UNIVERSITÉ DE LIÈGE
Faculté des Sciences Appliquées
Département d'Électricité, Électronique et Informatique
Institut Montefiore

Techniques for verification of concurrent systems with invariants

Thèse présentée par
Pascal Fontaine
en vue de l'obtention du titre
de Docteur en Sciences Appliquées

Année Académique 2003–2004

Acknowledgements

First, I wish to express my deepest gratitude to my advisor, Pascal Gribomont, for his continuous support, for his endless patience, for the clear short lessons in various fields of logic,... and for bringing my writing ability to a decent level.

I am grateful to the other members of the jury, Bernard Boigelot, Baudouin Le Charlier, Jean-François Raskin, Michael Rusinowitch, and Pierre Wolper, who accepted to read and evaluate this thesis.

I wish to express special thanks to Silvio Ranise for numerous fruitful discussions, for often pointing me to the key articles I missed, and for his careful reading of several parts of this thesis. I am also grateful to David Déharbe for letting me play with haRVey, and always be quick at answering my questions about the code. More generally I am grateful to the CASSIS and MOSEL groups at the LORIA institute for making my couple of visits both enjoyable and interesting.

The Montefiore Institute has been a very pleasant place to live and work during those years, and I would like to thank all people who made it pleasant. Special thanks go to Bernard Boigelot and Axel Legay, for numerous technical — and less technical but always interesting — discussions, and for their support. I am indebted to Axel Legay, who read large parts of this thesis and gave me valuable comments.

Last but not least, I would like to thank all members of my family and friends (those who are still here, and those who aren't). This work could not have been done without them. I especially thank Vanessa, who furthermore endured my bad temper and doubts, and turned them daily into more positive thoughts...

Table of Contents

Chapter 1

Introduction

Correctness of computerized systems becomes a major issue in the modern society. Everyone who uses a computer has experienced computer "bugs", sometimes losing important data or potentially disclosing personal data to malicious Internet users. Undoubtedly, even for small companies, computer security is a major concern, and has an impact on their economical life. Computer bugs are also assumed to be responsible for several lethal accidents, notably in plane crashes, failure of emergency services, malfunction of medical equipments...[1] Confidence in algorithm correctness can be gained by extensive testing. However, other methods exist, which hopefully allow to reach a higher level of confidence in algorithm correctness.

In this thesis, we propose several techniques related to one of those methods: verification using *invariants*. An assertion I is an invariant[2] of a computer system if every execution step of the system preserves I, that is, every execution step from a state satisfying I leads the system to a state also satisfying I. If I is an invariant of a system, and if it is initially true, then it remains true during the whole execution of the system. Properties that can be verified with invariants are *invariance properties*. Invariance properties are *safety* properties, i.e., properties that state that something bad never happens (see [123] for formal definitions). To check that a system verifies the invariance property P, it "suffices" to find an invariant I such that P is a logical consequence of I, and such that I is a logical consequence of the initial conditions. A famous successful application of system verification using invariants is the design and proof with the B-method [3] of the automatic pilot of the *Météor* underground line, for the *Régie autonome des transports parisiens* (RATP).

Another important technique for verifying computer systems is model-

[1] See [130] for more information about computer related failures and their consequences.
[2] Sometimes also called *inductive invariant*.

checking (see [52] for an introduction). An important advantage of model-checking over invariant-based methods is that it does not require to build invariants. Properties are verified directly and automatically. However, using invariants to verify systems is generally less restricting for the language of systems: a predilection application of model-checking is the verification of finite-state systems, and in particular, hardware systems, although there has been interesting advances to directly verify infinite-state systems (for instance [2, 185]).

Some methods rely on both model-checking and invariant-based verification. For instance, in the method of invisible invariants [14, 149], the proof of parameterized systems has two aspects. First, model-checking techniques are used on a finite instance of a parameterized system, in order to heuristically build an invariant candidate for the system. Second, this invariant candidate is validated using invariant validation techniques. The power of the method not only comes from the model-checking techniques, but also from the ability to automatically validate invariants for parameterized systems, i.e., containing quantified assertions. More generally, model-checking and invariant-based methods both share techniques, for instance Binary Decision Diagrams and SAT-provers (see Chapter 6).

The method of invisible invariants is one of the techniques for automatic invariant generation. Other techniques can be found in [5, 27, 34, 180]. Even when they do not provide sufficiently strong invariants to have the required invariance properties as consequences, it might be easier to build a sufficiently strong invariant with the help of those automatically found invariants. Invariant generation methods often rely heavily on automatic validation of verification conditions similar to those generated for invariant validation. In general, invariant construction is made easier with a good invariant validation tool. The invariant can be manually designed by iteratively checking and correcting invariant candidates. The validation tool provides important information when validating the invariant fails: it gives the assertion for which the proof failed, the transition which may cause this assertion to be falsified, and an example. With automatic tools for invariant validation, it is even possible to write an invariant by trial and error, each failure giving important indications about how to correct the invariant candidate.[3]

Another motivation for designing powerful techniques to automatically validate invariants is to provide provers for the numerous verification conditions generated when using techniques allowing to incrementally design systems by refinement, with their invariants (see for instance [4, 97, 99]).

[3] The number of iterations being a measure of the user's capability to understand the algorithm to be verified.

Automatic invariant validation is important in several ways. First, it is interesting in itself, used as a verification technique. Second, invariant validation is sometimes used with other techniques, to provide automatic verification tools. Third, some invariant generation methods rely on techniques to validate verification conditions similar to the ones generated by invariant validation. Also, a tool to automatically validate invariants simplifies the use of methods for the development of systems by incremental transformations. The aim of this thesis is to provide techniques to automatically validate invariants.

1.1 Overview of the thesis

The logical framework used in this thesis is many-sorted first-order logic with equality. Definitions of this framework, the notions, and notations used throughout the text are given in Chapter 2. An adaptation of the Herbrand's Theorem to this framework is also exposed. This theorem is essential for this work, and more specifically for Chapters 5 and 9. Chapter 3 is an in-depth study of all formula transformation operations useful for automatic validation of formulas, or required by methods presented in Chapters 5 to 9: first, term representation is briefly discussed, then Skolemization and conjunctive normal form transformations are presented. Those transformations are described in details in order to give a precise idea of how they can be implemented. Attention is paid to their efficiency and to the quality of the produced results.

In Chapter 4, we present techniques to reduce the problem of invariant validation to the validation of logic formulas, called *verification conditions* or *proof obligations*. As for Chapter 3, sufficient details are given to implement the transformations. Generation of verification conditions is usually presented either using weakest precondition calculus (for instance [62, 72]) or using transition relations (see for example [3, 4]). Our presentation unifies both concepts, by defining the weakest precondition computation using transition relations.

Chapter 5, contains one important original contribution of this thesis. Using notions introduced in Chapters 3 and 4, we identify large classes of systems and formulas for which the invariant validation problem is decidable. More precisely, we show that a class of formulas is decidable, and we give sufficient conditions for proof obligations issued from validation of invariants for concurrent parameterized systems — more specifically, systems that are composed of a parameterized number of identical processes or nearly identical processes[4] — to belong to this decidable class. The verification of such systems is the subject of many recent works (for instance [1, 25, 40, 120]). Our sufficient criteria are fulfilled by many systems. Results from this chapter have been presented in [80].

In the remaining of the thesis, we present general techniques for automatic validation of verification conditions. Proof obligations generated to validate systems naturally contain interpreted predicates and functions. For example, equality is needed to express even the simple property of mutual exclusion. Many leader election algorithms use total order on the set of process identifiers. Algorithms on ring topology cannot be modelled without using the successor function. Data structures such as lists, queues, arrays... with accessors and constructors may be needed. Real-time systems do simple arithmetic on time and delays, ... One way to deal with interpreted operators is to augment the formula with theory axioms. For instance it is well known that a first-order logic formula is satisfiable in logic with equality if and only if it is satisfiable in logic without equality, provided axioms for equality are added to it (see for example [75]). However, several problems arise when using theory axioms. First, theories may not perfectly describe the operators. Second, theories often contain axioms schemata, that is infinite sets of formulas of a given form: axioms for equality contain one replacement axiom for each function and predicate. Last but not least, those axioms can make the proving task difficult. For instance, it is clear that equality reasoning needs specialized methods (see [58, 138]). Dedicated inference rules, like the paramodulation rule, were introduced to avoid equality axioms. Later associative and commutative theories were also directly included in the reasoning process [138]. This approach has the ambition to treat quantified formulas,[5] but does not deal with all specialized theories as met in verification. Decision procedures for specialized quantified theories, like for example linear arithmetic [37, 184], is another approach. As drawback the decision procedure does not easily accept language extensions. The language of linear arithmetic with quantifiers is decidable, but it suffices to add one unary uninterpreted predicate to make it undecidable [104], and thus invalidate any decision procedure.

Our approach is different from both the preceding ones. The satisfiability problem for ground (i.e., quantifier-free) logic with equality is decidable [6], and decision procedures for many other theories can be combined to give a practical decision procedure for ground formulas in the language of the com-

[4] For example a process can compare its identifier with the identifier of another process. This somewhat breaks symmetry.

[5] Two successful implementations of theorem proving tools for first-order logic are described in [158, 163].

bination (see Chapter 8). We follow the idea of building a decision procedure for quantified logic on top of a decision procedure for ground logic, itself built on top of a decision procedure for propositional logic. Chapter 5 will show that verification conditions can often be validated through the validation of ground verification conditions. Generally speaking, verification condition validation can naturally be handled by considering ground instances. In Chapter 9, we will present a heuristic instantiation method for more general cases.

