$) derivation, where only the clause constraints and unifiers

are shown:



The constraint of the final conclusion is equivalent to $\Lambda_1\sigma_1, \Lambda_2\sigma_1, \Lambda_2\sigma_2, \Lambda_3\sigma_2$ with $\sigma_1 = \sigma\sigma''$ and $\sigma_2 = \sigma'\sigma''$. ∎

4.3.5 Minimal Models of Hierarchic Theories

For sufficiently complete clause sets, an alternative characterization of hierarchic models is given by the Hierarchic Model Lemma ([Kru13], Lemma 3.79):

Lemma 4.26 (Hierarchic Model Lemma [Kru13])
The hierarchic models of a sufficiently complete extension N are exactly the Σ_e-interpretations satisfying $\mathrm{sgi}(N) \cup E_{\mathcal{I}} \cup D_{\mathcal{I}}$ for some $\mathcal{I} \in \mathscr{C}_b$. ∎

Intuitively, the reason is the following: Let $\mathcal{M}$ be a Σ_e-interpretation. If $\mathcal{M} \models E_{\mathcal{I}}$, then there must be a homomorphism from $\mathcal{I}$ into the base universe of $\mathcal{M}$. If additionally $\mathcal{M} \models D_{\mathcal{I}}$, then this homomorphism must be injective (i.e., $\mathcal{M}$ does not collapse elements of the base universe). Finally, sufficient completeness of N guarantees that (i) the homomorphism is also surjective, because in $\mathcal{M}$, any non-base ground term of base sort is equal to a base ground term (i.e., $\mathcal{M}$ does not add extra elements to the base universe), and (ii) that $\mathcal{M}$ satisfies *all* ground instances of N, because any ground substitution is "equivalent" to a simple ground substitution. In sum, the restriction of $\mathcal{M}$ to the base signature is isomorphic to $\mathcal{I} \in \mathscr{C}_b$, and $\mathcal{M}$ satisfies all clauses in N, therefore $\mathcal{M}$ is a hierarchic model of N.

We will use hierarchic specifications to represent transition systems modulo background theories. The idea is to represent the reachability theory[5] $\mathcal{T}_{\mathrm{TS}}$ as a hierarchic specification, whose base specification describes the background theory. The background theory may itself be given as a hierarchic specification, and so on, all the way down to an initial theory $\mathcal{T}_0$:

$$\underset{(\Sigma_0, \mathscr{C}_0)}{\mathcal{T}_0} \quad \rightsquigarrow \quad \underset{(\Sigma_1, N_1)}{\mathcal{T}_1} \quad \rightsquigarrow \quad \ldots \quad \rightsquigarrow \quad \underset{(\Sigma_n, N_n)}{\mathcal{T}_n = \mathcal{T}_{\mathrm{TS}}}$$

In this scheme, each $\mathcal{T}_{i+1}$ is an extension of $\mathcal{T}_i$, and $\Sigma_i \subseteq \Sigma_{i+1}$. The initial theory $\mathcal{T}_0$ may have its models defined explicitly as a class of models, which need not be first-order axiomatizable, whereas all subsequent extensions are given by clausal axiomatizations.

[5] We will formally define reachability theories in Section 4.4.

When $\mathcal{T}_0$ is given by a set of models, the corresponding hierarchic superposition calculus $\mathrm{SUP}(\mathcal{T}_0)$ will be used (the only such theory we will consider is the theory of linear real arithmetic). Otherwise, the standard superposition calculus can be used, as we will show in the following.

As discussed in the introduction, we want $\mathcal{T}_{\mathrm{TS}}$ to have a unique minimal Herbrand model. Therefore, the successive $\mathcal{T}_i$ should also have this property. We will now discuss how this can be achieved.

The main result of this section can be summarized as follows: First, if $\mathcal{T}_i$ has a unique minimal Herbrand model (possibly among other Herbrand models), then any extension by a sufficiently complete Horn clause set N_{i+1} yields a theory $\mathcal{T}_{i+1}$ with a unique minimal Herbrand model. Secondly, if $\mathcal{T}_i$ has a unique Herbrand model, then any extension by a sufficiently complete clause set N_{i+1} that is Horn *modulo* $\mathcal{T}_i$ (Definition 4.29) yields a theory $\mathcal{T}_{i+1}$ with a unique minimal Herbrand model. A special case of the second result is that any sufficiently complete FOL(LA) theory that is Horn modulo LA has a unique minimal Herbrand model.

Model Intersection for Hierarchic Horn Specifications. We begin with the first case. Assume a hierarchic specification $(\mathsf{Sp}_b, \mathsf{Sp}_e)$, where the class of base models is closed under intersection,[6] i.e., whenever $\{\mathcal{B}_i\}_{i\in I}$ is a non-empty set of Herbrand interpretations in $\mathscr{C}_b$, then $\bigcap_{i\in I} \mathcal{B}_i$ is also in $\mathscr{C}_b$. Hence there is a unique minimal (with respect to subset inclusion) model in $\mathscr{C}_b$. The model intersection property also (trivially) holds if $\mathscr{C}_b$ consists of a single Herbrand interpretation (as is the case for any *complete* theory, like the theory of linear arithmetic).

Lemma 4.27
Let $\{\mathcal{B}_i\}_{i\in I}$ be a non-empty set of Herbrand interpretations over some signature Σ. Then $\bigcup_{i\in I} D_{\mathcal{B}_i} = D_{(\bigcap_{i\in I} \mathcal{B}_i)}$.

Proof. As $\mathcal{B}_i$ are Herbrand interpretations, we have $D_{\mathcal{B}_i} = \{s \not\simeq t \mid s,t \in T(\Sigma) \text{ and } s \neq t \text{ and } s \simeq t \notin \mathcal{B}_i\}$. Hence

$$\begin{aligned}\bigcup_{i\in I} D_{\mathcal{B}_i} &= \{s \not\simeq t \mid s,t \in T(\Sigma) \text{ and } s \neq t \text{ and } s \simeq t \notin \bigcap_{i\in I} \mathcal{B}_i\} \\ &= D_{(\bigcap_{i\in I} \mathcal{B}_i)} \ .\end{aligned}$$

□

Proposition 4.28
Let $(\mathsf{Sp}_b, \mathsf{Sp}_e)$ be a hierarchic specification where the class of base models is closed under intersection, and N_e is sufficiently complete and Horn. Then, writing $\mathcal{T}$ for Sp_b and Σ for $\Sigma_b \cup \Sigma_e$, the set $\mathrm{Mod}_{\mathcal{T},\Sigma}(N_e)$ is closed under intersection.

[6] Here we view Herbrand interpretations as sets of ground equations.

Proof. Let $\{\mathcal{I}_i\}_{i\in I}$ be a non-empty subset of $\mathrm{Mod}_{\mathcal{T},\Sigma}(N_e)$, and let $\mathcal{I} = \bigcap_{i\in I} \mathcal{I}_i$. We show that $\mathcal{I} \in \mathrm{Mod}_{\mathcal{T},\Sigma}(N_e)$. By Lemma 4.26, we know that there exist base models $\mathcal{B}_i \in \mathscr{C}_b$ such that

$$\mathcal{I}_i \models \mathrm{sgi}(N_e) \cup E_{\mathcal{B}_i} \cup D_{\mathcal{B}_i}.$$

for each $i \in I$. Let $\mathcal{B} = \bigcap_{i\in I} \mathcal{B}_i$. By assumption, $\mathcal{B} \in \mathscr{C}_b$. Furthermore, as N_e is Horn, so is $\mathrm{sgi}(N_e)$, and hence $\mathrm{Mod}_\Sigma(\mathrm{sgi}(N_e))$ is closed under intersection. Thus $\mathcal{I}_i \models \mathrm{sgi}(N_e)$ implies $\mathcal{I} \models \mathrm{sgi}(N_e)$. From $\mathcal{I}_i \models E_{\mathcal{B}_i}$, it follows that $E_{\mathcal{B}_i} \subseteq \mathcal{I}_i$, and hence $\bigcap_{i\in I} E_{\mathcal{B}_i} = E_{\bigcap \mathcal{B}_i} = E_{\mathcal{B}} \subseteq \mathcal{I}$. Finally, from $\mathcal{I}_i \models D_{\mathcal{B}_i}$, it follows that $\overline{D}_{\mathcal{B}_i} \cap \mathcal{I}_i = \emptyset$, and hence $(\bigcup_{i\in I} \overline{D}_{\mathcal{B}_i}) \cap \mathcal{I} = \emptyset$. Since $\bigcup_{i\in I} \overline{D}_{\mathcal{B}_i} = \overline{D}_{\bigcap \mathcal{B}_i} = \overline{D}_{\mathcal{B}}$, we get that $\mathcal{I} \models D_{\mathcal{B}}$. In sum, $\mathcal{I} \models \mathrm{sgi}(N_e) \cup E_{\mathcal{B}} \cup D_{\mathcal{B}}$. Applying Lemma 4.26 again, we obtain $\mathcal{I} \in \mathrm{Mod}_{\mathcal{T},\Sigma}(N_e)$. □

Horn Modulo Theory Specifications. Now we turn to the second case, namely base theories with unique Herbrand models.

Definition 4.29 (Horn modulo Theory)
Given a hierarchic specification $(\mathsf{Sp}_b, \mathsf{Sp}_e)$ we call the extension N_e *Horn modulo* Sp_b (or Horn modulo the base theory) if every clause in N_e contains at most one non-base atom in the succedent. ■

Proposition 4.30
Let $(\mathsf{Sp}_b, \mathsf{Sp}_e)$ be a hierarchic specification where Sp_b has a unique base model $\mathscr{C}_b = \{\mathcal{B}\}$, and N_e is sufficiently complete and Horn modulo Sp_b. Then, writing $\mathcal{T}$ for Sp_b and Σ for $\Sigma_b \cup \Sigma_e$, the set $\mathrm{Mod}_{\mathcal{T},\Sigma}(N_e)$ is closed under intersection.

Proof. Let $\{\mathcal{I}_i\}_{i\in I}$ be a non-empty subset of $\mathrm{Mod}_{\mathcal{T},\Sigma}(N_e)$, and let $\mathcal{I} = \bigcap_{i\in I} \mathcal{I}_i$. We show that $\mathcal{I} \in \mathrm{Mod}_{\mathcal{T},\Sigma}(N_e)$. By Lemma 4.26, we know that

$$\mathcal{I}_i \models \mathrm{sgi}(N_e) \cup E_{\mathcal{B}} \cup D_{\mathcal{B}}.$$

holds for each $i \in I$. From $\mathcal{I}_i \models E_{\mathcal{B}}$, it follows that $E_{\mathcal{B}} \subseteq \mathcal{I}_i$, so $E_{\mathcal{B}} \subseteq \mathcal{I}$ and hence $\mathcal{I} \models E_{\mathcal{B}}$. From $\mathcal{I}_i \models D_{\mathcal{B}}$, it follows that $\overline{D}_{\mathcal{B}} \cap \mathcal{I}_i = \emptyset$, so $\overline{D}_{\mathcal{B}} \cap \mathcal{I} = \emptyset$, and hence $\mathcal{I} \models D_{\mathcal{B}}$. Now assume for contradiction that $\mathcal{I} \not\models \mathrm{sgi}(N_e) \cup E_{\mathcal{B}} \cup D_{\mathcal{B}}$, which is equivalent to $\mathcal{I} \not\models \mathrm{sgi}(N_e)$. So let $C = A_1, \ldots, A_n, B_1, \ldots, B_k \to A, B_{k+1}, \ldots, B_m \in \mathrm{sgi}(N_e)$ where A, A_j are non-base, and the B_j are base atoms, such that $\mathcal{I} \not\models C$. Thus $\mathcal{I} \models \{A_1, \ldots, A_n, B_1, \ldots, B_k\}$ and $\mathcal{I} \models \{\neg A, \neg B_{k+1}, \ldots, \neg B_m\}$. The truth value of the B_j is fixed by $\mathcal{B}$, i.e., $\mathcal{B} \models B_j$ or $\mathcal{B} \models \neg B_j$, and $\mathcal{I}_i \models B_j$ if and only if $\mathcal{B} \models B_j$. Therefore $\mathcal{I}_i \models \{A_1, \ldots, A_n, B_1, \ldots, B_k, \neg B_{k+1}, \ldots, \neg B_m\}$ for all $i \in I$. So there must be $j \in I$ such that $\mathcal{I}_j \models \neg A$. But then $\mathcal{I}_j \not\models C$, a contradiction. □

Clausal Base Theories. While Propositions 4.28 and 4.30 fit nicely into the hierarchic superposition framework, they are also applicable to hierarchic specifications where the base theory is axiomatized by a set of first-order clauses. It is well-known that the class

of Herbrand models of a Horn clause set is closed under intersection and hence has, if any, a unique minimal element (see for example [BG91a]). Proposition 4.28 thus applies to hierarchic specifications with Horn base theories. Similarly, *predicate completion* techniques [Cla77, CN00, Hor10] can be applied to obtain clause sets with unique Herbrand models, making also Proposition 4.30 applicable.

We will show that for hierarchic specifications of the form $((\Sigma_b, N_b), (\Sigma_e, N_e))$, the Herbrand models of the clause set $N_b \cup N_e$ are precisely the hierarchic Herbrand models of N_e with respect to the base theory consisting of the Herbrand models of N_b, under the condition that N_e is sufficiently complete. In this case, standard superposition can be used for hierarchic reasoning, instead of SUP(T).

Proposition 4.31
Let $((\Sigma_b, N_b), (\Sigma_e, N_e))$ be a hierarchic specification where N_b and N_e are Horn, and N_e is sufficiently complete. Then, writing $\mathcal{T}$ for (Σ_b, N_b) and Σ for $\Sigma_b \cup \Sigma_e$, it holds that $\mathrm{Mod}_\Sigma(N_b \cup N_e) = \mathrm{Mod}_{\mathcal{T},\Sigma}(N_e)$.

Proof. Let $\Sigma = \Sigma_b \cup \Sigma_e$ and $N = N_b \cup N_e$.

$\supseteq$: Let $\mathcal{I} \in \mathrm{Mod}_{\mathcal{T},\Sigma}(N_e)$. By Lemma 4.26), we have $\mathcal{I} \models \mathrm{sgi}(N_e) \cup E_\mathcal{B} \cup D_\mathcal{B}$ for some $\mathcal{B} \in \mathrm{Mod}_{\Sigma_b}(N_b)$. By sufficient completeness of N_e, $\mathcal{I} \models \mathrm{sgi}(N_e)$ implies $\mathcal{I} \models \mathrm{gnd}(N_e)$ and hence $\mathcal{I} \models N_e$. From $\mathcal{I} \models E_\mathcal{B} \cup D_\mathcal{B}$ it follows that $\mathcal{I} \models \mathrm{gnd}(N_b)$ and hence $\mathcal{I} \models N_b$. Thus $\mathcal{I} \in \mathrm{Mod}_\Sigma(N)$.

$\subseteq$: Let $\mathcal{I} \in \mathrm{Mod}_\Sigma(N)$. We have $\mathcal{I} \models \mathrm{gnd}(N_b) \cup \mathrm{gnd}(N_e)$. From $\mathrm{sgi}(N_e) \subseteq \mathrm{gnd}(N_e)$ it follows that $\mathcal{I} \models \mathrm{sgi}(N_e)$. Since $\mathcal{I} \models \mathrm{gnd}(N_b)$, there is a $\mathcal{B} \in \mathrm{Mod}_{\Sigma_b}(N_b)$ such that $\mathcal{I} \models E_\mathcal{B} \cup D_\mathcal{B}$. Thus we get $\mathcal{I} \in \mathrm{Mod}_{\mathcal{T},\Sigma}(N_e)$ by Lemma 4.26). □

The following trivial example shows how the hierarchic model property is lost if N_e is not sufficiently complete: Let $\Sigma_b = \{a, b\}$, $N_b = \{a \not\simeq b\}$ and $\Sigma_e = \{c\}$, $N_e = \emptyset$, where a, b, c are of the same sort. Clearly N_e is not sufficiently complete, because it contains no equation for c. Indeed, the Herbrand interpretation over $\Sigma_b \cup \Sigma_e$ in which c is different from both a and b is a model of $N_b \cup N_e$, but not a hierarchic model.

Together with Propositions 4.28 and 4.30, Proposition 4.31 gives us a recipe for constructing theories with unique minimal Herbrand models, using repeated hierarchic combination. Starting with a base theory $\mathcal{T}_0$ that is given either by a set of Herbrand models or by a set of clausal axioms, we extend it with sufficiently complete Horn theories (or theories Horn modulo the previous theory, in case that theory has a unique model), all the while maintaining the unique minimal model property, until the final extension (Σ_n, N_n). Then hierarchic reasoning can be performed by using $\mathrm{SUP}(\mathcal{T}_0)$ on $N_1 \cup \ldots \cup N_n$ (in the case where $\mathcal{T}_0$ is given by a set of models), or by using standard superposition on $N_0 \cup N_1 \cup \ldots \cup N_n$ (in the clausal case).

4.4 Reachability Theories

We now formally define reachability theories. Assume we have a theory $\mathcal{T}$ with a unique minimal Herbrand model, possibly as the result of repeated hierarchic combination, as outlined in the previous section. Let the signature Σ_R consist of a single predicate symbol R, the *reachability predicate*. We call any R-atom a *state atom*.

Definition 4.32 (Reachability Theories)
Let $\mathcal{T}$ be a theory with a unique minimal Herbrand model. Let $n \geq 0, m \geq 1$. We call a clause of the form

$$l_1, \ldots, l_n \to A$$

an *initial clause*, a clause of the form

$$A_1, \ldots, A_m, l_1, \ldots, l_n \to B$$

a *transition clause*, and a clause of the form

$$B_1, \ldots, B_m, l_1, \ldots, l_n \to$$

a *goal clause*, whenever the l_i are $\Sigma_{\mathcal{T}}$-literals, and A, A_i and B, B_i are state atoms. A transition clause with $m = 1$ is called *binary*, and a goal clause with $m = 1$ is called *unit*. A *generalized ($\mathcal{T}$-)reachability theory* is a clause set consisting of initial and transition clauses, and all l_i are positive if $\mathcal{T}$ has more than one Herbrand model. The theory $\mathcal{T}$ is also called the *background theory* of the generalized reachability theory. We call N a *($\mathcal{T}$-)reachability theory* if all transition clauses in N are binary. A *(generalized) reachability query* is a (generalized) reachability theory additionally containing one or more goal clauses (which may be non-unit only for a generalized reachability query). ■

Note that in the above definition of initial, transition and goal clauses, the l_i are *literals* and not atoms, unlike in the standard clause notation $\Gamma \to \Delta$ where Γ, Δ are multisets of atoms. We use this notation to convey the intuition that the l_i are to be seen as conditions, somewhat like the *body* of a program clause in logic programming. Of course, these formulas are still clauses in the standard sense, and they can be written in standard notation by removing the negation from the negative l_i and moving the resulting atoms to the succedent.

We now associate a transition system with each reachability theory. The idea is that the states are the interpretations of the arguments of state atoms with respect to the minimal Herbrand model of $\mathcal{T}$.

Definition 4.33 ($\mathsf{s}_{\mathcal{T}}$, $S_{\mathcal{T}}$)
Let $\mathcal{I}_{\mathcal{T}}$ be the minimal Herbrand model of $\mathcal{T}$. We define the operator $\mathsf{s}_{\mathcal{T}}$ as

$$\mathsf{s}_{\mathcal{T}}(\nu, R(t_1, \ldots, t_n)) = (\mathcal{I}_{\mathcal{T}}(\nu)(t_1), \ldots, \mathcal{I}_{\mathcal{T}}(\nu)(t_n))$$

and the set of *states* with respect to $\mathcal{T}$ as

$$S_{\mathcal{T}} = \{\mathsf{s}_{\mathcal{T}}(\nu, A) \mid A \text{ is a state atom and } \nu \text{ is an assignment for } \mathcal{I}_{\mathcal{T}}\}.$$ ■

Definition 4.34 (Transition System of a Reachability Theory)
Given a $\mathcal{T}$-reachability theory N, the *transition system associated with* N is defined as $\mathrm{TS}(N) = (S_\mathcal{T}, \to, S_0)$ where

$$\begin{aligned} \to &= \{\, (\mathsf{s}_\mathcal{T}(\nu, A), \mathsf{s}_\mathcal{T}(\nu, B)) \mid A, l_1, \ldots, l_n \to B \in N \text{ and} \\ &\qquad\qquad \mathcal{I}_\mathcal{T}, \nu \models \{l_1, \ldots, l_n\} \,\}, \\ S_0 &= \{\, \mathsf{s}_\mathcal{T}(\nu, A) \mid l_1, \ldots, l_n \to A \in N \text{ and } \mathcal{I}_\mathcal{T}, \nu \models \{l_1, \ldots, l_n\} \,\}. \end{aligned}$$

■

Definition 4.35 (Reachability Problem)
The *reachability problem* for a class of reachability theories $\mathcal{R}$ is the problem of deciding, given $N \in \mathcal{R}$ and a subset S of the states of $\mathrm{TS}(N)$, whether some $s \in S$ is reachable in $\mathrm{TS}(N)$. ■

Proposition 4.36
The reachability problem for arbitrary reachability theories is undecidable.

Proof. Post's Correspondence Problem (PCP) [Pos46] is straightforwardly represented as a reachability theory (with empty background theory): Consider sequences of words $(\alpha_i)_{1 \le i \le n}$ and $(\beta_i)_{1 \le i \le n}$ where $\alpha_i = a_{i1} a_{i2} \ldots a_{ik_i}$ and $\beta_i = b_{i1} b_{i2} \ldots b_{im_i}$. We assume a constant symbol ϵ and a unary function symbol for every symbol a_i, b_j in the PCP alphabet. The term $a_{ik_i}(\ldots a_{i2}(a_{i1}(x))\ldots)$, which we abbreviate by $\alpha_i(x)$, represents the concatenation of x with α_i (similarly for β_i). The reachability theory N_PCP contains the clauses $\to R(\alpha_i(\epsilon), \beta_i(\epsilon))$ and $R(x, y) \to R(\alpha_i(x), \beta_i(y))$ for all $i \in [1, n]$. Solving the PCP reduces to deciding whether any ground instance of (x, x) is reachable in $\mathrm{TS}(N_\mathrm{PCP})$. □

Proposition 4.37
If $\mathcal{T}$ has a unique minimal Herbrand model, then any generalized $\mathcal{T}$-reachability theory N also has a unique minimal Herbrand model.

Proof. As Σ_R contains only one predicate symbol, N is trivially sufficiently complete. Furthermore, N is either Horn, or Horn modulo $\mathcal{T}$, if $\mathcal{T}$ has a unique Herbrand model. The statement follows from Propositions 4.28 and 4.30 together with the fact that N is satisfiable whenever $\mathcal{T}$ is (a trivial model is obtained by interpreting all ground state atoms as true). □

There is a direct correspondence between subsets of $S_\mathcal{T}$ and hierarchic Herbrand interpretations over Σ: As in any hierarchic Herbrand interpretation over Σ, the restriction to $\Sigma_\mathcal{T}$ must be isomorphic to $\mathcal{I}_\mathcal{T}$, the minimal Herbrand model of $\mathcal{T}$ (actually, it must even be equal to $\mathcal{I}_\mathcal{T}$, because the enrichment contains only a predicate symbol), different hierarchic Σ-interpretations can only differ in their interpretation of the predicate symbol R. The bijection between subsets $S \subseteq S_\mathcal{T}$ and hierarchic Herbrand Σ-interpretations $\mathcal{I}$ is given by $\mathcal{I}(R)(a_1, \ldots, a_n) = \{\mathrm{true}_\mathrm{R}\}$ iff $(a_1, \ldots, a_n) \in S$.

Definition 4.38
Let N be a generalized $\mathcal{T}$-reachability theory. The mapping T_N on subsets of $S_\mathcal{T}$ is defined as follows:

$$\begin{aligned} T_N(S) = \{\, \mathsf{s}_\mathcal{T}(\nu, B) \mid\ & A_1, \ldots, A_m, l_1, \ldots, l_n \to B \in N, \\ & \text{where } m, n \geq 0 \text{ and} \\ & \mathcal{I}_\mathcal{T}, \nu \models \{l_1, \ldots, l_n\}, \text{ and} \\ & \mathsf{s}_\mathcal{T}(\nu, A_i) \in S \text{ for all } i \in [1, m] \,\}. \end{aligned}$$

■

The following Propositions 4.39, 4.40 and 4.41 are generalizations of well-known results for Horn clause sets and their *immediate consequence operator* (see for example Propositions 6.3, 6.4 and Theorem 6.5, respectively, from [Llo93]).

Proposition 4.39
For any generalized $\mathcal{T}$-reachability theory N, the mapping T_N is continuous.

Proof. Let $\mathcal{S}$ be a directed subset of $\mathfrak{P}(S_\mathcal{T})$. Now we have that $s \in T_N(\mathrm{lub}(\mathcal{S}))$ iff there is $A_1, \ldots, A_m, l_1, \ldots, l_n \to B \in N$ with $m, n \geq 0$, such that $s = \mathsf{s}_\mathcal{T}(\nu, B)$ and $\mathcal{I}_\mathcal{T}, \nu \models \{l_1, \ldots, l_n\}$ and

$$\{\mathsf{s}_\mathcal{T}(\nu, A_1), \ldots, \mathsf{s}_\mathcal{T}(\nu, A_m)\} \subseteq \mathrm{lub}(\mathcal{S}), \tag{4.1}$$

for some assignment ν. Now, as $\mathcal{S}$ is directed, (4.1) is equivalent to

$$\{\mathsf{s}_\mathcal{T}(\nu, A_1), \ldots, \mathsf{s}_\mathcal{T}(\nu, A_m)\} \subseteq S,$$

for some $S \in \mathcal{S}$. Hence $s \in T_N(\mathrm{lub}(\mathcal{S}))$ iff $s \in T_N(S)$ iff $s \in \mathrm{lub}(T_N(\mathcal{S}))$. Thus $T_N(\mathrm{lub}(\mathcal{S})) = \mathrm{lub}(T_N(\mathcal{S}))$. □

Proposition 4.40
Let N be a generalized $\mathcal{T}$-reachability theory and $S \subseteq S_\mathcal{T}$ a set of states. Then S defines a hierarchic model of N if and only if $T_N(S) \subseteq S$.

Proof. Let $C = A_1, \ldots, A_m, l_1, \ldots, l_n \to B$ with $m, n \geq 0$ be an arbitrary clause in N. S satisfies C iff for any assignment ν with $\mathcal{I}_\mathcal{T}, \nu \models \{l_1, \ldots, l_n\}$, it holds that $\mathsf{s}_\mathcal{T}(\nu, B) \in S$ whenever $\{\mathsf{s}_\mathcal{T}(\nu, A_1), \ldots, \mathsf{s}_\mathcal{T}(\nu, A_m)\} \subseteq S$. This condition is equivalent to $T_N(S) \subseteq S$. □

Proposition 4.41
Let N be a generalized $\mathcal{T}$-reachability theory. Then $\mathcal{I}_N = \mathrm{lfp}(T_N) = T_N^\omega$, up to isomorphism.

Proof. As $\mathcal{I}_N$ is the minimal Herbrand model of N, it is the intersection of all Herbrand models of N, or, equivalently, the greatest lower bound of those models with respect to the complete lattice of all $\mathcal{T}$-hierarchic Herbrand interpretations over Σ. $\mathcal{I}_N = \mathrm{glb}\{S \subseteq S_\mathcal{T} \mid S \in \mathrm{Mod}_{\mathcal{T},\Sigma}(N)\}$ (here we use the fact that there is a bijection

between $\mathcal{T}$-hierarchic Herbrand interpretations over Σ and sets of states). By Proposition 4.40, this is equivalent to $\mathcal{I}_N = \mathrm{glb}\{S \subseteq S_\mathcal{T} \mid T_N(S) \subseteq S\}$, from which we get $\mathcal{I}_N = \mathrm{lfp}(T_N)$ by Proposition 4.8. Finally, applying Propositions 4.10 and 4.39, we get $\mathcal{I}_N = T_N^\omega$. □

The following proposition establishes the connection between the notions of minimal model of a reachability theory and the set of reachable states of the associated transition system.

Proposition 4.42
Let N be a $\mathcal{T}$-reachability theory, and let $\mathrm{TS}(N) = (S_\mathcal{T}, \to, S_0)$. Then $\mathcal{I}_N = \mathrm{Post}^*_{\mathrm{TS}(N)}(S_0)$.

Proof. Writing Post for $\mathrm{Post}_{\mathrm{TS}(N)}$, we first show $T_N^n = \bigcup_{i=1}^n \mathrm{Post}^{i-1}(S_0)$ by induction on n. For $n = 1$ we have $T_N^1 = T_N(\emptyset) = S_0 = \mathrm{Post}^0(S_0)$. For $n > 1$, we have

$$\begin{aligned} T_N^n &= T_N(T_N^{n-1}) \\ &= T\left(\textstyle\bigcup_{i=1}^{n-1} \mathrm{Post}^{i-1}(S_0)\right) \\ &= S_0 \cup \mathrm{Post}\left(\textstyle\bigcup_{i=1}^{n-1} \mathrm{Post}^{i-1}(S_0)\right) \\ &= S_0 \cup \textstyle\bigcup_{i=2}^{n} \mathrm{Post}^{i-1}(S_0) \\ &= \textstyle\bigcup_{i=1}^{n} \mathrm{Post}^{i-1}(S_0)\ . \end{aligned}$$

Consequently,

$$\begin{aligned} \mathcal{I}_N = T_N^\omega &= \textstyle\bigcup_{n\in\mathbb{N}} T_N^n \\ &= \textstyle\bigcup_{n\in\mathbb{N}} \bigcup_{i=1}^{n} \mathrm{Post}^{i-1}(S_0) \\ &= \textstyle\bigcup_{n\in\mathbb{N}} \mathrm{Post}^n(S_0) \\ &= \mathrm{Post}^*(S_0)\ . \end{aligned}$$

□

The following proposition generalizes a well-known result about arbitrary models of clause sets to hierarchic models.

Proposition 4.43
Let N be a clause set over Σ that is an extension of a base theory $\mathcal{T}$. If N has a hierarchic model, then N has a hierarchic Herbrand model.

Proof. Let $\mathcal{I} \in \mathrm{Mod}_\mathcal{T}(N)$. We construct $\mathcal{I}' \in \mathrm{Mod}_{\mathcal{T},\Sigma}(N)$. We can assume without loss of generality that $\mathcal{I}$ is a normal model, hence for any ground Σ-terms s, t, we have $\mathcal{I} \models s \simeq t$ iff $\mathcal{I}(s) = \mathcal{I}(t)$. By assumption, $\mathcal{I}|_{\Sigma_\mathcal{T}}$ is isomorphic to a Herbrand interpretation $\mathcal{B} \in \mathscr{C}_\mathcal{T}$, and $\mathcal{I} \models \mathrm{gnd}(N)$. Let $\cong_\mathcal{I}$ be the relation on ground Σ-terms defined by $s \cong_\mathcal{I} t$ iff $\mathcal{I}(s) = \mathcal{I}(t)$. Clearly, $\cong_\mathcal{I}$ is a congruence. Let $\mathcal{I}'$ be the Herbrand interpretation defined by $\cong_\mathcal{I}$. It is straightforward to show that also $\mathcal{I}'|_{\Sigma_\mathcal{T}}$ is isomorphic to $\mathcal{B}$ and $\mathcal{I}' \models \mathrm{gnd}(N)$. Hence $\mathcal{I}' \in \mathrm{Mod}_{\mathcal{T},\Sigma}(N)$. □

Corollary 4.44
Let N be a clause set over Σ that is an extension of a base theory $\mathcal{T}$. N is $\mathcal{T}$-unsatisfiable if and only if N has no hierarchic Herbrand model.

Proof. The "only if" direction is trivial, as $\mathrm{Mod}_{\mathcal{T},\Sigma}(N) \subseteq \mathrm{Mod}_{\mathcal{T}}(N)$. The "if" direction follows immediately from Proposition 4.43. □

Proposition 4.45
Let N be a clause set over Σ that is an extension of a base theory $\mathcal{T}$. Let ϕ be a quantifier-free formula over Σ. Then $N \models_{\mathcal{T}} \exists\phi$ if and only if $N \models_{\mathcal{T},\Sigma} \exists\phi$.

Proof.

$$
\begin{array}{lll}
 & N \models_{\mathcal{T}} \exists\phi & \\
\Leftrightarrow & \mathcal{I} \models \exists\phi \text{ for every } \mathcal{I} \in \mathrm{Mod}_{\mathcal{T}}(N) & \text{(by definition of } \models_{\mathcal{T}}) \\
\Leftrightarrow & \mathrm{Mod}_{\mathcal{T}}(N \cup \{\forall\neg\phi\}) = \emptyset & \\
\Leftrightarrow & \mathrm{Mod}_{\mathcal{T},\Sigma}(N \cup \{\forall\neg\phi\}) = \emptyset & \text{(by Corollary 4.44)} \\
\Leftrightarrow & \mathcal{I} \not\models \forall\neg\phi \text{ for every } \mathcal{I} \in \mathrm{Mod}_{\mathcal{T},\Sigma}(N) & \\
\Leftrightarrow & \mathcal{I} \models \exists\phi \text{ for every } \mathcal{I} \in \mathrm{Mod}_{\mathcal{T},\Sigma}(N) & \\
\Leftrightarrow & N \models_{\mathcal{T},\Sigma} \exists\phi &
\end{array}
$$

We can apply Corollary 4.44 in the third step because $\forall\neg\phi$ is logically equivalent to a set of Σ-clauses. □

Proposition 4.46
Let $\mathcal{I}, \mathcal{J}$ be two Herbrand interpretations over Σ such that $\mathcal{I} \subseteq \mathcal{J}$, and $\mathcal{I} \models \exists\phi$ where ϕ is a conjunction of positive literals. Then $\mathcal{J} \models \exists\phi$.

Proof. We assume without loss of generality that ϕ consists of a single equation $s \simeq t$ containing one variable x. $\mathcal{I} \models \exists x.s \simeq t$ iff there exists a in $U_{\mathcal{I}}$ (of the same sort as x) such that $\mathcal{I}, [x \mapsto a] \models s \simeq t$. Let σ be a substitution that maps x to any ground term t' such that $\mathcal{I}(t') = a$ (t' necessarily exists since $\mathcal{I}$ is term-generated). Then $s\sigma \simeq t\sigma \in \mathcal{I}$, hence also $s\sigma \simeq t\sigma \in \mathcal{J}$ and thus $\mathcal{J} \models \exists x.s \simeq t$. □

Theorem 4.47
Let N be a $\mathcal{T}$-reachability theory, and let $l_1, \ldots, l_n, A \rightarrow$ be a unit goal clause. Then $N \models_{\mathcal{T}} \exists(l_1 \wedge \ldots \wedge l_n \wedge A)$ if and only if $\mathsf{s}_{\mathcal{T}}(\nu, A)$ is a reachable state of $\mathrm{TS}(N)$, for some assignment ν with $\mathcal{I}_{\mathcal{T}}, \nu \models \{l_1, \ldots, l_n\}$.

Proof. Let $\phi = \exists(l_1 \wedge \ldots \wedge l_n \wedge A)$. By Proposition 4.45, $N \models_{\mathcal{T}} \phi$ is equivalent to $N \models_{\mathcal{T},\Sigma} \phi$. Now we distinguish two cases:

(i) All l_i are positive. Then by Proposition 4.46, $N \models_{\mathcal{T},\Sigma} \phi$ is equivalent to $\mathcal{I}_N \models \phi$.

(ii) Some l_i are negative, and $\mathcal{T}$ has a unique Herbrand model. Then $N \models_{\mathcal{T},\Sigma} \phi$ is equivalent to $\mathcal{I}_{\mathcal{T}}, \nu \models \{l_1, \ldots, l_n\}$ and $\mathsf{s}_{\mathcal{T}}(\nu, A) \in I_N$, for some assignment ν.

In both cases, we conclude $\mathcal{I}_T(\nu)(A) \in \mathcal{I}_N$, for some assignment ν, and the statement follows from Proposition 4.42. □

From Theorem 4.47, it follows that any complete method for computing $\models_{\mathcal{T}}$, like superposition or superposition modulo $\mathcal{T}$, can be used to analyze reachability in $\mathcal{T}$-reachability theories.

Proposition 4.48
Let $((\Sigma_b, N_b), (\Sigma_e, N_e))$ be a hierarchic specification where N_b and N_e are Horn, and N_e is sufficiently complete. Let ϕ be a quantifier-free formula over $\Sigma = \Sigma_b \cup \Sigma_e$. Then, writing $\mathcal{T}$ for (Σ_b, N_b), it holds that $N_e \models_{\mathcal{T}} \exists\phi$ if and only if $N_b \cup N_e \cup \{\forall\neg\phi\} \models \bot$.

Proof. By Proposition 4.46, $N_e \models_{\mathcal{T}} \exists\phi$ is equivalent to $N_e \models_{\mathcal{T},\Sigma} \exists\phi$. Then we have

$$
\begin{array}{ll}
& N_e \models_{\mathcal{T},\Sigma} \exists\phi \\
\Leftrightarrow & \mathcal{I} \models \exists\phi \text{ for every } \mathcal{I} \in \mathrm{Mod}_{\mathcal{T},\Sigma}(N) \\
\Leftrightarrow & \mathcal{I} \models \exists\phi \text{ for every } \mathcal{I} \in \mathrm{Mod}_{\Sigma}(N_b \cup N_e) \quad \text{(by Proposition 4.31)} \\
\Leftrightarrow & \mathrm{Mod}_{\Sigma}(N_b \cup N_e \cup \{\forall\neg\phi\}) = \emptyset \\
\Leftrightarrow & \mathrm{Mod}(N_b \cup N_e \cup \{\forall\neg\phi\}) = \emptyset \quad \text{(since } N_b \cup N_e \cup \{\forall\neg\phi\} \text{ is a clause set)} \\
\Leftrightarrow & N_b \cup N_e \cup \{\forall\neg\phi\} \models \bot
\end{array}
$$

□

4.4.1 Forward and Backward Encodings

We say that a $\mathcal{T}$-reachability theory *represents* a transition system T if $\mathrm{TS}(N) = T$. In that case, we also say that N is a *forward encoding* of T. Given a reachability query based on a forward encoding, there also exists a corresponding *backward encoding*, which is obtained by reversing the polarities of all state literals. The backward encoding $N^\flat$ represents a transition system $\mathrm{TS}(N^\flat)$ in which the transition relation is the inverse of the one of $\mathrm{TS}(N)$, and the states represented by the initial and goal clauses are exchanged.

Definition 4.49 (Backward Encoding)
Let N be a set of initial clauses, binary transition clauses and unit goal clauses. Then $N^\flat$ is the set

$$
\begin{array}{rcl}
N^\flat = & \{\ A, l_1, \ldots, l_n \to & \mid \quad l_1, \ldots, l_n \to A \in N\ \} \cup \\
& \{\ B, l_1, \ldots, l_n \to A & \mid\ A, l_1, \ldots, l_n \to B \in N\ \} \cup \\
& \{\quad l_1, \ldots, l_n \to B & \mid\ B, l_1, \ldots, l_n \to \quad \in N\ \}.
\end{array}
$$

■

Let $\mathrm{TS}(N) = (S_{\mathcal{T}}, \to, S_0)$ and $\mathrm{TS}(N^\flat) = (S_{\mathcal{T}}, \to^\flat, S_0^\flat)$, and let

$$S_G \ = \ \{\, \mathsf{s}_{\mathcal{T}}(\nu, B) \mid B, l_1, \ldots, l_n \to \in N \text{ and } \mathcal{I}_{\mathcal{T}}, \nu \models \{l_1, \ldots, l_n\}\,\}.$$

It follows immediately from Definition 4.32 that

$$\to^\flat = \to^{-1} \quad \text{and} \quad S_0^\flat = S_G$$

and thus

$$\mathrm{Post}_{\mathrm{TS}(N^\flat)} \;=\; \mathrm{Pre}_{\mathrm{TS}(N)}\,.$$

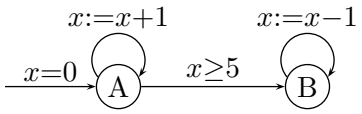
Applying Proposition 4.42 finally yields

$$\mathcal{I}_{N^\flat} \;=\; \mathrm{Post}^*_{\mathrm{TS}(N^\flat)}(S_0^\flat) \;=\; \mathrm{Pre}^*_{\mathrm{TS}(N)}(S_G).$$

Thus the minimal Herbrand model of the backward encoding is the set of states that can reach a goal state, also called backward-reachable from the goal states.

The superposition calculus can always be forced to perform either backward or forward traversal of the transition system's state space, by using a selection strategy that selects all negative state literals. Under the forward encoding, this yields forward traversal, and under the backward encoding, it yields backward traversal.

Example 4.50
Consider the following automaton:[7]



The reachability queries based on the forward and backward encodings, respectively, are given by

$$\begin{array}{rl} N: & \\ & \to A(0) \\ \underline{A(x)} & \to A(x+1) \\ \underline{A(x)}, x \geq 5 & \to B(x) \\ \underline{B(x)} & \to B(x-1) \\ \underline{B(3)} & \to \end{array} \qquad \begin{array}{rl} N^\flat: & \\ \underline{A(0)} & \to \\ \underline{A(x+1)} & \to A(x) \\ \underline{B(x)}, x \geq 5 & \to A(x) \\ \underline{B(x-1)} & \to B(x) \\ & \to B(3) \end{array}$$

where selected literals are underlined. The clauses $B(3) \to$ and $\to B(3)$ are the goal clauses. ∎

4.5 SUP(LA) for Timed Systems

In this section, we apply the concepts of hierarchic superposition (Section 4.3) and reachability theories (Section 4.4) to the analysis of timed automata and extended timed automata—timed automata with additional, unbounded integer variables. We

[7] See Section 4.6.4 for further examples.

will see that SUP(LA) constitutes a decision procedure for reachability in timed automata (Section 4.5.6). In the Section 4.6, we will augment the SUP(LA) calculus with an automatic induction rule, generalizing the latter result to extended timed automata (Section 4.6.3).

As we will focus on FOL(LA)—the hierarchic combination of first-order logic with linear arithmetic—we now introduce a few concepts that we will rely on in the rest of this section and in Section 4.6.

The base theory LA has a unique model, with universe $\mathbb{R}$, and a corresponding base sort $\mathbb{R}$. In the context of extended timed automata, we will also deal with integer variables, of sort $\mathbb{Z}$. While it would be desirable to treat $\mathbb{Z}$ as a subsort of $\mathbb{R}$, a formal integration of subsorts into the SUP(LA) framework has not yet been developed, and is beyond the scope of this work. We therefore stay in the multisorted framework introduced in Section 2.2, and treat $\mathbb{Z}$ and $\mathbb{R}$ as different sorts, while overloading all base operators like addition, subtraction and multiplication with constants to handle both reals and integers. For instance, the addition ($+$) symbol stands for five functions of sorts $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, $\mathbb{Z} \times \mathbb{Z} \to \mathbb{R}$, $\mathbb{R} \times \mathbb{Z} \to \mathbb{R}$, $\mathbb{Z} \times \mathbb{R} \to \mathbb{R}$ and $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$, respectively. Since unifiers in SUP(LA) only ever map base variables to base variables, this ensures that any term has a unique sort.

4.5.1 Timed Automata

Timed Automata were originally introduced in [AD94]. Our presentation of timed automata is partly based on [BK08].

To streamline the definitions, we define *instructions*, which are substitutions whose effect on clocks, if any, is to reset them to zero.

Definition 4.51 (Instructions)
Let $\mathcal{T}$ be an extension of $\mathbb{R}$, and let $X \subseteq \mathcal{X}$ be a set of variables with a subset $X_C \subseteq X$ of real-valued clock variables. An *instruction* for $\mathcal{T}$ and X is a substitution $A : X \to T_{\mathcal{T}}(\mathcal{X})$ such that $A(x) \in \{0, x\}$ if $x \in X_C$. The set of all instructions for $\mathcal{T}$ and X is denoted by $\mathit{Instr}_{\mathcal{T}}(X)$. Given an instruction A and $\vec{x} = x_1, \ldots, x_n$, we write $\vec{x}' \simeq A(\vec{x})$ for $x_1' \simeq A(x_1) \wedge \ldots \wedge x_n' \simeq A(x_n)$. Given an assignment ν for $\mathcal{T}$, we write $A(\nu)$ for the assignment mapping x to $\nu(A(x))$. ∎

Definition 4.52 (Clock Constraints)
A *clock constraint* over a set C of clocks is formed according to the grammar

$$g ::= \; x \circ c \;\mid\; x - y \circ c \;\mid\; g \wedge g$$

where $x, y \in C$, $c \in \mathbb{N}$ and $\circ \in \{<, \leq, \geq, >\}$.[8] The set of all clock constraints over C is denoted by $CC(C)$. An *atomic clock constraint* is a clock constraint containing no conjunction.

[8] We abbreviate $x \geq c \wedge x \leq c$ by $x \simeq c$.

We also use the notation $CC(X, Y)$ to denote the set of clock constraints built over atomic constraints with $x, y \in X$ and $c \in Y$. Thus $CC(C) = CC(C, \mathbb{N})$.[9] ∎

Definition 4.53 (Timed Automata)
A *timed automaton* is a tuple $(Loc, Act, C, \hookrightarrow, L_0, Inv)$ where Loc is a finite set of locations, with initial location $L_0 \in Loc$, Act is a finite set of actions, C is a finite set of clocks,

$$\hookrightarrow \subseteq Loc \times CC(C) \times Act \times \mathfrak{P}(C) \times Loc$$

is a transition relation[10], and $Inv : Loc \to CC(C)$ is a function assigning invariants to locations. ∎

For convenience, we write $L \overset{g:\alpha,D}{\hookrightarrow} L'$ for $(L, g, \alpha, D, L') \in \hookrightarrow$, and $L \overset{g,D}{\hookrightarrow} L'$ if $L \overset{g:\alpha,D}{\hookrightarrow} L'$ for some $\alpha \in Act$.

We assume that each state variable in an automaton—TA or ETA (see Section 4.5.3)—has an associated sort declaration $x : S$, where S is either $\mathbb{R}$ (for TA and ETA) or $\mathbb{Z}$ (for ETA only).

Definition 4.54 (Semantics of Timed Automata)
Let $TA = (Loc, Act, C, \hookrightarrow, L_0, Inv)$ be a timed automaton. The semantics of TA is given by the transition system $\mathrm{TS}(TA) = (S, Act', \to, s_0)$ with

- $S = Loc \times \{\nu \in Val(C) \mid \nu(x) \geq 0 \text{ for all } x \in C\}$
- $Act' = Act \cup \mathbb{R}_{\geq 0}$
- $s_0 = (L_0, \{\vec{x} \mapsto 0\})$
- the transition relation $\to$ is defined by the following two rules:
  - (i) discrete transitions: $(L, \nu) \xrightarrow{\alpha} (L', \nu')$ iff $L \overset{g:\alpha,D}{\hookrightarrow} L'$ and $\nu \models g$ and $\nu' = \nu[D \mapsto 0]$ and $\nu' \models Inv(L')$
  - (ii) delay transitions: $(L, \nu) \xrightarrow{\delta} (L, \nu + \delta)$ iff $\nu + \delta \models Inv(L)$ for $\delta \in \mathbb{R}_{\geq 0}$.

We write $\xrightarrow{\mathsf{disc}}$ for $\bigcup_{\alpha \in Act} \xrightarrow{\alpha}$ and $\xrightarrow{\mathsf{time}}$ for $\bigcup_{\delta \in \mathbb{R}_{\geq 0}} \xrightarrow{\delta}$. ∎

Clearly $\to = \xrightarrow{\mathsf{disc}} \cup \xrightarrow{\mathsf{time}}$.

Remark 4.55
Condition (ii) of Definition 4.54 exploits the convexity of the invariants, which holds because they are clock constraints. For non-convex invariants, one would have to require $\nu + \delta' \models Inv(L)$ to hold for all $0 \leq \delta' \leq \delta$. The fact that invariants are convex also allows us to give a sufficiently complete encoding of TA into FOL(LA), as the additional quantification over δ' would cause the introduction of a Skolem function ranging into the reals. The same applies to the encoding of ETA (Definition 4.61) and FPTA (Definition 4.117). ∎

[9] The notation will be used in Sections 4.5.5 and 4.7.
[10] $\mathfrak{P}(C)$ denotes the powerset of C.

Definition 4.56 (Parallel Composition of Timed Automata)
Let $TA_i = (Loc_i, Act_i, C_i, \hookrightarrow_i, L_{0,i}, Inv_i)$, $i = 1, 2$ be timed automata with $C_1 \cap C_2 = \emptyset$. Let $H = Act_1 \cap Act_2$. The *parallel composition* of TA_1, TA_2 is the timed automaton

$$TA_1 \parallel TA_2 = (Loc_1 \times Loc_2, Act_1 \cup Act_2, C_1 \cup C_2, \hookrightarrow, (L_{0,1}, L_{0,2}), Inv)$$

where $Inv(L_1, L_2) = Inv_1(L_1) \wedge Inv_2(L_2)$, and $(L_1, L_2) \overset{g:\alpha,D}{\hookrightarrow} (L_1', L_2')$ if and only if

(i) $\alpha \in H$ and $L_1 \overset{g_1:\alpha,D_1}{\hookrightarrow} L_1'$ and $L_2 \overset{g_2:\alpha,D_2}{\hookrightarrow} L_2'$ and $g = g_1 \wedge g_2$ and $D = D_1 \cup D_2$, or

(ii) $\alpha \notin H$ and $L_1 \overset{g:\alpha,D}{\hookrightarrow} L_1'$ and $L_2' = L_2$, or

(iii) $\alpha \notin H$ and $L_1' = L_1$ and $L_2 \overset{g:\alpha,D}{\hookrightarrow} L_2'$. ∎

An easy consequence of the above definitions is that

$$\mathrm{TS}(TA_1) \parallel \mathrm{TS}(TA_2) = \mathrm{TS}(TA_1 \parallel TA_2)$$

holds up to isomorphism, whenever T_1, T_2 have disjoint sets of clocks [BK08].

4.5.2 Reachability Theories for Timed Automata

Definition 4.57 (Reachability Theory for Timed Automata)
Let $TA = (Loc, Act, C, \hookrightarrow, L_0, Inv)$ be a timed automaton. Let $\mathcal{T}_{TA}$ be the extension of $\mathbb{R}$ with a new sort S_{Loc} and constant symbols $\{L | L \in Loc\}$ of sort S_{Loc}. The reachability theory for TA, denoted by N_{TA}, is the $\mathcal{T}_{TA}$-reachability theory consisting of the following clauses:[11]

$$\vec{x} \simeq 0 \;\rightarrow\; R(L_0, \vec{x})$$

$$R(L, \vec{x}),\ t \geq 0,\ \vec{x} \geq 0,\ \vec{x}' \simeq \vec{x}+t,\ Inv(L)[\vec{x}'/\vec{x}] \;\rightarrow\; R(L, \vec{x}') \quad \text{for all } L \in Loc$$

$$R(L, \vec{x}),\ g[\vec{x}],\ \vec{x}' \simeq A_D(\vec{x}),\ Inv(L')[\vec{x}'/\vec{x}] \;\rightarrow\; R(L', \vec{x}') \quad \text{for all } L \overset{g,D}{\hookrightarrow} L'$$

where $\vec{x}$ contains all variables in C, and $\vec{x}'$ contains the corresponding variables in C', and A_D is the instruction defined by $A_D(x) = 0$ if $x \in D$ and $A_D(x) = x$ otherwise. We call clauses of the second and third type *time-step clauses* and *discrete-step clauses*, respectively. ∎

Proposition 4.58 (Adequacy of the Encoding)
Let $TA = (Loc, Act, C, \hookrightarrow, L_0, Inv)$ be a timed automaton. Then $\mathrm{TS}(N_{TA}) = \mathrm{TS}(TA)$.

[11] See Remark 4.76 on page 93 about the constraint $\vec{x} \geq 0$ in the second clause.

Proof. First observe that the background theory $\mathcal{T}_{TA}$ has a unique Herbrand model $\mathcal{I}_{\mathcal{T}_{TA}}$, as it is the combination of two signature-disjoint theories with unique Herbrand models.[12] Let $\mathrm{TS}(TA) = (S, Act', \to, s_0)$ and $\mathrm{TS}(N_{TA}) = (S_{\mathcal{T}_{TA}}, \to_N, s_{0,N})$. First observe that $(L, \nu) = \mathsf{s}_{\mathcal{T}}(\nu, R(L, \vec{x}))$ for any $L \in Loc$ and assignment ν from C into $\mathbb{R}_{\geq 0}$, and thus $S = S_{\mathcal{T}_{TA}}$. Furthermore, we have

$$s_0 = (L_0, \{\vec{x} \mapsto 0\}) = \mathsf{s}_{\mathcal{T}}(\{\vec{x} \mapsto 0\}, R(L_0, \vec{x})) = s_{0,N}.$$

It remains to show $\to = \to_N$. Let $\xrightarrow{\mathsf{disc}}_N$, $\xrightarrow{\mathsf{time}}_N \subseteq \to_N$ be the relations induced by the discrete-step clauses and the time-step clauses of N_{TA}, respectively. Clearly $\to_N = \xrightarrow{\mathsf{disc}}_N \cup \xrightarrow{\mathsf{time}}_N$. Let $\mathcal{X}$ be a set of real-sorted variables such that $C \cup C' \cup \{t\} \subseteq \mathcal{X}$, and let $\tilde{\nu}$ be an assignment from $\mathcal{X}$ into $\mathbb{R}_{\geq 0}$, let $L, L' \in Loc$ and $s = (L, \tilde{\nu}(\vec{x})) = \mathsf{s}_{\mathcal{T}}(\tilde{\nu}, R(L, \vec{x}))$, $s' = (L', \tilde{\nu}(\vec{x}')) = \mathsf{s}_{\mathcal{T}}(\tilde{\nu}, R(L', \vec{x}'))$. Then

$$\begin{aligned}
s \xrightarrow{\mathsf{disc}} s' \;\text{ iff }\; & L \overset{g,D}{\hookrightarrow} L' \text{ and } \tilde{\nu} \models g[\vec{x}],\ \tilde{\nu} = \tilde{\nu}[D \mapsto 0],\ \tilde{\nu} \models Inv(L')[\vec{x}'/\vec{x}] \\
\text{iff }\; & R(L, \vec{x}),\, g[\vec{x}],\, \vec{x}' \simeq A_D(\vec{x}),\, Inv(L')[\vec{x}'/\vec{x}] \;\to\; R(L', \vec{x}') \in N_{TA} \\
& \text{and } \tilde{\nu} \models \{g[\vec{x}],\, \vec{x}' \simeq A_D(\vec{x}),\, Inv(L')[\vec{x}'/\vec{x}]\} \\
\text{iff }\; & s \xrightarrow{\mathsf{disc}}_N s'
\end{aligned}$$

and, assuming $L = L'$ and $\tilde{\nu}(t) = \delta \in \mathbb{R}_{\geq 0}$,

$$\begin{aligned}
s \xrightarrow{\mathsf{time}} s' \;\text{ iff }\; & \delta \in \mathbb{R}_{\geq 0},\ \tilde{\nu}(\vec{x}') = \tilde{\nu}(\vec{x}) + \delta,\ \tilde{\nu} \models Inv(L)[\vec{x}'/\vec{x}] \\
\text{iff }\; & \tilde{\nu}(t) \geq 0,\ \tilde{\nu}(\vec{x}') = \tilde{\nu}(\vec{x}) + \tilde{\nu}(t),\ \tilde{\nu} \models Inv(L)[\vec{x}'/\vec{x}] \\
\text{iff }\; & R(L, \vec{x}),\, t \geq 0,\, \vec{x} \geq 0,\, \vec{x}' \simeq \vec{x}{+}t,\, Inv(L)[\vec{x}'/\vec{x}] \;\to\; R(L, \vec{x}') \in N_{TA} \\
& \text{and } \tilde{\nu} \models \{t \geq 0,\, \vec{x}' \simeq \vec{x} + t,\, Inv(L)[\vec{x}'/\vec{x}]\} \\
\text{iff }\; & s \xrightarrow{\mathsf{time}}_N s'.
\end{aligned}$$

□

4.5.3 Extended Timed Automata

In this section, we consider timed automata extended with unbounded integer variables and corresponding guards and assignments.

The designation "extended timed automata" has been applied in the literature to different concepts: To TA with diagonal constraints (i.e., atomic clock constraints with two variables, which we include by default) [BL10], to TA whose clocks can be updated to integral values other than zero [BHR06], and to various extensions of TA with additional data variables [Fri98] or annotations [NWY99, FPY02, BKST97].

[12] Strictly speaking, $\mathcal{T}_{TA}$ has more than one Herbrand model, because interpretations identifying constants in Loc are also models of it. This can be fixed by adding to the extension the axioms $L_i \not\simeq L_j$ for all $L_i, L_j \in Loc$. This is however not necessary here, since there are no negative atoms containing location symbols in the antecedent of any clause of N_{TA}.

Definition 4.59 (Linear Integer Constraints)
A *linear integer constraint* over a set X of integer-valued variables is a conjunction of one or more linear inequalities of the form $a_1x_1 + \ldots + a_nx_n \leq a_0$, where $x_i \in X$, $a_i \in \mathbb{Z}$. The set of all linear integer constraints over X is denoted by $LIC(X)$. ∎

Definition 4.60 (ETA Guards and Instructions)
Let X be a set of variables with subsets X_C, X_D of real- and integer-valued variables, respectively.

The set of *ETA-guards* over X is defined as $Guard_{\text{ETA}}(X) = CC(X_C) \times LIC(X_D)$.

The set $Instr_{\text{ETA}}(X) \subseteq Instr_{\text{LA}}(X)$ of *ETA-instructions* contains all instructions A such that $A(x) \in \{x + a \mid a \in \mathbb{Z}\}$ if $x \in X_D$. ∎

Definition 4.61 (Extended Timed Automaton)
An *extended timed automaton* is a tuple $(Loc, Act, X, \hookrightarrow, L_0, g_0, Inv)$ where Loc is a finite set of locations, with initial location $L_0 \in Loc$, Act is a finite set of actions, $X = X_C \uplus X_D$ is a finite set of clocks and integer variables, respectively,

$$\hookrightarrow \subseteq Loc \times Guard_{\text{ETA}}(X) \times Act \times Instr_{\text{ETA}}(X) \times Loc$$

is a transition relation, $g_0 \in LIC(X_D)$ is the initial condition, and $Inv : Loc \to CC(X_C)$ is a function assigning invariants to locations. ∎

For convenience, we write $L \overset{g:\alpha,A}{\hookrightarrow} L'$ for $(L, g, \alpha, A, L') \in \hookrightarrow$, and $L \overset{g,A}{\hookrightarrow} L'$ if $L \overset{g:\alpha,A}{\hookrightarrow} L'$ for some $\alpha \in Act$.

Definition 4.62 (Semantics of Extended Timed Automata)
Let $ETA = (Loc, Act, X, \hookrightarrow, L_0, g_0, Inv)$ be an extended timed automaton. The semantics of ETA is given by the transition system $\text{TS}(ETA) = (S, Act', \to, S_0)$ with

- $S = Loc \times \{\nu \in Val(X) \mid \nu(x) \geq 0 \text{ for all } x \in X_C\}$
- $Act' = Act \cup \mathbb{R}_{\geq 0}$
- $S_0 = \{(L_0, \nu) \mid \nu(x) = 0 \text{ for all } x \in X_C \text{ and } \nu \models g_0\}$
- the transition relation $\to$ is defined by the following two rules:
  - (i) discrete transitions: $(L, \nu) \overset{\alpha}{\to} (L', \nu')$ iff $L \overset{g:\alpha,A}{\hookrightarrow} L'$ and $\nu \models g$ and $\nu' = A(\nu)$ and $\nu' \models Inv(L')$
  - (ii) delay transitions: $(L, \nu) \overset{\delta}{\to} (L, \nu')$ iff $\nu'(x) = \nu(x) + \delta$ for all $x \in X_C$ and $\nu'(x) = \nu(x)$ for all $x \in X_D$ and $\nu' \models Inv(L)$ for $\delta \in \mathbb{R}_{\geq 0}$.

We write $\overset{\mathsf{disc}}{\longrightarrow}$ for $\bigcup_{\alpha \in Act} \overset{\alpha}{\to}$ and $\overset{\mathsf{time}}{\longrightarrow}$ for $\bigcup_{\delta \in \mathbb{R}_{\geq 0}} \overset{\delta}{\to}$. ∎

Remark 4.63
A linear integer constraint and the restriction of an instruction to X_D can always be written in matrix form as $\mathbf{A}\vec{x} \leq \vec{b}$ and $\vec{x} \mapsto \mathbf{D}\vec{x} + \vec{c}$, respectively, where $\mathbf{D}$ is a diagonal matrix over $\{0, 1\}$ and $\vec{x}$ are the variables in X_D. Furthermore, the sequential execution of two transitions can again be expressed in this form: Executing transitions with guards $\mathbf{A}_1\vec{x} \leq \vec{b}_1$ and $\mathbf{A}_2\vec{x} \leq \vec{b}_2$ and instructions $\vec{x} \mapsto \mathbf{D}_1\vec{x} + \vec{c}_1$ and $\vec{x} \mapsto \mathbf{D}_2\vec{x} + \vec{c}_2$, respectively, has the same effect as executing a transition with guard $\mathbf{A}_2\mathbf{D}_1\vec{x} \leq \vec{b}_2 - \mathbf{A}_2\vec{c}_1$ and instruction $\vec{x} \mapsto \mathbf{D}_2\mathbf{D}_1\vec{x} + \mathbf{D}_2\vec{c}_1 + \vec{c}_2$. We will exploit this fact in Section 4.6.3. ■

4.5.4 Reachability Theories for Extended Timed Automata

Definition 4.64 (Reachability Theory for Extended Timed Automata)
Let $ETA = (Loc, Act, X, \hookrightarrow, L_0, g_0, Inv)$ be an extended timed automaton. Let $\mathcal{T}_{ETA}$ be the extension of $\mathbb{R}$ with a new sort S_{Loc} and constant symbols $\{L | L \in Loc\}$ of sort S_{Loc}. The reachability theory for ETA, denoted by N_{ETA}, is the $\mathcal{T}_{ETA}$-reachability theory consisting of the following clauses:

$$\vec{x} \simeq 0,\ g_0[\vec{z}] \ \rightarrow\ R(L_0, \vec{x}, \vec{z})$$

$$R(L, \vec{x}, \vec{z}),\ t \geq 0,\ \vec{x} \geq 0,\ \vec{x}' \simeq \vec{x} + t,\ Inv(L)[\vec{x}'/\vec{x}] \ \rightarrow\ R(L, \vec{x}', \vec{z}) \qquad \text{for all } L \in Loc$$

$$R(L, \vec{x}, \vec{z}),\ g[\vec{x}, \vec{z}],\ (\vec{x}', \vec{z}') \simeq A(\vec{x}, \vec{z}),\ Inv(L')[\vec{x}'/\vec{x}] \ \rightarrow\ R(L', \vec{x}', \vec{z}') \quad \text{for all } L \overset{g,A}{\hookrightarrow} L'$$

where $\vec{x}, \vec{z}$ are the variables in X_C, X_D, and $\vec{x}', \vec{z}'$ are the corresponding variables in X'_C, X'_D, respectively. ■

Proposition 4.65 (Adequacy of the Encoding)
Let $ETA = (Loc, Act, X, \hookrightarrow, L_0, g_0, Inv)$ be an extended timed automaton. Then $\mathrm{TS}(N_{ETA}) = \mathrm{TS}(ETA)$.

Proof. The proof is analogous to the one of Proposition 4.58. □

4.5.5 Parametric Clock Constraints

In this section, we introduce parametric clock constraints, a generalization of clock constraints (Definition 4.52). We present the notion of c-closed clock constraints, which will enable us, in the next section (Section 4.5.6), to prove that SUP(LA) finitely saturates reachability queries for timed automata. We then generalize the notion of c-closedness to that of κ-closedness for parametric clock constraints. Together with constraint induction (Section 4.6), the properties of κ-closed parametric constraints will enable us to prove an analogous termination result for reachability theories of extended timed automata.

Definition 4.66 (Parametric Clock Constraints)
Let X, K be finite sets of real-valued clock variables and natural-valued variables called *parameters*, respectively. A *linear expression* over K is an expression of the form $a_0 + a_1k_1 + \ldots + a_mk_m$ with $a_1, \ldots, a_m \in \mathbb{Z}$ and $k_1, \ldots, k_m \in K$. The set of linear expressions over K is denoted by $LE(K)$. A *parametric clock constraint* g over X, K is formed according to the grammar

$$g ::= \; x \circ \kappa \;\mid\; x - y \circ \kappa \;\mid\; g \wedge g$$

where $x, y \in X$, $\circ \in \{<, \leq, \geq, >\}$, and $\kappa \in LE(K)$. A parametric clock constraint is *atomic* if it contains no conjunction. The set of all parametric clock constraints over X, K is denoted by $CC(X, LE(K))$. ∎

Parametric clock constraints, originally introduced in the context of parametric timed automata [AHV93], are a generalization of clock constraints (Definition 4.52), in the sense that $CC(X) \subseteq CC(X, LE(\emptyset)) = CC(X, \mathbb{Z})$. The other direction does not hold, because the constants in $CC(X)$ are assumed to be natural numbers, but are integers in $CC(X, \mathbb{Z})$. This makes no difference for two-variable constraints, since $x - y \leq a$ is equivalent to $y - x \geq -a$. It does however make a difference for one-variable constraints: For instance, $x \leq a$ with $a < 0$ is in $CC(X, \mathbb{Z})$, but not in $CC(X)$.

We now first focus on non-parametric constraints and define the notion of c-closed constraints (Definition 4.68). Later we will consider parametric constraints and generalize c-closedness to κ-closedness (Def 4.71).

c-closed Clock Constraints

The notion of c-equivalence was originally introduced in [Tri98]. The definition below extends the original one to handle negative constants.[13]

Definition 4.67 (c-equivalence)
Let $c \in \mathbb{N}$. Two clock valuations $\nu, \nu' \in Val(X)$ are called *c-equivalent*, written $\nu \sim_c \nu'$, if

(i) for all $x \in X$, either

a) $\nu(x) = \nu'(x)$ or

b) $\nu(x), \nu'(x) > c$ or

c) $\nu(x), \nu'(x) < -c$,

and

[13] Additionally, the condition $|\nu(x) - \nu(y)|, |\nu'(x) - \nu'(y)| > c$ from [Tri98] has been replaced by conditions (ii) b), c), an improvement suggested by S. Tripakis.

(ii) for all $x, y \in X$, either

a) $\nu(x) - \nu(y) = \nu'(x) - \nu'(y)$ or

b) $\nu(x) - \nu(y), \nu'(x) - \nu'(y) > c$ or

c) $\nu(y) - \nu(x), \nu'(y) - \nu'(x) > c$. ■

The intuition is that two clock valuations are c-equivalent if they cannot be distinguished by any atomic (non-parametric) constraint containing only constants in the interval $[-c, c]$.

Definition 4.68 (c-closed Constraints)
A constraint $g \in CC(X, \mathbb{Z})$ is called *c-closed* if $\nu \models g$ and $\nu \sim_c \nu'$ imply $\nu' \models g$, for any $\nu, \nu' \in Val(X)$. ■

Proposition 4.69
An atomic constraint $x - \delta y \circ a \in CC(X, \mathbb{Z})$, $\delta \in \{0, 1\}$, is c-closed if and only if $|a| \leq c$.

Proof. We give a proof for $x - \delta y \leq a$, the other cases are proven analogously.

If: Assume $|a| \leq c$ and let $\nu, \nu' \in Val(X)$ such that $\nu(x) - \delta\nu(y) \leq a$ and $\nu \sim_c \nu'$. We show $\nu'(x) - \delta\nu'(y) \leq a$. If $\delta = 0$, then $\nu(x) \leq a \leq c$, and thus either $\nu(x) = \nu'(x)$ or $\nu(x), \nu'(x) < -c \leq a$. Hence $\nu'(x) \leq a$. If $\delta = 1$ then $\nu(x) - \nu(y) \leq a \leq c$, and thus either $\nu(x) - \nu(y) = \nu'(x) - \nu'(y)$ or $\nu(x) - \nu(y), \nu'(x) - \nu'(y) < -c \leq a$. Hence $\nu'(x) - \nu'(y) \leq a$.

Only if: Assume $|a| > c$. Pick any $\nu, \nu' \in Val(X)$ such that $\nu(y) = \nu'(y) = 0$, and

(i) $c < \nu(x) \leq a$ and $c < a < \nu'(x)$ if $a \geq 0$, or

(ii) $\nu(x) \leq a$ and $a < \nu'(x) < -c$ if $a < 0$.

Clearly $\nu \sim_c \nu'$ holds, since $\nu(x), \nu'(x) > c$ or $\nu(x), \nu(x) < -c$. On the other hand, $\nu(x) - \delta\nu(y) = \nu(x) \leq a$, but $\nu'(x) - \delta\nu'(y) = \nu'(x) > a$. Hence $x - \delta y \leq a$ is not c-closed. □

Proposition 4.70 (Properties of c-closed Constraints)
Let $g, g' \in CC(X, \mathbb{Z})$, and $c, c' \in \mathbb{N}$.

(i) If g is c-closed and $c' > c$, then g is c'-closed.

(ii) If g and g' are c-closed, then $g \wedge g'$ is c-closed.

(iii) For any $g \in CC(X, \mathbb{Z})$, there exists $c \in \mathbb{N}$ such that g is c-closed.

(iv) There are only finitely many non-equivalent c-closed constraints in $CC(X, \mathbb{Z})$.

Proof. (i) Straightforward, by observing that $\nu \sim_{c'} \nu'$ implies $\nu \sim_c \nu'$. (ii) Assume g, g' are c-closed, $\nu \models g \wedge g'$ and $\nu \sim_c \nu'$. Then $\nu \models g$ and $\nu \models g'$, hence $\nu' \models g$ and $\nu' \models g'$ thus $\nu' \models g \wedge g'$. (iii) Take any c greater than the absolute value of all right-hand sides of atomic constraints in g. (iv) By Proposition 4.69, there are only finitely many c-closed atomic constraints, and a constraint g is c-closed iff there is an equivalent constraint g' whose atomic constraints are all c-closed. Hence there can be only finitely many non-equivalent c-closed constraints. □

κ-closed Parametric Clock Constraints

We now consider parametric constraints. For parametric constraints, the notion of c-closedness is too weak, and we therefore generalize it. Let $\preceq$ be the partial order on linear expressions over K defined by

$$a_0 + a_1k_1 + \ldots + a_nk_n \;\preceq\; b_0 + b_1k_1 + \ldots + b_nk_n$$

if and only if $|a_i| \leq |b_i|$ for all $i \in [0, n]$. Obviously, for any κ, there are only finitely many κ' such that $\kappa' \preceq \kappa$.

Definition 4.71 (κ-closed Parametric Constraints)
Let $g \in CC(X, LE(K))$, and let κ be a linear expression over K having only coefficients in $\mathbb{N}$. We say that g is *κ-closed* if $g\sigma$ is $\kappa\sigma$-closed, for all $\sigma \in K \to \mathbb{N}$. ■

The following proposition generalizes Proposition 4.69 to κ-closed constraints:

Proposition 4.72
An atomic constraint $x - \delta y \circ \kappa' \in CC(X, LE(K))$ is κ-closed if and only if $\kappa' \preceq \kappa$.

Proof. Let $\kappa = a_0 + a_1k_1 + \ldots + a_mk_m$ and $\kappa' = b_0 + b_1k_1 + \ldots + b_mk_m$.

If: Assume $\kappa' \preceq \kappa$. Then $|b_i| \leq |a_i|$ hence $|b_i| \leq a_i$, as $a_i \in \mathbb{N}$. Thus for any $\sigma \in K \to \mathbb{N}$, we have $|\kappa'\sigma| \leq \kappa\sigma$, hence by Proposition 4.69, $x - \delta y \circ \kappa'\sigma$ is $\kappa\sigma$-closed.