Chapters 6 to 8 are dedicated to the study of the satisfiability decision procedure for sets of ground formulas. First, Chapter 6 describes how it is possible to reduce efficiently the satisfiability problem for sets of general formulas to the satisfiability of sets of literals, using Boolean methods like Binary Decision Diagrams (BDDs) and SAT-solvers. Although SAT-solvers are impressively powerful tools to decide propositional formulas, the fact that they return full models is a major practical problem for integrating them into solvers for richer quantifier-free logics. As BDDs return partial models, the amount of work for provers for sets of first-order literals is much smaller when they are associated with BDDs than with SAT-provers. We propose in Section 6.4 an efficient algorithm to circumvent this problem. We have implemented this algorithm into the haRVey prover [67], and obtained good results on practical examples.

A set of first-order logic ground literals[6] without equality is unsatisfiable if and only if the set contains an atom and its negation. For instance the set $\{p(a), \neg p(b)\}$ is satisfiable whereas $\{p(a), \neg p(a)\}$ is not. When equality is used, an unsatisfiable set may not contain an unsatisfiable pair of literals: consider as an example the set $\{p(a), \neg p(b), a = b\}$. Congruence closure allows to decide satisfiability of sets of literals with equalities, uninterpreted functions and predicates. In collaboration with propositional methods, it can provide a technique to validate ground formulas with only uninterpreted predicates and functions. For this collaboration to be efficient (see Chapter 6), the prover for sets of literals should be proof-producing, i.e., the algorithm should not only decide the satisfiability of the set of literals, but should also return the subset of literals necessary to prove unsatisfiability. Another useful facility is the ability to accept literals incrementally, that is, literals are given one by one to the procedure, and satisfiability is checked after each addition with only incremental work. In Chapter 7, we present a new incremental, proof producing algorithm for deciding sets of literals

[6]The satisfiability problem for ground formulas is decidable if (and only if) it is decidable for literal conjunctions. Indeed every ground formula can be transformed into disjunctive normal form.

with equality and uninterpreted predicates and functions. Usually, congruence closure algorithms do not have those functionalities, and furthermore do not directly handle inequalities and predicates. Despite of these facts, our algorithm has the same complexity as the best known congruence closure algorithms. An earlier version of this algorithm has been presented in [79].

The Nelson-Oppen combination framework [135] aims at creating a decision procedure for the union of two theories on disjoint languages, from the decision procedures for the quantifier-free fragment in each theory. For instance the set of literals

$$L = \{x \leq y,\ y \leq x + f(x),\ P(h(x) - h(y)),\ \neg P(0),\ f(x) = 0\}$$

contains uninterpreted predicates and functions augmented with linear arithmetic on integer. The combination scheme provides a decision procedure for such sets of literals, from a decision procedure for uninterpreted predicates and functions (i.e., the empty first-order theory) and a decision procedure for linear arithmetic. This is possible because languages are disjoint, the only shared symbols being the equality and variables. However, the standard version of the Nelson-Oppen technique tackles only one-sorted, *stably infinite* first-order theories. A theory is stably infinite if every satisfiable quantifier-free formula in its language admits a model with a domain of infinite cardinality. The stably infinite requirement is often not acceptable in a verification context, since it does not allow to consider theories with only finite models. Another restriction on theories comes from the first-order requirement: for instance, it is not possible to consider a theory with only finite models of unbounded size, useful for instance to describe the sort of processes in a parameterized algorithm.

The Nelson-Oppen scheme has recently been adapted to a many-sorted framework [177], and to handle non-stably infinite theories [176]. Those two enhancements were presented independently. In Chapter 8, we propose a unifying version in the continuity of both previous ones, which further relaxes the stably infinite requirement. Notably, some non-stably infinite theories can now be combined with the theory of arrays. Also, the combination scheme is presented here using a semantic notion of theory, allowing to handle non first-order theories. Some results from this Chapter have been presented in [81].

Finally, in Chapter 9, we will present a heuristic instantiation method to reduce the problem of satisfiability of quantified verification conditions to the problem of satisfiability of sets of ground instances. Its efficiency is due to the fact that the verification conditions are often large, but easy to validate since a small number of instances of quantified assertions are usually enough for the validation. This heuristic has given good results on many practical verification conditions generated by the invariant validation tool CAVEAT.

Chapter 2

Notations, definitions, and general theorems

We introduce here the logical framework and notations used throughout this work. The classical notions of logical language, term, formula, interpretation, satisfiability,... are defined.

The chosen framework is many-sorted first-order logic. The reason to opt for a first-order logic comes from a tradeoff between expressive power and possibilities for an automated treatment. Writing transition functions, properties, or state sets (see Chapter 4) into first-order logic is a constraint, but to a reasonable extent. Many (infinite state, parameterized) algorithms may be studied and verified within a first-order framework (see for example [80, 102, 101, 122, 154]). Quantifiers allow to directly express, for instance, that properties are fulfilled by every process in a system with a parameterized number of symmetric processes. Sometimes, more expressive languages are preferred, even if this means that proofs require user interaction [145, 164, 9, 8].

Using a many-sorted framework is natural in a verification context. Program variables usually have a predefined purpose, and a type (specifying a restricted set of possible values) related to this purpose. They can be partitioned into classes such that comparing two variables of different classes does not make much sense. The variable sorts specify the operations allowed with them at a nearly syntactical level. Verification conditions may thus reflect these constraints. The results obtained within a many-sorted framework are stronger than those obtained within an unsorted framework (see Chapters 5 and 8).

Many-sorted first-order logic [82, 71] can be translated to unsorted first-order logic, and many results for unsorted first-order logic can be transferred to many-sorted logic. This is true in particular for the Compactness, Löwenheim-Skolem, Löwenheim-Skolem-Tarski and Herbrand's Theorems (see for example [75, 71] for the standard version of those theorem). If recasting those theorems in many-sorted logic is not fundamental, specialized versions may be stronger than the classic ones. We give here a many-sorted version of the Herbrand's Theorem for several reasons: first, it is a strong refinement of its classical counterpart, second, the Herbrand's Theorem for logic with equality is not part of common knowledge[1], and finally, the Herbrand's Theorem is the basis for dealing with quantifiers in CAVEAT (see Chapters 5 and 9).

Definition 1 (Many-sorted first-order language). *A* many-sorted first-order language *is a tuple* $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, r, d \rangle$ *such that*

- $\mathcal{S}$ *is a countable non-empty set of sorts (or types);*
- $\mathcal{V} = \bigcup_{\tau \in \mathcal{S}} \mathcal{V}_\tau$ *is the (countable) union of disjoint countable sets* $\mathcal{V}_\tau$ *of variables of sort* τ*;*
- $\mathcal{F}$ *is a countably infinite set of function symbols;*
- $\mathcal{P}$ *is a countably infinite set of predicate symbols;*
- r $(\mathcal{F} \cup \mathcal{P} \to \mathbb{N})$ *assigns an arity to each function symbol in* $\mathcal{F}$ *and each predicate symbol in* $\mathcal{P}$*;*
- d $(\mathcal{F} \cup \mathcal{P} \to \mathcal{S}^\star)$ *assigns a sort in* $\mathcal{S}^{r(f)+1}$ *to each function symbol* $f \in \mathcal{F}$ *and a sort in* $\mathcal{S}^{r(p)}$ *to each predicate symbol* $p \in \mathcal{P}$.

Nullary predicates are propositions*, and nullary functions are* constants*. The* signature *of* $\mathcal{L}$ *is the tuple* $\langle \mathcal{S}, \mathcal{F}, \mathcal{P}, r, d \rangle$.

It is assumed that symbols are not overloaded. That is, a given function or predicate cannot be assigned to different sorts. However, it is handy to use the same symbol with arguments of different sorts. In that case it suffices to consider that such symbols are implicitly "decorated" with their sort declaration. One example is the equality symbol "=", which is decorated into one symbol for every sort.

Sometimes it is convenient to consider the *restriction* or *subset* of a given many-sorted language $\mathcal{L}$: a restriction $\mathcal{L}'$ of $\mathcal{L}$ is a many-sorted language such that the sets of sorts, variables, function and predicate symbols are subsets of the corresponding sets of $\mathcal{L}$.

Terms, atoms (atomic formulas), literals, and formulas of a many-sorted language are defined the usual way:

[1] A version of Herbrand's Theorem for unsorted first-order logic with equality may be found in [84].

Definition 2 (Terms). *Given a sort τ, the sets of* τ-terms *in a many-sorted language $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, r, d \rangle$ are the smallest sets such that*

- *for each $x \in \mathcal{V}_\tau$, x is a τ-term;*
- *for each function symbol $f \in \mathcal{F}$ of sort $\langle \tau_1, \ldots \tau_n, \tau \rangle$, $f(t_1, \ldots t_n)$ is a τ-term if $t_1, \ldots t_n$ are $\tau_1, \ldots \tau_n$-terms respectively.*

Sort$(t) = \tau$ *if t is a τ-term.*

The *depth* of t is zero if t is a variable or a constant, and is $1 + d$ if t is $f(t_1, \ldots t_n)$, where d is the largest depth of terms t_i ($i \in \{1, \ldots n\}$). In Section 3.3, one concern will be the depth of Skolem terms.