Only if: Assume $\kappa' \not\preceq \kappa$. Then $|b_n| > a_n$ for some $n \in [0, m]$. Now let $\sigma \in K \to \mathbb{N}$ be such that $\sigma(k_n) = 1$ and $\sigma(k_i) = 0$ for all $i \neq n$. Then $|\kappa'\sigma| = |b_n| > a_n = \kappa\sigma$. Hence by Proposition 4.69, $x - \delta y \circ \kappa'\sigma$ is not $\kappa\sigma$-closed. Hence $x - \delta y \circ \kappa'$ is not κ-closed. □

Proposition 4.73 (Properties of κ-closed Constraints)
Let $g, g' \in CC(X, LE(K))$, and κ, κ' be linear expressions over K with coefficients in $\mathbb{N}$.

(i) If g is κ-closed and $\kappa \preceq \kappa'$, then g is κ'-closed.

(ii) If g and g' are κ-closed, then $g \wedge g'$ is κ-closed.

(iii) For any $g \in CC(X, LE(K))$, there exists κ such that g is κ-closed.

(iv) There are only finitely many non-equivalent κ-closed constraints in $CC(X, LE(K))$.

Proof. (i) Follows by Proposition 4.72. (ii) Assume g, g' are κ-closed. Let $\sigma \in K \to \mathbb{N}$ be arbitrary. Then $g\sigma, g'\sigma$ are $\kappa\sigma$-closed. Assume $\nu \models (g \wedge g')\sigma$ and $\nu \sim_{\kappa\sigma} \nu'$. Then $\nu \models g\sigma$ and $\nu \models g'\sigma$, hence $\nu' \models g\sigma$ and $\nu' \models g'\sigma$. Thus $\nu' \models (g \wedge g')\sigma$ and hence $(g \wedge g)\sigma$ is $\kappa\sigma$-closed. (iii) Take any κ such that $\kappa' \preceq \kappa$ for every right-hand side κ' of atomic constraints of g. (iv) By Proposition 4.72, there are only finitely many κ-closed atomic constraints, and a constraint g is κ-closed iff there is an equivalent constraint g' whose atomic constraints are all κ-closed. Hence there can be only finitely many non-equivalent κ-closed constraints. □

Operations on Parametric Clock Constraints

Definition 4.74 (tpre(g), $[Y]g$)
Let $g \in CC(X, LE(K))$, and $G = \mathrm{Sol}(g)$. Let $\nu \in \mathit{Val}(X \uplus K)$ and $t \in \mathbb{R}$. Let $Y \subseteq X$. The assignment $\nu +_Y t$ is defined by $(\nu + t)(x) = \nu(x) + t$ if $x \in Y$ and $(\nu + t)(x) = \nu(x)$ otherwise. We write $\nu + t$ instead of $\nu +_X t$. The assignment $\nu[Y]$ is defined by $\nu[Y](x) = 0$ if $x \in Y$ and $\nu[Y](x) = \nu(x)$ otherwise. We write $\nu[y]$ instead of $\nu[\{y\}]$. We write $\nu \geq_Y 0$ if $\nu(x) \geq 0$ for all $x \in Y$. We write $\nu \geq 0$ instead of $\nu \geq_X 0$. We define the *time-predecessor* (also called *backward diagonal projection*) as

$$\mathsf{tpre}(G) = \{\nu \mid \nu \geq_X 0 \text{ and } t \geq 0 \text{ and } \nu +_X t \in G\}$$

and

$$[Y]G = \{\nu \mid \nu[Y] \in G\}$$

By $\mathsf{tpre}(g)$ we denote any constraint obtained by eliminating the existential quantifier from the formula $\exists t.\vec{x} \geq 0 \wedge t \geq 0 \wedge g[\vec{x}+t/\vec{x}]$ by a conjunction-preserving procedure (like Fourier-Motzkin elimination), and discarding any true ground conjuncts, and possibly discarding implied conjuncts. By $[Y]g$ we denote the constraint obtained by replacing by zero all occurrences in g of variables from Y and discarding any true ground conjuncts, and possibly discarding implied conjuncts. ■

It is easy to see that both $\mathrm{Sol}(\mathsf{tpre}(g)) = \mathsf{tpre}(\mathrm{Sol}(g))$ and $\mathrm{Sol}([Y]g) = [Y]\,\mathrm{Sol}(g)$ hold.

For any conjunction g of constraints, let $g|_X$ denote the conjunction of all atomic constraints from g that contain variables from X. If $g \in CC(X, \mathbb{Z})$, then $\mathsf{tpre}(g)|_X = \mathsf{tpre}(g) \in CC(X, \mathbb{Z})$ if $\mathsf{tpre}(g)$ is satisfiable, since all conjuncts resulting from the quantifier elimination and that do not contain a variable from X are ground and thus discarded. By the same argument, we have $([Y]g)|_X = [Y]g \in CC(X, \mathbb{Z})$ if $[Y]g$ is satisfiable.

On the other hand, if $g \in CC(X, LE(K))$, then in general $\mathsf{tpre}(g) = \mathsf{tpre}(g)|_X \wedge \phi$, and $[Y]g = ([Y]g)|_X \wedge \phi'$, where ϕ, ϕ' contain parameters from K. For example, if $g = k_1 \leq x \leq k_2$, then $\mathsf{tpre}(g) = 0 \leq x \leq k_2 \wedge k_1 \leq k_2$ and $[x](g) = k_1 \leq 0 \leq k_2$. For

any $\sigma : K \to \mathbb{N}$ however, we have $\mathsf{tpre}(g\sigma) = \mathsf{tpre}(g\sigma)|_X = (\mathsf{tpre}(g)|_X)\sigma$ and $[Y](g\sigma) = (([Y]g)|_X)\sigma$, since $\phi\sigma, \phi'\sigma$ are ground.

Example 4.75
Figure 4.4 shows the geometric interpretation of $\mathsf{tpre}(\Lambda)$ for $\Lambda = a_1 \leq x \leq a_2, b_1 \leq y \leq b_2$, assuming that $0 \leq a_1 \leq a_2$ and $0 \leq b_1 \leq b_2$. The constraint $\mathsf{tpre}(\Lambda)$ is obtained by eliminating the existential quantifier from $\exists t.x \geq 0 \wedge y \geq 0 \wedge t \geq 0 \wedge a_1 \leq x + t \leq a_2 \wedge b_1 \leq y + t \leq b_2$, yielding $0 \leq x \leq a_2, 0 \leq y \leq b_2, a_1 - b_2 \leq x - y \leq a_2 - b_1$ after removal of the true conjuncts $a_1 \leq a_2$ and $b_1 \leq b_2$.

Figure 4.5 shows the geometric interpretation of $[y]\Lambda$ for $\Lambda = a_1 \leq x \leq a_2, y \leq b_1$. The constraint $[y]\Lambda$ is obtained from Λ by substituting zero for y, yielding $a_1 \leq x \leq a_2$ after removal of the true conjunct $0 \leq b_1$.

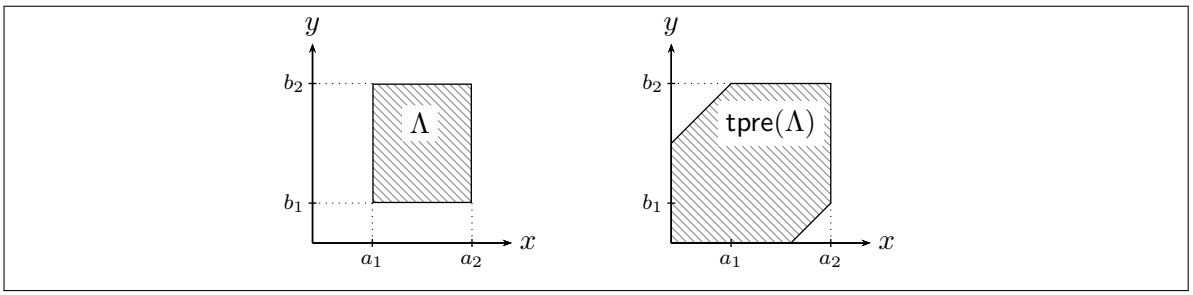


Figure 4.4: Illustration of $\mathsf{tpre}(\Lambda)$

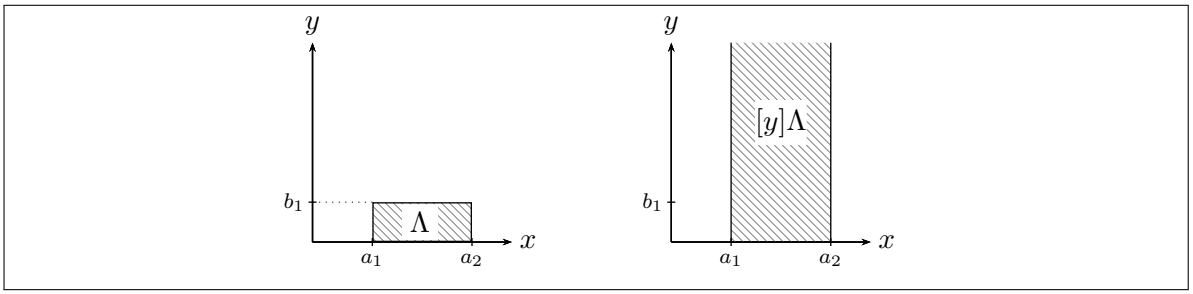


Figure 4.5: Illustration of $[y]\Lambda$

■

Remark 4.76
The original definition of c-equivalence in [Tri98] omits the condition (i) c) from Definition 4.67, but is sufficiently strong to ensure the properties from Proposition 4.70 for standard clock constraints $CC(X)$. When considering $CC(X, \mathbb{Z})$ however, where clocks may be compared to negative constants, the absence of condition (i) c) means that, say, $x \leq -1,\ x \leq -2,\ x \leq -3,\ \ldots$ are all c-closed, for any $c \geq 0$, hence there are infinitely many non-equivalent c-closed constraints.

The addition of condition (i) c) requires us to adapt the definition of $\mathsf{tpre}(G)$: Usually, the time-predecessor is simply defined as

$$\mathsf{tpre}(G) = \{\nu \mid t \geq 0 \text{ and } \nu +_X t \in G\}$$

which preserves the c-closedness (according to the old definition from [Tri98]) of standard clock constraints; it does not, however, preserve the c-closedness based on Definition 4.67, as $\nu(x) < -c$ does not imply $\nu(x)+t < -c$. This is why we add the condition $\nu \geq_X 0$ in the definition of $\mathsf{tpre}(G)$, ensuring that the operation preserves c-closedness of constraints with respect to our definition of c-equivalence, as we prove in Proposition 4.77. It is natural to restrict the time-predecessors to non-negative values, as clocks can only take non-negative values anyway.

As a final step, the addition of the condition $\nu \geq_X 0$ in $\mathsf{tpre}(G)$ means that the constraint $\vec{x} \geq 0$ also has to be added to the time-reachability clauses of TA (Definition 4.57) and ETA (Definition 4.64). While this extra constraint is crucial in the case of ETA reachability theories, it is not necessary for TA, since termination of saturation for TA theories can be proven using the old definition of c-equivalence, as we did in earlier work [FW12]. For the sake of a uniform presentation, we rely on the definitions of this section for both TA and ETA. ■

Proposition 4.77
Let $g \in CC(X, \mathbb{Z})$ be c-closed. Then $\mathsf{tpre}(g)$ is c-closed.

Proof. Let $\nu, \nu' \in \mathit{Val}(X)$ such that $\nu \models \mathsf{tpre}(g)$ and $\nu \sim_c \nu'$. Then $\nu \geq 0$ and $\nu + t \models g$ for some $t \geq 0$. We show $\nu + t \sim_c \nu' + t$. Let $x, y \in X$ be arbitrary. It suffices to consider case (i) of Definition 4.67, since $(\nu + t)(x) - (\nu + t)(y) = \nu(x) + t - \nu(y) - t = \nu(x) - \nu(y)$. If $\nu(x) = \nu'(x)$, then also $\nu(x) + t = \nu'(x) + t$, and if $\nu(x), \nu'(x) > c$ then also $\nu(x) + t, \nu'(x) + t > c$. The case $\nu(x), \nu'(x) < -c$ is ruled out by $\nu \geq 0$. □

Proposition 4.78
Let $g \in CC(X)$ be c-closed and let $Y \subseteq X$. Then $[Y]g$ is c-closed.

Proof. Let $\nu, \nu' \in \mathit{Val}(X)$ such that $\nu \models [Y]g$ and $\nu \sim_c \nu'$. Then $\nu[Y] \models g$. We show $\nu[Y] \sim_c \nu'[Y]$. Let $x, y \in X$ be arbitrary. We write $\nu, \nu' \models \mathrm{a}(x)$ if ν, ν' and x satisfy condition (i) a) of Definition 4.67, and analogously we use $\mathrm{b}(x), \mathrm{c}(x), \mathrm{a}(x, y)$, $\mathrm{b}(x, y)$ and $\mathrm{b}(y, x)$ for the other conditions.[14] It follows from $\nu \sim_c \nu'$ that ν, ν' must satisfy one of $\mathrm{a}(x), \mathrm{b}(x), \mathrm{c}(x)$, and one of $\mathrm{a}(x, y), \mathrm{b}(x, y), \mathrm{b}(y, x)$, for all $x, y \in X$. The table below shows the corresponding case analysis:

[14] Condition (ii) c) of Def. 4.67 is the same as (ii) b) with x and y swapped, therefore we denote it by $\mathrm{b}(y, x)$ instead of $\mathrm{c}(x, y)$.

if $\nu, \nu' \models$	and		then $\nu[Y], \nu'[Y] \models$
—	$x \in Y$		$\mathrm{a}(x)$
$\mathrm{a}(x)$	$x \notin Y$		$\mathrm{a}(x)$
$\mathrm{b}(x)$	$x \notin Y$		$\mathrm{b}(x)$
$\mathrm{c}(x)$	$x \notin Y$		$\mathrm{c}(x)$
—	$x, y \in Y$		$\mathrm{a}(x, y)$
$\mathrm{a}(x, y)$	$x, y \notin Y$		$\mathrm{a}(x, y)$
$\mathrm{b}(x, y)$	$x, y \notin Y$		$\mathrm{b}(x, y)$
—	$x \in Y, y \notin Y,$	$\mathrm{a}(y)$	$\mathrm{a}(x, y)$
		$\mathrm{b}(y)$	$\mathrm{b}(y, x)$
		$\mathrm{c}(y)$	$\mathrm{b}(x, y)$

For example, if $x \in Y, y \notin Y$ and $b(y)$, then $\nu(y) - \nu(x) = \nu(y) > c$ and $\nu'(y) - \nu'(x) = \nu'(y) > c$. The other cases are proved analogously. □

Proposition 4.79
Let $g \in CC(X, LE(K))$ be κ-closed and let $Y \subseteq X$. Then $\mathsf{tpre}(g)|_X$ and $([Y]g)|_X$ are κ-closed.

Proof. Let $\sigma : K \to \mathbb{N}$ be arbitrary. Since g is κ-closed, $g\sigma$ is $\kappa\sigma$-closed. By Proposition 4.77, $\mathsf{tpre}(g\sigma) = (\mathsf{tpre}(g)|_X)\sigma$ is $\kappa\sigma$-closed. Hence $\mathsf{tpre}(g)|_X$ is κ-closed. By Proposition 4.78, $[Y](g\sigma) = (([Y]g)|_X)\sigma$ is $\kappa\sigma$-closed. Hence $([Y]g)|_X$ is κ-closed. □

4.5.6 SUP(LA) as a Decision Procedure for Timed Automata

In this section, we will show that SUP(LA) can be turned into a decision procedure for reachability in timed automata. The result we will establish is that, under a suitable derivation strategy, any SUP(LA) derivation from $N_{TA}^{\flat}$—a reachability query based on the backward encoding of *TA*—terminates.

The termination argument relies on the fact that there are only finitely many non-equivalent c-closed clock constraints over a given set of clock variables (Proposition 4.70), by showing that for a suitable constant $c \in \mathbb{N}$, the constraints of all derived clauses are c-closed.

Proposition 4.80
Consider an ordered resolution inference

$$\mathcal{I}\frac{t{\geq}0,\ \vec{x}' \simeq \vec{x}{+}t,\ Inv(L)[\vec{x}'/\vec{x}] \parallel R(L, \vec{x}') \to R(L, \vec{x}) \qquad \Lambda \parallel \ \to R(L, \vec{x}')}{\Lambda' \parallel \ \to R(L, \vec{x})}$$

between a time-reachability clause and a goal clause. Assume Λ and $Inv(L)$ are c-closed, for some $c \in \mathbb{N}$. Then Λ' is c-closed.

Proof. Λ' is equivalent to $\exists t.(t \geq 0 \wedge Inv(L)[\vec{x}+t/\vec{x}] \wedge \Lambda[\vec{x}+t/\vec{x}'])$ or equivalently $\mathsf{tpre}(Inv(L)[\vec{x}] \wedge \Lambda[\vec{x}/\vec{x}'])$. By Proposition 4.70, $Inv(L)[\vec{x}] \wedge \Lambda[\vec{x}/\vec{x}']$ is c-closed and thus Λ' is c-closed by Proposition 4.77. □

Proposition 4.81
Consider an ordered resolution inference

$$\mathcal{I}\frac{g[\vec{x}],\, \vec{x}' \simeq A_D(\vec{x}),\, Inv(L')[\vec{x}'/\vec{x}] \;\|\; R(L',\vec{x}') \to R(L,\vec{x}) \qquad \Lambda \,\|\, \to R(L,\vec{x}')}{\Lambda' \,\|\, \to R(L,\vec{x})}$$

between a discrete-step clause and a goal clause. Assume Λ, g and $Inv(L)$ are c-closed, for some $c \in \mathbb{N}$. Then Λ' is c-closed.

Proof. Λ' is equivalent to $g[\vec{x}] \wedge Inv(L')[A_D(\vec{x})/\vec{x}] \wedge \Lambda[A_D(\vec{x})/\vec{x}']$ or equivalently $g[\vec{x}] \wedge [D](Inv(L')[\vec{x}] \wedge \Lambda[\vec{x}/\vec{x}'])$. It follows by Propositions 4.70 and 4.78 that Λ' is c-closed. □

Derivation Strategy

We apply the following derivation strategy:

(i) all negative literals are selected;

(ii) the only inference rule is ordered resolution;

(iii) reduction rules include subsumption deletion.

An immediate consequence of the selection strategy is that ordered resolution can only derive positive unit clauses of the form $\Lambda \,\|\, \to R(L,\vec{x})$, which we call *goal clauses*,[15] where $L \in Loc$ is a location constant.

Theorem 4.82
Let TA be a timed automaton and $N_{TA}^{\flat}$ be a reachability query based on the backward encoding of TA. Then any derivation from $N_{TA}^{\flat}$ following the above strategy terminates.

Proof. Assume for contradiction that $N_{TA}^{\flat} = N_0 \triangleright N_1 \triangleright N_2 \triangleright \ldots$ is an infinite derivation, where $N_{i+1} = N_i \cup \{C_{i+1}\}$ such that C_i is a positive unit clause not subsumed by any clause in N_i. By Proposition 4.70, there exists $c \in \mathbb{N}$ such that all guards and invariants in TA, as well as the clause constraints of all goal clauses in $N_{TA}^{\flat}$ are c-closed. It follows from Propositions 4.80 and 4.81 that any derived C_i has a c-closed constraint as well. As Loc is finite, there must be an infinite subsequence of positive unit clauses all containing the same location constant, and as there are only finitely many non-equivalent c-closed constraints (by Proposition 4.70), there exists an index k such that any C_i with $i \geq k$ is subsumed by some C_j with $j < k$. Now either C_j itself or some clause subsuming it, and hence also subsuming C_i, is contained in N_j, a contradiction. □

[15] This terminology is consistent with Definition 4.32, since we work with a backward encoding.

4.6 Constraint Induction

In this section, we introduce a new SUP(T) rule, called *constraint induction*. The focus of the rule is on the clause constraints (hence the name), and it relies on the computation of the transitive closure of constraints involved in *loops* that arise in the SUP(T) search space.

In the following, we first give a general definition of the rule that applies to arbitrary incarnations of SUP(T) (Proposition 4.83) and then show an effective instance of the rule based on loop detection (Section 4.6.1).

For the transitive closure computation, we restrict ourselves to the special case of SUP(LA) (Section 4.6.2). The constraint induction rule strictly increases the power of the SUP(LA) calculus, as evidenced by the resulting decision procedure for reachability in ETA (Section 4.6.3) and by improved termination and performance on other examples as well (Section 4.6.4).

Proposition 4.83 (Constraint Induction)
Let Λ be a $\mathcal{T}$-formula with free variables $\vec{x}, \vec{x}'$, and $\phi[\vec{x}]$ a formula with free variables $\vec{x}$. Suppose there exists a $\mathcal{T}$-formula Λ^+ with free variables $\vec{x}, \vec{x}'$, such that

$$\mathrm{Sol}_{\mathcal{I}}(\Lambda^+, \vec{x}\vec{x}') = \mathrm{Sol}_{\mathcal{I}}(\Lambda, \vec{x}\vec{x}')^+ \tag{4.2}$$

holds in any $\mathcal{T}$-model $\mathcal{I}$. Then

$$\forall \vec{x}, \vec{x}'.\ \Lambda \wedge \phi[\vec{x}] \to \phi[\vec{x}'] \ \models_{\mathcal{T}}\ \forall \vec{x}, \vec{x}'.\ \Lambda^+ \wedge \phi[\vec{x}] \to \phi[\vec{x}'].$$

Proof. Let $\mathcal{I}$ be a $\mathcal{T}$-model such that

$$\mathcal{I} \models \forall \vec{x}, \vec{x}'.\ \Lambda \wedge \phi[\vec{x}] \to \phi[\vec{x}']. \tag{4.3}$$

Let ν be an arbitrary assignment for $\mathcal{I}$, and $\vec{a}, \vec{a}'$ be sequences of elements in $U_{\mathcal{I}}$, such that

$$\mathcal{I}, \nu[\vec{x} \mapsto \vec{a}, \vec{x}' \mapsto \vec{a}'] \models \Lambda^+[\vec{x}, \vec{x}'] \wedge \phi[\vec{x}].$$

By (4.2), there exists $k \geq 2$ and $\vec{a}_1, \ldots, \vec{a}_k$ in $U_{\mathcal{I}}$, with $\vec{a} = \vec{a}_1, \vec{a}' = \vec{a}_k$, such that

$$\mathcal{I}, \nu[\vec{x}_1 \mapsto \vec{a}_1, \ldots, \vec{x}_k \mapsto \vec{a}_k] \models \Lambda[\vec{x}_1, \vec{x}_2] \wedge \ldots \wedge \Lambda[\vec{x}_{k-1}, \vec{x}_k] \wedge \phi[\vec{x}_1].$$

Using (4.3), it follows by induction on i that

$$\mathcal{I}, \nu[\vec{x}_1 \mapsto \vec{a}_1, \ldots, \vec{x}_k \mapsto \vec{a}_k] \models \phi[\vec{x}_i]$$

for all $i \in [2, k]$. Thus

$$\mathcal{I}, \nu[\vec{x}_1 \mapsto \vec{a}_1, \ldots, \vec{x}_k \mapsto \vec{a}_k] \models \Lambda^+[\vec{x}_1, \vec{x}_k] \wedge \phi[\vec{x}_1] \to \phi[\vec{x}_k].$$

or equivalently

$$\mathcal{I}, \nu[\vec{x} \mapsto \vec{a}, \vec{x}' \mapsto \vec{a}'] \models \Lambda^+[\vec{x}, \vec{x}'] \wedge \phi[\vec{x}] \to \phi[\vec{x}'].$$

as $\vec{x}_2, \ldots, \vec{x}_{k-1}$ don't occur in the formula. Thus we conclude that

$$\mathcal{I} \models \Lambda^+[\vec{x}, \vec{x}'] \wedge \phi[\vec{x}] \rightarrow \phi[\vec{x}'].$$

□

Proposition 4.83 can be straightforwardly turned into an inference rule for SUP($\mathcal{T}$), which we call constraint induction. The rule is shown in Figure 4.6.[16]

Constraint Induction:

$$\mathcal{I}\frac{C_1, \ldots, C_n}{\begin{array}{c} D_1 \\ \vdots \\ D_m \end{array}}$$

where

(i) $C_1, \ldots, C_n \models_{\mathcal{T}} \forall \vec{x}, \vec{x}'.\ \Lambda[\vec{x}, \vec{x}'] \wedge \phi[\vec{x}] \rightarrow \phi[\vec{x}']$

(ii) $\{D_1, \ldots, D_m\} = \mathrm{cnf}(\forall \vec{x}, \vec{x}'.\ \Lambda^+[\vec{x}, \vec{x}'] \wedge \phi[\vec{x}] \rightarrow \phi[\vec{x}'])$

if Λ^+ exists.

Figure 4.6: Constraint induction rule

Remark 4.84
The constraint Λ^+ typically represents a relation of the form

$$\exists m.\ m \geq 1 \wedge \Lambda'[m, \vec{x}, \vec{x}']$$

where m is an integer-sorted variable. For example, the transitive closure of $x' = x + 1$ is equivalent to $\exists m.\ m \geq 1 \wedge x' = x + m$. This existential quantification however comes "for free" in abstracted FOL($\mathcal{T}$) clauses, as long as m does not occur in the free part of the clause. For instance,

$$m \geq 1, x' = x + m \parallel P(x) \rightarrow P(x')$$

is logically equivalent to

$$(\exists m.\ m \geq 1, x' = x + m) \parallel P(x) \rightarrow P(x').$$

We refer to the integer-sorted variables introduced by constraint induction as *loop counters*. ■

[16] In Definition 2.31, inference rules have only a single conclusion. The generalization to multiple conclusions, as required for constraint induction, is straightforward.

The constraint induction rule as shown in Figure 4.6 is not effective, because, firstly, condition (i) is undecidable in general and secondly, the "transitive closure formula" Λ^+ may not be expressible in the language of $\mathcal{T}$.

To address the first problem, we will now consider an instance of the constraint induction rule which relies on the detection of "loops" in the search space. This technique applies to the general SUP(T) framework with arbitrary base theories. The second problem is specific to the base theory and will be addressed in Section 4.6.2 for the case of linear arithmetic.

4.6.1 Constraint Induction by Loop Detection

Suppose we have a SUP($\mathcal{T}$)-derivation

$$\Lambda \parallel C,\ D_1,\ \ldots,\ D_m \ \vdash\ \Lambda' \parallel C.$$

where the first premise and the conclusion have the same free part and differ only in their constraints. Since the applicability of inference rules in SUP($\mathcal{T}$) does not depend on the constraints of the premises (except in the case of constraint refutation), such a situation can lead to an infinite derivation

$$\begin{gathered}\Lambda' \parallel C,\ D_1,\ \ldots,\ D_m \ \vdash\ \Lambda'' \parallel C\\ \Lambda'' \parallel C,\ D_1,\ \ldots,\ D_m \ \vdash\ \Lambda''' \parallel C\\ \ldots\end{gathered}$$

unless some clauses become redundant, or constraint refutation derives the empty clause. We call such a situation a *loop* in the proof search space. Constraint induction makes it possible to avoid such infinite looping by deriving a clause that "summarizes" (or "accelerates") the loop.

Proposition 4.85
Let $\Lambda_0 \parallel C_0$, $\Lambda_1 \parallel C_1$, $\ldots$, $\Lambda_m \parallel C_m \ \vdash\ \Lambda \parallel C_0$ be a derivation in SUP($\mathcal{T}$) (without variable elimination), where $\Lambda_0 \parallel C_0$ is used exactly once. Then there exists a conjunctive constraint $\Lambda_\Delta[\vec{x}, \vec{x}']$ such that

$$\Lambda_1 \parallel C_1,\ \ldots,\ \Lambda_m \parallel C_m \ \models_{\mathcal{T}}\ \forall \vec{x}, \vec{x}'.\ \Lambda_\Delta[\vec{x}, \vec{x}'] \wedge C_0[\vec{x}] \rightarrow C_0[\vec{x}'].$$

Proof. Since the applicability of inference rules in SUP($\mathcal{T}$) does not depend on the constraints of the premises, we can replace Λ_0 by $\top$ (the empty constraint), and again obtain a valid SUP($\mathcal{T}$)-derivation

$$\top \parallel C_0,\ \Lambda_1 \parallel D_1,\ \ldots,\ \Lambda_m \parallel D_m \ \vdash\ \hat{\Lambda} \parallel C_0$$

where $\hat{\Lambda}$ is a conjunction of instances of $\Lambda_1, \ldots, \Lambda_m$, with unifiers applied.[17] By soundness of SUP($\mathcal{T}$), we get

$$\models_{\mathcal{T}} \forall \vec{x}_0, \ldots, \vec{x}_m.\ C_0[\vec{x}_0] \wedge \hat{C}_1[\vec{x}_0, \vec{x}_1] \wedge \ldots \wedge \hat{C}_m[\vec{x}_{m-1}, \vec{x}_m] \rightarrow \left(\hat{\Lambda}[\vec{x}_0, \ldots, \vec{x}_m] \rightarrow C_0[\vec{x}_m]\right)$$

where

$$\hat{C}_i = \bigwedge_{j \in I_i} ((\Lambda_i \rightarrow C_i)\sigma_j)$$
$$\hat{\Lambda} = \bigwedge_{i \in [1,m]} \bigwedge_{j \in I_i} \Lambda_i \sigma_j$$

for some non-empty sets of indices[18] I_i. It follows that

$$\models_{\mathcal{T}} \forall \vec{x}_0, \ldots, \vec{x}_m.\ \hat{C}_1[\vec{x}_0, \vec{x}_1] \wedge \ldots \wedge \hat{C}_m[\vec{x}_{m-1}, \vec{x}_m] \rightarrow \left(\hat{\Lambda}[\vec{x}_0, \ldots, \vec{x}_m] \wedge C_0[\vec{x}_0] \rightarrow C_0[\vec{x}_m]\right).$$

As $\Lambda_i \,\|\, C_i \models_{\mathcal{T}} \forall \hat{C}_i$ for all $i \in [1, m]$, we get

$$\Lambda_1 \,\|\, C_1,\ \ldots,\ \Lambda_m \,\|\, C_m \ \models_{\mathcal{T}}\ \Lambda_\Delta[\vec{x}, \vec{x}'] \wedge C_0[\vec{x}] \rightarrow C_0[\vec{x}'].$$

by taking $\Lambda_\Delta = \exists \vec{x}_1, \ldots, \vec{x}_{m-1}.\ \hat{\Lambda}[\vec{x}, \vec{x}_1, \ldots, \vec{x}_{m-1}, \vec{x}']$. □

Example 4.86
Consider the FOL(LA) clauses

(1) $x \simeq 1 \,\|\, \rightarrow P(x)$
(2) $x' \simeq x + 2 \,\|\, P(x) \rightarrow Q(x')$
(3) $x' \simeq x - 1 \,\|\, Q(x) \rightarrow P(x')$

Resolution of (1) with (2) and (3) yields

(4) $x \simeq 1, x' \simeq x + 2, x'' \simeq x' - 1 \,\|\, \rightarrow P(x'')$

or

(4') $x \simeq 2 \,\|\, \rightarrow P(x)$

after constraint simplification and variable normalization. Indeed it holds that

$$(2), (3) \ \models_{\text{LA}}\ x' \simeq x{+}1 \wedge P(x) \rightarrow P(x').$$

Now the clause

[17] See the remark on page 71.
[18] See the remark on page 71.

(5) $$k \geq 1,\, x' \simeq x + k \parallel P(x) \to P(x')$$

where k is an integer-sorted variable, can be derived by constraint induction. Note that (5) depends only on (2) and (3), and not on (1) or (4)—the latter two clauses only serve as witnesses to the existence of the loop. ∎

Remark 4.87
The constraint induction rule presented here is stronger than the one from our earlier work [FKW12], which would produce a clause depending also on the initial clause, e.g.,

(5') $$k \geq 1,\, x \simeq k + 1 \parallel \quad \to P(x)$$

in the example above. This clause is clearly weaker than (5), as it can be obtained by resolution of (1) and (5). ∎

Remark 4.88
In Proposition 4.85, the restriction of $\Lambda_0 \parallel C_0$ being used exactly once in the derivation of $\Lambda \parallel C_0$ is required to ensure that condition (i) of the constraint induction rule is satisfied. In order to handle arbitrary derivations, the proposition would need to be strengthened to deal with formulas of the form

$$\Lambda \wedge \phi[\vec{x}_1] \wedge \ldots \wedge \phi[\vec{x}_n] \to \phi[\vec{x}'].$$

For instance, from $x' \simeq x + y \wedge \phi[x] \wedge \phi[y] \to \phi[x']$ one could then obtain $k \geq 1 \wedge x' \simeq 2^{k-1}(x+y) \wedge \phi[x] \wedge \phi[y] \to \phi[x']$. This generalization requires a more involved theory however, and we leave it as future work. ∎

There is a practical problem that remains to be solved if we want to integrate constraint induction into the SUP($\mathcal{T}$) calculus: The constraint Λ_Δ cannot be simply read off from the conclusion of the loop. Indeed, consider clause (4) in Example 4.86: We cannot tell which of the variables x, x' is the "original" variable that should be kept in order to define Λ_Δ. The situation becomes even worse when constraint simplification and variable normalization are used (as is the case in any practical implementation): Clause (4') could just as well have been derived by resolution of (1) with the clause $x' \simeq 2 \parallel P(x) \to P(x')$. So the information needed to reconstruct Λ_Δ is lost by constraint simplification and variable normalization.

We can recover this information by using the following trick: We replace the initial constraint by a constraint containing fresh base constant symbols, and "replay" the loop with the modified constraint, as shown in Algorithm 4.1. The purpose of the constant symbols is to "pin down" the variables of the first occurrence of C_0 so that they don't get eliminated.

The procedure can be called whenever a clause $\Lambda' \parallel C$ is derived which has an ancestor $\Lambda \parallel C$ with the same free part. The constraint Λ_0 is $x_1 \simeq c_1, \ldots, x_n \simeq c_n$ where c_i is a

Algorithm 4.1: ReplayLoop

Input: Partial SUP($\mathcal{T}$) derivation $\mathcal{D} = \Lambda \parallel C,\ D_1,\ \ldots,\ D_m \vdash\ \Lambda' \parallel C$
Output: Constraint $\Lambda_\Delta[\vec{x}, \vec{x}']$

1 $\Lambda_0 := \vec{x} \simeq \vec{c}$ *for fresh base constants* $\vec{c}$;
2 $C_0[\vec{x}] := C[\vec{x}]$;
3 **for** $i := 1$ **to** m **do**
4 $\quad \Lambda_i \parallel C_i := \inf_{\mathcal{D},i}(\Lambda_{i-1} \parallel C_{i-1}, D_{i,1}, \ldots, D_{i,k_i})$;
5 **return** $\Lambda_m[\vec{x}/\vec{c},\ \vec{x}'/\vec{x}]$;

fresh constant of the same sort as x_i, where $x_1, \ldots, x_n$ are the base variables of $\Lambda \parallel C$. On lines 3 and 4, the derivation $\mathcal{D}$ is replayed, starting with $\Lambda_0 \parallel C_0$. The ith inference of $\mathcal{D}$ is denoted by $\inf_{\mathcal{D},i}$, and $D_{i,1}, \ldots, D_{i,k_i}$ are the remaining premises. This replaying of inferences is always possible, as the applicability of inference rules in SUP($\mathcal{T}$) does not depend on the constraints of the premises.

Example 4.89
Consider again the clauses from Example 4.86. When clause (4') is derived, we notice that clause (1) is among its ancestors and has the same free part. So we execute ReplayLoop, producing the following clauses:

$$\begin{array}{ll} \Lambda_0 \parallel C_0 & x \simeq c \parallel\ \rightarrow P(x) \\ \Lambda_1 \parallel C_1 = \mathrm{Res}(\Lambda_0 \parallel C_0,\ (2)) & x \simeq c + 2 \parallel\ \rightarrow Q(x) \\ \Lambda_2 \parallel C_2 = \mathrm{Res}(\Lambda_1 \parallel C_1,\ (3)) & x \simeq c + 1 \parallel\ \rightarrow P(x) \end{array}$$

The returned constraint is $(x \simeq c + 1)[x/c, x'/x] = x' \simeq x + 1$, as expected, so constraint induction can be applied to produce (5). ∎

4.6.2 Computing the Transitive Closure of LA Constraints

The second problem we have to solve in order to make the constraint induction rule effective concerns the computation of the transitive closure constraint Λ^+.

We show how this can be realized for the case of FOL(LA).

Proposition 4.90
Let $a, c \in \mathbb{R} \cup \{-\infty\}$ and $b, d \in \mathbb{R} \cup \{\infty\}$ such that $a \leq b$ and $c \leq d$. If $R(x, x') \subseteq \mathbb{R}^2$ is defined by $a \leq x \leq b \wedge c \leq x' - x \leq d$ then for any $m \geq 1$, $R^m(x, x')$ is defined by

$$\bigwedge_{i=0}^{m-1} x \geq a - id \ \wedge\ \bigwedge_{i=0}^{m-1} x \leq b - ic \ \wedge\ \bigwedge_{i=1}^{m-1} x' \geq a + ic \ \wedge\ \bigwedge_{i=1}^{m-1} x' \leq b + id$$

$$\wedge\ mc \leq x' - x \leq md\ .$$

In particular, $R^m(x, x')$ is equivalent to

$$a \leq x \leq b - (m{-}1)c \ \wedge\ a + (m{-}1)c \leq x' \leq b + d \ \wedge\ mc \leq x' - x \leq md$$

if $c, d \geq 0$, and equivalent to

$$a \leq x \leq b \ \wedge\ a + c \leq x' \leq b + d \ \wedge\ mc \leq x' - x \leq md$$

if $c < 0$ and $d \geq 0$, and equivalent to

$$a - (m{-}1)d \leq x \leq b \ \wedge\ a + c \leq x' \leq b + (m{-}1)d \ \wedge\ mc \leq x' - x \leq md$$

if $c, d < 0$.

Proof. The general form is established by a straightforward though tedious induction on m. The special forms are obtained by taking greatest upper bounds and smallest lower bounds. For instance, if $d \geq 0$, then $x \geq a$ implies $x \geq a - id$ for all $i \geq 0$, whereas if $d < 0$, then $x \geq a - id$, $i \geq 0$ implies $x \geq a$. □

It is straightforward to extend Proposition 4.90 to constraints with strict inequalities.

Example 4.91
Consider the constraint $\Lambda = x_1 \geq 3,\ x_1' \geq x_1 + 2,\ x_2 + 1 > x_2'$. It represents the product relation

$$\mathrm{Sol}(\Lambda) \ =\ (x_1 \geq 3 \ \wedge\ x_1' \geq x_1 + 2) \ \cdot\ x_2 + 1 > x_2' \ .$$

By Proposition 4.90, we get

$$\begin{aligned}
\mathrm{Sol}(\Lambda)^m \ &=\ (x_1 \geq 3 \ \wedge\ x_1' \geq x_1 + 2)^m \ \cdot\ (x_2 + 1 > x_2')^m \\
&=\ (x_1 \geq 3 \ \wedge\ x_1' \geq 5 \ \wedge\ x_1' - x_1 \geq 2m) \ \cdot\ (x_2' - x_2 \leq m) \\
&=\ x_1 \geq 3 \ \wedge\ x_1' \geq 5 \ \wedge\ x_1' - x_1 \geq 2m \ \wedge\ x_2' - x_2 \leq m
\end{aligned}$$

with $a = 3, c = 2, b = d = \infty$ for the first component relation and $d = 1, b = \infty, a = c = -\infty$ for the second component relation.

Thus Λ^+ is

$$\exists m.\ m \geq 1 \ \wedge\ x_1 \geq 3 \ \wedge\ x_1' \geq 5 \ \wedge\ x_1' - x_1 \geq 2m \ \wedge\ x_2' - x_2 \leq m$$

where x_1, x_2 are real-sorted variables, and m is an integer-sorted variable. Assuming that we have concluded

$$N \models_{\mathcal{T}} \Lambda \wedge P(x_1, x_2) \to P(x_1', x_2')$$

by loop detection, we can thus derive the clause

$$m \geq 1,\ x_1 \geq 3,\ x_1' \geq 5,\ x_1' - x_1 \geq 2m,\ x_2' - x_2 \leq m \parallel P(x_1, x_2) \to P(x_1', x_2')$$

by constraint induction. ■

Proposition 4.92
Let $\vec{x} = x_1, \ldots, x_n$, let $\mathbf{A} \in \mathbb{Z}^{m \times n}$ be a matrix, $\vec{b} \in \mathbb{Z}^m$, and let $\mathbf{D}$ be a $n \times n$ diagonal matrix over $\{0, 1\}$, and $\vec{c} \in \mathbb{Z}^n$. If $R(\vec{x}, \vec{x}') \in \mathbb{Z}^{2n}$ is defined by

$$\mathbf{A}\vec{x} \leq \vec{b} \ \wedge \ \vec{x}' = \mathbf{D}\vec{x} + \vec{c}$$

then $R^m(\vec{x}, \vec{x}')$, for any $m \geq 1$, is defined by

$$\begin{aligned} &\big((m = 1 \wedge \mathbf{A}\vec{x} \leq \vec{b}) \\ \vee \ &(m > 1 \wedge \mathbf{A}\vec{x} \leq \vec{b} \\ &\qquad \wedge \mathbf{A}(\mathbf{D}\vec{x} + \vec{c}) \leq \vec{b} \\ &\qquad \wedge \mathbf{A}(\mathbf{D}\vec{x} + (m-2)\mathbf{D}\vec{c} + \vec{c}) \leq \vec{b})\big) \\ \wedge \ &\vec{x}' = \mathbf{D}\vec{x} + (m-1)\mathbf{D}\vec{c} + \vec{c} \end{aligned}$$

Proof. The proof relies on the convexity of the constraints and on the fact that $\mathbf{D}$ is idempotent. It can be found in [BW94]. □

Example 4.93
Consider the LA constraint $\Lambda = x \leq y, \ x' \simeq x + 1, \ y' \simeq y$. In the matrix notation of Proposition 4.92, this is

$$\begin{pmatrix} 1 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq 0 \quad \wedge \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Following Proposition 4.92, $\mathrm{Sol}(\Lambda)^m$ is defined by

$$\begin{aligned} &\big((m = 1 \wedge x \leq y) \\ \vee \ &(m > 1 \wedge x \leq y \\ &\qquad \wedge x + 1 \leq y \\ &\qquad \wedge x + m - 1 \leq y)\big) \\ \wedge \ &x' = x + m \ \wedge \ y' = y \end{aligned}$$

which can be simplified to

$$m \geq 1 \ \wedge \ x + m - 1 \leq y \ \wedge \ x' = x + m \ \wedge \ y' = y \ .$$

Assuming that we have concluded

$$N \models_{\mathcal{T}} \Lambda \wedge P(x, y) \rightarrow P(x', y')$$

by loop detection, we can thus derive the clause

$$m \geq 1, \ x + m - 1 \leq y, \ x' \simeq x + m, \ y' \simeq y \parallel P(x, y) \rightarrow P(x', y')$$

by constraint induction. ■

Remark 4.94
In general, the transitive closure of constraints of the type defined in Proposition 4.92 cannot be simplified into a purely conjunctive form. If the constraint remains disjunctive, then the clausification performed by the constraint induction rule yields two clauses. Consider for instance $x \leq y \wedge x' = x + 1 \wedge y' = 3$, whose transitive closure is $((m = 1 \wedge x \leq y) \vee (m > 1 \wedge x \leq y \wedge x + m - 1 \leq 3)) \wedge x' = x + m \wedge y' = 3$. ∎

Remark 4.95 (Transitive closure of parametric constraints)
Proposition 4.92 is also applicable when $\mathbf{A}$, $\vec{b}$ and $\vec{c}$ contain parameters, since the proof makes no assumption about their values. Example 4.107 in Section 4.6.4 shows an application of constraint induction to a parametric system, based on Proposition 4.92. Extending Proposition 4.90 to parametric constraints is more complicated, since the simplification of the initial conjunction depends on the signs of a, b, c and d. In order to justify the simplification when some of a, b, c, d are parameters, suitable assumptions need to made about them. Such assumptions must be represented by additional base clauses (i.e., constrained empty clauses). For instance, assuming c and d are parameters, the constraint $c, d \geq 0$ is represented by the two clauses $c < 0 \parallel \Box$ and $d < 0 \parallel \Box$. When computing the transitive closure of a parametric constraint, the clause set has to be searched for such base clauses in order to justify the simplification. ∎

Strengthening Constraints

In general, the constraint returned by ReplayLoop (Algorithm 4.1) may not be of the type required by Propositions 4.90 and 4.92, or it may even be such that its transitive closure cannot be expressed as a linear constraint at all. In this case, constraint induction may still be applicable to a strengthened version of the constraint.

Consider for instance $\Lambda = x' \simeq 2x, y' \simeq x + y$ and assume we have established $N \models_{\text{LA}} \Lambda \wedge \phi[x, y] \rightarrow \phi[x', y']$ by loop detection. It follows that

$$k \geq 1 \wedge x' \simeq 2^k x \wedge y' \simeq (2^k - 1)x + y \wedge \phi[x, y] \rightarrow \phi[x', y'], \tag{4.4}$$

is also a consequence of N, but this formula is not expressible in the language of FOL(LA). However, $N \models_{\text{LA}} \Lambda' \wedge \phi[x, y] \rightarrow \phi[x', y']$ also holds, for any strengthening Λ' of Λ. If we choose (say) $\Lambda' = \Lambda \wedge x \simeq 1$, then the transitive closure can be computed as per Proposition 4.92, yielding

$$k \geq 1 \wedge x \simeq 1 \wedge x' \simeq 2 \wedge y' \simeq y + k \wedge \phi[x, y] \rightarrow \phi[x', y'], \tag{4.5}$$

which can be represented by FOL(LA) clauses. Evidently, (4.5) is weaker than (4.4), but the former may still be useful. We will exploit this technique in Section 4.6.3 to ensure that constraint induction produces only clauses with clock constraints: We will strengthen any Λ of the form

$$x + ay + b \circ x', \; y' \simeq 0$$

to

$$x + b \circ x',\ y' \simeq 0,\ y \simeq 0$$

by adding $y \simeq 0$, before computing the transitive closure according to Proposition 4.90.

4.6.3 SUP(LA) with Constraint Induction as a Decision Procedure for ETA

We now define acceleratable cycles of extended timed automata. The notion of acceleratable cycle was introduced in [HL02]

Definition 4.96 (Acceleratable Cycle)
Let $ETA = (Loc, Act, X, \hookrightarrow, L_0, g_0, Inv)$ be an extended timed automaton. A *cycle* over $L^{(0)}, \ldots, L^{(n-1)} \in Loc$, $n \geq 1$, is a sequence of edges $e_0, \ldots, e_{n-1} \in \hookrightarrow$ such that

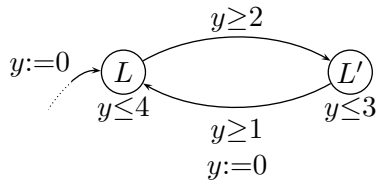
$$e_i = (L^{(i)}, g^{(i)}, \alpha^{(i)}, A^{(i)}, L^{(i+1 \bmod n)}).$$

The cycle is called *simple* if $L^{(i)} \neq L^{(j)}$ whenever $i \neq j$. The cycle is called *acceleratable* if it is simple and all guards and invariants on the cycle contain at most a single clock variable, which is the same for all guards and invariants on the cycle, and this clock is reset on all incoming edges to $L^{(0)}$. The clock being reset on an acceleratable cycle is called the *clock of the cycle*, and $L^{(0)}$ is called the *reset location*. ∎

Note that acceleratability of a cycle places no restrictions on the integer guards and instructions occurring on the cycle.

In [HL02], it is shown that any acceleratable cycle has an interval $[a, b]$, called the *window* of the cycle, that contains all possible traversal times of the cycle, independently of any path prefix. That is, starting from the reset location with the clock of the cycle set to zero, traversing the cycle once takes between a and b time units, and for any $t \in [a, b]$, there is a path (of the underlying transition system) through the cycle that takes exactly t time units.

Example 4.97
Consider the following cycle in an ETA:



The cycle is acceleratable, L is the reset location and y is the clock of the cycle. Assume we start in L with $y = 0$. According to the invariant, up to 4 time units can be spent in L. But in order to make the transition to L', y must not be larger than 3, for otherwise

the invariant of L' would be violated. Moreover, the guard $y \geq 1$ is redundant, since y must be at least 2 when going from L' back to L. However, the automaton can stay in L' until y becomes equal to 3 (i.e., at most one time unit). In conclusion, the total time needed to traverse the cycle is in the interval $[2, 3]$, and this interval is determined only by the guard $y \geq 2$ and the invariant $y \leq 3$. Conversely, for any $t \in [2, 3]$, the traversal of the cycle can be scheduled to take exactly t time units. ∎

The cycle in Example 4.97 has a single reset location L. In general, an acceleratable cycle may have more than one reset location, i.e., more than one location whose incoming edges reset the cycle's clock. Such a cycle can be decomposed into maximal segments of the form $(e_k, e_{k+1}, \ldots, e_{k'})$, where the cycle's clock is reset on edge $e_{k'}$, but on none of the edges $e_k, \ldots, e_{k'-1}$. If edge e_i has a guard of the form $y \geq a_i$, and each location $L^{(i)}$ has an invariant of the form $y \leq b_i$ (possibly $b_i = \infty$ to represent the invariant true), then the time required to traverse the segment $(e_k, e_{k+1}, \ldots, e_{k'})$ lies in the interval $[\max\{a_k, a_{k+1}, \ldots, a_{k'}\}, b_{k'}]$. The window of the whole cycle is then $[a, b]$, where a is the sum over all lower bounds and b is the sum over all upper bounds of the individual segments' intervals. More details can be found in [HL02].

Definition 4.98 (Integer-Flat ETA)
An extended timed automaton *ETA* is called *integer-flat* if every strongly connected component of its underlying graph that contains an integer assignment on some edge, is an acceleratable cycle. ∎

In particular, integer-flatness rules out nested cycles with integer assignments, which could be used to encode multiplication.

Given an integer-flat ETA, we fix a topological sorting[19] π of the SCCs in the underlying graph of *ETA*, and let $\pi(L) \in \{1, 2, \ldots\}$ be the index of the SCC that L belongs to. We extend π to positive unit clauses by $\pi(\Lambda \parallel \to R(L, \ldots)) = \pi(L)$.

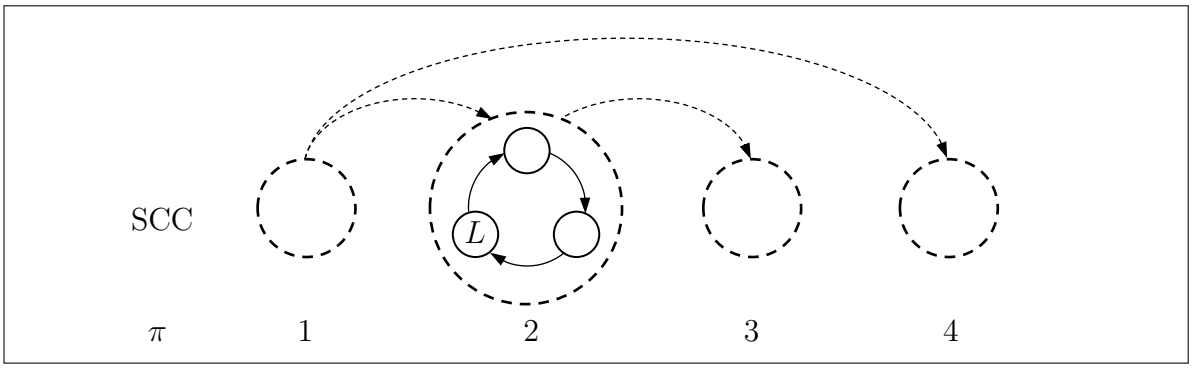


Figure 4.7: Illustration of location numbering with $\pi(L) = 2$

[19] Any directed graph is a DAG of its strongly connected components, and any DAG admits a topological sorting.

We will now see how SUP(LA) with constraint induction can be turned into a decision procedure for reachability in extended timed automata. We fix an extended timed automaton $ETA = (Loc, Act, X, \hookrightarrow, L_0, g_0, Inv)$ for the rest of this section. We start with a reachability query in the backward encoding $N^{\flat}_{ETA}$, where we assume that all clauses have been purified.

Definition 4.99 (C_L, C_e)
We adopt the following notation for the clauses in $N^{\flat}_{ETA}$: We write C_L for the time-reachability clause

$$t{\geq}0,\ \vec{x} \geq 0,\ \vec{x}' \simeq \vec{x}{+}t,\ Inv(L)[\vec{x}'/\vec{x}] \ \parallel\ R(L, \vec{x}', \vec{z}) \to R(L, \vec{x}, \vec{z})$$

corresponding to location $L \in Loc$, and C_e for the discrete-step clause

$$g[\vec{x}, \vec{z}],\ (\vec{x}', \vec{z}') \simeq A(\vec{x}, \vec{z}),\ Inv(L')[\vec{x}'/\vec{x}] \ \parallel\ R(L', \vec{x}', \vec{z}') \to R(L, \vec{x}, \vec{z})$$

corresponding to transition $e = (L, g, \alpha, A, L') \in \hookrightarrow$. ∎

Derivation Strategy

We apply the following derivation strategy:

(i) all negative literals are selected;

(ii) the only inference rules used are

  a) ordered resolution, and

  b) constraint induction by loop detection, using the rules from Propositions 4.90 and 4.92 (together with the strengthening explained in Section 4.6.2) for transitive closure computation;

(iii) forward subsumption deletion is the only reduction rule.[20]

An immediate consequence of the selection strategy is that ordered resolution can only derive positive unit clauses (goal clauses) of the form $\Lambda \parallel \to R(l, \vec{x})$, where l is a location constant or variable. The constraint induction rule derives only binary clauses, which we call *acceleration clauses*. More generally, any derivation following the above strategy is of the form $N^{\flat}_{ETA} = N_0 \triangleright N_1 \triangleright N_2 \triangleright \ldots$ where

$$N_{i+1} = \begin{cases} N_i \cup \{C_i\} & \text{or} \\ N_i \cup \{C_i, C'_i, \ldots\} \end{cases}$$

such that

[20] Forward subsumption is *sufficient* for termination. Other reduction rules can be used as well, as long as they don't interfere with constraint induction, like non-strict backward subsumption—see the discussion on page 116.

- if $N_{i+1} = N_i \cup \{C_i\}$, then C_i is either a goal clause derived by resolution from another goal clause and a binary clause in N_i, or an acceleration clause derived by constraint induction from binary clauses in N_i, and
- if $N_i \cup \{C_i, C_i', \ldots\}$, then $C_i, C_i', \ldots$ are acceleration clauses[21] derived by constraint induction from binary clauses in N_i, and
- neither of $C_i, C_i', \ldots$ is subsumed by any clause in N_i.

We use the notation $C^{(1)} \triangleright C^{(2)} \triangleright C^{(3)} \triangleright \ldots$ to denote a sequence of clauses in a derivation where each $C^{(j+1)}$ is the conclusion of an inference with main premise $C^{(j)}$.

For positive unit clauses C, C', and binary clauses $D_1, \ldots, D_n$, we write

$$C \vdash_{D_1,\ldots,D_n} C'$$

to denote the fact that C' is the result of successively resolving C with $D_1, \ldots, D_n$, i.e., $C' = \mathrm{Res}(\ldots \mathrm{Res}(C, D_1), \ldots, D_n)$.

Relation between Clauses and Sets of States

We now extend the mapping $\mathsf{s}_{\mathcal{T}}$ defined in Section 4.4 to unary and binary FOL(LA) clauses. Here, the theory $\mathcal{T}$ is fixed to be $\mathcal{T}_{ETA}$, so we omit the subscript from $\mathsf{s}_{\mathcal{T}_{ETA}}$. The mapping s is defined as follows:

$$\begin{array}{lcl} \mathsf{s}(\Lambda \parallel \rightarrow R(l, \vec{x})) & = & \{\, \mathsf{s}(\nu, R(l, \vec{x})) \mid \nu \models_{\mathrm{LA}} \Lambda \,\} \\ \mathsf{s}(\Lambda \parallel R(l', \vec{x}') \rightarrow R(l, \vec{x})) & = & \{\, (\mathsf{s}(\nu, R(l', \vec{x}')), \mathsf{s}(\nu, R(l, \vec{x}))) \mid \nu \models_{\mathrm{LA}} \Lambda \,\} \end{array}$$

where l, l' stand for either location constants or variables.[22] Under this mapping, unit clauses represent sets of states of TS(ETA), while binary clauses represent binary relations on states, or, equivalently, functions from sets of states to sets of states.

Proposition 4.100
Consider an arbitrary ordered resolution inference

$$\mathcal{I}\frac{C_1 = (\Lambda_1 \parallel \rightarrow R(l', \vec{x}')) \qquad C_2 = (\Lambda_2 \parallel R(l', \vec{x}') \rightarrow R(l, \vec{x}))}{C_3 = (\Lambda_1 \Lambda_2 \parallel \rightarrow R(l, \vec{x}))}$$

between clauses in or derived from $N_{ETA}^{\flat}$, where the unifier has already been applied. It holds that $\mathsf{s}(C_3) = (\mathsf{s}(C_2))(\mathsf{s}(C_1))$.

[21] Constraint induction based on Propositions 4.90, 4.92 produces at most two acceleration clauses.
[22] Either way, we have $\nu(l) \in \mathit{Loc}$, no matter whether l is a variable of "location sort", or a constant $L \in \mathit{Loc}$.

Proof.

$$\begin{aligned}\mathsf{s}(C_3) &= \{\mathsf{s}(\nu, R(l,\vec{x})) \mid \nu \models_{\text{LA}} \Lambda_1, \Lambda_2\}\\ &= \{\mathsf{s}(\nu, R(l,\vec{x})) \mid (\mathsf{s}(\nu, R(l',\vec{x}')), \mathsf{s}(\nu, R(l,\vec{x}))) \in \mathsf{s}(C_2),\ (\mathsf{s}(\nu, R(l',\vec{x}'))) \in \mathsf{s}(C_1)\}\\ &= (\mathsf{s}(C_2))(\mathsf{s}(C_1)).\end{aligned}$$

□

Proposition 4.101
A clause $C_1 = (\Lambda_1 \parallel \rightarrow R(l,\vec{x}))$ subsumes a clause $C_2 = (\Lambda_2 \parallel \rightarrow R(l,\vec{x}))$ if and only if $\mathsf{s}(C_2) \subseteq \mathsf{s}(C_1)$; it subsumes a clause $C_3 = (\Lambda_3 \parallel R(l',\vec{x}') \rightarrow R(l,\vec{x}))$ if and only if $\text{im}(\mathsf{s}(C_3)) \subseteq \mathsf{s}(C_1)$.

Proof. Straightforward. □

Definition 4.102
We define

$$\text{Pre}_L(Q) = \{s \mid s = (L,\nu), s' = (L,\nu'), s \xrightarrow{\mathsf{time}} s', s' \in Q\}$$
$$\text{Pre}_e(Q) = \{s \mid s = (L,\nu), s' = (L',\nu'), s \xrightarrow{e} s', s' \in Q\}$$

for any $L \in Loc$ and $e = (L, g, \alpha, A, L') \in \hookrightarrow$.

For a cycle $\text{cyc} = e_0, \ldots, e_{n-1}$ over $L^{(0)}, \ldots, L^{(n-1)}$, we define

$$\text{Pre}_{\text{cyc}} = \text{Pre}_{e_{n-1}} \circ \text{Pre}_{L^{(n-1)}} \circ \ldots \circ \text{Pre}_{e_0} \circ \text{Pre}_{L^{(0)}}$$

and $\text{Pre}^{\geq k}_{\text{cyc}} = \bigcup_{i \geq k} \text{Pre}^i_{\text{cyc}}$. ■

It is easy to see that $\mathsf{s}(C_L) = \text{Pre}_L$ and $\mathsf{s}(C_e) = \text{Pre}_e$.

Cycle Acceleration by Constraint Induction

Consider an acceleratable cycle $e_0, \ldots, e_{n-1}$ over $L^{(0)}, \ldots, L^{(n-1)}$ in *ETA*, with reset location $L = L^{(0)}$, and suppose that

$$\Lambda \parallel \rightarrow R(L, \ldots) \quad \vdash_{C_{e_{n-1}},\, C_{L^{(n-1)}},\, C_{e_{n-2}},\, \ldots,\, C_{e_0},\, C_{L^{(0)}}} \quad \Lambda' \parallel \rightarrow R(L, \ldots) \ .$$

Now constraint induction is potentially applicable. ReplayLoop (Algorithm 4.1) returns a constraint equivalent to

$$\exists \vec{x}_1, \ldots, \vec{x}_{2n-1}. \bigwedge_{1 \leq m \leq n} \left(\Lambda_{e_{n-m}}[\vec{x}_{2m-2}, \vec{x}_{2m-1}] \wedge \Lambda_{L^{(n-m)}}[\vec{x}_{2m-1}, \vec{x}_{2m}]\right)$$

where Λ_{e_i} and $\Lambda_{L^{(i)}}$ are the constraints of C_{e_i} and $C_{L^{(i)}}$, respectively. It is straightforward to check that this constraint (after renaming $\vec{x}_0$ to $\vec{x}'$ and $\vec{x}_n$ to $\vec{x}$) is equivalent to

$$y \leq b \ \wedge\ y' \simeq 0 \ \wedge\ \vec{x}' \geq \vec{x} \ \wedge\ a \leq \vec{x}' - \vec{x} + y \leq b \ \wedge\ g[\vec{z}] \ \wedge\ \vec{z}' \simeq A(\vec{z}) \tag{4.6}$$

where y is the clock of the cycle, $\vec{x}$ are the variables in $X_C \setminus \{y\}$, $\vec{z}$ are the variables in X_D, $[a,b]$ is the window of the cycle, and g and A are a linear integer constraint and an instruction, respectively, corresponding to the composition of integer guards and instructions on the cycle (see Remark 4.63). Note that each atomic constraint in the conjunction contains either only clock variables, or only integer variables, and thus the whole constraint describes a product of relations on clock and integer variables. The clock part of the constraint is of the form mentioned in Section 4.6.2 and we strengthen it by $y \simeq 0$, yielding $y \simeq 0 \ \wedge \ y' \simeq 0 \ \wedge \ a \leq \vec{x}' - \vec{x} \leq b$ (which implies $\vec{x}' \geq \vec{x}$) before computing the transitive closure. The integer part of the constraint is of the form required by Proposition 4.92. Hence one or two clauses—depending on g and A—can be derived by constraint induction. We denote these clauses by $C_L^{\geq 1}$ and $C_L^{=1}, C_L^{\geq 2}$, respectively. They have the following form:

$$C_L^{\geq 1}: \quad k \geq 1,\ y \simeq 0,\ y' \simeq 0,\ ak \leq \vec{x}' - \vec{x} \leq bk,\ g^{(k)}[\vec{z}],\ \vec{z}' \simeq A^{(k)}(\vec{z}) \ \parallel \ R(L,\vec{u}') \to R(L,\vec{u})$$

$$C_L^{=1}: \quad y \simeq 0,\ y' \simeq 0,\ a \leq \vec{x}' - \vec{x} \leq b,\ g[\vec{z}],\ \vec{z}' \simeq A(\vec{z}) \ \parallel \ R(L,\vec{u}') \to R(L,\vec{u})$$

$$C_L^{\geq 2}: \quad k \geq 2,\ y \simeq 0,\ y' \simeq 0,\ ak \leq \vec{x}' - \vec{x} \leq bk,\ g^{(k)}[\vec{z}],\ \vec{z}' \simeq A^{(k)}(\vec{z}) \ \parallel \ R(L,\vec{u}') \to R(L,\vec{u})$$

where $\vec{u}$ are all variables in X.

The purpose of the strengthening by $y \simeq 0$ is to ensure that the clock part of the constraints of acceleration clauses is a parametric clock constraint. The transitive closure could also be computed directly from (4.6), without strengthening, yielding

$$k \geq 1,\ y \leq b,\ y' \simeq 0,\ \vec{x}' \geq \vec{x},\ ak \leq \vec{x}' - \vec{x} + y \leq bk, g^{(k)}[\vec{z}],\ \vec{z}' \simeq A^{(k)}(\vec{z}).$$

This constraint is still linear, but it is not a parametric clock constraint, because of the occurrence of three clock variables in $ak \leq \vec{x}' - \vec{x} + y \leq bk$.

While the strengthening ensures that only clock constraints are produced, it weakens the obtained acceleration clauses: For instance, $\mathsf{s}(C_L^{\geq 1})$ does not return *all* predecessor states reachable by any $k \geq 1$ backward traversals of the cycle (like $\mathrm{Pre}_{\mathrm{cyc}}^{\geq 1}$), but only those satisfying $y \simeq 0$. This is not a problem however, because the "lost" information is eventually recovered, when resolution with the clauses $C_{e_{n-1}}, \ \ldots, \ C_{e_0}, \ C_{L^{(0)}}$ of the cycle is performed: If

$$C \ \vdash_{C_L^{\geq 1},\ C_{e_{n-1}},\ \ldots,\ C_{L^{(0)}}} \ C'$$

then

$$\mathsf{s}(C') \ = \ \mathrm{Pre}_{\mathrm{cyc}}^{\geq 2}(\mathsf{s}(C)).$$

Fairness of the derivation ensures that C' is eventually derived. The situation is illustrated in Figure 4.8.

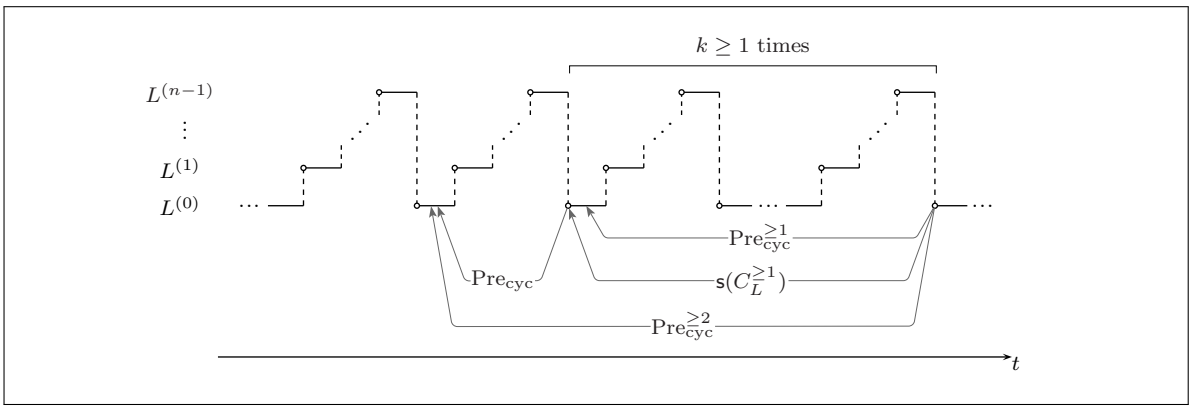


Figure 4.8: Location diagram: Repeated traversal of an acceleratable cycle

Termination of Saturation

We are now ready to show termination of our derivation strategy on $N^{\flat}_{ETA}$. For this we will rely on the notion of κ-closed parametric clock constraints (Definition 4.71) and on the following key invariants:

(I1) Any derived goal clause is of the form

$$\phi_1[\vec{x}, \vec{k}],\ \phi_2[\vec{z}, \vec{k}],\ \phi_3[\vec{k}]\ \parallel\ \rightarrow R(L, \vec{u})$$

where $\vec{x} \subseteq X_C$, $\vec{z} \subseteq X_D$, $\vec{k} \subseteq K$, $\vec{u} = \vec{x} \cup \vec{z}$, and ϕ_1 is a parametric clock constraint, and ϕ_2, ϕ_3 are linear constraints, assuming that any goal clause in $N^{\flat}_{ETA}$ has this form.

(I2) Whenever a unit clause C' is derived from another unit clause C, then $\pi(C') \leq \pi(C)$.

(I1) is established by a straightforward though tedious induction on the derivation. (I2) is an immediate consequence of the shape of binary clauses in the backward encoding.

Proposition 4.103
Let $\Lambda = \phi_1,\ \phi_2,\ \phi_3$ as per Invariant (I1), and let κ be a linear expression over K such that ϕ_1 and all guards and invariants in *ETA* are κ-closed. Consider a resolution step

$$\Lambda \parallel\ \rightarrow R(L, \vec{u})\ \vdash\ \Lambda' \parallel\ \rightarrow R(L', \vec{u})$$

by resolution with some C_e or C_L. Then $\Lambda' = \phi'_1,\ \phi',\ \phi'_2,\ \phi_3$ where

- $\phi'_1 \in CC(X, LE(K))$ is κ-closed, and
- ϕ' is a linear constraint over K that is uniquely determined by ϕ_1 and the guards and invariants in *ETA*, and

- ϕ_2' is the conjunction of ϕ_2 with zero or more integer guards in *ETA*.

Proof. Consider the case $C = C_L$. Then

$$\begin{aligned}\Lambda' &= t \geq 0,\ \vec{x} \geq 0,\ \mathit{Inv}(L)[\vec{x}'/\vec{x}],\ \Lambda[\vec{x}+t/\vec{x}] \\ &= \mathsf{tpre}(\phi_1 \wedge \mathit{Inv}(L)),\ \phi_2,\ \phi_3 \\ &= \mathsf{tpre}(\phi_1 \wedge \mathit{Inv}(L))|_X,\ \phi',\ \phi_2,\ \phi_3\end{aligned}$$

where ϕ' is a linear constraint over K that is uniquely determined[23] by ϕ_1 and $\mathit{Inv}(L)$. By assumption, ϕ_1 and $\mathit{Inv}(L)$ are κ-closed. Hence by Propositions 4.73 and 4.79, $\phi_1' = \mathsf{tpre}(\phi_1 \wedge \mathit{Inv}(L))|_X$ is also κ-closed. Now consider $C = C_e$. In this case,

$$\begin{aligned}\Lambda' &= g[\vec{x},\vec{z}],\ \vec{x}' \simeq A_D(\vec{x}),\ \mathit{Inv}(L')[\vec{x}'/\vec{x}],\ \Lambda[\vec{x}'/\vec{x}] \\ &= g|_X,\ g|_Z,\ [D](\phi_1 \wedge \mathit{Inv}(l')),\ \phi_2,\ \phi_3 \\ &= g|_X,\ [D](\phi_1 \wedge \mathit{Inv}(l'))|_X,\ \phi',\ \phi_2,\ g|_Z,\ \phi_3\end{aligned}$$

where again ϕ' is a linear constraint over K uniquely determined by ϕ_1 and $\mathit{Inv}(L')$. Observe that $g = g|_X \wedge g|_Z$ by definition of $\mathit{Guard}_{\mathrm{ETA}}$.[24] By assumption, g, ϕ_1 and $\mathit{Inv}(L')$ are κ-closed.[25] Hence by Propositions 4.73 and 4.79, $\phi_1' = g \wedge [D](\phi_1 \wedge \mathit{Inv}(l'))|_X$ is also κ-closed. □

Let $\mathcal{D}(N^\flat_{ETA})$ denote the set of all fair derivations from $N^\flat_{ETA}$ in SUP(LA) with constraint induction, following the above derivation strategy.

Lemma 4.104
Consider an acceleratable cycle with reset location L, and let N be a clause set containing the acceleration clauses for L. Then any $\mathcal{D}(N^\flat_{ETA})$-derivation starting in N can produce only finitely many L-clauses.[26]

Proof. Assume for contradiction that $N \ni D_1 \triangleright D_2 \triangleright \ldots$ is an infinite sequence of L-clauses. Then $\mathsf{s}(D_{i+1}) = \mathrm{Pre}_{\mathrm{cyc}}(\mathsf{s}(D_i))$. By fairness, some D_j is resolved with the acceleration clauses, and with the clauses of the cycle, yielding D_j' such that $\mathsf{s}(D_j') = \mathrm{Pre}^{\geq 2}_{\mathrm{cyc}}(\mathsf{s}(D_j))$. The clause D_j' subsumes all clauses $D_{j+2}, D_{j+3}, \ldots$, contradicting the assumption. □

Theorem 4.105
Assume *ETA* is integer-flat. Then all derivations in $\mathcal{D}(N^\flat_{ETA})$ are finite.