Definition 3 (Atomic Formulas). *The set of* atomic formulas *is the smallest set such that*

- *for each predicate symbol $p \in \mathcal{P}$ of sort $\langle \tau_1, \ldots \tau_n \rangle$, $p(t_1, \ldots t_n)$ is an atomic formula if $t_1, \ldots t_n$ are $\tau_1, \ldots \tau_n$-terms respectively;*
- *for each pair t, t' of τ-terms, $t = t'$ is an atomic formula.*

A literal *is either an atomic formula, or the negation $\neg p$ of an atomic formula p; notation $t \neq t'$ stands for $\neg(t = t')$; universally true and false atoms are respectively noted $\top$ and $\bot$.*

Definition 4 (Formulas). *The set of* formulas *is the smallest set such that*

- *every atomic formula is a formula;*
- *for each formula φ, $\neg\varphi$ is a formula;*
- *for each formula φ and each variable $x \in \mathcal{V}$, $\forall x\, \varphi$, $\exists x\, \varphi$ are formulas;*
- *for each formulas φ_1 and φ_2, $(\varphi_1 \circ \varphi_2)$ is a formula if $\circ \in \{\vee, \wedge, \Rightarrow, \equiv\}$.*

A pair of parentheses can be omitted when there is no ambiguity, for example using associativity of conjunctions, or associativity of disjunctions. A pair of parentheses can also be added to improve readability. The *formula depth* is the natural extension of the term depth: the formula depth of an atomic formula corresponds to its term depth, otherwise the formula depth of G is $1 + d$ where d is the largest depth of the proper subformulas of G.

Among all formulas, two forms are of special interest: a *clause* is a disjunction ($\vee$) of literals, and a *conjunctive normal form* is a conjunction ($\wedge$) of clauses. Many proof methods assume formulas are in conjunctive normal form (see for instance Chapter 6); in those cases a preliminary step to check satisfiability of a formula is to translate it into conjunctive normal form (see Section 3.4).

The language is often augmented with the if-then-else connector. It does not extend the expressive power: formula 'if A then B else C' is logically equivalent to formula $(A \Rightarrow B) \wedge (\neg A \Rightarrow C)$. Using DAG representation with maximal sharing (see Section 3.1), formulas containing such connectors can be translated linearly to formulas using only classical connectors.[2]

The set of variables of formula φ (or term) in a many-sorted language $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, r, d \rangle$ is the subset $\mathrm{Var}(\varphi)$ of $\mathcal{V}$ used in the formula (term). The set of *free* variables of a formula φ is $\mathrm{Free}(\varphi)$ defined inductively on the structure of the formula: the free variables of an atomic formula are the variables of the formula, the free variables of a Boolean combination of formulas is the union of the free variables of the formulas in the combination, and the set of free variables for $\exists x\, \varphi$ and $\forall x\, \varphi$ contains all free variables in φ but x.

A formula is *quantifier-free* if it does not contain the symbols $\exists$ and $\forall$. A formula (or term) φ is *ground* if it does not contain any variable. Ground formulas are always quantifier-free. A formula φ is *closed* if it does not contain free variables.

Definition 5 (Prenex form). *A formula is in* prenex form *if it is written as*

$$Q_1 x_1 \dots Q_n x_n (\varphi)$$

where $Q_1, \dots Q_n \in \{\exists, \forall\}$, $x_1, \dots x_n \in \mathcal{V}$, and φ is quantifier-free. The formula φ is the matrix *of the prenex form.*

Definition 6 (Skolem form). *A formula is in* Skolem form *if it is in prenex form without existential quantifiers.*

In Section 3.3, Skolemization transformations are introduced. They permit to transform a formula into Skolem form, preserving satisfiability. In Section 3.3 a more flexible notion of Skolem form is introduced, which does not assume that the formula is in prenex form.

Definition 7 (Interpretation and model). *An* interpretation *of a formula in a many-sorted first-order language $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, r, d \rangle$ is a pair $\mathcal{I} = \langle D, I \rangle$ where*

- *D assigns a non-empty domain D_τ (set) to each sort $\tau \in \mathcal{S}$. Those sets are not necessarily disjoint;*

[2] The if-then-else construction is also sometimes used inside terms (See Section 3.5).

- *I assigns an element in D_τ to each variable of sort τ;*
- *I assigns a function $D_{\tau_1} \times \ldots D_{\tau_n} \longrightarrow D_\tau$ to each function symbol $f \in \mathcal{F}$ of sort $\langle \tau_1, \ldots \tau_n, \tau \rangle$;*
- *I assigns a function $D_{\tau_1} \times \ldots D_{\tau_n} \longrightarrow \{\top, \bot\}$ to each predicate symbol $p \in \mathcal{P}$ of sort $\langle \tau_1, \ldots \tau_n \rangle$;*
- *the identity is assigned to the equality sign (=).*

An interpretation $\mathcal{I}$ assigns a value $\mathcal{I}[t]$ in D_τ to each τ-term t. Similarly, $\mathcal{I}$ assigns a value[3] *$\mathcal{I}[\varphi]$ in $\{\top, \bot\}$ to each formula φ. An interpretation $\mathcal{I}$ is a* model *of formula φ (noted $\mathcal{I} \Vdash \varphi$) if $\mathcal{I}[\varphi] = \top$, and it is a* model *of the set of formulas S (noted $\mathcal{I} \Vdash S$) if it is a model of every formula in S.*

Interpretation $\mathcal{I}_{x_1/d_1,\ldots x_n/d_n}$ is defined to be equal to $\mathcal{I}$ except that it interprets each variable x_i as d_i, where $d_i \in D_\tau$ if x_i is of sort τ. Given an interpretation $\mathcal{I}$ in language $\mathcal{L}$, a *restriction* of $\mathcal{I}$ to a subset of $\mathcal{L}$ is the interpretation equal to $\mathcal{I}$ for every domain, function symbol, predicate symbol in the subset of $\mathcal{L}$.

Two formulas G and H are said to be logically equivalent ($G \longleftrightarrow H$) if they have the same models. Formula G is a logical consequence of formula H ($H \models G$) if every model of H is a model of G. A formula G is a logical consequence of a set of formulas S ($S \models G$) if every model of S is a model of G.

Definition 8 (Satisfiability). *A formula (a set of formulas) is* satisfiable *if there exists a model for it. If not, it is said to be* unsatisfiable. *A formula is* valid *(or is a* tautology*) if every interpretation is a model for it.*

Interpreted predicates and functions will often be used in this document. This means that the interpretations considered will be a subset of the set of all possible interpretations. In this context, a formula will be satisfiable if there exists an interpretation in the considered subset which makes the formula true. If every interpretation in the subset is a model of the formula, it is valid. For example, a given sort τ may be the sort of integers, and the symbol "+" may be considered as being the addition on integers. A formula would be unsatisfiable if and only if every interpretations such that $D_\tau = \mathbb{N}$ and such that $I[+]$ is the addition makes the formula false.

A fundamental concept for Skolemization and structural normal form (see Chapter 3) is the *polarity* of a subformula inside a formula:

[3] We assume the usual meaning for connectors.

Definition 9 (Subformula polarity). *If G is a formula, the sets of subformulas $\Sigma^+(G)$ and $\Sigma^-(G)$ are the smallest sets such that*

- $G \in \Sigma^+(G)$;
- *if* $\neg M \in \Sigma^+(G)$ *(resp.* $\Sigma^-(G)$*), then* $M \in \Sigma^-(G)$ *(resp.* $\Sigma^+(G)$*);*
- *if* $A \wedge B \in \Sigma^\pm(G)$ *or* $A \vee B \in \Sigma^\pm(G)$*, then* $\{A, B\} \subset \Sigma^\pm(G)$*;*
- *if* $A \Rightarrow B \in \Sigma^\pm(G)$*, then* $A \in \Sigma^\mp(G)$ *and* $B \in \Sigma^\pm(G)$*;*
- *if* $A \equiv B \in \Sigma^+(G) \cup \Sigma^-(G)$*, then* $\{A, B\} \subset \Sigma^+(G)$ *and* $\{A, B\} \subset \Sigma^-(G)$*;*
- *if* $\forall x\, A \in \Sigma^\pm(G)$ *or* $\exists x\, A \in \Sigma^\pm(G)$*, then* $A \in \Sigma^\pm(G)$*.*

The set of subformulas $\Sigma(G)$ is the union of $\Sigma^+(G)$ and $\Sigma^-(G)$.

Given a formula G, the set $\Sigma^+(G)$ ($\Sigma^-(G)$) is the set of all subformulas with positive (respectively negative) polarity in G. The set $\Sigma(G)$ is the set of all subformulas of G. Notice that $\Sigma^+(G)$ and $\Sigma^-(G)$ are not disjoint in general. Subformula occurrences dominated by "$\equiv$" occur positively as well as negatively in G by definition. In presence of if-then-else connectors, if the formula if A then B else C belongs to $\Sigma^\pm(G)$, then $A \in \Sigma^+(G) \cap \Sigma^-(G)$ and $\{B, C\} \subset \Sigma^\pm(G)$.

Subformula polarities are exploited through the following theorem [75]:

Theorem 1 (Implicational Replacement Theorem). *Let $G(H)$ be a formula such that $H \notin \Sigma^-(G(H))$ (respectively $H \notin \Sigma^+(G(H))$). If $\mathcal{I} \Vdash A \Rightarrow B$ (resp. $\mathcal{I} \Vdash B \Rightarrow A$), then $\mathcal{I} \Vdash G(A) \Rightarrow G(B)$.*

Proof. This is easily proved by induction on the depth of the formula. □

The Herbrand's Theorem used in conjunction with the Compactness Theorem[4] provides the basis for the completeness of methods to study first-order logic formulas (see for example [75]). Both the Completeness and Herbrand's Theorem still hold in a many-sorted framework. However, recasting the Herbrand's Theorem into many-sorted logics provides refined results.