Proof. Let $N^\flat_{ETA} = N_0 \triangleright N_1 \triangleright N_2 \triangleright \ldots$ be a derivation in $\mathcal{D}(N^\flat_{ETA})$ and assume for contradiction that the derivation is infinite. By (I2), the derivation must contain an infinite sequence of unit clauses

$$C^{(1)} \triangleright C^{(2)} \triangleright C^{(3)} \triangleright \ldots \tag{4.7}$$

[23] See discussion after Definition 4.74, page 92.
[24] See page 92 for the definition of $g|_X$.
[25] Note that g does not refer to the integer variables $\vec{z}$, since we are in a clock SCC.
[26] An L-clause is a clause of the form $\Lambda \parallel \rightarrow R(L,\vec{x})$.

where $\pi(C^{(i+1)}) = \pi(C^{(i)})$ for all $i \geq 1$, and no $C^{(i)}$ is subsumed by any $C^{(j)}$ with $j < i$. We distinguish two cases, depending on whether the location in $C^{(1)}$ lies on an integer cycle or a clock SCC:

Assume the location in $C^{(1)}$ lies on an integer cycle with reset location $L^{(0)}$. Then (4.7) contains an infinite subsequence of $L^{(0)}$-clauses. By fairness, the acceleration clause for $L^{(0)}$ is derived in some N_i. But Lemma 4.104 implies that there can be no infinite sequence of $L^{(0)}$-clauses starting in N_i, a contradiction.

Now assume $C^{(1)}$ refers to a clock SCC. Then each $C^{(i+1)}$ is derived from $C^{(i)}$ by resolution with some C_e or C_L. Let ϕ_1, ϕ_2, ϕ_3 be the constraint of clause $C^{(1)}$ as per Invariant (I1). By Proposition 4.73 (i) and (iii), there exists a linear expression κ over K such that ϕ_1 and all guards and invariants in ETA are κ-closed: By (iii), each individual constraint is κ'-closed for some κ', and by (i), we can choose κ such that $\kappa' \preceq \kappa$ for all κ'. By Proposition 4.103, the constraint of $C^{(k)}$, $k \geq 1$, is of the form

$$\phi_1^{(k)},\ \phi',\ \phi'',\ \ldots,\ \phi^{(k-1)},\ \phi_2^{(k-1)},\ \phi_3$$

where

- $\phi_1^{(k)}$ is κ-closed,
- $\phi', \phi'', \ldots$ are linear constraints over K that are uniquely determined by κ-closed clock constraints, and
- $\phi_2^{(k-1)}$ is the conjunction of ϕ_2 with integer guards in ETA.

As there are only finitely many non-equivalent κ-closed clock constraints, there can be only finitely many non-equivalent $\phi^{(i)}$; as there are only finitely many integer guards in ETA, there can be only finitely many non-equivalent $\phi_2^{(k-1)}$. Hence, there can be only finitely many non-equivalent $C^{(k)}$, $k \geq 1$, contradicting the assumption that (4.7) is infinite. □

4.6.4 Implementation and Results

We have implemented constraint induction by loop detection together with the transitive closure computation for LA constraints as an extension to the SPASS(LA) theorem prover.

The implementation is based on the constraint induction rule presented in [FKW12], which differs from the rule presented in this section in the fact that the derived acceleration clause also depends on the initial clause of the loop (see Remark 4.87).

The implementation can (and will hopefully soon be) adapted to the improved constraint induction rule without changing the general framework, which we describe in this section.

Another minor difference is that the current implementation, due to limitations of the underlying SPASS(LA) implementation, cannot yet handle arithmetic constants (i.e., parameters) occurring in the constraints of a loop, as would be necessary to handle the Producer-Consumer problem from Example 4.107. We therefore refer to [FKW12] for the experimental evaluation, and limit ourselves to a general discussion of the examples.

Implementation

In order to find a partial derivation

$$\Lambda \parallel C,\ D_1,\ \ldots,\ D_m\ \vdash\ \Lambda' \parallel C.$$

to which constraint induction is potentially applicable, we proceed as follows: Whenever a new clause C with a non-empty constraint has been derived, the term index is queried to find all clauses with same free part as C. Then the ancestors of C are checked in order of decreasing derivation depth to check whether one of the retrieved clauses is an ancestor of C. This traversal is stopped as soon as one of the potential partner clauses has been reached— in which case the constraint induction rule is applied—, or when the minimum of the derivation depths of all potential partner clauses has been reached.

Algorithm 4.2: ConstraintInduction

Input: clause Given, index WorkedOff

```
1 Derived := ∅;
2 Candidates := GetVariants(Given, WorkedOff);
3 foreach Candidate in Candidates do
4     if Derivation := GetDerivation(Candidate,Given) then
5         Λ := ReplayLoop(Derivation);
6         if Λ+ := Closure(Λ) then
7             Derived := Derived ∪ Purify(cnf(Λ+ ∧ Given → Given[x⃗′/x⃗]));
8 return Derived;
```

Algorithm 4.2 shows the implementation of constraint induction by loop detection. The procedure GetVariants returns all variants of the Given clause stored in the Worked-Off index. The procedure GetDerivation checks whether the first clause is an ancestor of the second one, and if so, returns the corresponding partial derivation. The procedure Closure computes the transitive closure of an LA constraint as explained in Section 4.6.2. If the given constraint does not have the required form, Closure returns false.

GetDerivation relies on the assumption that all clauses in the partial derivation of Given from Candidate are still available. This means that redundant clauses cannot in general be deleted, but must be kept in memory.

Additionally, since the Candidate clauses are taken from the WorkedOff set, non-strict backward subsumption should be deactivated. Non-strict backward subsumption removes a clause from the WorkedOff set if a newly derived clause subsumes it non-strictly, implying that the new clause is a variant of the one in WorkedOff. If this subsumption is applied, potential candidate clauses could be deleted from WorkedOff right after the derivation of a clause that would later be selected as Given clause for constraint induction.

Finally, to enable efficient reconstruction of the derivation, each clause now has pointers to its parent clauses, instead of just its parents' clause numbers.

All these requirements increase the memory consumption, and could be avoided if clauses could be (recursively) reconstructed based on information about their origin, as is the case in the Waldmeister [HJL99] prover, for example. Given the current architecture of SPASS(LA), implementing such a system would require a significant programming effort.

Example Problems

The following two examples show how constraint induction can enable termination of saturation for satisfiable clause sets, thereby proving the unreachability of certain (bad) states.

Example 4.106 (Extended timed automaton)
Consider the extended timed automaton in Figure 4.9, where $\vec{x} = x_1, x_2$ are clocks and $\vec{z} = z_1, z_2$ are integer variables, and the initial condition is $x_1 \simeq 0, x_2 \simeq 0, z_1 \simeq 0, z_2 \simeq 10$. We want to check whether location L_2 is reachable with a valuation such that $z_1 \geq z_2$ and $x_2 < 12$. Since x_2 is never reset to zero, its value represents the total time elapsed since first entering L_1. As the cycle at L_1 must be traversed four times before z_1 has overtaken z_2, and each cycle traversal takes at least three time units, such a state is not reachable. The backward encoding of this ETA together with the negated conjecture

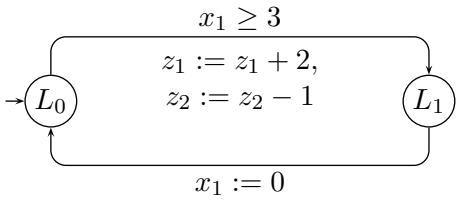


Figure 4.9: An extended timed automaton

(last clause) is

$$\begin{aligned}
\vec{x} \simeq 0,\ z_1 \simeq 0,\ z_2 \simeq 10 \ &\|\ R(L_0, \vec{x}, \vec{z}) \ \rightarrow \\
\vec{x} \geq 0,\ t \geq 0,\ \vec{x}' \simeq \vec{x} + t \ &\|\ R(L_0, \vec{x}', \vec{z}) \rightarrow R(L_0, \vec{x}, \vec{z}) \\
x_1 \geq 3,\ z_1 \simeq z_1 + 2,\ z_2 \simeq z_2 - 1 \ &\|\ R(L_1, \vec{x}, \vec{z}') \rightarrow R(L_0, \vec{x}, \vec{z}) \\
\vec{x} \geq 0,\ t \geq 0,\ \vec{x}' \simeq \vec{x} + t \ &\|\ R(L_1, \vec{x}', \vec{z}) \rightarrow R(L_1, \vec{x}, \vec{z}) \\
x_1' \simeq 0,\ x_2' \simeq x_2 \ &\|\ R(L_0, \vec{x}', \vec{z}) \rightarrow R(L_1, \vec{x}, \vec{z}) \\
z_1 \geq z_2,\ x_2 < 12 \ &\|\ \rightarrow R(L_1, \vec{x}, \vec{z})
\end{aligned}$$

The clause set is satisfiable. Without the constraint induction rule, saturation does not terminate. With constraint induction activated, the acceleration clause

$$k \geq 1,\ x_1 \simeq 0,\ x_1' \simeq 0,\ x_1' - x_1 \geq 3k,\ z_1' \simeq z_1 + 2k,\ z_2' \simeq z_2 - k \,\|\, L_0(\vec{x}', \vec{z}') \rightarrow L_0(\vec{x}, \vec{z})$$

is derived as soon as the cycle has been traversed once, enabling the termination of saturation. ∎

Example 4.107 (Parametric Producer-Consumer Problem)
Figure 4.10 depicts a simple model of the producer-consumer problem, consisting of a producer P and a consumer C sharing a common buffer of capacity $\text{cap} > 0$. Here the buffer is modelled as a shared integer variable x representing the number of elements in the buffer, initially zero. The producer can add an element to the buffer if it is not full, and the consumer can remove an element from the buffer if it is not empty. The

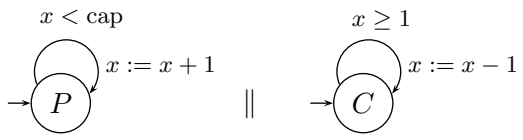


Figure 4.10: Producer and Consumer

combined system is encoded by the following three clauses:

$$\begin{aligned}
x \simeq 0 \ &\|\ \rightarrow S(x) \\
x < \text{cap},\ x' \simeq x + 1 \ &\|\ S(x) \rightarrow S(x') \\
x \geq 1,\ x' \simeq x - 1 \ &\|\ S(x) \rightarrow S(x')
\end{aligned}$$

where cap is a parameter. The second and third clauses give rise to loops which can be accelerated by constraint induction, yielding the clauses

$$n \geq 1,\ x + n - 1 < \text{cap},\ x' \simeq x + n \ \|\ S(x) \rightarrow S(x')$$

and

$$n \geq 1,\ x - n + 1 \geq 1,\ x' \simeq x - n \ \|\ S(x) \rightarrow S(x')$$

respectively.[27] It can now be checked by finite saturation that the buffer can never overflow, by adding the conjecture

$$x \simeq 0,\ x' > \mathrm{cap} \ \parallel \ S(x) \to S(x') \ .$$

Similarly, the impossibility of buffer underflow can be checked by adding the conjecture

$$x \simeq 0,\ x' < 0 \ \parallel \ S(x) \to S(x') \ .$$

Without constraint induction, saturation of these clause sets does not terminate. ■

The next example shows that the induction rule is also useful for speeding up proof search and finding shorter proofs in the case of unsatisfiable clause sets.

Example 4.108 (Water tank controller)
In this example we consider a family of problems parameterized by natural numbers $c_{\mathsf{maxin}}, c_{\mathsf{out}}$ and c_{thresh}.[28] Figure 4.11 depicts a water tank controller [AKW09] monitoring the water level x in a water tank. There is a non-controllable inflow from the outside of the system that adds at most c_{maxin} units per cycle to the water tank, and a controllable outflow valve that can reduce the content of the tank by c_{out} units per cycle. The controller is meant to keep the level of the water tank below $c_{\mathsf{thresh}} + c_{\mathsf{maxin}}$ units, provided the initial level is at most $c_{\mathsf{thresh}} + c_{\mathsf{maxin}}$. The problem instance in [AKW09] has fixed values $c_{\mathsf{thresh}} = 200$, $c_{\mathsf{maxin}} = 40$ and $c_{\mathsf{out}} = 40$.

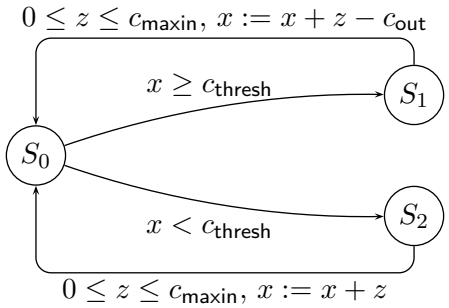


Figure 4.11: Water tank controller

The system is modelled by the following clauses:

$$\begin{aligned}
x \simeq 0 \ &\parallel \ &&\to S_0(x) \\
x \ge c_{\mathsf{thresh}} \ &\parallel \ && S_0(x) \to S_1(x) \\
x < c_{\mathsf{thresh}} \ &\parallel \ && S_0(x) \to S_2(x) \\
-c_{\mathsf{out}} \le x' - x \le c_{\mathsf{maxin}} - c_{\mathsf{out}} \ &\parallel \ && S_1(x) \to S_0(x') \\
0 \le x' - x \le c_{\mathsf{maxin}} \ &\parallel \ && S_2(x) \to S_0(x') \ .
\end{aligned}$$

[27] The acceleration clauses are computed based on Proposition 4.92, so no assumption on cap is needed.
[28] $c_{\mathsf{maxin}}, c_{\mathsf{out}}$ and c_{thresh} are natural numbers, not parameters.

Consider the negated conjecture

$$x \simeq c_{\mathsf{thresh}} \parallel S_0(x) \rightarrow$$

which, when added to the above clauses, yields an unsatisfiable clause set, since the water level c_{thresh} is indeed reachable. The number of clauses derived to establish unsatisfiability, and thus also the length of the reachability proof, directly depends on the ratio of c_{thresh} and c_{maxin}, since without constraint induction, the shortest possible proof consists of $c_{\mathsf{thresh}}/c_{\mathsf{maxin}}$ traversals of the cycle $S_0 \rightarrow S_2 \rightarrow S_0$.

This loop is represented by the third and fifth clauses, and it can be accelerated by constraint induction, yielding the clause

$$k \geq 1,\ x < c_{\mathsf{thresh}},\ x' \leq c_{\mathsf{thresh}} + c_{\mathsf{maxin}},\ 0 \leq x' - x \leq c_{\mathsf{maxin}} k \ \parallel \ S_0(x) \rightarrow S_0(x')$$

Resolving this clause with the first clause and the conjecture yields the empty clause

$$k \geq 1,\ 0 < c_{\mathsf{thresh}},\ 0 \leq c_{\mathsf{maxin}},\ 0 \leq c_{\mathsf{thresh}} \leq c_{\mathsf{maxin}} k \ \parallel \ \square\ .$$

The constraint is satisfiable, thus we have a proof of constant length, independently of the constants c_{thresh} and c_{maxin}. ∎

4.7 First-Order Probabilistic Timed Automata

In the preceding parts of this chapter, we have seen how superposition modulo linear arithmetic can be used as a (decision) procedure for reachability in timed systems, by encoding the system's transition relation into FOL(LA).

For the (extended) timed automata we have considered until now, a relatively restricted fragment of FOL(LA) was sufficient. But the SUP(LA) calculus is complete for much more expressive fragments of FOL(LA), namely any class of sufficiently complete clause sets. It is therefore a natural question to ask to what extent the reachability-by-saturation approach developed so far can be applied to modelling formalisms richer than (extended) timed automata.

In this section, we give one answer to this question, by developing the model of *first-order probabilistic timed automata*, or FPTA, which generalize probabilistic timed automata (PTA) [Jen96, KNSS02], themselves a combination of probabilistic and timed automata, by first-order background theories (Section 4.7.2).

Extending the reachability-by-saturation approach to FPTA poses two new challenges: On the one hand, the resulting reachability theories are no longer trivial enrichments of LA, but in addition have to accommodate the automaton's first-order background theory. We will make use of the results established in Section 4.3.5 to address this challenge, by placing suitable restrictions on FPTA background theories.

On the other hand, probabilistic systems give rise to the notion of *quantitative* reachability properties. While *qualitative* reachability—which we have dealt with until now—concerns the question whether a set of states is reachable or not, quantitative reachability asks about the probability of reaching certain states. In order to compute reachability probabilities, it is no longer sufficient to consider a single path from an initial state to a goal state—corresponding to a reachability proof in our setting—but several (and for completeness even all) such paths need to be enumerated. We will use a clause labelling scheme similar to the one of Chapter 3 to enumerate reachability proofs.

For the final step from reachability proofs to reachability probabilities, we will rely on existing model checking techniques for PTA, by "instantiating" the FPTA with the reachability proofs in order to obtain a finite PTA. Since probabilistic timed systems exhibit a combination of probabilistic and non-deterministic behavior, one distinguishes between *maximum* and *minimum* reachability probabilities, reflecting the possible ways in which non-determinism can be resolved. The original FPTA and the constructed PTA will agree on maximal reachability probabilities, enabling the verification of safety properties of the FPTA (see Section 4.7.3).

To illustrate the idea of instantiating an automaton using constraints, consider Figure 4.12, showing part of an FPTA (left) with a clock variable x and an integer variable y. The guard $P(y, z)$ is a first-order atom and contains an *auxiliary* variable z used for

binding. Assuming that $z \leq 2 \to P(z, 1)$ and $z > 2 \to P(z, 2)$ hold in the background theory, the instantiation yields a PTA, the corresponding part of which is shown on the right.

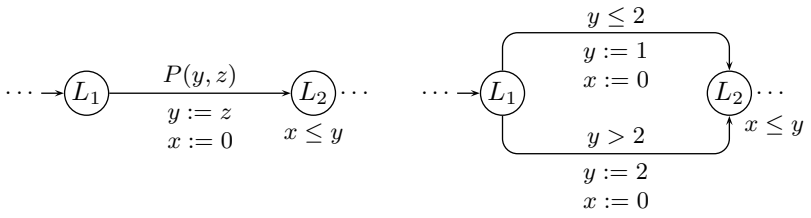


Figure 4.12: Instantiating an FPTA (left) into a PTA (right)

State variables do not have to be of arithmetic sort, but can have arbitrary sorts, allowing, e.g., the representation of messages as terms. A more involved example will be presented in Section 4.7.7.

4.7.1 Preliminaries

Definition 4.109 (Discrete Probability Distributions)
A *discrete probability distribution* over a countable set Q is a function $\mu : Q \to [0, 1]$ such that $\sum_{q \in Q} \mu(q) = 1$. We denote the set of discrete probability distributions over Q by $\mathit{Dist}(Q)$. The *support* of a distribution $\mu \in \mathit{Dist}(Q)$, written as $\mathrm{supp}(\mu)$, is the largest set $Q' \subseteq Q$ such that $\mu(q) > 0$ for all $q \in Q'$. We call μ *finite* if $\mathrm{supp}(\mu)$ is finite. The *point distribution* for $q \in Q$, written as $\circledast(q)$, is the distribution with $\mathrm{supp}(\circledast(q)) = \{q\}$. ■

Definition 4.110 (Markov Decision Processes)
A *Markov decision process* (MDP) is a tuple $M = (S, \mathit{Act}, \twoheadrightarrow, s_0)$ consisting of a set S of states, a set Act of actions, a nondeterministic-probabilistic transition relation $\twoheadrightarrow \subseteq S \times \mathit{Act} \times \mathit{Dist}(S)$, and an initial state[29] $s_0 \in S$. We always require M to be *action-deterministic*: For every $s \in S$ and $\alpha \in \mathit{Act}$, there is at most one μ with $(s, \alpha, \mu) \in \twoheadrightarrow$. ■

We write $s \overset{\alpha}{\twoheadrightarrow} s'$ to mean that $(s, \alpha, \mu) \in \twoheadrightarrow$ for some (unique) μ with $\mu(s') > 0$. Because of action-determinism, we can also view the transition relation of an MDP as a transition probability function $\mathbf{P} : S \times \mathit{Act} \times S \to [0, 1]$ defined as

$$\mathbf{P}(s, \alpha, s') = \begin{cases} \mu(s') & \text{if } (s, \alpha, \mu) \in \twoheadrightarrow, \\ 0 & \text{otherwise.} \end{cases}$$

[29] MDP are often defined with an initial distribution $\mu_0 \in \mathit{Dist}(S)$. Requiring a single initial state does not restrict expressivity, as the initial distribution can always be achieved by a corresponding transition from the initial state.

For technical reasons, we also introduce MDP with update labels:

Definition 4.111 (Markov Decision Process with Update Labels)
An *Markov decision process with update labels* is a tuple $M = (S, Act, U, \twoheadrightarrow, s_0)$ where S and Act are sets of states and actions, respectively, U is a set of *update labels*, and $\twoheadrightarrow \subseteq S \times Act \times Dist(U \times S)$, and $s_0 \in S$ is the initial state. As for standard MDP, we always require M to be *action-deterministic*: For every $s \in S$ and $\alpha \in Act$, there is at most one μ with $(s, \alpha, \mu) \in \twoheadrightarrow$. ∎

We write $s \overset{\alpha,u}{\twoheadrightarrow} s'$ to mean that $(s, \alpha, \mu) \in \twoheadrightarrow$ for some (unique) μ with with $\mu(u, s') > 0$. The transition probability function $\mathbf{P} : S \times Act \times S \to [0, 1]$ for an MDP with update labels is defined as

$$\mathbf{P}(s, \alpha, s') = \begin{cases} \sum_{u \in U} \mu(u, s') & \text{if } (s, \alpha, \mu) \in \twoheadrightarrow , \\ 0 & \text{otherwise.} \end{cases}$$

An MDP with update labels is straightforwardly transformed into a standard MDP by dropping the update labels and replacing every $\mu \in Dist(U \times S)$ by μ' with $\mu'(s) = \sum_{u \in U} \mu(u, s)$. When talking about reachability probabilities in an MDP with update labels, we always mean the respective probabilities in the associated standard MDP.

To simplify notation, we omit the update label from a point distribution, i.e., we identify $(s, \alpha, \circledast((u, s'))) \in \twoheadrightarrow$ with $(s, \alpha, \circledast(s'))$ and write $s \overset{\alpha}{\twoheadrightarrow} s'$ instead of $s \overset{\alpha,u}{\twoheadrightarrow} s'$.

Definition 4.112 (Underlying Transition System of an MDP)
Let $M = (S, Act, \twoheadrightarrow, s_0)$ be an MDP. The *underlying transition system* of M is $\mathrm{TS}(M) = (S, Act, \to, s_0)$ where $s \overset{\alpha}{\to} s'$ if and only if $s \overset{\alpha}{\twoheadrightarrow} s'$. If $M = (S, Act, U, \twoheadrightarrow, s_0)$ is an MDP with update labels, then its underlying transition system is $\mathrm{TS}(M) = (S, Act', \to, s_0)$ with $Act' = Act \times U$ and $s \overset{\alpha,u}{\to} s'$ if and only if $s \overset{\alpha,u}{\twoheadrightarrow} s'$. A (finite or infinite) *path* in an MDP is a path in its underlying transition system. ∎

When M is an MDP (with or without update labels), we write Pre_M and Post_M for $\mathrm{Pre}_{\mathrm{TS}(M)}$ and $\mathrm{Post}_{\mathrm{TS}(M)}$, respectively.

Reachability Probabilities in MDP Markov decision processes extend Markov chains with nondeterministic choice. Computation in an MDP in state s takes place by first nondeterministically selecting an action $\alpha \in Act$ such that $(s, \alpha, \mu) \in \twoheadrightarrow$ and then choosing a destination state s' with probability $\mu(s') > 0$. Different nondeterministic choices thus induce different reachability probabilities. This is captured by the notion of *scheduler* (or *adversary*, or *strategy*), a function which takes as input a finite path from s_0 to s, and outputs an action $\alpha \in Act$ with $(s, \alpha, \mu) \in \twoheadrightarrow$. Given an MDP M, a scheduler $\mathfrak{S}$ induces a Markov chain $M^{\mathfrak{S}}$ whose states are finite paths of M. A probability measure $\Pr^{\mathfrak{S}}$ can then be defined for $M^{\mathfrak{S}}$ in a standard way, see [BK08] for details. The *maximum*

and *minimum reachability probability* in M for a set of target states $B \subseteq S$, starting from $s \in S$, are then respectively defined as

$$\Pr^{\max}(s \models \Diamond B) \;=\; \sup_{\mathfrak{S}} \Pr^{\mathfrak{S}}(s \models \Diamond B)$$

and

$$\Pr^{\min}(s \models \Diamond B) \;=\; \inf_{\mathfrak{S}} \Pr^{\mathfrak{S}}(s \models \Diamond B)$$

where $\mathfrak{S}$ ranges over all possible schedulers. Fortunately, minimal and maximal reachability probabilities are already attained by *memoryless* schedulers, whose decision on what action to take next only depends on the current state, and not on the path leading to it. This allows minimal and maximal probabilities to be characterized as the unique solutions of a suitable equation system. In particular, the maximal reachability probability $\Pr^{\max}(s \models \Diamond B)$ is given by the unique solution $(x_s)_{s \in S}$ to the following equation system[30] [BK08]:

(i) If $s \in B$, then $x_s = 1$.

(ii) If $\mathrm{Post}^*(s) \cap B = \emptyset$, then $x_s = 0$.

(iii) If $s \notin B$ and $\mathrm{Post}^*(s) \cap B \neq \emptyset$, then

$$x_s = \max \Big\{ \sum_{t \in S} \mathbf{P}(s, \alpha, t) \cdot x_t \mid \alpha \in \mathit{Act} \Big\} .$$

Probabilistic Timed Automata Probabilistic timed automata (PTA) [Jen96, KNSS02] combine discrete probabilistic choice, real time and nondeterminism. They arise as the natural orthogonal combination of probabilistic [Seg02] and timed automata. They can also be thought of as extending Markov decision processes with the ability of resetting and testing the value of clocks [Spr04].

Here we assume that PTA can have discrete-valued variables with finite domains in addition to clock variables, and that the language of transition guards is extended accordingly. Such PTA are sometimes called VPTA [HH09]. This generalization facilitates modelling, but does not increase expressivity, as any VPTA can be transformed into a standard PTA by encoding the values of the extra variables in the locations and substituting their values in all expressions where they occur. We therefore refer to VPTA simply as PTA. We also assume that, as we did for (E)TA, each state variable in X has an associated sort declaration $x : S$, where S is an arithmetic sort ($\mathbb{R}$ or $\mathbb{N}$), or some finite set of values. For simplicity, we assume that the domains of discrete variables consist of integers. We abuse the notation *Val* to denote the set of assignments mapping each variable to a value of the appropriate sort.

[30] We don't need the notion of *enabled actions* [BK08] because by definition, $\sum_{t \in S} \mathbf{P}(s, \alpha, t) \in \{0, 1\}$ for all $\alpha \in \mathit{Act}$.

Definition 4.113 (PTA Guards and Instructions)
Let X be a set of variables with subsets X_C, X_D of real-valued variables and integer-valued variables, respectively.

The $Guard_{\mathrm{PTA}}(X)$ set of *PTA-guards* over X consists of conjunctions of clock constraints over X_C and Boolean combinations of atomic constraints over X_D, of the form $t_1 \circ t_2$, where t_1, t_2 are LA terms over X_D and $\circ \in \{<, \leq, \simeq, \geq, >\}$.

The set $Instr_{\mathrm{PTA}}(X) \subseteq Instr_{\mathrm{LA}}(X)$ of *PTA-instructions* contains all instructions which map variables in X_D to LA terms over X_D. ∎

Definition 4.114 (Probabilistic Timed Automata)
A *probabilistic timed automaton* (PTA) is a tuple $(Loc, Act, X, \hookrightarrow, L_0, \nu_0, Inv)$ where Loc is a finite set of locations with initial location $L_0 \in Loc$, Act is a finite set of actions, $X = X_C \uplus X_D$ is a finite set of clocks and discrete variables, respectively,

$$\hookrightarrow \subseteq Loc \times Guard_{\mathrm{PTA}}(X) \times Act \times Dist(Instr_{\mathrm{PTA}}(X) \times Loc)$$

is the *probabilistic edge relation*, $\nu_0 \in Val(X)$ is the initial valuation with $\nu_0(x) = 0$ for all $x \in X_C$, and

$$Inv : Loc \to CC(X_C, X_D)$$

is a function assigning invariants to locations. ∎

The semantics of a PTA is defined in terms of a Markov decision process:

Definition 4.115 (Semantics of Probabilistic Timed Automata)
Let $PTA = (Loc, Act, X, \hookrightarrow, L_0, \nu_0, Inv)$ be a PTA. The semantics of PTA is the Markov decision process $\mathrm{MDP}(PTA) = (S, Act', \twoheadrightarrow, s_0)$ with $S = Loc \times Val(X)$, $Act' = Act \cup \mathbb{R}_{\geq 0}$, and $\twoheadrightarrow \subseteq S \times Act' \times Dist(S)$ being the smallest relation such that

(i) $(s, \delta, \circledast(s')) \in \twoheadrightarrow$ if $s = (L, \nu)$, $s' = (L, \nu')$, $\delta \in \mathbb{R}_{\geq 0}$ and $\nu'(x) = \nu(x) + \delta$ for $x \in X_C$ and $\nu'(x) = \nu(x)$ otherwise, and $\nu' \models Inv(L)$;

(ii) $(s, \alpha, \dot{\mu}) \in \twoheadrightarrow$ if $s = (L, \nu)$ and there is $(L, g, \alpha, \mu) \in \hookrightarrow$ such that $\nu \models g$ and $A(\nu) \models Inv(L')$ for all $(A, L') \in \mathrm{supp}(\mu)$, and $\dot{\mu}$ is defined by

$$\dot{\mu}(L', \nu') = \sum_{A:\, \nu' = A(\nu)} \mu(A, L') \ . \qquad ∎$$

The summation in the definition of discrete transitions (ii) is needed for the case where multiple assignments may result in the same target state (L', ν').

4.7.2 First-Order Probabilistic Timed Automata

We will now extend PTA with a first-order background theory $\mathcal{T}$, first-order state variables and first-order transition guards, resulting in *first-order probabilistic timed automata* (FPTA). In order to keep FPTA amenable to analysis (which will be presented in Section 4.7.3), we make the following assumptions about the background theory $\mathcal{T}$:

(i) $\mathcal{T}$ is Horn modulo LA (see Definition 4.29) and hence has a unique minimal Herbrand model $\mathcal{I}_\mathcal{T}$;

(ii) $\mathcal{T}$ is a non-equational extension of LA, hence any distinct non-base ground terms are different under $\mathcal{T}$, and thus there is a bijection between ground substitutions and assignments.

We call any theory satisfying these assumptions an *FPTA background theory*.

Condition (ii) is required for the construction of a max-reachability equivalent PTA, which will be defined in Section 4.7.3. The reason is that, while guards in FPTA are evaluated with respect to the theory $\mathcal{T}$, the guards in a PTA are evaluated with respect to LA only. In particular, distinct non-base ground terms occurring in a PTA are treated as distinct constants (from some finite variable domain). Since we cannot in general assume that ground terms in SUP(LA) derivations are minimal (with respect to some reduction ordering), condition (ii) makes sure that distinct non-base ground terms are indeed different under $\mathcal{T}$. This will become more clear when we define the PTA P_E in Definition 4.131. The restriction can be dropped if one is only interested in qualitative reachability, because then a single reachability proof (or a finite saturation of the reachability theory without an empty clause) is already conclusive, and no PTA needs to be constructed.

Definition 4.116 ($\mathcal{T}$-guards, $\mathcal{T}$-instructions)
A *$\mathcal{T}$-guard* over $X \subseteq \mathcal{X}$ is formed according to the grammar

$$g ::= \ cc \ \mid \ l \ \mid \ g \wedge g$$

where $cc \in CC(X_C, X_D)$ and l is a $\Sigma_\mathcal{T}$-literal over $\mathcal{X}$, which may be negative only if $\mathcal{T}$ has a unique Herbrand model.[31] The set of all $\mathcal{T}$-guards over X is denoted by $Guard_\mathcal{T}(X)$.

A *$\mathcal{T}$-instruction* over $X \subseteq \mathcal{X}$ is a substitution $A : X \to T_{\Sigma_\mathcal{T}}(\mathcal{X})$ such that $A(x) \in \{0, x\}$ if $x \in X_C$. The set of all $\mathcal{T}$-instructions over X is denoted by $Instr_\mathcal{T}(X)$. Given $A \in Instr_\mathcal{T}(X)$, we define the *auxiliary variables* and the *argument variables* of A as

$$\begin{aligned} \mathrm{auxvar}(A) &= \mathrm{var}(\mathrm{cdom}(A)) \setminus X \\ \mathrm{argvar}(A) &= (\mathrm{var}(\mathrm{cdom}(A)) \setminus \mathrm{cdom}(A)) \cap X \end{aligned}$$

[31] See Definition 4.32.

respectively.

A $\mathcal{T}$-instruction A is *closed* with respect to a $\mathcal{T}$-guard g if $\text{auxvar}(A) \subseteq \text{var}(g)$. ∎

The intuition behind auxiliary and argument variables is that auxiliary variables are all variables that occur on the right-hand side of an assignment but are not state variables, while argument variables are states variables that occur below a function symbol on the right-hand side of an assignment.

Definition 4.117 (First-Order Probabilistic Timed Automata)
A *first-order probabilistic timed automaton* (FPTA) is a tuple $(\mathcal{T}, Loc, Act, X, \hookrightarrow, L_0, g_0, Inv)$ where $\mathcal{T}$ is an FPTA background theory, Loc is a finite set of locations, with initial location $L_0 \in Loc$, Act is a finite set of actions, $X = X_C \uplus X_D$ is a finite set of clocks and discrete variables, respectively,

$$\hookrightarrow \subseteq Loc \times Guard_{\mathcal{T}}(X) \times Act \times Dist(Instr_{\mathcal{T}}(X) \times Loc)$$

is the *probabilistic edge relation*, $g_0 \in Guard_{\mathcal{T}}(X)$ is the initial condition, and

$$Inv : Loc \to CC(X_C, X_D)$$

is a function assigning invariants to locations. We additionally require that

(i) g_0 is a conjunction of equations,[32] one for each $x \in X$, and in particular contains $x \simeq 0$ for each $x \in X_C$, and

(ii) for every $(L, g, \alpha, \mu) \in \hookrightarrow$ and all A, L' with $\mu(A, L') > 0$, A is closed with respect to g.

The sets of auxiliary and argument variables for $p = (L, g, \alpha, \mu) \in \hookrightarrow$ are defined as

$$\text{auxvar}(p) = \bigcup_{(A,L') \in \text{supp}(\mu)} \text{auxvar}(A)$$
$$\text{argvar}(p) = \bigcup_{(A,L') \in \text{supp}(\mu)} \text{argvar}(A) \ .$$

respectively.

An FPTA is *action-deterministic* if, for every $L \in Loc$ and $\alpha \in Act$, there is at most one $(L, g, \alpha, \mu) \in \hookrightarrow$. ∎

For convenience, we write

- $L \overset{g:\alpha,\mu,A}{\hookrightarrow} L'$ if $(L, g, \alpha, \mu) \in \hookrightarrow$ with $\mu(A, L') > 0$,
- $L \overset{g:\alpha,\mu}{\hookrightarrow} L'$ if $L \overset{g:\alpha,\mu,A}{\hookrightarrow} L'$ for some A,

[32] So g_0 describes a unique assignment.

- $L \overset{g:\alpha,A}{\hookrightarrow} L'$ if $L \overset{g:\alpha,\mu,A}{\hookrightarrow} L'$ for some μ,
- $L \overset{\alpha,A}{\hookrightarrow} L'$ if $L \overset{g:\alpha,\mu,A}{\hookrightarrow} L'$ for some g and μ.

An FPTA is *well-formed*, if every enabled edge can be taken without violating the invariant of any target location. Just as PTA, any FPTA can be transformed into a well-formed one by strengthening the guard on every probabilistic edge with a condition ensuring that the invariants of all target locations will be satisfied after taking the corresponding transition, see [KNSW07]. Since this transformation has no effect on the semantics of the automaton, we assume all FPTA to be well-formed from now on.

Parallel Composition for FPTA

To facilitate higher-level modelling of complex systems, it is often useful to describe them as the parallel composition of several independently specified but interacting components. To this end, we provide a parallel composition operator on FPTA via synchronization on shared actions. The parallel composition operator is based on the parallel composition operator for PTA [KNPS06], but with the added possibility for the components to exchange messages. Message exchange is implemented by giving component automata the possibility to read other components' state variables (in a way similar to, e.g., the language of the PRISM model checker [KNP11]). To make this possible, we allow component FPTA inside a parallel composition to violate condition (ii) of Definition 4.117, i.e., the codomain of an assignment may refer to variables that do not occur in the guard, as long as these variables are state variables of other component automata. The final product automaton, on the other hand, must satisfy the condition, as the semantics of FPTA is only defined when all assignments are closed.

For simplicity, we assume all components in a parallel composition to have the same background theory. Several background theories can be combined into a single one, following the approach from Section 4.3.5.

Definition 4.118 (Parallel Composition of FPTA)
Let $P_i = (\mathcal{T}, Loc_i, Act_i, X_i, \hookrightarrow_i, L_{0,i}, g_{0,i}, Inv_i)$, $i \in \{1,2\}$ be two FPTA with $X_1 \cap X_2 = \emptyset$. The *parallel composition* of P_1 and P_2 is defined as

$$P_1 \parallel P_2 = (\mathcal{T}, Loc_1 \times Loc_2, Act_1 \cup Act_2, X_1 \cup X_2, \hookrightarrow, (L_{0,1}, L_{0,2}), (g_{0,1} \wedge g_{0,2}), Inv)$$

where for all $(L_1, L_2) \in Loc_1 \times Loc_2$ we have

$$Inv(L_1, L_2) = Inv_1(L_1) \wedge Inv_2(L_2)$$

and $((L_1, L_2), g, \alpha, \mu) \in \hookrightarrow$ if and only if

(i) $\alpha \in Act_1 \setminus Act_2$ and $(L_1, g, \alpha, \mu_1) \in \hookrightarrow_1$ and $\mu = \mu_1 \otimes \circledast(\emptyset, L_2)$, or

(ii) $\alpha \in Act_2 \setminus Act_1$ and $(L_2, g, \alpha, \mu_2) \in \hookrightarrow_2$ and $\mu = \circledast(\emptyset, L_1) \otimes \mu_2$, or
(iii) $\alpha \in Act_2 \cap Act_1$ and $(L_i, g_i, \alpha, \mu_i) \in \hookrightarrow_i$ for $i \in \{1, 2\}$ and $g = g_1 \wedge g_2$ and $\mu = \mu_1 \otimes \mu_2$

where $\otimes$ is the product operation on probability distributions, in this case

$$(\mu_1 \otimes \mu_2)(A_1 \cup A_2, (L_1, L_2)) = \mu_1(A_1, L_1) \cdot \mu_2(A_2, L_2) \ . \qquad \blacksquare$$

It would also be possible to add message passing over channels to FPTA, like for channel systems [BK08], but we leave this as future work.

Semantics of FPTA

The extra variables in FPTA guards, which are not state variables (like the variable z in the introductory example at the beginning of Section 4.7), produce additional non-determinism at the semantic level, because the background theory can "choose" to instantiate them in different ways. As we will define the semantics of FPTA in terms of MDP—which we require to be action-deterministic (see Definition 4.110)—the set of actions has to be extended when mapping an FPTA to its corresponding MDP. For this reason we introduce the set $Act_{\mathcal{T}}$ of actions indexed by $\mathcal{T}$-assignments:

Definition 4.119 ($Act_{\mathcal{T}}$)
Given a set Act of actions and a theory $\mathcal{T}$, we define the set

$$Act_{\mathcal{T}} = \{\alpha_\nu \mid \alpha \in Act, \nu \in Val_{\mathcal{T}}(Y), Y \subseteq \mathcal{X}\}$$

of actions indexed by $\mathcal{T}$-assignments. $\blacksquare$

The semantics of an FPTA is defined in terms of a Markov decision process with update labels—as opposed to PTA semantics (Definition 4.115), where we did not require update labels:

Definition 4.120 (Semantics of First-Order Probabilistic Timed Automata)
Let $P = (\mathcal{T}, Loc, Act, X, \hookrightarrow, L_0, g_0, Inv)$ be an FPTA. The semantics of P is the MDP $\mathrm{MDP}(P) = (S, Act', U, \twoheadrightarrow, s_0)$ with $S = Loc \times Val_{\mathcal{T}}(X)$, $s_0 = (L_0, \nu_0)$ with ν_0 being the unique valuation satisfying g_0, $Act' = Act_{\mathcal{T}} \cup \mathbb{R}_{\geq 0}$, update labels $U = Instr_{\mathcal{T}}(X)$, and $\twoheadrightarrow \subseteq S \times Act' \times Dist(U \times S)$ being the smallest relation such that

(i) $(s, \delta, \circledast(s')) \in \twoheadrightarrow$ if $s = (L, \nu)$, $s' = (L, \nu')$, $\delta \in \mathbb{R}_{\geq 0}$ and $\nu'(x) = \nu(x) + \delta$ for $x \in X_C$ and $\nu'(x) = \nu(x)$ otherwise, and $\mathcal{I}_{\mathcal{T}}, \nu' \models Inv(L)$;

(ii) $(s, \alpha_{\nu_{\mathrm{aux}}}, \mu_{\nu_{\mathrm{aux}}}) \in \twoheadrightarrow$ if $s = (L, \nu)$ and there is $p = (L, g, \alpha, \nu) \in \hookrightarrow$ and $\nu_{\mathrm{aux}} \in Val_{\mathcal{T}}(\mathrm{auxvar}(p))$, $\nu'' \in Val_{\mathcal{T}}(\mathrm{var}(g) \setminus (X \cup \mathrm{auxvar}(p)))$ such that $\nu \cup \nu_{\mathrm{aux}} \cup \nu'' \in \mathrm{Sol}_{\mathcal{T}}(g)$, and for all $(A, L') \in \mathrm{supp}(\mu)$ it holds that $\nu' \models_{\mathcal{T}} Inv(L')$ and $\mu_{\nu_{\mathrm{aux}}}(A, (L', \nu')) = \mu(A, L')$ for $\nu' = A(\nu \cup \nu_{\mathrm{aux}} \cup \nu'')|_X$. $\blacksquare$

The update labels in MDP(P) prevent the collapsing of assignments which result in the same target state (L', ν'). This ensures that MDP(P) is isomorphic to the transition system of the reachability theory of P, as shown in Proposition 4.125.

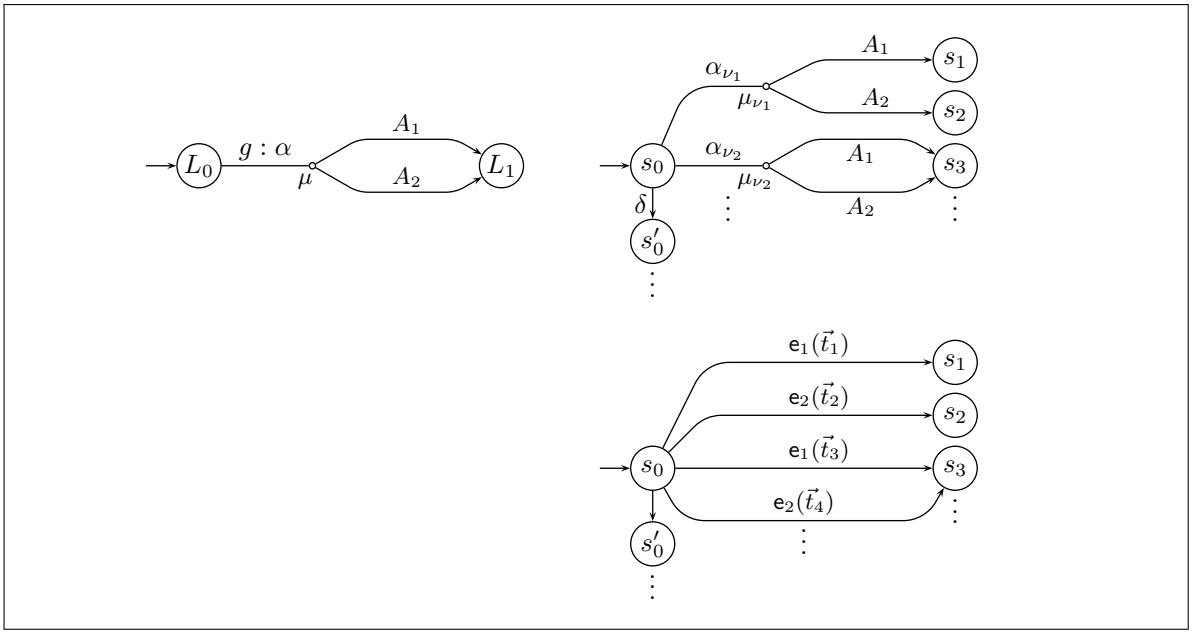


Figure 4.13: Illustration of an FPTA P, MDP(P) and TS(N_P)

4.7.3 Labelled Superposition for Max Reachability

We now present a saturation-based approach to enumerate reachability proofs in FPTA, and to instantiate the FPTA using these proofs to obtain a PTA that can be model-checked.

The first step will be to generalize the notion of reachability theory (Definition 4.32) to *labelled reachability theories* (Definition 4.121), which consist of labelled clauses. The labels act as identifiers for transitions. We then give an encoding of FPTA into a labelled reachability theory (Definition 4.124).

The second step is to extend the SUP(LA) calculus to handle the clause labels, in such a way that saturation leads to the enumeration of all the paths needed for the instantiation of a max-reachability equivalent PTA. We call this property *path completeness* (see Proposition 4.138). The extension of SUP(LA) is similar to the splitting calculus of Chapter 3. However, reduction rules like subsumption deletion and tautology deletion have to be restricted in order to preserve path completeness.

The final step consists in constructing a PTA from the initial FPTA and the collected reachability proofs, in such a way that maximal reachability probabilities are preserved.

The method can thus be used to validate safety properties of the FPTA: Showing that $\Pr^{\max}(s \models \Diamond B) \leq \epsilon$ establishes that $\Pr^{\mathfrak{S}}(s \models \Box\neg B) \geq 1 - \epsilon$ for any scheduler $\mathfrak{S}$, where B stands for a set of bad states.

Of course, the labels can be omitted, and the procedure then becomes the classical (qualitative) reachability procedure.

Labelled Reachability Theories

For this section, we assume the same setting as in Section 4.4, namely a theory $\mathcal{T}$ with a unique minimal Herbrand model. We again assume a signature Σ_R containing a single reachability predicate symbol R, and additionally an *edge sort* S_E and function symbols called *edge identifiers*, ranging into S_E. An *edge term* is a term of the form $f_e(t_1, \ldots, t_n)$ where $f_e \in \Sigma_E$ and the t_i are $\Sigma_{\mathcal{T}}$-terms.

Definition 4.121 (Labelled Reachability Theories)
Let $\mathcal{T}$ be a theory with a unique minimal Herbrand model. A set of labelled clauses N is a *labelled ($\mathcal{T}$-)reachability theory* if the corresponding set of unlabelled clauses is a $\mathcal{T}$-reachability theory, and every initial clause has label $\emptyset$ and every transition clause has either label $\emptyset$, or a label of the form $\{e\}$, where e is an edge term, such that all variables of e are contained in the remainder of the clause. ∎

Example 4.122
The following clauses constitute a labelled reachability theory for the automaton from Example 4.50 (page 82):

$$\begin{aligned}
\emptyset&: && \to A(0)\\
\{\mathsf{e}_1(x)\}&: && A(x) \to A(x+1)\\
\{\mathsf{e}_2(x)\}&: && A(x), x \geq 5 \to B(x)\\
\{\mathsf{e}_3(x)\}&: && B(x) \to B(x-1)\ .
\end{aligned}$$

∎

Definition 4.123 (Transition System of a Labelled Reachability Theory)
Given a labelled $\mathcal{T}$-reachability theory N, the *transition system associated with* N is defined as $\mathrm{TS}(N) = (S_{\mathcal{T}}, Act, \to, S_0)$ with

$$\begin{aligned}
Act &= \{\, \mathcal{I}_{\mathcal{T}}(\nu)(e) \mid e \text{ is an edge term} \,\} \cup \{\tau\},\\
\to &= \{\, (\mathsf{s}_{\mathcal{T}}(\nu, A),\ \mathrm{lab}(\ell),\ \mathsf{s}_{\mathcal{T}}(\nu, B)) \mid \ell : A, l_1, \ldots, l_n \to B \in N \text{ and}\\
&\qquad\qquad \mathcal{I}_{\mathcal{T}}, \nu \models \{l_1, \ldots, l_n\} \,\},\\
S_0 &= \{\, \mathsf{s}_{\mathcal{T}}(\nu, A) \mid \emptyset : l_1, \ldots, l_n \to A \in N \text{ and } \mathcal{I}_{\mathcal{T}}, \nu \models \{l_1, \ldots, l_n\} \,\}.
\end{aligned}$$

where τ is a special *silent action*, distinct from all other symbols, and $\mathrm{lab}(\emptyset) = \tau$ and $\mathrm{lab}(\{e\}) = \mathcal{I}_{\mathcal{T}}(\nu)(e)$. ∎

We are now ready to define the labelled reachability theory of an FPTA. The idea is the same as for (E)TA, with the exception that each discrete-step clause now has a label referring to the edge it encodes:

Definition 4.124 (Reachability Theories for First-Order Probabilistic Timed Automata)
Let $P = (\mathcal{T}, Loc, Act, X, \hookrightarrow, L_0, g_0, Inv)$ be an FPTA. Let $\mathcal{T}_P$ be the extension of $\mathcal{T}$ with a new sort S_{Loc} and constant symbols $\{L | L \in Loc\}$ of sort S_{Loc}. The edge identifiers in Σ_E are all tuples (L, α, A, L') such that $L \overset{\alpha,A}{\hookrightarrow} L'$. The reachability theory for P, denoted by N_P, is the labelled $\mathcal{T}_P$-reachability theory consisting of the following clauses:

$$\emptyset : g_0[\vec{x}\vec{z}] \;\rightarrow\; R(L_0, \vec{x}\vec{z})$$

$$\emptyset : R(L, \vec{x}\vec{z}),\ t{\geq}0,\ \vec{x}' \simeq \vec{x}{+}t,\ Inv(L)[\vec{x}'/\vec{x}] \;\rightarrow\; R(L, \vec{x}'\vec{z}) \qquad \text{for all } L \in Loc$$

$$\{\mathsf{e}(\vec{x}\vec{y}\vec{z})\} : R(L, \vec{x}\vec{y}),\ g,\ (\vec{x}'\vec{y}') \simeq A(\vec{x}\vec{y}),\ Inv(L')[\vec{x}'\vec{y}'/\vec{x}\vec{y}] \;\rightarrow\; R(L', \vec{x}'\vec{y}')$$
$$\text{for all } L \overset{g:\alpha,\mu,A}{\hookrightarrow} L',\ \text{with } \mathsf{e} = (L, \alpha, A, L')$$

where $\vec{x}, \vec{y}$ are the variables in X_C, X_D, and $\vec{x}', \vec{y}'$ are the corresponding variables in X'_C, X'_D, respectively, and $\vec{z} = \text{auxvar}(L, g, \alpha, \mu)$. ∎

Proposition 4.125
$\text{TS}(N_P)$ is isomorphic to $\text{TS}(\text{MDP}(P))$.

Proof. The proof is similar to the proof of adequacy of the TA encoding (Prop. 4.58). First observe that $\text{TS}(\text{MDP}(P))$ and $\text{TS}(N_P)$ have same set of states. Then we show isomorphism between transitions: Let $\rightarrow$ be the transition relation of $\text{TS}(\text{MDP}(P))$ and $\rightarrow_N$ be the transition relation of $\text{TS}(N)$. For the time steps, observe that $(s, \delta, s') \in \rightarrow$ if and only if $(s, \delta, \circledast(s')) \in \twoheadrightarrow$, which by Definitions 4.120, 4.124 and 4.123 is equivalent to $(s, \tau, s') \in \rightarrow_N$. For the discrete steps, observe that $(s, \alpha_\nu, s') \in \rightarrow$ if and only if $(s, \alpha_\nu, \mu_\nu) \in \twoheadrightarrow$, which again Definitions 4.120, 4.124 and 4.123 is equivalent to $(s, e, s') \in \rightarrow_N$. □

Extending SUP(LA) to Labelled Clauses

In order for SUP(LA) to deal with the clause labels, we extend the rules of the calculus accordingly, in a way similar to the splitting calculus of Chapter 3: Given any SUP(LA) inference

$$\mathcal{I}\frac{\Lambda_1 \parallel \Gamma_1 \rightarrow \Delta_1 \quad \dots \quad \Lambda_n \parallel \Gamma_n \rightarrow \Delta_n}{(\Lambda \parallel \Gamma \rightarrow \Delta)\sigma}$$

with most general simple unifier σ, the corresponding labelled inference is

$$\mathcal{I}\frac{\ell_1 : \Lambda_1 \parallel \Gamma_1 \rightarrow \Delta_1 \quad \dots \quad \ell_n : \Lambda_n \parallel \Gamma_n \rightarrow \Delta_n}{(\ell_1 \cup \dots \cup \ell_n : \Lambda \parallel \Gamma \rightarrow \Delta)\sigma} \quad .$$

We call the resulting calculus LSUP(LA).

At the beginning of Section 4.3.4, we noted that real quantifier elimination was implicitly used in SUP(LA) to ensure that only those real-valued variables which occur in the free part of an abstracted FOL(LA) clause remain in the constraint. In LSUP(LA), we assume that only those variables are eliminated which occur *neither* in the free part *nor* in the clause label.

It is straightforward to generalize the hierarchic lifting lemma [Kru13] from SUP(LA) to LSUP(LA):

Lemma 4.126 (Lifting Lemma for LSUP(LA))
Let $\ell_1 : C_1, \ldots, \ell_n : C_n$ be two variable-disjoint clauses, and let θ be a simple substitution such that

$$\mathcal{I}\frac{C_1\theta \quad \ldots \quad C_n\theta}{C'}$$

is a non-redundant ground SUP(LA) inference. Then

$$\mathcal{I}\frac{(\ell_1 : C_1)\theta \quad \ldots \quad (\ell_n : C_n)\theta}{(\ell_1 \cup \ldots \cup \ell_n)\theta : C'}$$

is a corresponding ground LSUP(LA) inference, and there is a corresponding non-redundant LSUP(LA) inference

$$\mathcal{I}\frac{\ell_1 : C_1 \quad \ldots \quad \ell_n : C_n}{\ell'' : C''}$$

and a simple ground substitution τ such that $(\ell'' : C'')\tau = (\ell_1 \cup \ldots \cup \ell_n)\theta : C'$.

Proof. The proof proceeds exactly as in [Kru13] with the labels being carried along and substitutions applied to them, as the labels play no role in the applicability of inference rules. □

4.7.4 Reduction Rules for LSUP(LA)

Reduction rules like subsumption deletion remove redundant clauses from the search space, and are an important ingredient in superposition-based theorem proving.

In the context of reachability proof enumeration with LSUP(LA), however, standard subsumption deletion can cause incompleteness, in the sense that some paths from the initial to a goal state will be missed, when it is applied to clauses containing state atoms. The same holds for tautology deletion. Figure 4.14 illustrates the problem.

On the left-hand side, the clause $\ell, \{e_3, e_4\} : A \to$ subsumes $\{e_1\} : A \to B$ but deleting the latter can mean that no path using edge e_1 is derived.

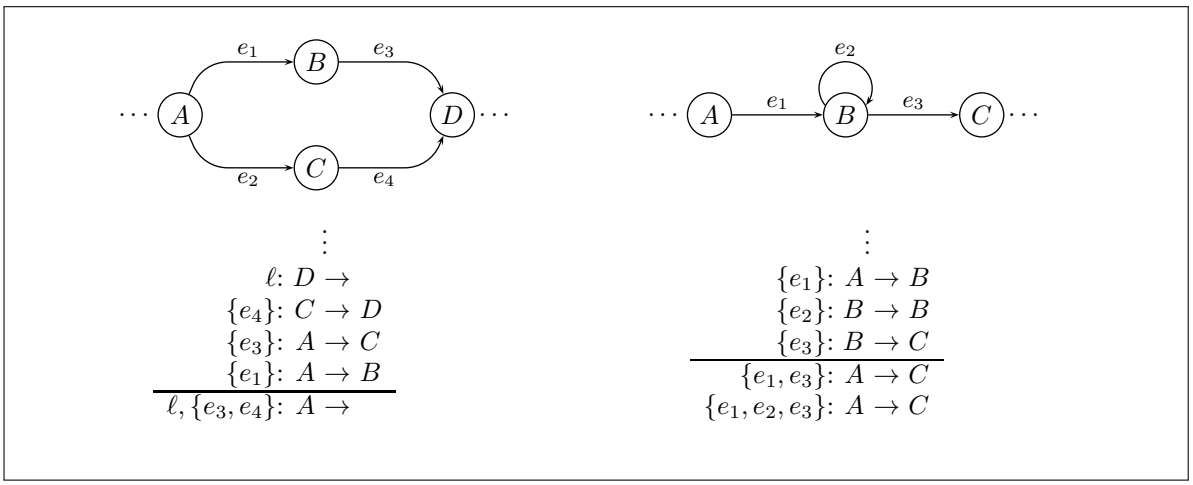


Figure 4.14: Incompleteness caused by subsumption and tautology deletion

On the right-hand side, there are two problems: Clause $\{e_2\} : B \to B$ is a tautology (logically), but deleting it can mean that no path using edge e_2 is derived. Secondly, $\{e_1, e_3\} : A \to C$ subsumes $\{e_1, e_2, e_3\} : A \to C$, but deleting the latter means losing all paths which contain the edges e_1, e_2, e_3.

Let's first consider subsumption deletion. The first problem suggests that a ground clause should only subsume another ground clause if both clauses have the same state literals, i.e., the literals built from state atoms. For the theory literals, the standard subset notion of subsumption can be used. On the non-ground level, a clause C_1 should subsume another clause C_2 if all (simple) ground instances of C_1 subsume all (simple) ground instances of C_2. Putting things together, we get the subsumption deletion rule shown in Figure 4.15, where State(C) denote the state literals of a clause C, and Theory(C) denote the theory literals, i.e., all literals that are not state literals.

Conditions (ii) and (iii) of subsumption deletion are the standard conditions for SUP(LA) subsumption deletion. The other conditions ensure the lifting of the requirements discussed above.

The tautology deletion rule is identical to the corresponding SUP(LA) rule, except that state literals are not considered when determining whether the clause is a tautology.

4.7.5 Instantiating the FPTA

For the rest of this section, we fix an arbitrary action-deterministic FPTA

$$P = (\mathcal{T}, Loc, Act, X, \hookrightarrow, L_0, g_0, Inv) \ .$$

Subsumption Deletion:

$$\mathcal{R}\frac{\ell_1 : \Lambda_1 \parallel C_1 \qquad \ell_2 : \Lambda_2 \parallel C_2}{\ell_1 : \Lambda_1 \parallel C_1}$$

where σ is a simple matcher such that

(i) $\ell_2 \subseteq \ell_1\sigma$, and

(ii) $\models_{\text{LA}} \forall\vec{x}.\exists\vec{y}.(\Lambda_2 \to \Lambda_1\sigma)$, where $\vec{x}$ are all base variables in $\ell_2 : \Lambda_2 \parallel C_2$, and $\vec{y} = \text{var}(\Lambda_1\sigma) \setminus \vec{x}$

(iii) $\text{State}(C_1\sigma) = \text{State}(C_2)$, and

(iv) $\text{Theory}(C_1\sigma) \subseteq \text{Theory}(C_2)$.

Tautology Deletion:

$$\mathcal{R}\frac{\ell : \Lambda \parallel C}{}$$

where $\models_{\mathcal{T}} \text{Theory}(C)$ or $\exists\vec{x}.\Lambda \models_{\text{LA}} \bot$, for $\vec{x} = \text{var}(\Lambda)$.

Figure 4.15: Subsumption deletion and tautology deletion for LSUP(LA)

We identify each labelled, constrained empty clause $\ell : \Lambda \parallel \square$ with the pair (ℓ, Λ), and each set E of empty clauses with the set $\bigcup_{(\ell,\Lambda)\in E}\bigcup_{e\in\ell}\{(e,\Lambda)\}$. For a set E of empty clauses, we write $E_{L,\alpha,A,L'}$ for $\{(\mathsf{e}(\vec{t}),\Lambda) \in E \mid \mathsf{e} = (L,\alpha,A,L')\}$ and $E_{L,\alpha}$ for $\bigcup_{A,L'} E_{L,\alpha,A,L'}$.

Definition 4.127 (Edge Constraints)
Let $p = (L, g, \alpha, \mu) \in \hookrightarrow$ such that $\mu(A, L') > 0$, and let $\vec{u} = u_1, \ldots, u_m = X \cup \text{auxvar}(p)$. Let $\mathsf{e} = (L, \alpha, A, L')$ be the corresponding edge identifier, and consider an edge term $e = \mathsf{e}(t_1, \ldots, t_m)$ and an LA constraint Λ. We define

$$\varphi(e,\Lambda) \;=\; \exists\vec{y}.\Lambda\sigma_e \;\wedge \bigwedge_{t_i\sigma\neq u_i} u_i \simeq t_i\sigma_e$$

where $\vec{y} = \text{var}(\Lambda\sigma_e) \setminus \text{var}(e\sigma_e)$, and σ_e is the *normalizing renaming* defined as $\sigma_e = \sigma_m \circ \ldots \circ \sigma_1$, for

$$\sigma_i = \begin{cases} \{t_i \mapsto u_i\} & \text{if } t_i \text{ is a variable,} \\ \emptyset & \text{otherwise.} \end{cases}$$

Given a set E of empty clauses, the *edge constraints* represented by E are defined as

$$\varphi(E) = \bigcup_{(e,\Lambda)\in E} \varphi(e,\Lambda)\ . \qquad \blacksquare$$

Example 4.128
For $\vec{u} = u_1u_2u_3$, we get

$$\begin{aligned}\varphi(\mathsf{e}(x,x,f(x)),\,(x{\geq}y{+}1,\,y{\geq}2)) &= \exists y.(u_1 \geq y{+}1,\, y \geq 2) \wedge u_2 \simeq u_1 \wedge u_3 \simeq f(u_1)\\ &= u_1 \geq 3 \wedge u_2 \simeq u_1 \wedge u_3 \simeq f(u_1)\end{aligned}$$

with the normalizing renaming $\{x \mapsto u_1\}$. ∎

Definition 4.129 (Assignment-deterministic Edge Constraints)
Let $\vec{x} = X$ and consider $p = (L, g, \alpha, \mu) \in{\hookrightarrow}$ such that $\mu(A, L') > 0$. An edge constraint $\phi \in \varphi(E_{L,\alpha,A,L'})$ is called *assignment-deterministic* if

$$\models_{\mathrm{LA}} (\exists \vec{y}.\phi) \leftrightarrow \bigwedge_{i=1}^{k} z_i \simeq c_i$$

where $\vec{z} = \mathrm{auxvar}(p) \cup \mathrm{argvar}(p)$ and $\vec{y} = \mathrm{var}(\phi) \setminus \vec{z}$, and the c_i are ground terms of the same sort as z_i.[33] If ϕ is assignment-deterministic, we write σ_ϕ for the substitution that maps each z_i to c_i. ∎

That is, an edge constraint is assignment-deterministic if it forces each argument variable and each auxiliary variable of its transition to take a fixed value.

Example 4.130
Suppose $X = \{x_1, x_2\}$ and consider $p = (L, g, \alpha, \mu) \in{\hookrightarrow}$ such that $\mu(A, L') > 0$. Assume that $\mathrm{argvar}(p) = \{x_2\}$ and $\mathrm{auxvar}(p) = \{z\}$. Let e be the corresponding edge identifier, and consider the set of empty clauses $E = \{(\mathsf{e}(x_1, a, f(a)), x_1 \geq 1), (\mathsf{e}(x_1, b, f(b)), x_1 \geq 2)\}$. Then $\varphi(E) = \{\phi_1, \phi_2\}$ with

$$\begin{aligned}\phi_1 &= x_1 \geq 1 \wedge x_2 \simeq a \wedge z \simeq f(a)\\ \phi_2 &= x_1 \geq 2 \wedge x_2 \simeq b \wedge z \simeq f(b)\ .\end{aligned}$$

Both ϕ_1, ϕ_2 are assignment-deterministic, and $\sigma_{\phi_1} = \{x_2 \mapsto a,\, z \mapsto f(a)\}$, $\sigma_{\phi_2} = \{x_2 \mapsto b,\, z \mapsto f(b)\}$. ∎

Now we come to the instantiation of FPTA by edge constraints. Given an FPTA P and a set E of empty clauses derived from $N_P^\flat$ by LSUP(LA), we construct the PTA P_E.

Definition 4.131 (Probabilistic Timed Automaton P_E)
Let E be a finite, assignment-deterministic set of empty clauses. Let $E^\sigma_{L,\alpha}$ denote the largest subset of $E_{L,\alpha}$ such that $\sigma_\phi = \sigma$ holds for all $\phi \in \varphi(E^\sigma_{L,\alpha})$. The instantiation of P by E is the probabilistic timed automaton $P_E = (Loc \uplus \{\bot\}, Act_E, X, \hookrightarrow_E, L_0, \nu_0, Inv)$ where $\hookrightarrow_E$ is the smallest relation such that $(L, g_\sigma, \alpha_{\sigma_{\mathrm{aux}}}, \mu_\sigma) \in{\hookrightarrow_E}$ if there are

[33] Non-arithmetic symbols are treated as constants.

- $(L, g, \alpha, \mu) \in \hookrightarrow$ and
- ground substitutions $\sigma_{\text{arg}}, \sigma_{\text{aux}}$ with domains $\text{argvar}(p)$ and $\text{auxvar}(p)$, respectively, and $\sigma = \sigma_{\text{arg}} \circ \sigma_{\text{aux}}$,

such that $E^{\sigma}_{L,\alpha} \neq \emptyset$, and g_σ is equivalent to

$$\bigvee \exists \vec{z}.\varphi(E^{\sigma}_{L,\alpha}) \quad \text{with} \quad \vec{z} = \text{var}(\varphi(E^{\sigma}_{L,\alpha})) \setminus X \ ,$$

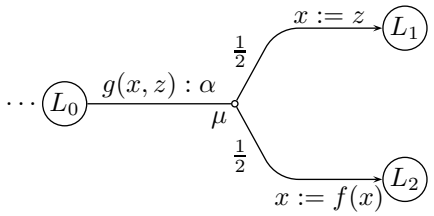
and μ_σ is defined by

$$\begin{aligned} \mu_\sigma(A, L') &= \sum_{\substack{(A',L') \in Q: \\ A'\sigma = A}} \mu(A', L') \\ \mu_\sigma(\emptyset, \bot) &= \sum_{(A,L') \in \text{supp}(\mu) \setminus Q} \mu(A, L') \end{aligned}$$

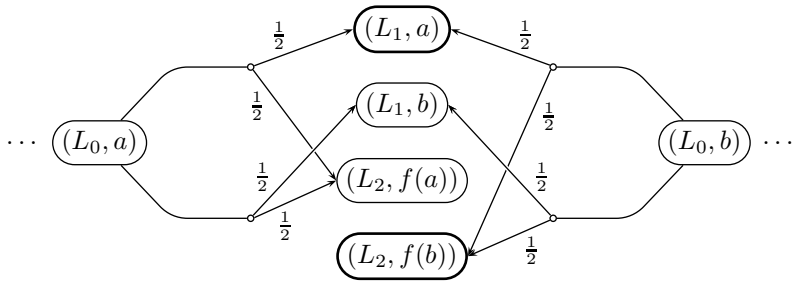
where $Q = \{(A, L') \mid E^{\sigma}_{L,\alpha} \cap E_{L,\alpha,A,L'} \neq \emptyset\}$. Finally, Act_E is the set of all actions occurring in $\hookrightarrow_E$. ■

Since all σ_ϕ are ground substitutions, the codomain of assignments in P_E consists only of state variables and ground terms. Taking the domain of discrete variables to consist of all values they are assigned to on any transition of P_E, we obtain finite discrete domains for the non-clock variables, as required for PTA (see discussion on page 123).

Example 4.132
We illustrate the construction with a very simple example of an FPTA P without clock variables. Suppose $X = X_D = \{x\}$ and consider the transition $p = (L_0, g[x, z], \alpha, \mu)$ shown below:



Assume that $\Sigma_{\mathcal{T}}$ contains only the constants a and b, and that $\mathcal{T}$ is axiomatized by $g(a, a) \wedge g(a, b) \wedge g(b, a) \wedge g(b, b)$ and suppose we are interested in reachability of the set of states $B = \{(L_1, a), (L_2, f(b))\}$. The corresponding part of MDP(P) looks as follows:



We have $\mathrm{auxvar}(p) = \{z\}$ and $\mathrm{argvar}(p) = \{x\}$, and the clauses of N_P corresponding to p are

$$\{\mathsf{e}_1(x,z)\} : R(L_0,x), g[x,z] \to R(L_1,z)$$
$$\{\mathsf{e}_2(x,z)\} : R(L_0,x), g[x,z] \to R(L_2,f(x))$$

with $\mathsf{e}_1 = (L_0, \alpha, \{x := z\}, L_1)$ and $\mathsf{e}_2 = (L_0, \alpha, \{x := f(x)\}, L_2)$. Saturation of

$$N_P \cup \{\emptyset : R(L_1,a) \to\ ,\ \emptyset : R(L_2,f(b)) \to\ \}$$

yields the set of empty clauses

$$E = \{\, \mathsf{e}_1(a,a),\ \mathsf{e}_1(b,a),\ \mathsf{e}_2(b,a),\ \mathsf{e}_2(b,b)\,\}$$

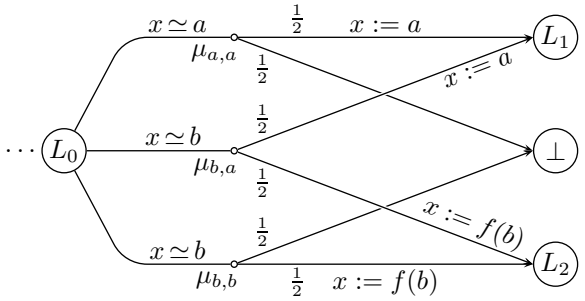
which is partitioned into

$$E^{\{x\mapsto a,\, z\mapsto a\}} = \{\mathsf{e}_1(a,a)\},$$
$$E^{\{x\mapsto b,\, z\mapsto a\}} = \{\mathsf{e}_1(b,a), \mathsf{e}_2(b,a)\},$$
$$E^{\{x\mapsto b,\, z\mapsto b\}} = \{\mathsf{e}_2(b,b)\}\ .$$

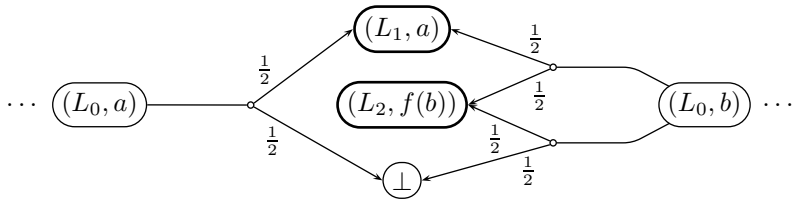
Accordingly, $\hookrightarrow_E$ contains the three transitions

$$(L_0,\ x \simeq a,\ \alpha,\ \mu_{a,a}),\ (L_0,\ x \simeq b,\ \alpha,\ \mu_{b,a}),\ (L_0,\ x \simeq b,\ \alpha,\ \mu_{b,b})$$

and the corresponding part of P_E looks as follows:



The corresponding part of $\mathrm{MDP}(P_E)$ looks like this:



In this example we can see why the construction can preserve only maximum reachability probabilities: Since the states (L_1, b) and $(L_2, f(a))$ don't lie in $\mathrm{Pre}^*(B)$, the corresponding edge instances are not derived. The transition from (L_0, a) to (L_1, b) and $(L_2, f(a))$, which is present in $\mathrm{MDP}(P)$, is thus missing from $\mathrm{MDP}(P_E)$. Therefore $\Pr^{\min}((L_0, a) \models \Diamond B) = 0$ in $\mathrm{MDP}(P)$, but $\Pr^{\min}((L_0, a) \models \Diamond B) = \frac{1}{2}$ in $\mathrm{MDP}(P_E)$. On the other hand, the maximum probability of reaching B is 1 in both $\mathrm{MDP}(P)$ and $\mathrm{MDP}(P_E)$.