Definition 10 (Herbrand sets). *Given a many-sorted first-order language $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, r, d \rangle$ and a formula φ in language $\mathcal{L}$, the family H of* Herbrand sets *for φ assigns for each sort $\tau \in \mathcal{S}$, the smallest set H_τ such that*

[4] The Compactness Theorem guarantees that any unsatisfiable set of formulas has a finite unsatisfiable subset.

- *for each constant $a \in \mathcal{F}$ of sort τ used in φ, $a \in H_\tau$. If no constant of sort τ is used in φ, then H_τ includes one new arbitrary constant;*

- *for each function $f \in \mathcal{F}$ of sort $\langle \tau_1, \dots \tau_n, \tau \rangle$, and for all elements $t_1 \in H_{\tau_1}, \dots, t_n \in H_{\tau_n}$, the term $f(t_1, \dots t_n)$ is in H_τ;*

For instance, the Herbrand set of formula $\forall x \, [p\,(f\,(x)) \vee a = g\,(x)]$ in a one-sorted language is $\{a, f(a), g(a), f(g(a)), g(f(a)), \dots\}$.

Definition 11 (Congruence). *Given a set of terms T, a* congruence relation $\leftrightarrow_{\mathcal{C}}$ *on T is a reflexive, symmetric and transitive relation such that if $f(t_1, \dots t_n), f(t'_1, \dots t'_n) \in T$ and if $t_i \leftrightarrow_{\mathcal{C}} t'_i$ for each $i \in \{1, \dots n\}$, then $f(t_1, \dots t_n) \leftrightarrow_{\mathcal{C}} f(t'_1, \dots t'_n)$.*

Each interpretation $\mathcal{I}$ defines a congruence $\leftrightarrow_{\mathcal{I}}$ on the set of terms: $t \leftrightarrow_{\mathcal{I}} t'$ if and only if $\mathcal{I} \Vdash t = t'$.

Definition 12 (Herbrand interpretation). *Let φ be a formula in a many-sorted first-order language. Let H be the family of Herbrand sets for φ. An* Herbrand interpretation *$\mathcal{I} = \langle D, I \rangle$ in language $\mathcal{L}$ for formula φ, is an interpretation such that*

- *there exists a congruence $\leftrightarrow_{\mathcal{C}}$ such that D assigns to sort τ the quotient set of H_τ by congruence $\leftrightarrow_{\mathcal{C}}$;*

- *$\mathcal{I}[a] = \mathcal{C}_a$ for any constant a in φ, where $\mathcal{C}_a$ is the class associated with a in congruence $\leftrightarrow_{\mathcal{C}}$;*

- *for any term $f(t_1, \dots t_m)$, if $\mathcal{I}[t_i] = \mathcal{C}_{t'_i}$ for each $i \in \{1, \dots n\}$, then $\mathcal{I}[f(t_1, \dots t_n)] = \mathcal{C}_{f(t'_1, \dots t'_n)}$, where $\mathcal{C}_t$ is the class associated with t in $\mathcal{C}$.*

For any ground term t and any Herbrand interpretation $\mathcal{I}$ for a formula containing t, $\mathcal{I}[t] = \mathcal{C}_t$.

Theorem 2 (Herbrand). *A closed Skolem formula in a many-sorted first-order language is satisfiable if and only if it has a Herbrand model.*

Proof. The condition is trivially sufficient. To prove it is also necessary, assume $\varphi = \forall x_1, \dots \forall x_n \, \varphi'$ is a satisfiable closed Skolem formula in a many-sorted first-order language $\mathcal{L}$ where φ' is quantifier-free, and $\mathcal{I}$ is a model for φ. Interpretation $\mathcal{I}$ defines a congruence on the set of ground terms $\leftrightarrow_{\mathcal{I}}$ such that $t \leftrightarrow_{\mathcal{I}} t'$ if and only if $\mathcal{I} \Vdash t = t'$.

For any ground term t, interpretation $\mathcal{H}$ is defined such that $\mathcal{H}[t] = \mathcal{C}_t$ where $\mathcal{C}_t$ is the class of t in relation $\leftrightarrow_{\mathcal{I}}$. For any ground predicate $p(t_1, \dots t_n)$, $\mathcal{H}$ is such that $\mathcal{H}[p(t_1, \dots t_n)] = \mathcal{I}[p(t_1, \dots p_n)]$.

Given an interpretation $\mathcal{J}$, interpretation $\mathcal{J}_{x_1/t_1,\dots x_n/t_n}$ where each t_i is a ground term of the sort of x_i is defined to be $\mathcal{I}_{x_1/\mathcal{J}[t_1],\dots x_n/\mathcal{J}[t_n]}$. Then $\mathcal{H}[\forall x_1,\dots \forall x_n\, \varphi']$ is true if and only if $\mathcal{H}_{x_1/t_1,\dots x_n/t_n}[\varphi']$ is true for all ground terms $t_1,\dots t_n$ of appropriate sorts. By construction,

$$\mathcal{H}_{x_1/t_1,\dots x_n/t_n}[\varphi'] = \mathcal{I}_{x_1/t_1,\dots x_n/t_n}[\varphi'].$$

To conclude, it suffices to notice that $\mathcal{I}_{x_1/t_1,\dots x_n/t_n}[\varphi']$ is true for all terms $t_1,\dots t_n$ of appropriate sort as $\mathcal{I}$ is a model for φ. $\square$

Notice that Theorem 2 holds only for closed formulas: a Skolem formula with free variables does not have the same Herbrand sets that the same formula where every free variable has been replaced by a new constant.

Corollary 1. *A closed Skolem formula $\varphi = \forall x_1,\dots \forall x_n\, \varphi'$ in a many-sorted first-order language is satisfiable if and only if it there exists a finite conjunction of ground instances $\varphi'(t_1,\dots t_n)$, where $t_1,\dots t_n$ are ground terms of appropriate sort.*

Chapter 3

Formula transformations

Techniques for automated verification of proof obligations often require formulas to be in a given form. Formulas can be put in the required form using the transformations presented in this chapter. Those transformations may have an important influence on the proving process, and thus require a careful design. First, formula representation is discussed, as it is essential to evaluate the cost of every transformation. Then basic formula simplifications are briefly outlined.

A first and crucial step in the automated treatment of quantified formulas is Skolemization, permitting to eliminate every quantifier having an existential semantic (in a satisfiability check context). This transformation is used in Chapters 5 and 9. A survey of modern techniques for Skolemization is given in Section 3.3.

Getting a conjunctive normal form associated with a formula is also essential, as many techniques are only applicable to conjunctive normal forms. As for Skolemization, the naïve method is well known, but its complexity makes it unsuitable for practical cases. Section 3.4 introduces satisfiability preserving (p-)definitional normal form transformations. They are presented only in the quantifier-free framework, as they are here used only in this case (see Chapters 6 to 8).

The convenience of if-then-else terms has already been mentioned. This construct is used in Chapter 4. Common benchmarks for satisfiability modulo theories [156] intensively rely on such terms. We finally introduce an elimination method of those if-then-else constructs, also in the quantifier-free case.

In this chapter, a special care is taken to study the transformations in presence of interpreted predicates. We not only show that the transformations are satisfiability-preserving, but also state clearly what can be expected to be preserved in the models of the formulas. In practice, it means interpreted predicates are preserved by those transformations.

3.1 Formula representation

Formulas can obviously be represented by trees. Nodes are labelled by predicate, function, or connector names, and children are representations of the subformulas or subterms. Leaves are thus either Boolean constants ($\top$, $\bot$), Boolean propositions, individual constants, or variables.

Some Boolean connectors are associative. Rather than writing $(a \wedge b) \wedge c$, one preferably writes $a \wedge b \wedge c$. Representations by trees can reflect the associative property of connectors and functions: corresponding nodes may also have a variable number of children. As a result formulas can be further simplified (see Section 3.2).

Typical formulas use multiple instances of the same subformulas, atoms, and terms. Using Directed Acyclic Graphs (DAG for short) can be valuable in this context. Several instances of the same subformula, atom, or term are represented by the same node.[1] In this context, a node may have more than one parent. For instance, the parents of a shared term are the expressions in which this shared term is a direct subexpression.

DAG representations can be exponentially smaller than the size of the tree representations. Consider formulas $\varphi_{0,n}$, with $\varphi_{i,n} = \alpha_i(\varphi_{i+1,n}, \varphi_{i+1,n})$ if $i < n$, and $\varphi_{n,n} = a$.[2] The number of nodes of the tree representation of $\varphi_{0,n}$ (where every $\varphi_{i,n}$ has been replaced by its definition) is exponential with respect to n, whereas a DAG representation of $\varphi_{0,n}$ may be linear with respect to n.

Among all DAG representations of a formula or term, the one which uses *maximal sharing* (or *perfect sharing*, or *aggressive sharing*) is often considered [117, 158]. Maximal sharing is used when all syntactically identical nodes are merged. The maximal sharing DAG representation is the smallest DAG representation. Furthermore two identical terms (or subformulas) are represented by the same node; this means that testing for syntactical equality of two terms can be done by checking if the nodes are identical, which comes in practice to check for pointer equality. This is done in constant time, whereas testing for term syntactical equality in a context of non maximal sharing representation is done in linear time with respect to the size of term representations.

[1] The same goal can be achieved by renaming common parts of formulas [141].

[2] For each i, α_i is a Boolean connector, a function identifier, or a predicate identifier, such that each $\varphi_{i,n}$ is a well-formed formula or term.

In order to use maximal sharing, one must be able to quickly identify expressions which are already represented in memory. This means to be able to find nodes which have already been introduced. The canonical solution for this is to use indexing techniques. Using hash tables, every introduced node is added in a table indexed by a hash key calculated from the label and from the hash keys of the children. A good hash function will distribute nodes evenly in the table, and allows to find in constant time whether or not a given node has already an equivalent in memory. Two nodes are equivalent only if they have the same label.[3]

DAG representations are smaller and allow to test syntactical equality in constant time. However some care must be taken to reimplement algorithms in order to deal with directed acyclic graphs, even in cases where the corresponding efficient algorithms operating on trees are trivial. Linear algorithms for tree representation can often be translated into linear algorithms[4] for DAG representation. It is the case for all algorithms such that expressions are substituted functionally, i.e., every subexpression e is substituted by $f(e)$ without dependence on the context. Another example is searching in the DAG for a given atom: depth first search in DAGs is linear in size of the DAGs (like depth first search in trees). Unfortunately, this is not the case for all algorithms; consider an algorithm which would rename quantified variables such that no two quantified variable (in tree representation) would be the same.[5] This is linear with respect to the tree representation although the output DAG may be exponentially larger than the input DAG.

Finally, let us mention that term representation and indexing is a critical issue in automated reasoning as in various other fields as, for instance, logic and functional programming. More on term representation and indexing can be found in [152].

3.2 Formula simplification

Formula simplification is not a well-defined task: one could for example require that simplifying a valid formula gives the Boolean atom “true”. From this point of view, the simplification task completely includes the verification task. In particular, there is no perfect simplification procedure for undecidable languages. To better state what *simplification* is in our context, one could say that it is rewriting a formula to a simpler equivalent formula, i.e.,

[3]To prevent collisions of nodes which do not have the same label, a hash table can be associated to every possible node label rather than using one large hash table for all nodes.

[4]If reading and writing into hash tables are assumed to be constant-time operations.

[5]In Section 3.3, this is called *rectifying* a formula.

such that the total number of nodes and edges in the formula representation is decreased. This is somewhat dependent on the way formulas are represented: as an example, rules which guarantee simplification for tree representations may not be suitable for DAG representations, because they may decrease sharing. It is not required for the simplification task to be a perfect one, and notably it may not reduce valid formulas to true, and unsatisfiable formulas to false.

We further require for simplification to be harmless and "without cost". Harmless means it should not make the verifying job harder, it should not remove structural information. In fact, it is often observed (for instance in [117]) that the use of some transformation rules can be counter-productive. "Without cost" essentially means linear,[6] as the global complexity is unaffected by one supplementary pass in the DAG representation of the formula. When these two requirements are met (harmless and without cost) there is no reason not to use simplification, even if it may not be very useful.

There are several reasons to want some kind of simplification. First, simplification may be necessary to guarantee completeness of the proving method; some propositional decision procedures for instance do not handle Boolean constants, and it is required to eliminate them before any proving attempt. Second, sometimes formulas can be handled by the prover after they have been simplified, but not before [110]: simplification may dramatically decrease the size of a formula. Third, automatic formula generation (see for instance Chapter 4.) and transformation may introduce many subformulas which can be rewritten in a more compact and natural way using very simple rules. Finally, verification often means human interaction, and it is much more valuable for human expertise to present formulas in a simpler form.

Figure 3.1 presents rewrite rules to eliminate Boolean constants inside formulas, for n-ary connectors $\wedge$ and $\vee$ and binary connectors $\Rightarrow$ and $\equiv$. It is not difficult to show that when no rule can be applied anymore the obtained formula is either a Boolean constant, or a formula which does not contain any Boolean constant.[7] Furthermore, the rewrite system (see for example [60, 59] for more information on rewrite systems) in Figure 3.1 is terminating. This means that any derivation will eventually end in a state where no more rule can be applied. It is also confluent, that is the formula obtained when no more rule can be applied does not depend on the derivation. Confluent and terminating rewrite systems are called convergent.

Using the simple rules in Figures A.1 and A.4 in Appendix A — respec-

[6] It is opposed to "perfect": if a simplification process is without cost, one cannot expect it would reduce valid and unsatisfiable formulas to their corresponding Boolean constants.

[7] Provided the only Boolean connectors used are $\wedge$, $\vee$, $\Rightarrow$ and $\equiv$, and provided no if-then-else like construction is used inside terms.

$$
\begin{array}{rclcrcl}
 & & & & \top \wedge A_1 \wedge \ldots \wedge A_n & \rightarrow & A_1 \wedge \ldots \wedge A_n \\
\neg\top & \rightarrow & \bot & & \bot \wedge A_1 \wedge \ldots \wedge A_n & \rightarrow & \bot \\
\neg\bot & \rightarrow & \top & & \top \vee A_1 \vee \ldots \vee A_n & \rightarrow & \top \\
 & & & & \bot \vee A_1 \vee \ldots \vee A_n & \rightarrow & A_1 \vee \ldots \vee A_n
\end{array}
$$

$$
\begin{array}{rclcrcl}
A \Rightarrow \top & \rightarrow & \top & & \text{if } \top \text{ then } A \text{ else } B & \rightarrow & A \\
A \Rightarrow \bot & \rightarrow & \neg A & & \text{if } \bot \text{ then } A \text{ else } B & \rightarrow & B \\
\top \Rightarrow A & \rightarrow & A & & \text{if } C \text{ then } \top \text{ else } B & \rightarrow & C \vee B \\
\bot \Rightarrow A & \rightarrow & \top & & \text{if } C \text{ then } \bot \text{ else } B & \rightarrow & \neg C \wedge B \\
A \equiv \top & \rightarrow & A & & \text{if } C \text{ then } A \text{ else } \top & \rightarrow & \neg C \vee A \\
A \equiv \bot & \rightarrow & \neg A & & \text{if } C \text{ then } A \text{ else } \bot & \rightarrow & C \wedge A
\end{array}
$$

Figure 3.1: Rules for $\top$ and $\bot$ elimination

tively to simplify some negations and repeated subformulas — can safely reduce the size of formulas, in particular by allowing to detect very local inconsistencies and tautologies. The rewrite system obtained by considering the Boolean simplification rules in Figures 3.1, A.1 and A.4 and the term simplification rules in Figures A.2 and A.3 remains convergent. Notice that those simplification rules may be used either at the construction of each node of a formula, or as a completely separate treatment step.

Convergence of a rewrite system is not easily obtained. As an example, the rewrite system furthermore including the rule $\neg(A \Rightarrow \neg B) \rightarrow A \wedge B$ is not confluent anymore. In other words, several totally simplified formulas exist, depending on the order of rule application. For example, $\neg(p \Rightarrow \neg q) \vee (p \Rightarrow \neg q)$ might be simplified either to $\top$ if the simplification rules for disjunctions is given precedence or to $(p \wedge q) \vee (p \Rightarrow \neg q)$ if the last rule is applied first. This means the rewrite system must be supplied with some kind of rule application order.

The simplification rules should be chosen depending on the application field, and on their cost within the entire system, for instance, with respect to the representation of formulas, or in regard of the validation method in use. In [117], the author introduces some kind of restricted unit resolution as a simplification step, and in [141] some rules for simplification of equalities with one quantified term are presented. Finally, notice that it may be valuable to use associativity for conjunction and disjunctions. On the other hand, systematic use of associativity may lead to augment the total number of nodes and edges in the DAG representation of a formula, by decreasing sharing.

3.3 Skolemization

In refutation theorem proving,[8] a prenex form without existential quantifiers is in Skolem form.[9] For example the formula

$$\forall x \forall y \left[\left(P\left(x, f\left(x,y\right)\right) \wedge Q\left(g\left(x,y\right)\right)\right) \wedge \left(P\left(a,y\right) \vee Q\left(y\right)\right)\right] \tag{3.1}$$

is a Skolem form. When dealing with closed Skolem forms, (universal) quantifiers may be omitted without ambiguity. Formula

$$\left(P\left(x, f\left(x,y\right)\right) \wedge Q\left(g\left(x,y\right)\right)\right) \wedge \left(P\left(a,y\right) \vee Q\left(y\right)\right)$$

is the preceding Skolem form where all variables (i.e., x and y) are implicitly universally quantified.

Skolemization is a satisfiability preserving process aimed at transforming a formula to Skolem form. The simplest Skolemization procedure first puts formulas in prenex form by logical equivalence preserving transformations. For example the formula

$$\forall x \left[\exists u\, P\left(x,u\right) \wedge \exists z\, Q\left(z\right)\right] \wedge \forall y \left[P\left(a,y\right) \vee Q\left(y\right)\right] \tag{3.2}$$

is logically equivalent to the prenex form

$$\forall x \forall y \exists u \exists z \left[\left(P\left(x,u\right) \wedge Q\left(z\right)\right) \wedge \left(P\left(a,y\right) \vee Q\left(y\right)\right)\right].$$

Then a satisfiability preserving process eliminates existential quantifiers: every existential quantifier is removed, and the corresponding variable is replaced by a Skolem term, i.e., a new (Skolem) function with arity equal to the number of preceding universal quantifiers, applied to all preceding universally quantified variables. A Skolem term for u is $f(x,y)$: f is a new function symbol, x and y are variables of universal quantifiers having $\exists u$ in their scope. Formula (3.1) is a Skolem form associated with formula (3.2). This last step does not preserves logical equivalence notably because it extends the formula signature. But it preserves satisfiability. When checking for formula satisfiability (i.e., in refutation context), a satisfiability preserving transformation may be safely applied.