The example also illustrates why the argument variables (in this case x) have to be taken into account for assignment determinism: Had we partitioned E only into $E^{\{z \mapsto a\}}$ and $E^{\{z \mapsto b\}}$, then P_E would contain transitions with guard $x \simeq a \vee x \simeq b$ and assignment $x := f(x)$. Such assignments, however, are not expressible in the language of PTA. ∎

Definition 4.133 (Coverage of a Path)
Let π be a path in $\mathrm{MDP}(P)$. A set E of empty clauses *covers* π if for every discrete step

$$(L, \nu) \overset{\alpha_{\nu_{\mathrm{aux}}}, A}{\twoheadrightarrow} (L', \nu')$$

of π there is a constraint $\phi \in \varphi(E_{L,\alpha,A,L'})$ such that $\nu \cup \nu_{\mathrm{aux}} \models_{\mathrm{LA}} \phi$. ∎

Because $\mathcal{T}$ contains no non-base equations, and any non-base term t occurs only in equations $x \simeq t$ in E, $\nu \cup \nu_{\mathrm{aux}} \models_{\mathrm{LA}} \phi$ is equivalent to $\mathcal{I}_{\mathcal{T}}, (\nu \cup \nu_{\mathrm{aux}}) \models \phi$.

Proposition 4.134
Consider an edge term $\mathsf{e}(t_1, \ldots, t_m)$ and an LA constraint Λ, and let σ be a grounding substitution for $\mathsf{e}(t_1, \ldots, t_m)$ and $\exists \vec{y}.\Lambda$, $\vec{y} = \mathrm{var}(\Lambda) \setminus \mathrm{var}(t_1, \ldots, t_m)$, such that $(\exists \vec{y}.\Lambda)\sigma$ is true. Then there exists a substitution $\theta \geq \sigma$ such that $\models_{\mathrm{LA}} \varphi(\mathsf{e}(t_1, \ldots, t_m), \Lambda)\theta$.

Proof. We have

$$\begin{aligned} \varphi(\mathsf{e}(t_1, \ldots, t_m), \Lambda)\theta &= (\exists \vec{y}.\Lambda)\theta \wedge \bigwedge x_i\theta \simeq t_i\theta \\ &= (\exists \vec{y}.\Lambda)\sigma \wedge \bigwedge x_i\theta \simeq t_i\sigma \end{aligned}$$

so we get $\models_{\mathrm{LA}} \varphi(\mathsf{e}(t_1, \ldots, t_m), \Lambda)\theta$ by choosing $\theta = \sigma \circ \{x_i \mapsto t_i\sigma \mid x_i \notin \mathrm{dom}(\sigma)\}$. □

Example 4.135
Consider $\mathsf{e}(x_1, f(y))$ and $\Lambda = x_1 \geq 0$. Let $\sigma = \{x_1 \mapsto 1, y \mapsto a\}$. Now choose $\theta = \sigma \circ \{x_2 \mapsto f(a)\}$, then $\varphi(\mathsf{e}(x_1, f(y)), x_1 \geq 0)\theta = (x_1 \geq 0 \wedge x_2 \simeq f(y))\theta = 1 \geq 0 \wedge f(a) \simeq f(a)$. ∎

Path Enumeration by Saturation

Consider an FPTA P and its labelled reachability theory N_P, where all clauses have been abstracted (see Definition 4.17). Let G be a set of goal clauses representing a set of goal states B. We will use LSUP(LA) saturation to enumerate paths from s_0 to B. We apply the following derivation strategy: After purification, all negative theory literals are selected. We denote the resulting set of purified clauses by N_P^B. The transition clauses in N_P^B have the form

$$\ell : \Lambda \;\|\; R(\ldots), l_1, \ldots, l_n \;\rightarrow\; R(\ldots)$$

where ℓ is the label, consisting of edge terms, Λ is the LA constraint, $R(\ldots)$ are state atoms, and the l_i are negative $\mathcal{T}$-literals, all of which are selected. Initial and goal clauses have an analogous form, with only a positive or a negative state atom, respectively.

Remark 4.136 (Discrete-step Clauses after Purification)
Consider a discrete-step clause from N_P

$$\{\mathsf{e}(\vec{x}\vec{y})\} : R(L, \vec{x}),\; g[\vec{x}\vec{y}\vec{z}],\; \vec{x}' \simeq A(\vec{x}),\; Inv(L')[\vec{x}'_b] \;\rightarrow\; R(L', \vec{x}')$$

where $X = \vec{x} = \vec{x}_b \cup \vec{x}_n$ and $\mathrm{auxvar}(p) = \vec{y} = \vec{y}_b \cup \vec{y}_n$, where subscripts b and n denote base and non-base variables, respectively. In the backward encoding $N_P^\flat$ and after purification, this yields

$$\{\mathsf{e}(\vec{x}\vec{y})\} : \Lambda_g[\vec{x}_b\vec{y}_b\vec{z}_b],\; \Lambda_A[\vec{x}_b\vec{x}'_b\vec{y}_b],\; \Lambda_{Inv}[\vec{x}'_b] \;\|\; R(L', A'(\vec{x})),\; g'[\vec{x}\vec{y}\vec{z}] \;\rightarrow\; R(L, \vec{x})$$

where Λ_g and Λ_A are the base literals resulting from purification of g and $\vec{x}' \simeq A(\vec{x})$, respectively, and Λ_{Inv} are the literals of $Inv(L')[\vec{x}'_b]$ (which are all base) and A' is the remainder of A after purification. For example, if $A = \{x_1 \mapsto 0, x_2 \mapsto f(a)\}$ where x_1, x_2 are base and x_3 is non-base, then $\Lambda_A = x'_1 \simeq 0$ and $A' = \{x_1 \mapsto x'_1, x_2 \mapsto f(a)\}$. ∎

The following two propositions establish the connection between the empty clauses and edge constraints produced by the saturation of N_P^B and the corresponding transitions of P. Path soundness (Proposition 4.137) states that the edge constraints imply their corresponding guards in P, while path completeness (Proposition 4.138) states that any path of P reaching a goal state is covered by some empty clause.

Proposition 4.137 (Path soundness)
Let $p = (L, g, \alpha, \mu) \in \hookrightarrow$ and $\phi \in \varphi(E_{L,\alpha,A,L'})$. Then

$$\mathcal{I}_\mathcal{T} \models \forall \vec{x}.\; \phi \rightarrow \exists \vec{z}.\; g$$

where $\vec{z} = \mathrm{var}(g) \setminus (X \cup \mathrm{auxvar}(p))$.

Proof. As noted in Remark 4.136, the clause corresponding to $\mathsf{e} = (L, \alpha, A, L')$ is of the form[34]

$$C = \{\mathsf{e}(\vec{x}\vec{y})\} : \Lambda_g[\vec{x}_b\vec{y}_b\vec{z}_b],\ \Lambda_A[\vec{x}_b\vec{x}'_b\vec{y}_b],\ \Lambda_{Inv}[\vec{x}'_b]\ \|\ R',\ g'[\vec{x}\vec{y}\vec{z}]\ \rightarrow\ R$$

where the literals of g' are selected. Consider any refutation using C. Before C can interact with any other clause of $N_P^\flat$, the g' literals have to be resolved away by inferences with theory clauses, eventually yielding a clause of the form

$$(\{\mathsf{e}(\vec{x}\vec{y})\} : \Lambda' \|\ R' \rightarrow\ R)\tau$$

for some simple substitution τ, and Λ' is equivalent to the conjunction of $\Lambda_g, \Lambda_A, \Lambda_{Inv}$ and some Λ'' such that $\mathcal{T} \models \forall.\Lambda''\tau \rightarrow g'\tau$. In the empty clause (ℓ, Λ) eventually derived, we have $\mathsf{e}(\vec{x}\vec{y})\tau\rho \in \ell$ for some substitution ρ, and the base equations of $\varphi(\mathsf{e}(\vec{x}\vec{y})\tau\rho, \Lambda)$ imply $\exists\vec{z}_b.\Lambda_g$, and also imply $\exists\vec{z}.\Lambda''\tau$, modulo appropriate renaming. It is then straightforward to show that

$$\models_{\mathcal{T}} \forall\vec{x}\vec{y}.\varphi(\mathsf{e}(\vec{x}\vec{y})\tau\rho, \Lambda) \rightarrow \exists\vec{z}.(\Lambda_g \wedge g'\tau)$$

hence

$$\models_{\mathcal{T}} \forall\vec{x}\vec{y}.\varphi(\mathsf{e}(\vec{x}\vec{y})\tau\rho, \Lambda) \rightarrow \exists\vec{z}.(\Lambda_g \wedge g')\ .$$

Since g is equivalent to $\Lambda_g \wedge g'$, it follows that

$$\models_{\mathcal{T}} \forall\vec{x}\vec{y}.\varphi(\mathsf{e}(\vec{x}\vec{y})\tau\rho, \Lambda) \rightarrow \exists\vec{z}.g\ . \qquad \square$$

Proposition 4.138 (Path completeness)
Let π be a finite path of MDP(P) ending in B. Then there is a derivation of an empty clause (ℓ, Λ) from N_P^B such that (ℓ, Λ) covers π.

Proof. By Proposition 4.125, π corresponds to a path of TS(N_P). So consider the set $N_\pi \subseteq \mathrm{gnd}(N_P^B)$ of ground clauses corresponding to π, and consider an arbitrary discrete step

$$(L, \nu) \overset{\alpha_{\nu_{\mathrm{aux}}}, A}{\twoheadrightarrow} (L', \nu')$$

of π. Following Remark 4.136, the corresponding ground clause is $C\theta$ with

$$C\ =\ \mathsf{e}(\vec{x}\vec{y}) : \Lambda_{\mathsf{e}} \,\|\, R(L', A'(\vec{x})),\ g'[\vec{x}\vec{y}\vec{z}] \rightarrow R(L, \vec{x})$$

where $\theta = \sigma\sigma_{\mathrm{aux}}\sigma''$ with ground substitutions σ, σ_{aux} and σ'' corresponding to ν, ν_{aux} and ν'' as in Definition 4.120. By Theorem 4.47, derivation of a ground empty clause $(l\tau, \Lambda\tau)$ from N_π, using $C\theta$. By definition of LSUP(LA), it follows that $\mathsf{e}(\vec{x}\vec{y})\theta \in \ell\tau$. Applying the lifting lemma (Lemma 4.126), there is a corresponding non-ground derivation from N_P^B of the empty clause (ℓ, Λ), using C, and there is a substitution ρ such that $\mathsf{e}(\vec{x}\vec{y})\rho \in \ell$,

[34] Writing R for $R(L, \vec{x})$ and R' for $R(L', A'(\vec{x}))$.

and $\mathsf{e}(\vec{x}\vec{y})\rho\tau = \mathsf{e}(\vec{x}\vec{y})\theta$, and $\models_{\text{LA}} \Lambda \to \Lambda_e\rho$. As $\varphi(\mathsf{e}(\vec{x}\vec{y})\rho, \Lambda) \in \varphi(\ell, \Lambda)$, it remains to show that[35]

$$\models_{\text{LA}} \varphi(\mathsf{e}(\vec{x}\vec{y})\rho, \Lambda)\theta. \tag{1}$$

Now

$$\models_{\text{LA}} \varphi(\mathsf{e}(\vec{x}\vec{y})\rho, \Lambda)\theta \leftrightarrow \varphi(\mathsf{e}(\vec{x}\vec{y})\rho, \Lambda)\tau$$

as $\mathsf{e}(\vec{x}\vec{y})\rho\tau = \mathsf{e}(\vec{x}\vec{y})\theta$ and $\Lambda\tau$ is ground. As $\Lambda\tau$ is true, we can apply Proposition 4.134 to deduce (1). □

The following two Propositions establish that P and P_E agree on all one-step transition probabilities between states that are both reachable from the initial state and that can reach a goal state in B. We write Post_P, Pre_P, $\mathbf{P}_P$ for $\text{Post}_{\text{MDP}(P)}$, $\text{Pre}_{\text{MDP}(P)}$, $\mathbf{P}_{\text{MDP}(P)}$, respectively (and analogously for P_E).

Proposition 4.139
Consider two states $s, s' \in \text{Post}_P^*(s_0) \cap \text{Pre}_P^*(B)$ such that $s' \in \text{Post}_P(s)$. Then for all $\alpha_{\nu_{\text{aux}}} \in Act_{\mathcal{T}}$, there is $\alpha_{\sigma_{\text{aux}}} \in Act_{P_E}$ such that σ_{aux} is equivalent to ν_{aux} and

$$\mathbf{P}_P(s, \alpha_{\nu_{\text{aux}}}, s') = \mathbf{P}_{P_E}(s, \alpha_{\sigma_{\text{aux}}}, s') \ .$$

Proof. Let $s = (L, \nu)$ and $s' = (L', \nu')$, and let

$$s \overset{\alpha_{\nu_{\text{aux}}}, A}{\twoheadrightarrow} s'$$

be a step of MDP(P). Let $p = (L, g, \alpha, \mu) \in \hookrightarrow$ be the corresponding transition of P. Let $A = A^{(1)}, \ldots, A^{(n)}$ be all assignments such that

$$s \overset{\alpha_{\nu_{\text{aux}}}, A^{(i)}}{\twoheadrightarrow} s' \ .$$

It holds that

$$\mathbf{P}_P(s, \alpha_{\nu_{\text{aux}}}, s') = \sum_i \mu(A^{(i)}, L') \ .$$

Since each step from s to s' lies on a path from s_0 to B, each such step is covered by E (Proposition 4.138), i.e., there are constraints $\phi^{(i)} \in \varphi(E^{\sigma}_{L,\alpha})$, for $\sigma = \sigma_{\text{arg}} \circ \sigma_{\text{aux}}$, such that $\nu, \nu_{\text{aux}} \models \phi^{(i)}$ for all i, and σ_{arg} is compatible with ν and σ_{aux} is equivalent to ν_{aux}. Hence, by Definition 4.131, there is $(L, g_\sigma, \alpha_{\sigma_{\text{aux}}}, \mu_\sigma) \in \hookrightarrow_E$ such that $\nu \models g_\sigma$ and

$$\mu_\sigma(A', L') = \sum_{A^{(i)}\sigma = A'} \mu(A^{(i)}, L') \ .$$

By definition of MDP(P_E), we get

$$\begin{aligned} \mathbf{P}_{P_E}(s, \alpha_{\sigma_{\text{aux}}}, s') &= \sum_{A'} \mu_\sigma(A', L') \\ &= \sum_i \mu(A^{(i)}, L') \\ &= \mathbf{P}_P(s, \alpha_{\nu_{\text{aux}}}, s') \ . \end{aligned}$$

□

[35] Again, non-arithmetic terms are treated as constants.

Now for the other direction:

Proposition 4.140
Consider two states $s, s' \in \mathrm{MDP}(P_E)$ such that $s' \in \mathrm{Post}_{P_E}(s)$. Then for all $\alpha_{\sigma_{\mathrm{aux}}} \in Act_{P_E}$ there is $\alpha_{\nu_{\mathrm{aux}}} \in Act_{\mathcal{T}}$, such that σ_{aux} is equivalent to ν_{aux} and

$$\mathbf{P}_P(s, \alpha_{\nu_{\mathrm{aux}}}, s') = \mathbf{P}_{P_E}(s, \alpha_{\sigma_{\mathrm{aux}}}, s') \ .$$

Proof. Let $s = (L, \nu)$ and $s' = (L', \nu')$, and consider the step

$$s \overset{\alpha_{\sigma_{\mathrm{aux}}}}{\twoheadrightarrow} s'$$

in $\mathrm{MDP}(P_E)$, and let $(L, g_\sigma, \alpha_{\sigma_{\mathrm{aux}}}, \mu_\sigma) \in \hookrightarrow_E$ with $\sigma = \sigma_{\mathrm{arg}} \circ \sigma_{\mathrm{aux}}$ be the corresponding transition in P_E, and $(L, g, \alpha, \mu) \in \hookrightarrow$ be the corresponding transition in P. By Proposition 4.137, it follows that $\mathcal{I}_{\mathcal{T}} \models g_\sigma \to \exists \vec{z}.\ g$, where $\vec{z}$ are the variables of g not occurring in g_σ. Hence there are transitions

$$s \overset{\alpha_{\nu_{\mathrm{aux}}}, A^{(i)}}{\twoheadrightarrow} s'$$

in $\mathrm{MDP}(P)$. Then

$$\begin{aligned} \mathbf{P}_P(s, \alpha_{\nu_{\mathrm{aux}}}, s') &= \sum \mu(A^{(i)}, L') \\ &= \mathbf{P}_{P_E}(s, \alpha_{\sigma_{\mathrm{aux}}}, s') \ . \end{aligned}$$

□

Theorem 4.141
P and P_E agree on maximal reachability probabilities of B.

Proof. It follows by induction from Propositions 4.139 and 4.140 that

$$\mathrm{Post}^*_P(s_0) \cap \mathrm{Pre}^*_P(B) = \mathrm{Post}^*_{P_E}(s_0) \cap \mathrm{Pre}^*_{P_E}(B) \ .$$

Now consider the equation system for max reachability (page 123): On the one hand, the value of x_{s_0} depends only the x_s with $s \in \mathrm{Post}^*(s_0)$, and on the other hand, $x_s = 0$ holds for all $s \notin \mathrm{Pre}^*(B)$. Hence x_{s_0} is uniquely determined by the x_s with $s \in \mathrm{Post}^*(s_0) \cap \mathrm{Pre}^*(B)$. By Propositions 4.139 and 4.140, we have

$$\mathbf{P}_P(s, \alpha_{\nu_{\mathrm{aux}}}, s') = \mathbf{P}_{P_E}(s, \alpha_{\sigma_{\mathrm{aux}}}, s') \ .$$

for all $s, s' \in \mathrm{Post}^*(s_0) \cap \mathrm{Pre}^*(B)$. Hence $\Pr^{\max}_P(s_0 \models \Diamond B) = \Pr^{\max}_{P_E}(s_0 \models \Diamond B)$. □

4.7.6 Implementation

We have implemented the approach presented in this section as an extension to the SPASS(LA) theorem prover. The extension accepts a system of FPTA described in a

format based on the standard SPASS syntax [WDF+09], together with a set of reachability conjectures. It computes the product FPTA (Definition 4.118), constructs the labelled reachability theory (Definition 4.124), and saturates the resulting clause set using the LSUP(LA) calculus (Section 4.7.3). If saturation terminates, PTA P_E (Definition 4.131) is constructed, which is then output in the Modest format for use with the mcpta model checker [HH09].

The reachability conjectures are translated into corresponding reachability property goals in the Modest property specification language, and the PTA P_E is augmented so as to trigger a special action reach in the corresponding states. In this way, the output of the SPASS(LA) extension can be passed directly to mcpta to compute the respective reachability probabilities. Figure 4.16 gives a high-level overview of the pipeline.

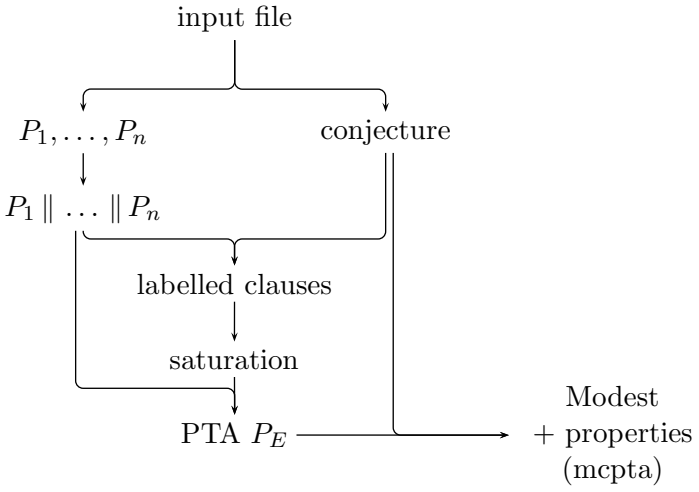


Figure 4.16: Pipeline for FPTA reachability analysis

The mcpta model checker applies an integral time semantics, and then uses the probabilistic model checker PRISM [KNP11] as a back-end, which finally builds and analyzes a probabilistic automata model. The most recent version of PRISM natively supports PTA, which it did not at the time we developed the pipeline. Using PRISM directly as a back-end is part of future work.

4.7.7 Experimental Results: Analyzing DHCP

We tested our implementation on a simple model of a DHCP [Dro97] dialog between a client and a server over a faulty network. The first-order structures of the FPTA represent messages as terms starting from the IP-layer. We omit detailed modeling of IP-addresses and the needed masking operations in order to keep the model small and to focus on timing and probability aspects. Our model could be extended that way (see

the first-order LAN model available at `http://www.spass-prover.org/prototypes/`). Nevertheless, the term structure enables a reasonably detailed model of the network stack. The IP-address lease database of the server is modeled in form of first-order atoms. For each participant (client, server) the model contains three layers – in the form of FPTA: The DHCP protocol layer, a resend mechanism, and the respective connected faulty networks, as shown in Figure 4.17. The inter-network communication (dashed arrows) has been shortcut to keep the example simple. The networks have a fixed latency and message loss probability.

The FPTA on the protocol layer closely follow the steps of the DHCP protocol from the init state to a bound state on the client side and the respective steps on the server side. The resend and network automata are identical for both server and client. The resend automaton tries two resends with a time out of 3 time units to the network before it gives up. The network forwards a message with probability 0.9 and causes a latency delay of 1 time unit.

The input file contains six FPTA, modelling the server and the client with their respective resend mechanisms and networks, the background theory – in this case describing only the lease table – and the reachability conjecture for the client's bound state.

The composed FPTA as computed by SPASS(LA) has 1230 locations, and the FOL(LA) encoding consists of 5811 clauses. The number of generated edge terms is 4578. SPASS(LA) takes about 13 minutes (on recent Linux driven Xeon X5460 hardware) to finitely saturate the clause set, deriving about 16000 clauses, of which 81 are labelled empty clauses corresponding to reachability proofs. Using these proofs, SPASS(LA) produces a PTA with 27 locations, which is then passed to mcpta. The PTA model constructed by mcpta has 8524 states, and takes less than a second to construct.

As example quantities, we are able to calculate that the probability of successfully obtaining a DHCP lease within 0.4 time units (meaning no message is lost) is 0.656, and the probability of obtaining one after 3.6 time units (allowing for message loss at every stage of the protocol) is 0.996.

The input file containing the FPTA and the input file to mcpta produced by SPASS from the saturation are also available at `http://www.spass-prover.org/prototypes/`.

The *client network* is described by the following FPTA:

```
begin_automaton(FPTA,ClientNetwork).
  variables[clock(vcntime), discr(vcnmsg)].
  actions[cdmsg, csend].
  locations[Init,
            (Wait, le(vcntime, 1)),
            Ok,
            Fail].
```

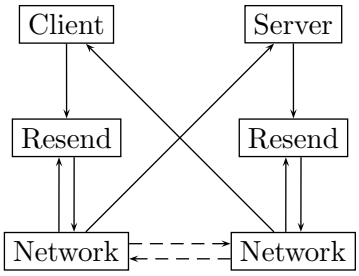


Figure 4.17: The DHCP example consisting of 6 FPTA

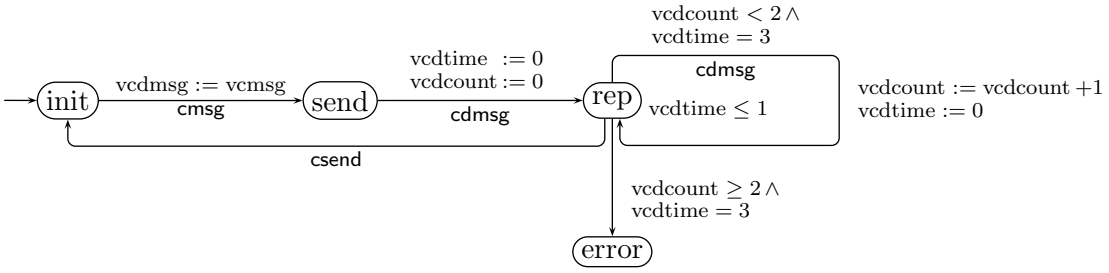


Figure 4.18: FPTA for the DHCP client's resend mechanism

```
  initial(Init(0,      % vcntime
               null)). % vcnmsg
  branch(Init,
         external([vcdmsg], true),
         cdmsg,
         Wait,
         [vcnmsg=vcdmsg, vcntime=0]).
  pbranch(Wait,
          equal(vcntime, 1),
          [edge(9, Ok), edge(1, Fail)]).
  branch(Ok,
         true,
         csend,
         Init).
  branch(Fail,
         true,
         Init).
end_automaton.
```

The `variables` keyword declares the automaton's state variables, which are either `clock` variables, or discrete variables (`discr`), i.e. of integer or first-order sort. The `actions`

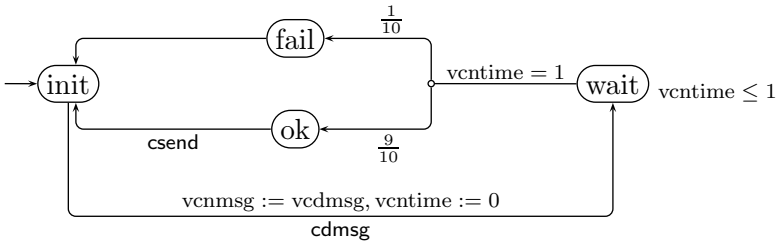


Figure 4.19: FPTA for the DHCP client's faulty network

keyword declares the automaton's action alphabet. The `locations` keyword declares the automaton's locations, optionally with their invariant (like for `Wait`). The keyword `branch` declares a transition without probabilistic choice, i.e., the target location is entered with probability 1. The `pbranch` keyword declares a transition with a probability distribution. Both take the source location, and the guard as arguments, followed by the target location and the assignment (for `branch`), or the the distribution over the target locations and assignments (for `pbranch`). The keyword `external` in guards is used to declare a variable as being the state variable of another component FPTA in the composition.[36] The keyword `bind`, which doesn't appear in the example above, but can be found in the input files under `http://www.spass-prover.org/prototypes/`, declares a variable as being not a state variable. Such a variable is either an auxiliary variable if it appears in some assignment, or acts as an existentially quantified variable otherwise.[37]

The *client* starts with his vcmsg variable containing the ground term

$$\begin{array}{l}\text{ippacket}(\text{networkip}, \text{broadcastip}, \text{udp},\\ \quad \text{udppacket}(\text{port68}, \text{port67},\\ \quad\quad \text{dhcpdata}(\text{broadcast}, \text{networkip}, \text{networkip}, \text{networkip}, \text{networkip}, \text{clientmac},\\ \quad\quad\quad \text{dhcpoptions}(\text{DHCPdiscover}, \text{networkip}, \text{networkip}))))\end{array}$$

representing the initial DHCP *discover* message.

The background theory consists of the single ground atom

$$\text{Leasefree}(\text{leasefreetuple}(\text{clientip}, \text{clientmac}))$$

modelling the server's lease table.

Because clock variables only occur in equations, like vcdtime = 3 and vcntime = 1, there is actually no nondeterminism in this model, only probabilistic choice. The maximum reachability probabilities computed are thus also minimum reachability probabilities.

[36] See the section on parallel composition for FPTA on page 127.
[37] See the semantics of FPTA, Definition 4.120.

Comparison With Previous Work In [FHW10] we erroneously claimed that P and P_E agreed on maximal *and minimal* reachability probabilities. As shown in Example 4.132, only maximal reachability probabilities are preserved by the construction.

4.7.8 Future Work

A natural question is what restrictions on background theories can yield decidable classes of FPTA. Here one must distinguish between "qualitative decidability", meaning that the reachability problem is decidable, and "quantitative decidability", meaning that proof enumeration can be made to terminate.

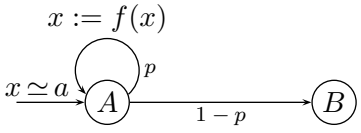
A related question is whether the proof enumeration presented in this section can be combined with constraint induction, which may aid in termination, like in the case of ETA (Section 4.6.3). This could be difficult using the labelling and proof enumeration scheme, but see Section 4.8 for an alternative approach which might better fit with constraint induction.

Even if proof enumeration cannot be guaranteed to terminate, reachability probabilities could be generated "on-the-fly", during saturation: Using the constrained empty clauses derived so far, a PTA could be generated incrementally. Using such an approach, a sequence of lower bounds for the maximum reachability probability could be derived, converging to the actual value in the limit.

Sections 4.8 and 4.9 present some further ideas for future work.

4.8 Outlook: Language-based Reachability Analysis

In general, the state space of an FPTA is infinite, because of the clocks, which always have an infinite domain, and because of potentially infinite domains of discrete variables. While the labelling scheme introduced in Section 4.7 exploits the expressivity of FOL(LA) to finitely represent infinitely many clock valuations, infinite discrete domains can quickly lead to non-termination. As a simple example, consider the automaton with a single discrete variable x:



The encoding under the labelling scheme from Section 4.7 would be of the form

$$\begin{aligned} \emptyset\colon &\quad \to A(a) \\ \{\mathsf{e}_1(x)\}\colon A(x) &\to A(f(x)) \\ \{\mathsf{e}_2(x)\}\colon A(x) &\to B(x) \\ \emptyset\colon B(x) &\to \end{aligned}$$

There are infinitely many paths from the initial state $A(a)$ to states of the form $B(t)$, and saturation by the labelled calculus of Section 4.7.3 doesn't terminate, because all edge instances $\mathsf{e}_1(f^n(a))$ and $\mathsf{e}_2(f^n(a))$, $n \geq 0$, are being enumerated.

We now briefly sketch a different labelling scheme that can finitely saturate examples such as the one above. It is based on the idea of replacing probabilities by symbols and describing paths through the automaton as regular expressions over those symbols. A similar idea was used in [Daw05] to symbolically compute exact reachability probabilities in finite discrete-timed Markov chains. Here, we use it to obtain a regular expression describing all paths from the initial state to a goal state in a potentially infinite transition system.

As we only provide a preliminary sketch of the approach, we restrict ourselves to purely probabilistic automata. We can think of these as FPTA without clocks, without nondeterministic branching and without a background theory (except for the theory of equality). We leave it as future work to extend this approach to full FPTA.

Let P be an automaton as described above, and let Act consist of the non-zero probabilities in P. Consider the underlying transition system $\mathrm{TS}(P) = (S, Act, \to, s_0)$. Given a set B of goal states, we say that P *accepts* a word $w \in Act^*$ if there is a path $s_0 \xrightarrow{w} s$ for some $s \in B$. We write $\mathcal{L}(P)$ for the language accepted by P.

The reachability probability $\Pr(s_0 \models \Diamond B)$ can be computed whenever $\mathcal{L}(P)$ is regular, since any finite-state machine accepting $\mathcal{L}(P)$ is isomorphic to a finite Markov chain which agrees with P on $\Pr(s_0 \models \Diamond B)$.

We now sketch a saturation-based (incomplete) procedure to compute $\mathcal{L}(P)$, if it is regular. Figure 4.20 shows the rules of the calculus.

Definition 4.142 (Star Expressions)
A *star expression* over Act is a regular expression formed according to the grammar

$$e ::= \epsilon \mid \alpha \mid e^* \mid e_1 e_2$$

where $\alpha \in Act$. By $\mathcal{L}(e)$ we denote the language represented by e. ∎

We only consider clauses of the form $e\colon \to A$, $e\colon A \to$, $e\colon A \to B$, and $e\colon \Box$, where e is a star expression.

We define the semantics of such clauses with respect to a transition system and a set B of goal states:

Definition 4.143 (Semantics of Labelled Clauses)
Let $\mathrm{TS} = (S, Act, \to, S_0)$ be a transition system such that S consists of ground atoms, and $B \subseteq S$ be a set of goal states. The semantics of labelled clauses is defined as follows: Let $w \in Act^*$. Then

$\mathrm{TS} \models w : \to A$	if for any ground $A\sigma$, there exists $s \in S_0$ with $A\sigma \in \mathrm{Post}(s, w)$;
$\mathrm{TS} \models w : A \to B$	if for any ground $A\sigma, B\sigma$: $B\sigma \in \mathrm{Post}(A\sigma, w)$;
$\mathrm{TS} \models w : A \to$	if for any ground $A\sigma$, there exists $s \in B$ with $s \in \mathrm{Post}(A\sigma, w)$;
$\mathrm{TS} \models w : \square$	if there exist $s \in S_0, s' \in B$ with $s' \in \mathrm{Post}(s, w)$.

If e is a star expression over Act, then $\mathrm{TS} \models e : C$ if $\mathrm{TS} \models w : C$ for all $w \in \mathcal{L}(e)$. ∎

As a consequence of the above definition, we get that $\mathrm{TS} \models e : \square$ if and only if $\mathcal{L}(e) \subseteq \mathcal{L}(P)$.

For a set of states $Q \subseteq S$ and a star expression e, we let $\mathrm{Post}_{\mathrm{TS}}(Q, e) = \bigcup_{w \in \mathcal{L}(e)} \mathrm{Post}_{\mathrm{TS}}(Q, w)$, and analogously for $\mathrm{Pre}_{\mathrm{TS}}$.

The side conditions of the Loop rules are undecidable in general, but sufficient conditions can be checked syntactically. Consider for example the condition $\mathrm{Post}_{\mathrm{TS}}(\mathrm{gnd}(A), e) \subseteq \mathrm{gnd}(B)$. If $e = \epsilon$ is the empty word, then the condition reduces to $\mathrm{gnd}(A) \subseteq \mathrm{gnd}(B)$, i.e., A is an instance of B. If $e = \alpha \in Act$, then a sufficient condition is the existence of a clause $\alpha : A' \to B'$ in N, such that $A' = A\sigma$ and $B'\sigma' = B\sigma$, for some substitutions σ, σ'. More complex expressions e can be reduced to these base cases by observing that, for arbitrary sets of states $Q, Q \subseteq S$,

- $\mathrm{Post}_{\mathrm{TS}}(Q, \alpha e) \subseteq Q'$ if there exists $Q'' \subseteq S$ such that $\mathrm{Post}_{\mathrm{TS}}(Q, \alpha) \subseteq Q''$ and $\mathrm{Post}_{\mathrm{TS}}(Q'', e) \subseteq Q'$;
- $\mathrm{Post}_{\mathrm{TS}}(Q, e^* e') \subseteq Q'$ if $\mathrm{Post}_{\mathrm{TS}}(Q, e) \subseteq Q$ and $\mathrm{Post}_{\mathrm{TS}}(Q, e') \subseteq Q'$.

If saturation terminates, we derive a set of labelled empty clauses $e_1 : \square, \ldots, e_n : \square$ such that $\mathcal{L}(P) = \mathcal{L}(e_1 | \ldots | e_n)$. Regular expressions over probabilities can be evaluated to probabilities by applying the function val, presented in [Daw05], and defined by

$$\begin{aligned} \mathrm{val}(p) &= p \\ \mathrm{val}(e_1 e_2) &= \mathrm{val}(e_1) \cdot \mathrm{val}(e_2) \\ \mathrm{val}(e_1 | e_2) &= \mathrm{val}(e_1) + \mathrm{val}(e_2) \\ \mathrm{val}(e^*) &= \frac{1}{1 - \mathrm{val}(e)} \ . \end{aligned}$$

Example 4.144
Consider again the automaton from the introduction. It can now be encoded as follows (with $q = 1 - p$):

Resolution:

$$\mathcal{I}\frac{e_1 : \to A \quad e_2 : A' \to B}{e_1 e_2 : (\to B)\sigma} \qquad \mathcal{I}\frac{e_1 : B \to A \quad e_2 : A' \to}{e_1 e_2 : (B \to)\sigma}$$

$$\mathcal{I}\frac{e_1 : B \to A \quad e_2 : A' \to C}{e_1 e_2 : (B \to C)\sigma} \qquad \mathcal{I}\frac{e_1 : \to A \quad e_2 : A' \to}{e_1 e_2 : \square}$$

where $\sigma = \mathrm{mgu}(A, A')$.