The prenex Skolem form for a formula is not unique. First there is a choice for Skolem functions and constants. Second the prenex form is not unique. As an example, another Skolem form associated with formula (3.2) is

$$\forall x \forall y \left[\left(P(x, f(x)) \wedge Q(g(x))\right) \wedge \left(P(a,y) \vee Q(y)\right)\right]. \tag{3.3}$$

[8] That is when trying to prove unsatisfiability of formulas.

[9] Whereas in the context of validation theorem proving, a prenex form without universal quantifiers is in Skolem form.

The Skolem terms in Formula (3.3) do not depend on variable y, and this constitutes an improvement compared to Formula (3.1). By putting existential quantifiers first in the prenex form, it is possible to get Skolem functions of smaller arities, which also means smaller[10] Herbrand domain, and easier proofs. A good Skolemization process should produce the smaller number of Skolem functions (i.e., prefer Skolem constants, and reuse functions), with the smaller arities.

The way a formula is Skolemized can greatly modify the way the prover will react. As for simplification, criteria for perfect Skolemizing would impose that Skolemizing an unsatisfiable formula leads to the Boolean constant false. Perfect Skolemizing is thus undecidable. We will focus here on efficient, simple, syntactical methods. An improvement will only be used if it cannot be counterproductive. Methods exist which produce better Skolem forms [93, 140] but with higher cost,[11] and sometimes with negative results.

Comparing different Skolemization methods implies having a well defined set of proof rules and compare the length of the smallest proofs for the Skolemized formulas. Another way to compare Skolemization methods is by using Herbrand complexity [17]. This is not an easy task, and it is very dependent on the logic and the set of rules used. To estimate the relative power of Skolemization methods, we accept that Skolemization process P is better than P', if for every formula G, $P(G) = P'(G)[t'_1/t_1, \ldots t'_n/t_n]$ with Skolem terms t_i, t'_i such that arity of t_i is smaller than arity of t'_i, and such that variables in t_i are all variables used in t'_i. It is also required that if t'_i and t'_j are an application of the same function, then so it is for t_i and t_j. Intuitively better Skolemization should produce less functions (and constants), and of smaller arities. This will first advocate the use of structural Skolem form over prenex Skolem form, then motivate the use of Andrews' and antiprenex normal forms, presented in the next section.

3.3.1 Standard Skolemizations

Prenex Skolemization is an introductory naïve method. Standard Skolemization methods (that is structural and Andrews' Skolemization) do not require to first put formulas in prenex form. Instead, it relies on a way to identify quantifiers with existential semantic and those with universal semantic.

Definition 13 (Strong and weak quantifiers). *Let H be a formula.*

[10] Herbrand domains are either finite or countably infinite sets. In this last case subsets of given term depth are finite. "Smaller" here means that it contains less terms with large depth.

[11] And requiring the use of theorem proving techniques.

- *If* $\forall x\, G \in \Sigma^+(H)$ *or* $\exists x\, G \in \Sigma^-(H)$*, then the quantifier over* x *is a* weak quantifier*;*

- *If* $\exists x\, G \in \Sigma^+(H)$ *or* $\forall x\, G \in \Sigma^-(H)$*, then the quantifier over* x *is a* strong quantifier*.*[12]

In refutation context, strong quantifiers have an existential semantic. This basically means that strong quantifiers would become existential quantifiers when prenexing the formulas. Strong quantifiers are the quantifiers to be eliminated by Skolemization.

When a formula contains equivalence ($\equiv$) or if-then-else connectors, a subformula may have both negative and positive polarities, thus a quantifier may be strong and weak at the same time. Usually Skolemization is presented on formulas without those connectors; formulas should first be freed of those connectors. However, systematic elimination of those connectors may have negative effects on the size of produced Skolem forms. Replacement of those connectors should be used sparingly: there is no need to eliminate an equivalence if both subformulas are quantifier free. In the same way, it is not needed to transform if-then-else connectors if the condition is quantifier-free. If an if-then-else formula is used with a unique polarity in G, both then-subformula and else-subformula are used with a unique polarity in G. In conclusion a formula does not contain *problematic connectors* if both sides of equivalence subformulas are quantifier-free, and if conditions of all if-then-else subformulas are quantifier-free. Problematic connectors can always be eliminated using these rules:

$$\begin{array}{rcl} \text{if } C \text{ then } A \text{ else } B & \to & (C \Rightarrow A) \wedge (\neg C \Rightarrow B) \\ A \equiv B & \to & (A \wedge B) \vee (\neg A \wedge \neg B) \end{array}$$

Elimination of those connectors does not entirely solve the problem. In a representation with sharing of $(C \Rightarrow A) \wedge (\neg C \Rightarrow B)$, subformula C is used with both polarities. Skolemization process should thus consider subformulas with each polarity separately. If a formula does not contain problematic connectors, and if subformulas are considered with one polarity at a time, then every quantifier is weak or strong but not both.

It has been previously shown that the Skolem term for a quantified variable depends on the preceding quantifiers. When formulas are not necessary in prenex form, this order relationship between quantifiers is expressed using their scope:

[12] In validation context, definitions for strong and weak quantifiers are inverted. Checking validity of G is checking satisfiability of $\neg G$. As a result all polarities are inverted.

Definition 14 (Quantifier scope). *If H is a formula, if $Qx\,G$ is a subformula of H with $Q \in \{\forall, \exists\}$, and if G' is a subformula of G, then G' is in the scope of Qx.*

A subformula may be in the scope of two quantifiers on the same variable. This may be a source for ambiguities. This problem can be eliminated efficiently by renaming quantified variables without modifying the DAG structure. Another particularly delicate problem occurs when using maximal sharing DAG representation: a formula may have an instance used inside the scope of a quantifier, and another used outside the scope of the same quantifier. A common way to avoid these problems is working on rectified formulas:

Definition 15 (Rectified formula). *A formula is rectified if for every pair of subformulas $Q_1 v_1 G(v_1)$ and $Q_2 v_2 H(v_2)$ occurring at different positions in the syntactic tree for the formula, $v_1 \neq v_2$.*

Definition 15 essentially means that different quantifiers bind different variables, in a representation with no sharing. If problematic connectors are eliminated, this requirement completely solves the problem of having the same quantifier being both weak and strong. It also eliminates ambiguous cases for quantifier scopes. However, it may have dramatic effects on the size of the produced Skolem forms because it induces many subformula duplications. In the following, it is assumed that if one quantifier is in the scope of another they do not quantify the same variable. But formulas are not required to be rectified.

Definition 16 (Skolemization step). *Let H be a closed formula such that $Qx\,G(x) \in \Sigma(H)$ and $\{x, y_1 \dots y_n\} = \mathrm{Free}(G(x))$.*

H' is obtained from H by an inner Skolemization step *on $Qx\,G(x)$ if H' is obtained from H by replacing every occurrence of $Qx\,G(x)$ where Qx is a strong quantifier by $G(f(y_1, \dots y_n))$ where f is a new n-ary (Skolem) function symbol.*

H' is obtained from H by an outer Skolemization step *on $Qx\,G(x)$ if H' is obtained from H by replacing every occurrence of $Qx\,G(x)$ where Qx is a strong quantifier and where Qx is in the scope of the quantifiers $Qz_1, \dots Qz_m$ by $G(f(z_1, \dots z_m))$ where f is a new m-ary (Skolem) function symbol.*

The important variables for inner Skolemization are the one used inside the scope of the quantifier being Skolemized. In opposite, the variables important for outer Skolemization are the one used in quantifiers whose scope contains the quantifier being Skolemized.

In Definition 16, if a Skolemization step is applied on a quantifier which is not in the scope of any strong quantifier, then the set $\{y_1, \dots y_n\}$ in the preceding definition is a subset of $\{z_1, \dots z_m\}$. Inner Skolemization produces Skolem functions of smaller arity, in that case.

Lemma 1. *Let $G(x)$ be a formula such that* $\mathrm{Free}(G(x)) \subset \{x, y_1, \dots y_n\}$ *and let f be a new n-ary function symbol. Let $\mathcal{I}$ be any interpretation. There exist interpretations $\mathcal{I}_1$ and $\mathcal{I}_2$ differing only from $\mathcal{I}$ on the interpretation of symbol f such that*

- $\mathcal{I}_1 \Vdash \exists x\, G(x) \Rightarrow G(f(y_1, \dots y_n))$;
- $\mathcal{I}_2 \Vdash G(f(y_1, \dots y_n)) \Rightarrow \forall x\, G(x)$.

Proof. Let $\mathcal{I} = \langle D, I \rangle$. A suitable interpretation $\mathcal{I}_1 = \langle D, I_1 \rangle$ is such that I_1 and I differ only for function $I_1[f]$. This function can be defined as follow: for any $d_1 \in D_{\mathrm{Sort}(y_1)}, \dots d_n \in D_{\mathrm{Sort}(y_n)}$,

- if $\mathcal{I}_{y_1/d_1, \dots y_n/d_n} \Vdash \exists x\, G(x)$, then there exists an element $d \in D_{\mathrm{Sort}(x)}$ such that $\mathcal{I}_{x/d, y_1/d_1, \dots y_n/d_n} \Vdash G(x)$. In that case $I_1[f](d_1, \dots d_n) = d$;
- if $\mathcal{I}_{y_1/d_1, \dots y_n/d_n} \nVdash \exists x\, G(x)$, then $I_1[f](d_1, \dots d_n) = d$ where d is any element of $D_{\mathrm{Sort}(x)}$.

$\mathcal{I}_2$ can be constructed in a similar way, or preceding case can be reused with $G(x) = \neg G'(x)$. ☐

Theorem 3. *If H' is obtained by a Skolemization step from H, then H' is satisfiable if and only if H is satisfiable. For any model of H there exists a model of H' (and conversely) differing only on the interpretation of the Skolem symbol.*

Proof. Suppose $\exists x\, G(x)$ is the formula replaced by the Skolemization step, and $f(y_1, \dots y_n)$ is the Skolem term (the universal case is treated similarly).

Let $\mathcal{I}$ be a model of H. Using Lemma 1 there exists $\mathcal{I}'$ such that $\mathcal{I}$ and $\mathcal{I}'$ only differ on the interpretation of symbol f and such that

$$\mathcal{I}' \Vdash \exists x\, G(x) \Rightarrow G(f(y_1, \dots y_n)).$$

From Theorem 1, it follows that $\mathcal{I}' \Vdash H \Rightarrow H'$, and finally $\mathcal{I}' \Vdash H'$.

Now assume that $\mathcal{J}$ is a model of H'. Then $\mathcal{J}$ is also a model of H. Indeed $G(f(y_1, \dots y_n)) \Rightarrow \exists x\, G(x)$ is valid. Using Theorem 1 again $\mathcal{J} \Vdash H' \Rightarrow H$. Finally $\mathcal{J} \Vdash H$. ☐

Definition 17 (Skolemization). *Let H be a closed formula. If H does not contain strong quantifiers, then* $\mathrm{sk}(H) = H$*. If $Qx\, G(x) \in \Sigma(H)$ where Qx is a strong quantifier over x, then* $\mathrm{sk}(H) = \mathrm{sk}(H')$ *where H' is obtained from H by a Skolemization step.*

Skolemization is a well-defined transformation; each step removes one strong quantifier. *Structural* and *Andrews'* Skolem [7] forms of a formula are obtained by Skolemization using respectively only outer and inner Skolemization steps.

Theorem 4. *If H' is obtained by Skolemization from H, then H' is satisfiable if and only if H is satisfiable. For any model of H there exists a model of H' (and conversely) differing only on the interpretation of the Skolem symbols.*

Proof. It results from a finite number of applications of Theorem 3 □

If every Skolemization step is applied on quantifiers which are not in the scope of strong quantifiers, then every Skolem term is of depth one. Indeed, every introduced Skolem term $f(y_1, \dots y_n)$ is such that $y_1, \dots y_n$ are weakly quantified. Consider formula $\forall x \exists y \forall u \exists v\, p(x, y, u, v)$: Skolemizing first the existential quantifier over y leads to Skolem form $\forall x \forall u\, p(x, f(x), u, g(x, u))$, whereas choosing the other quantifier gives $\forall x \forall u\, p(x, f(x), u, g(x, f(x), u))$.

Applying Skolemization on outer quantifiers first is a sufficient condition to get Skolem terms with a maximal term depth of one. However, this condition is not necessary. Also notice that Skolemizing the outermost quantifiers first produces more Skolem functions. Assume that a formula contains subformulas $\exists x\, A(\exists y\, G(y))$ and $\exists x\, B(\exists y\, G(y))$, where quantifiers over x and y are not weak. First (inner) Skolemizing y would only introduce one Skolem function for this variable. Skolemizing both quantifiers over x at first will require two Skolem function for y if ever $G(y)$ contains x. Avoiding Skolem terms of depth greater than one will not always compensate the multiplication of Skolem functions. However Skolemizing the outermost quantifiers first generally produces good results.

Standard Skolemization is better than prenex Skolemization, according to the pragmatic criteria given in the beginning of this section (page 23). Indeed, prenexing requires rectified formulas. To every Skolemization process for the prenex formula corresponds a Skolemization process for the original formula, and vice versa. For every step, if the strong quantifier to be eliminated is in the scope of one quantifier in the process for the original formula, it also is for the prenex formula. Standard Skolemization thus produces Skolem terms of smaller arity.

Consider again Formula (3.2). A prenex Skolem form is (3.1), whereas formula (3.3) can be obtained by structural Skolemization. Formula (3.3)

$$
\begin{array}{rcll}
\exists x\,[\varphi \wedge \psi] & \to & \exists x\,\varphi \wedge \psi & \text{if } x \notin \mathrm{Free}\,(\psi) \\
\exists x\,[\varphi \vee \psi] & \to & \exists x\,\varphi \vee \psi & \text{if } x \notin \mathrm{Free}\,(\psi) \\
\forall x\,[\varphi \wedge \psi] & \to & \forall x\,\varphi \wedge \psi & \text{if } x \notin \mathrm{Free}\,(\psi) \\
\forall x\,[\varphi \vee \psi] & \to & \forall x\,\varphi \vee \psi & \text{if } x \notin \mathrm{Free}\,(\psi) \\
\exists x\,[\varphi \vee \psi] & \to & \exists x\,\varphi \vee \exists x\,\psi & \text{if } x \in \mathrm{Free}\,(\varphi) \text{ and } x \in \mathrm{Free}\,(\psi) \\
\forall x\,[\varphi \wedge \psi] & \to & \forall x\,\varphi \wedge \forall x\,\psi & \text{if } x \in \mathrm{Free}\,(\varphi) \text{ and } x \in \mathrm{Free}\,(\psi)
\end{array}
$$

Figure 3.2: Rules for antiprenexing

could also have been obtained by careful prenex Skolemization. A standard Skolem form is not always a prenex Skolem form. Prenex Skolemization of formula $\forall x \exists y A(x, y) \wedge \forall u \exists v B(u, v)$ generally introduces at least one Skolem term depending on x and u. A standard Skolem form does not.

Finally notice that Andrews' Skolemization produces better results than structural Skolemization, if outer quantifiers are Skolemized first. To every Andrews' Skolemization process of a formula corresponds a structural Skolemization process, and vice versa. For every step, inner Skolemization produces terms of smaller arity. An Andrews' Skolem form for formula (3.2) is

$$\forall x\,[P\,(x, f(x)) \wedge Q\,(b)] \wedge \forall y\,[P\,(a, y) \vee Q\,(y)] \tag{3.4}$$

It contains one more constant, but one unary function less than the structural Skolem form (formula (3.3)).

3.3.2 Prenexing, improving and being efficient

Antiprenexing is a kind of inverse transformation of prenexing. It has been previously shown that prenexing before Skolemizing was counterproductive because this produces Skolem functions of greater arity. In the same way antiprenexing [31] may have a positive influence on proof length [70].[13]

To produce better Skolem forms, the scope of (strong) quantifiers should be reduced as much as possible before applying Skolemization. This may reduce the number of free variables in their scope. Produced Skolem terms will also have smaller arities. This is achieved by systematically applying rules from Figure 3.2, and using commutation of quantifiers of the same kind. It is called *miniscoping*. As for prenexing, the miniscope form is not unique. It depends on the miniscoping order of quantifiers.

[13] Theorem 3.1 in [70]: applying antiprenexing may yield exponentially shorter resolution refutations regardless whether structural or Andrews' Skolemization is applied.

Miniscoping may also have a negative effect. Last two rules from Figure 3.2 may duplicate strong quantifiers. As a result there may also be duplication of Skolem functions which in turn may lead to an increase of proof length. Antiprenexing is miniscoping without duplication of strong quantifiers. Last two rules of Figure 3.2 should only be applied on quantifiers which are not strong.

Finally when no more rule can be applied, it may be useful to apply the following rules on strong quantifiers

$$\begin{array}{lcll} \exists x\, \varphi \vee \exists y\, \psi & \rightarrow & \exists x\, [\varphi \vee \psi[y/x]] & \text{if } \mathrm{Sort}(x) = \mathrm{Sort}(y) \\ \forall x\, \varphi \wedge \forall y\, \psi & \rightarrow & \forall x\, [\varphi \wedge \psi[y/x]] & \text{if } \mathrm{Sort}(x) = \mathrm{Sort}(y) \end{array}$$

as this further decreases the number of Skolem functions.

Among the syntactic[14] advanced Skolemization techniques, it is important to mention *Strong Skolemization* [141, 139]. It applies on formulas $\exists x\, (\varphi_1 \wedge \dots \varphi_n)$ where $\exists x$ is a strong quantifier.[15] It produces stronger[16] Skolem forms, but it may introduce new weak quantifiers. Also it is not clear whether a formula should be first put in antiprenex form or if strong Skolemization should be applied directly. It has been shown in [140] that Strong Skolemization sometimes makes refutation easier but its advantage in general is not clear.

To conclude this discussion about Skolemization, it should be noticed that it cannot be efficient. Consider closed formulas ψ_n, with

$$\begin{array}{rcl} \psi_n & = & \varphi_{n,n} \\ \varphi_{n,i} & = & \exists x_i\, (p_i(x_i) \wedge \varphi_{n,i-1}) \vee \forall x_i\, (q_i(x_i) \vee \varphi_{n,i-1}) \text{ if } i > 0 \\ \varphi_{n,0} & = & r_n\, (x_1, \dots x_n)\,. \end{array}$$

Their DAG representations are linear with respect to n. Their (prenex, structural, Andrews') Skolem form DAG representation are exponential. Indeed Skolem form for φ_n will contain 2^n Skolem terms $r_n\, (t_1, \dots t_n)$, where t_i will either be x_i or a Skolem term.