Backward Loop:

$$\mathcal{I}\frac{e_1 : A \to \quad e_2 e_1 : A' \to}{e_2^* e_1 : (A \to)\sigma}$$

if $\sigma = \mathrm{mgu}(A, A')$ and $\mathrm{TS} \models \mathrm{gnd}(A\sigma) \xrightarrow{e_2} \mathrm{gnd}(A\sigma)$ or $\mathrm{Post}_{\mathrm{TS}}(\mathrm{gnd}(A\sigma), e_2) \subseteq \mathrm{gnd}(A\sigma)$.

Forward Loop:

$$\mathcal{I}\frac{e_1 : \to A \quad e_1 e_2 : \to A'}{e_1 e_2^* : (\to A)\sigma}$$

if $\sigma = \mathrm{mgu}(A, A')$ and $\mathrm{gnd}(A\sigma) \subseteq \mathrm{Pre}_{\mathrm{TS}}(\mathrm{gnd}(A\sigma), e_2)$.

Inner Loop:

$$\mathcal{I}\frac{e_1 e_2 : A \to B \quad e_1 e_3 e_2 : A' \to B'}{e_1 e_3^* e_2 : (A \to B)\sigma}$$

where $\sigma = \mathrm{mgu}(A, A')$ and every clause in $\mathrm{Pre}_{\mathrm{TS}}(B\sigma, e_2)$ is an instance of a clause in $\mathrm{Post}_{\mathrm{TS}}(A\sigma, e_1)$.

Subsumption:

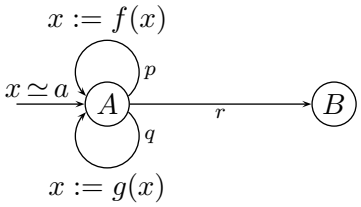
Clause $e : C$ subsumes $e' : D$ if $C\sigma = D$ for some substitution σ, and $\mathcal{L}(e') \subseteq \mathcal{L}(e)$.

Figure 4.20: Resolution with respect to a fixed transition system TS

(1)	ϵ :	$\to A(a)$
(2)	p :	$A(x) \to A(f(x))$
(3)	q :	$A(x) \to B(x)$
(4)	ϵ :	$B(x) \to$
Res(3,4)=(5)	q :	$A(x) \to$
Res(2,5)=(6)	pq :	$A(x) \to$
Backward Loop(5,6)=(7)	p^*q :	$A(x) \to$
Res(1,7)=(8)	p^*q :	$\square$

The clause set is saturated, as the clauses $q : \square$ and $pq : \square$ which could be derived by resolution from (5),(1) and (5),(2), respectively, are subsumed by clause (8). So $\mathcal{L}(P) = \mathcal{L}(p^*q)$ and the probability to reach any $B(x)$ from $A(a)$ is (as expected) $\mathrm{val}(p^*(1-p)) = 1$. ■

Example 4.145
Consider the automaton shown below:



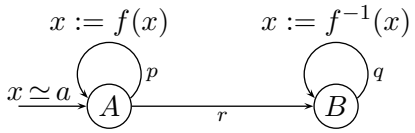
It can be encoded as follows, using a reduction ordering with $B \succ A$ (maximal literals are underlined):

(1)	ϵ :	$\to \underline{A(a)}$
(2)	p :	$A(x) \to \underline{A(f(x))}$
(3)	q :	$A(x) \to \underline{A(g(x))}$
(4)	r :	$A(x) \to \underline{B(x)}$
(5)	ϵ :	$\underline{B(x)} \to$
Res(4,5)=(6)	r :	$\underline{A(x)} \to$
Res(2,6)=(7)	pr :	$\underline{A(x)} \to$
Backward Loop(6,7)=(8)	p^*r :	$\underline{A(x)} \to$
Res(3,8)=(9)	qp^*r :	$\underline{A(x)} \to$
Backward Loop(8,9)=(10)	q^*p^*r :	$\underline{A(x)} \to$
Backward Loop(6,10)=(11)	$(q^*p^*)^*r$:	$\underline{A(x)} \to$
Res(1,11)=(12)	$(q^*p^*)^*r$:	$\square$

The clause set is saturated, and we deduce $\mathcal{L}(P) = \mathcal{L}((q^*p^*)^*r)$, thus the probability to reach any $B(x)$ from $A(a)$ is

$$\mathrm{val}((q^*p^*)^*r) = \frac{r}{1 - \frac{1}{(1-p)(1-q)}} \ .$$ ■

Example 4.146
As a counterexample, consider the following automaton:



An encoding using a reduction ordering with $B \succ A$ would be:

$$
\begin{array}{lll}
(1) & \epsilon : & \rightarrow \underline{A(a)} \\
(2) & p : & A(x) \rightarrow \underline{A(f(x))} \\
(3) & r : & A(x) \rightarrow \underline{B(x)} \\
(4) & q : & \underline{B(f(x))} \rightarrow \underline{B(x)} \\
(5) & \epsilon : & \underline{B(x)} \rightarrow
\end{array}
$$

with goal clause $\epsilon : B(x) \rightarrow$. By successive resolution with (3) and (5), clauses of the form $p^n r q^n : A(x) \rightarrow B(x)$ can be derived, for all $n \geq 0$, which don't subsume each other, and to which the rule Inner Loop is not applicable. Hence saturation doesn't terminate. This is not surprising, as the language $p^n r q^n$ describing the paths to $B(x)$ is not regular. ■

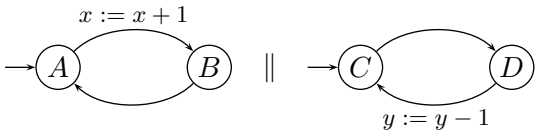
4.8.1 Discussion

The ideas presented in this section are still in a very preliminary stage. Many questions remain to be investigated, among which are the nature of the transition system TS with respect to which the calculus operates (intuitively, this should be the minimal model of the "positive" clauses in the clause set, i.e., the clauses of the form $\alpha :\rightarrow A$ and $\alpha : A \rightarrow B$), the integration of background theories, the generalization to nondeterministic and timed systems, and effective redundancy handling. We hope that this approach will be further developed in the near future.

4.9 Outlook: Avoiding Building the Product Automaton

So far, when dealing with the parallel composition of automata, we have only considered the reachability theory of the product automaton. The size of the product automaton, and hence of the encoding, grows exponentially with the number of components. More compact encodings can be obtained by treating the component automata separately.

Concurrent Encoding Consider the two automata below, which don't share any action, hence operate completely asynchronously.



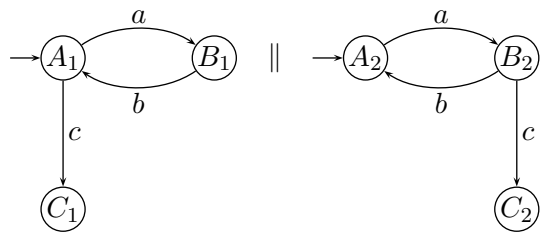
The product automaton has 4 locations and 8 transitions, hence its reachability theory consists of 9 clauses (including one for the initial state). Assuming both automata start with their respective state variable set to zero, the same system can be represented by the following 5 clauses:

$$\begin{aligned} &\to R(A, C, 0, 0) \\ R(A, l, x, y) &\to R(B, l, x + 1, y) \\ R(B, l, x, y) &\to R(A, l, x, y) \\ R(l, C, x, y) &\to R(l, D, x, y - 1) \\ R(l, D, x, y) &\to R(l, C, x, y) \ . \end{aligned}$$

Let's call such an encoding a *concurrent encoding*, as opposed to the *product encoding* we have considered so far. The concurrent encoding is used, for instance, in [DP01]. Depending on the degree of synchronization of the system, the concurrent encoding may be exponentially smaller than the product encoding, since in the totally asynchronous case, its size is proportional to the sum of the sizes of the encodings of the individual components. It is also clear that the concurrent encoding may yield shorter reachability proofs, since several transitions in one component which do not synchronize with any other transition, can be combined into a single transition, corresponding to a single clause, which can act as a lemma and be reused multiple times—it is easy to construct examples where the shortest proof from the product encoding is exponentially longer than the shortest proof from the concurrent encoding.

Timed systems, however, are very synchronous by nature, since every time step needs to be synchronized across all components. So even using a concurrent encoding, there will be exponentially many (in the number of components) time-step clauses. But even for timed systems, the concurrent encoding may still give an advantage, when there are many independent transitions.

Generalized Reachability Theories Finally, let us briefly mention an encoding using *generalized* reachability theories (Definition 4.32). To illustrate it, consider the following fully synchronous system:



An encoding into a generalized reachability theory would be the following:

$$\begin{aligned}
&\rightarrow R(A_1)\\
&\rightarrow R(A_2)\\
R(A_1), R(A_2) &\rightarrow R(B_1)\\
R(A_1), R(A_2) &\rightarrow R(B_2)\\
R(B_1), R(B_2) &\rightarrow R(A_1)\\
R(B_1), R(B_2) &\rightarrow R(A_2)\\
R(A_1), R(B_2) &\rightarrow R(C_1)\\
R(A_1), R(B_2) &\rightarrow R(C_2)\ .
\end{aligned}$$

An advantage of using this encoding may be that in case of a satisfiable clause set, i.e., unreachable goal states, saturation may be even faster than in the concurrent encoding. However, this encoding yields an over-approximation of reachability, in the sense that the set of reachable states of all component automata is a subset of the extension of the reachability predicate in the minimal model, but is in general not equal to it. For instance, the states C_1 and C_2 are not reachable in the above system, but $R(C_1)$ and $R(C_2)$ hold in the minimal model of the theory. So the encoding may be used to quickly establish the unreachability of unsafe states by saturation, but reachability proofs can be spurious.

Furthermore, the encoding can be used to model asynchronous systems exchanging messages by means of a special predicate Sent, in the style of [Wei99a]. For instance, a transition from A to B, sending a message m, would be modelled by the clause

$$R(A) \rightarrow R(B), \mathrm{Sent}(m)$$

while a transition from A to B depending on message m having been previously sent, would be modelled by the clause

$$R(A), \mathrm{Sent}(m) \rightarrow R(B)\ .$$

The encoding would cease to be an over-approximation when considered as a description of an unbounded number of copies of the system running in parallel, as is customary in the analysis of security protocols.

Conclusion The use of different encoding schemes and their effects in the context of the approach presented in this chapter, including optimizations to the SUP(LA) calculus to take full advantage of the concurrent encoding, are beyond the scope of this work, and we leave them to future investigation.

4.10 Discussion and Related Work

Reasoning in the combination of first-order logic and background theories such as linear arithmetic, the analysis of timed systems, and automatic induction are all active research areas, and we here give a brief overview of the current state of the art and how the work presented in this thesis relates to it.

FOL(T) Decision Procedures. While many superposition-based decision procedures for fragments of pure first-order logic exist [Nie96, JMW98, Wei99a, JRV06], the expressivity of FOL(T) makes the identification of interesting decidable fragments more challenging.[38]

Kruglov and Weidenbach [KW12] present a SUP(T)-based decision procedure for the FOL(T) fragment called *Bernays-Schönfinkel-Horn class with equality and ground base sort terms*, or BSHE(GBST), where every non-constant function symbol from the underlying FOL signature ranges into a base sort, and every term of base sort is ground.

Bonacina, Lynch and de Moura [BLdM09, BLdM11] present a decision procedure using DPLL(Γ+T), which is a deep integration of a superposition-based inference system Γ with a DPLL(T)-based [NOT06] SMT-solver, and the additional introduction of "speculative" axioms to enforce termination in satisfiable cases. These act as hypothetical clauses, which are treated in a special way by the prover, and thus bear a certain resemblance with our labelled clauses.

Our SUP(LA)-based decision procedures for TA and ETA can be viewed as a contribution to the research about FOL(T) decision procedures.

Analysis of Timed and Infinite-State Systems. Decidability of the reachability problem for timed automata was originally shown using the region graph, a finite quotient of the state space [ACD90]. The size of the region graph is exponential in the largest time constant occurring in the automaton and the number of clocks, and it is thus of theoretical interest only. In practice, reachability analysis for timed automata is based on the zone graph, in which zones, i.e., sets of clock valuations, are represented by difference bound matrices (DBM) [BY03]. In order to represent discrete values, like locations or additional discrete variables, one can distinguish semi-symbolic from fully symbolic

[38] See [FW12] for a discussion of the combination of BSH with linear arithmetic.

approaches. In a semi-symbolic approach, clock valuations are represented symbolically, while the discrete part of the system is represented explicitly. This approach is well-suited for systems with a small discrete state space. Fully symbolic approaches represent the discrete part of the state space symbolically as well, using data structures like BDDs. The encoding and derivation strategy for timed automata presented in Section 4.5.6 behaves like a semi-symbolic, backward fixpoint computation, while for ETA, in Section 4.6.3, the valuations of discrete variables are represented symbolically in the clause constraints.

The essential difference between such model checking-based approaches and the one presented here is that SUP(LA) is a more general method, supporting full quantification and integrating powerful redundancy criteria, and we obtain our TA decision procedure by applying this general-purpose calculus practically out-of-the-box to a straightforward encoding of the automaton.

The idea of encoding the reachability problem for timed automata and their extensions into logical formulas, as we do in this chapter, has also been explored by other authors: Comon and Jurski [CJ99] present an encoding of the binary reachability relation of timed automata into the decidable additive theory of real numbers, thereby providing yet another proof of the decidability of the reachability problem. The size of the resulting formulas makes the method unsuitable for practical applications, while the complexity of our method is comparable to that of zone-based backward model checking. On the other hand, their result relies on an earlier decidability result for counter automata [CJ98], whose variables, called counters, can be of real or integer type. We have yet to investigate whether this result implies the decidability of the extended timed automata presented in Section 4.5.3.

Fribourg [Fri98] applies Revesz's procedure [Rev93] to analyze reachability in timed automata, where clocks can also act as data registers. The method relies on the introduction of an additional, *universal* clock.

On the other hand, extensions of (probabilistic) timed automata with additional state variables and data structures have been considered in the literature as well: Lanotte et al. present Systems of Data Management Timed Automata (SDMTA) [LMST10], which are timed automata which can construct, send and receive terms over (finite sets of) elementary messages (constants), natural numbers and functions. They show that reachability is decidable for systems of DMTA which have only one integer variable. The result is proven by a reduction to Vector Addition Systems, not by using a standard saturation-based calculus like in our work. The authors mention an extension to probabilistic automata as future work.

A well-established method for dealing with infinite models is predicate abstraction [GS97], where the state space is partitioned into finitely many abstract states, represented by predicates over the system variables. The coarseness of the abstraction can be adapted automatically in a process called counterexample-guided abstraction refinement (CEGAR) [CGJ$^+$00]. CEGAR has recently been adapted to probabilistic systems

as well [HWZ08]. Wachter, Zhang, Hermanns and Hahn [WZH07, HHWZ10] have developed the model checker PASS based on probabilistic CEGAR and SMT, which can handle probabilistic programs in a guarded command language with unbounded integer and real variables, and which can also be extended to other data types handled by the SMT solvers, like bitvectors and arrays. Extending this approach to probabilistic *timed* automata poses some additional challenges, because a naive treatment of clock variables can make abstraction refinement impractical. Recently, Fioriti and Hermanns [FH12] have proposed heuristics allowing the probabilistic CEGAR approach to be applied to PTA as well. The idea is to exclude clocks from the abstraction process, and to only abstract the values of data variables.

Induction and Automatic Invariant Generation Inductive reasoning often plays an important role in the the verification of infinite-state systems, such as imperative programs or automata with unbounded data structures, and inductive theorem proving is an active research area. In general, proof by induction is an inherently interactive process, since it requires human ingenuity to come up with suitable inductive hypotheses. However, semi-automatic and fully automatic methods are applicable in certain cases.

Among semi-automatic approaches to inductive theorem proving one can distinguish explicit induction techniques [Bun01, BMS10], implicit induction schemes used in rewrite-based theorem provers [BR95a, BR95b], and "inductionless induction" [Com01, HW08] or proof by consistency [KM87]. All the above methods usually rely on inductive lemmas or schemata provided by the user, or they require the clause set to be pre-saturated, and thus do not produce inductive invariants on-the-fly during proof search.

On the other hand, fully automatic approaches to inductive reasoning in non-saturated set have been recently developed as well: Peltier has proposed an approach to reasoning on parametric or schematized formulas [Pel01, EP12] and integrated it into the superposition calculus [KP13a, KP13b]. This technique, like ours, is based on a loop detection rule, however, the language of formulas is more restricted than FOL(T). Horbach and Weidenbach [HW09] have presented a rule called *melting* for induction in the context of superposition for fixed domains [HW08], which is also based on loop detection. It is however a purely syntactic rule, whereas constraint induction takes the semantics of the base theory into account.

In the context of symbolic model checking, a form of induction called *acceleration* is often employed when exploring infinite-state systems [BW94, WB98, FL02, HL02, BIK10]. Acceleration consists in computing the transitive closure of a sequence of transitions of the system being analyzed, with the goal of speeding up the fixpoint computation of reachable states, or to make it terminate in the first place. Our constraint induction rule can be viewed as a combination of acceleration and induction in proof search.

5 Conclusion

In this thesis, we have presented novel contributions to the area of superposition-based theorem proving.

On the one hand, we have presented a modular formalization of explicit splitting with backtracking for superposition, and shown it to be sound and complete. In the process, we have defined a notion of derivation fairness suited to splitting with backtracking.

We have developed a novel approach to conflict-driven clause learning for splitting, which can produce local as well as global non-ground lemmas, going beyond the existing local ground lemma learning technique. The approach relies on clause labels to track variable instantiations. Experimental evaluation of our implementation shows that it improves the performance of superposition with splitting in SPASS.

On the other hand, we have developed an approach for reachability analysis based on hierarchic superposition SUP(T). We have shown SUP(LA) to be a decision procedure for reachability in timed automata.

We have introduced a novel inference rule for SUP(T), called constraint induction, which performs inductive reasoning by loop detection during proof search.

Moreover, we have presented a practical realization of constraint induction for SUP(LA). Furthermore, we have shown that SUP(LA) with constraint induction constitutes a decision procedure for reachability in timed automata extended with unbounded integer variables.

We have developed the formalism of first-order probabilistic timed automata (FPTA), combining probabilistic timed automata (PTA) with first-order background theories, and we have presented a SUP(LA)-based labelled calculus for proof enumeration and shown how to construct a max-reachability-equivalent PTA using the saturation of an FPTA reachability theory.

Finally, we have sketched a different approach to reachability proof enumeration, using a clause labelling scheme based on regular expressions.

Bibliography

[ABH+08] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. A generalized framework for conflict analysis. In Hans Kleine Büning and Xishun Zhao, editors, *SAT*, volume 4996 of *Lecture Notes in Computer Science*, pages 21–27. Springer, 2008.

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414 –425, June 1990.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183235, 1994.

[AHV93] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, page 592601, New York, NY, USA, 1993. ACM.

[AKW09] Ernst Althaus, Evgeny Kruglov, and Christoph Weidenbach. Superposition modulo linear arithmetic SUP(LA). In Silvio Ghilardi and Roberto Sebastiani, editors, *7th international Symposium on Frontiers of Combining Systems*, volume 5749 of *Lecture Notes in Artificial Intelligence*, pages 84–99, Trento, Italy, September 2009. Springer.

[BFT06] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Lemma learning in the model evolution calculus. In Miki Hermann and Andrei Voronkov, editors, *LPAR*, volume 4246 of *Lecture Notes in Computer Science*, pages 572–586. Springer, 2006.

[BG91a] Leo Bachmair and Harald Ganzinger. Perfect model semantics for logic programs with equality. In *Proceedings International Conference on Logic Programming '91*, pages 645–659, Paris, France, 1991. MIT Press.

[BG91b] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. MPI-Report MPI-I-91-208, Max-Planck-Institut für Informatik, Saarbrücken, Germany, September 1991.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994. Revised version of Technical Report MPI-I-91-208, 1991.

[BG01] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 2, pages 19–99. Elsevier and MIT Press, 2001.

[BGW92] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Theorem proving for hierarchic first-order theories. In Hélène Kirchner and Giorgio Levi, editors, *ALP*, volume 632 of *Lecture Notes in Computer Science*, pages 420–434. Springer, 1992.

[BGW93] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Superposition with simplification as a decision procedure for the monadic class with equality. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Proceedings of the Third Kurt Gödel Colloquium on Computational Logic and Proof Theory, KGC '93*, volume 713 of *Lecture Notes in Computer Science*, pages 83–96, London, UK, August 1993. Springer.

[BGW94] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Refutational theorem proving for hierarchic first-order theories. *Applicable Algebra in Engineering, Communication and Computing, AAECC*, 5(3/4):193212, 1994.

[BHR06] P. Bouyer, S. Haddad, and P.-A. Reynier. Extended timed automata and time petri nets. In *Sixth International Conference on Application of Concurrency to System Design, 2006. ACSD 2006*, pages 91–100, 2006.

[BIK10] Marius Bozga, Radu Iosif, and Filip Konecný. Fast acceleration of ultimately periodic relations. In *CAV*, pages 227–242, 2010.

[BK08] C. Baier and J.P. Katoen. *Principles of Model Checking*. Mit Press, 2008.

[BKS04] 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.

[BKST97] K. Brink, J. van Katwijk, R. F. Lutje Spelberg, and W. J. Toetenel. Analyzing schedulability of astral specifications using extended timed automata. In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *Euro-Par'97 Parallel Processing*, number 1300 in Lecture Notes in Computer Science, pages 1290–1297. Springer Berlin Heidelberg, January 1997.

[BL10] Patricia Bouyer and François Laroussinie. Model checking timed automata. In Stephan Merz and Nicolas Navet, editors, *Modeling and Verification of Real-Time Systems*, pages 111–140. ISTE, 2010.

[BLdM09] Maria Paola Bonacina, Christopher Lynch, and Leonardo Mendonça de Moura. On deciding satisfiability by DPLL($\Gamma + \mathcal{T}$) and unsound theorem proving. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2009.

[BLdM11] Maria Paola Bonacina, Christopher Lynch, and Leonardo Mendonça de Moura. On deciding satisfiability by theorem proving with speculative inferences. *J. Autom. Reasoning*, 47(2):161–189, 2011.

[BMS10] David Baelde, Dale Miller, and Zachary Snow. Focused inductive theorem proving. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2010.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[Bon13] Maria Paola Bonacina, editor. *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*. Springer, 2013.

[BP13] James P. Bridge and Lawrence C. Paulson. Case splitting in an automatic theorem prover for real-valued special functions. *J. Autom. Reasoning*, 50(1):99–117, 2013.

[BPT12] Peter Baumgartner, Björn Pelzer, and Cesare Tinelli. Model evolution with equality - revised and implemented. *J. Symb. Comput.*, 47(9):1011–1045, 2012.

[BR95a] Adel Bouhoula and Michaël Rusinowitch. Implicit induction in conditional theories. *J. Autom. Reasoning*, 14(2):189–235, 1995.

[BR95b] Adel Bouhoula and Michaël Rusinowitch. Spike: A system for automatic inductive proofs. In Vangalur S. Alagar and Maurice Nivat, editors, *AMAST*, volume 936 of *Lecture Notes in Computer Science*, pages 576–577. Springer, 1995.

[BT08] Peter Baumgartner and Cesare Tinelli. The model evolution calculus as a first-order DPLL method. *Artif. Intell.*, 172(4-5):591–632, 2008.

[Bun01] Alan Bundy. The automation of proof by mathematical induction. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 845–911. Elsevier and MIT Press, 2001.

[BW94] Bernard Boigelot and Pierre Wolper. Symbolic verification with periodic sets. In *CAV*, pages 55–67, 1994.

[BY03] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[CJ98] Hubert Comon and Yan Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 268–279. Springer, 1998.

[CJ99] Hubert Comon and Yan Jurski. Timed automata and the theory of real numbers. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR*, volume 1664 of *Lecture Notes in Computer Science*, pages 242–257. Springer, 1999.

[Cla77] Keith L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322, New York, 1977. Plenum Press.

[CN00] Hubert Comon and Robert Nieuwenhuis. Induction = I-axiomatization + first-order consistency. *Information and Computation*, 159(1/2):151–186, May 2000.

[Com01] Hubert Comon. Inductionless induction. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 14, pages 913–962. Elsevier and MIT Press, 2001.

[Daw05] Conrado Daws. Symbolic and parametric model checking of discrete-time markov chains. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004*, number 3407 in Lecture Notes in Computer Science, pages 280–294. Springer Berlin Heidelberg, January 2005.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[dMB11] Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[dN01] Hans de Nivelle. Splitting through new proposition symbols. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 172–185. Springer, 2001.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[DP01] Giorgio Delzanno and Andreas Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, August 2001.

[Dro97] R. Droms. Rfc 2131: Dynamic host configuration protocol. *The Internet Engineering Task Force (IETF)*, 1997. Obsoletes RFC 1541. Status: DRAFT STANDARD.

[DW09] Klaus Denecke and Shelly L. Wismath. *Universal Algebra and Coalgebra*. World Scientific, March 2009.

[EP12] Mnacho Echenim and Nicolas Peltier. Reasoning on schemata of formulæ. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *AISC/MKM/Calculemus*, volume 7362 of *Lecture Notes in Computer Science*, pages 310–325. Springer, 2012.

[FH12] Luis María Ferrer Fioriti and Holger Hermanns. Heuristics for probabilistic timed automata with abstraction refinement. In Jens B. Schmitt, editor, *MMB/DFT*, volume 7201 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2012.

[FHW10] Arnaud Fietzke, Holger Hermanns, and Christoph Weidenbach. Superposition-based analysis of first-order probabilistic timed automata. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 of *Lecture Notes in Computer Science*, pages 302–316. Springer Berlin Heidelberg, 2010.

[FKW12] Arnaud Fietzke, Evgeny Kruglov, and Christoph Weidenbach. Automatic generation of invariants for circular derivations in SUP(LA). In Nikolaj Bjørner and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 7180 of *Lecture Notes in Computer Science*, pages 197–211. Springer Berlin Heidelberg, 2012.

[FL02] Alain Finkel and Jérôme Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *FSTTCS*, pages 145–156, 2002.

[FLHT01] Christian G. Fermüller, Alexander Leitsch, Ullrich Hustadt, and Tanel Tamet. Resolution decision procedures. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 25, pages 1791–1849. Elsevier, 2001.

[FPY02] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 2280 in Lecture Notes in Computer Science, pages 67–82. Springer Berlin Heidelberg, January 2002.

[Fri98] Laurent Fribourg. A closed-form evaluation for extended timed automata. Technical report, CNRS & Ecole Normale Suprieure de Cachan, 1998.

[FW08] Arnaud Fietzke and Christoph Weidenbach. Labelled splitting. In *IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning*, page 459474, Berlin, Heidelberg, 2008. Springer-Verlag.

[FW09] Arnaud Fietzke and Christoph Weidenbach. Labelled splitting. *Ann. Math. Artif. Intell.*, 55(1-2):3–34, 2009.

[FW12] Arnaud Fietzke and Christoph Weidenbach. Superposition as a decision procedure for timed automata. *Mathematics in Computer Science*, 6(4):409–425, December 2012.

[Gre68] Cordell Green. *Theorem Proving by Resolution as a Basic for Question-Answering Systems*. Stanford Rsearch Institute, 1968.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[Hal91] Joseph Y. Halpern. Presburger arithmetic with unarr predicates is $\mathrm{pi}_1{}^1$ complete. *J. Symb. Log.*, 56(2):637–642, 1991.

[HH09] Arnd Hartmanns and Holger Hermanns. A modest approach to checking probabilistic timed automata. In *QEST*, pages 187–196, 2009.

[HHWZ10] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. PASS: Abstraction refinement for infinite probabilistic models. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 353–357. Springer, 2010.

[HJL99] Thomas Hillenbrand, Andreas Jaeger, and Bernd Löchner. System description: Waldmeister - improvements in performance and ease of use. In *CADE*, pages 232–236, 1999.

[HL02] Martijn Hendriks and Kim Guldstrand Larsen. Exact acceleration of real-time model checking. *Electr. Notes Theor. Comput. Sci.*, 65(6), 2002.

[Hor10] Matthias Horbach. Disunification for ultimately periodic interpretations. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 290–311. Springer, 2010.

[HR87] Jieh Hsiang and Michaël Rusinowitch. On word problems in equational theories. In Thomas Ottmann, editor, *Proceedings of the 14th International Colloquium on Automata, Languages and Programming, ICALP'87*, volume 267 of *Lecture Notes in Computer Science*, pages 54–71, Karlsruhe, Germany, July 1987.

[HV13] Krystof Hoder and Andrei Voronkov. The 481 ways to split a clause and deal with propositional variables. In Bonacina [Bon13], pages 450–464.

[HW07] Thomas Hillenbrand and Christoph Weidenbach. Superposition for finite domains. Research Report MPI–I–2007–RG1–002, Max-Planck Institute for Informatics, Saarbrücken, Germany, April 2007.

[HW08] Matthias Horbach and Christoph Weidenbach. Superposition for fixed domains. In Michael Kaminski and Simone Martini, editors, *Proceedings of the 17th Annual Conference of the European Association for Computer Science Logic, CSL 08*, volume 5213 of *Lecture Notes in Computer Science*, pages 293–307, Berlin / Heidelberg, September 2008. Springer.

[HW09] Matthias Horbach and Christoph Weidenbach. Deciding the inductive validity of $\forall\exists^*$ queries. In Erich Grädel and Reinhard Kahle, editors, *Proceedings of the 18th Annual Conference of the European Association for Computer Science Logic, CSL 2009*, volume 5771 of *Lecture Notes in Computer Science*, pages 332–347, Berlin / Heidelberg, September 2009. Springer.

[HWZ08] Holger Hermanns, Björn Wachter, and Lijun Zhang. Probabilistic CEGAR. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2008.

[Jen96] Henrik E Jensen. Model checking probabilistic real time systems. In B. Bjerner, M. Larsson, and B. Nordström, editors, *Proceedings of the 7th Nordic Workshop on Programming Theory*, volume 86, pages 247–261, 1996.

[JMW98] Florent Jacquemard, Christoph Meyer, and Christoph Weidenbach. Unification in extensions of shallow equational theories. In Tobias Nipkow, editor, *Rewriting Techniques and Applications, 9th International Conference, RTA-98*, volume 1379 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 1998.

[JRV06] Florent Jacquemard, Michaël Rusinowitch, and Laurent Vigneron. Tree automata with equality constraints modulo equational theories. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2006.

[KB70] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[KM87] Deepak Kapur and David R. Musser. Proof by consistency. *Artificial Intelligence*, 31(2):125–157, 1987.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[KNPS06] Marta Kwiatkowska, Gethin Norman, David Parker, and Jeremy Sproston. Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods in System Design*, 29(1):3378, 2006.

[KNSS02] Marta Z. Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 282(1):101150, 2002.

[KNSW07] Marta Z. Kwiatkowska, Gethin Norman, Jeremy Sproston, and Fuzhi Wang. Symbolic model checking for probabilistic timed automata. *Information and Computation*, 205(7):10271077, 2007.

[KP13a] Abdelkader Kersani and Nicolas Peltier. Combining superposition and induction: A practical realization. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *FroCos*, volume 8152 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2013.

[KP13b] Abdelkader Kersani and Nicolas Peltier. Completeness and decidability results for first-order clauses with indices. In Bonacina [Bon13], pages 58–75.

[Kru13] Evgeny Kruglov. *Superposition Modulo Theory.* PhD thesis, Max Planck Institute for Informatics, 2013.

[KW12] Evgeny Kruglov and Christoph Weidenbach. Superposition decides the first-order logic fragment over ground theories. *Mathematics in Computer Science*, 6(4):427–456, 2012.

[LAWRS07] 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.

[Llo93] John W. Lloyd. *Foundations of Logic Programming.* Springer, 1993.

[LMST10] Ruggero Lanotte, Andrea Maggiolo-Schettini, and Angelo Troina. Reachability results for timed automata with unbounded data structures. *Acta Inf.*, 47(5-6):279–311, 2010.

[McC03] William McCune. Otter 3.3 reference manual. *CoRR*, cs.SC/0310056, 2003.

[MW97] William McCune and Larry Wos. Otter - the CADE-13 competition incarnations. *J. Autom. Reasoning*, 18(2):211–220, 1997.

[Nie96] Robert Nieuwenhuis. Basic paramodulation and decidable theories (extended abstract). In *Proceedings 11th IEEE Symposium on Logic in Computer Science, LICS'96*, pages 473–482. IEEE Computer Society Press, 1996.

[NOT06] 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.

[NR01] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 7, pages 371–443. Elsevier and MIT Press, 2001.

[NWY99] Christer Norström, Anders Wall, and Wang Yi. Timed automata as task models for event-driven systems. In *RTCSA*, pages 182–189. IEEE Computer Society, 1999.

[Pau98] Lauwrence C. Paulson. The inductive approach to verifying cryptographic protocols. In *Journal of Computer Security*, volume 6, pages 85–128. IOS Press, 1998.

[Pel01] Nicolas Peltier. A general method for using schematizations in automated deduction. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *IJCAR*, volume 2083 of *Lecture Notes in Computer Science*, pages 578–592. Springer, 2001.

[Pos46] Emil L. Post. A variant of a recursively unsolvable problem. *J. Symbolic Logic*, 12(2):255–56, 1946.

[Rev93] Peter Z. Revesz. A closed-form evaluation for datalog queries with integer (gap)-order constraints. *Theor. Comput. Sci.*, 116(1&2):117–149, 1993.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Rus91] Michaël Rusinowitch. Theorem-proving with resolution and superposition. *J. Symb. Comput.*, 11(1/2):21–49, 1991.

[RV01] Alexandre Riazanov and Andrei Voronkov. Splitting without backtracking. In Bernhard Nebel, editor, *IJCAI*, pages 611–617. Morgan Kaufmann, 2001.

[RW69] George A. Robinson and Larry Wos. Paramodulation and theorem-proving in first-order theories with equality. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 4*, pages 135–150. Edinburgh University Press, 1969.

[Seg02] Robert Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, University of Birmingham, 2002.

[SLM09] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[Spr04] Jeremy Sproston. Model checking for probabilistic timed systems. In Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, Joost-Pieter Katoen, and Markus Siegle, editors, *Validation of Stochastic Systems*, number 2925

in Lecture Notes in Computer Science, pages 189–229. Springer Berlin Heidelberg, January 2004.

[SS96] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[Sut09] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[Tri98] Stavros Tripakis. *L'Analyse Formelle des Systèmes Temporisés en Pratique*. PhD thesis, Université Joseph Fourier – Grenoble 1, 1998.

[WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *CAV*, pages 88–97, 1998.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In Renate Schmidt, editor, *Proceedings of the 22nd International Conference on Automated Deduction, CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.

[Wei99a] Christoph Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction, CADE-16*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 378–382. Springer, 1999.

[Wei99b] Volker Weispfenning. Mixed real-integer linear quantifier elimination. In Sam Dooley, editor, *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation, Vancouver, BC (ISSAC 99)*, pages 129–136. ACM Press, New York, July 1999.

[Wei01] Christoph Weidenbach. Combining superposition, sorts and splitting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 27, pages 1965–2012. Elsevier, 2001.

[WGR96] Christoph Weidenbach, Bernd Gaede, and Georg Rock. SPASS & FLOTTER version 0.42. In Michael A. McRobbie and John K. Slaney, editors, *CADE*, volume 1104 of *Lecture Notes in Computer Science*, pages 141–145. Springer, 1996.

[WZH07] Björn Wachter, Lijun Zhang, and Holger Hermanns. Probabilistic model checking modulo theories. In *QEST*, pages 129–140. IEEE Computer Society, 2007.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

Index