3.4 Conjunctive normal form

Putting a Boolean formula in conjunctive normal form may be seen as a trivial transformation step. But it may have an exponential cost in the worst

[14] As opposed to techniques requiring theorem proving.

[15] A similar rule exists for strong universal quantifier over a disjunction.

[16] The Andrews' Skolem form is a direct logical consequence of the Strong Skolem form. It is thus believed (and observed in some practical cases) that refutation of the Strong Skolem form is easier.

case. A standard naive transformation procedure would use the de Morgan's rules and the distributive property of $\vee$:

$$A \vee (B \wedge C) \longleftrightarrow (A \vee B) \wedge (A \vee C)\,.$$

This leads to possible formula size explosion. Consider as an example formula

$$(a_1 \wedge b_1) \vee \ldots \vee (a_n \wedge b_n)$$

where $a_1, b_1, \ldots a_n, b_n$ are distinct Boolean propositions. A logically equivalent conjunctive normal form obtained using the distributive property of disjunction over conjunctions would contain 2^n clauses, each clause containing n elements.

Notice that any method to build a logically equivalent conjunctive (disjunctive) normal form provides a direct decision procedure for tautology (resp. satisfiability) checking in propositional logic. In fact, a conjunctive normal form is valid if and only if each clause in the conjunction is itself valid. And a clause is valid (in propositional logic) if and only if it contains a complementary pair of literals. This last problem has a polynomial complexity. The global problem (tautology checking) is known to be co-NP-complete. Under those consideration, it would be surprising[17] to have a polynomial conjunctive normal form transformation *such that the normal form and the original formula are logically equivalent*.

In the following, *structural* or *structure-preserving* normal forms are introduced. The simple key idea is the use of renaming, i.e., introducing new propositional atoms to rename subformulas. This allows to achieve linear complexity[18] with respect to the size of input formulas. Of course, contrary to standard normal form transformations, the conjunctive normal form and the original formula are not logically equivalent anymore. But those transformations preserve satisfiability.[19]

Conjunctive structural normal form transformations do not only preserve satisfiability. As their name suggests they also preserve structure. Standard transformations are not reversible. Subformulas are spread among different clauses due to the distributivity law. Structural transformations make use of labels for (nearly) each subformula of the input formula. The set of clauses given to automated theorem provers somewhat still contains the structure of the original formula. If a subformula of the input formula has a special meaning, the structure of the normal form still highlights this meaning. A

[17] That is, it would prove P = NP.

[18] The transformation is linear in Boolean and quantified-free first-order logic, and quadratic on quantified first-order formulas.

[19] In opposite, disjunctive structural normal form transformations preserve validity.

comparative proof length study can be found[20] in [17]. Rather than doing a theoretical proof length analysis for structural and non-structural normal forms, a simple example will be given at the end of next section which will show that even in propositional logic, proof length can suffer badly from non-structural normal form transformations. Finally, as a last argument in favour of structural normal forms, a proof of the structural normal form can easily be translated into a proof for the original formula.

A chronological account about structural normal form transformations can be found in [17] and [32]. Two independent works [69, 148] are generally retained as first formalisation and complexity study for structural normal form transformations. Two extensive and thorough articles [17, 141] on the subject can be found in [161]. Structural normal form transformations are commonly used in automated theorem proving, when conjunctive normal forms are needed [33, 23, 21, 140].

Several kinds of structural normal forms exist. They differ in the number of introduced labels, as well as the number of clauses. We next present a formal presentation of two of those flavours. The first one — definitional normal form — is somewhat the more natural one, and the necessary way to the second one, i.e., p-definitional normal form. A p-definitional normal form is usually preferred to a definitional form because it produces less clauses. Indeed a p-definitional normal form is a subset of the set of clauses obtained for a definitional normal form.

Normal forms are commonly presented for full first-order logic [17]. In the present work, normal form transformation are used only on the level of ground formulas.[21] For this reason and in order to lighten notations, structural normal forms are presented for ground logic only. But it should be noticed that definitional and p-definitional normal forms exist for quantified formulas and that their complexity is quadratic (vs. linear for ground logic) due to quantifiers.

[20] This study holds for quantified first-order logic. The length of the shortest proof of the definitional normal form is polynomially related to the length of shortest proof of the original formula, in a given calculus (containing the usual rules). An upper bound of the proof length for structural normal form as well as a lower bound for standard normal form is given for several series of formulas of increasing length: the proof length for structural normal form is increasing (at most) exponentially with respect to the size of formulas, whereas a non-elementary increase is observed for standard normal form.

[21] Structural normal form transformations on the level quantified formula introduce new predicates. Those new predicates make the job of the instantiation procedure harder (see Chapter 9).

3.4.1 Definitional form

For this normal form transformation, as for the following one, a label L_G (i.e., a new Boolean proposition) is introduced for every subformula G. The definitional form [69] contains clauses such that any interpretation would give the same value to a label and the formula it refers to. It contains clauses corresponding to $L_G \equiv G_L$ for every G, where G_L is G where all direct subformulas are replaced by their label. The normal form contains clauses that perfectly *define* the labels according to the formula they refer to.

Definition 18. *If H is a ground formula, for any $G \in \Sigma(H)$, a label L_G (i.e., a new Boolean proposition) is introduced. The definitional form of H is the formula*

$$\delta(H) = \bigwedge_{G \in \Sigma(H)} \left(C_G^+ \wedge C_G^-\right)$$

where C_G^+ and C_G^- are given[22] *in Figure 3.3.*

Remark: if $H = A \circ B$ is a ground formula, where $\circ$ is any Boolean connector, then $\delta(H)$ and $\delta(A) \wedge \delta(B) \wedge C_H^+ \wedge C_H^-$ hold the same clauses. Also, if G is a subformula of H, then every clause in $\delta(G)$ is a clause of $\delta(H)$.

Theorem 5. *A ground formula H is satisfiable if and only if $L_H \wedge \delta(H)$ is satisfiable, where L_H is the label corresponding to H. For any model of H there exists a model of $L_H \wedge \delta(H)$ (and conversely) differing only on the interpretation of the introduced labels.*

Proof. By structural induction, any model of H can be extended in a model of $L_H \wedge \delta(H)$, whereas any model of $L_H \wedge \delta(H)$ is also a model of H. ☐

Consider again formula

$$H =_{\text{def}} (a_1 \wedge b_1) \vee \ldots \vee (a_n \wedge b_n)$$

as an example. The definitional normal form $\delta(H)$ is[23]

$$\begin{array}{ll} & (L_1 \vee \neg a_1 \vee \neg b_1) \wedge (\neg L_1 \vee a_1) \wedge (\neg L_1 \vee b_1) \\ & \vdots \\ \wedge & (L_n \vee \neg a_n \vee \neg b_n) \wedge (\neg L_n \vee a_n) \wedge (\neg L_n \vee b_n) \\ \wedge & (\neg L_H \vee L_1 \vee \ldots L_n) \\ \wedge & (L_H \vee \neg L_1) \wedge \ldots \ (L_H \vee \neg L_n) \end{array}$$

[22] Definitions of C_G^+ and C_G^- are inverted with regard to [17]: the authors consider validity, we rather consider satisfiability.

[23] We assumed a rule extension for the n-ary connector $\vee$. Redundant labels are also eliminated.

G	C_G^+ and C_G^-
atomic	$C_G^- : L_G \vee \neg G$ $C_G^+ : \neg L_G \vee G$
$\neg M$	$C_G^- : L_G \vee L_M$ $C_G^+ : \neg L_G \vee \neg L_M$
$A \wedge B$	$C_G^- : L_G \vee \neg L_A \vee \neg L_B$ $C_G^+ : (\neg L_G \vee L_A) \wedge (\neg L_G \vee L_B)$
$A \vee B$	$C_G^- : (L_G \vee \neg L_A) \wedge (L_G \vee \neg L_B)$ $C_G^+ : \neg L_G \vee L_A \vee L_B$
$A \Rightarrow B$	$C_G^- : (L_G \vee L_A) \wedge (L_G \vee \neg L_B)$ $C_G^+ : \neg L_G \vee \neg L_A \vee L_B$
$A \equiv B$	$C_G^- : (L_G \vee L_A \vee L_B) \wedge (L_G \vee \neg L_A \vee \neg L_B)$ $C_G^+ : (\neg L_G \vee \neg L_A \vee L_B) \wedge (\neg L_G \vee L_A \vee \neg L_B)$
if A then B else C	$C_G^- : (L_G \vee L_A \vee \neg L_C) \wedge (L_G \vee \neg L_A \vee \neg L_B)$ $C_G^+ : (\neg L_G \vee L_A \vee L_C) \wedge (\neg L_G \vee \neg L_A \vee L_B)$

Figure 3.3: Transformation rules for quantifier-free logic

where $L_1, \ldots L_n$ are labels associated respectively with $(a_1 \vee b_1), \ldots (a_1 \vee b_1)$, and L_H is the label associated with H. Any model of $\delta(H)$ makes L_H true if and only if L_i is true for at least one $i \in \{1, \ldots n\}$. Furthermore, if $\delta(H)$ is true, then L_i is true if and only if both a_i and b_i are true. So every model of $\delta(H) \wedge L_H$ corresponds to a model for H, and vice-versa. The normal form contains $4n + 1$ clauses, $10n + 1$ literals, and $n + 1$ new labels. The largest clause contains $n + 1$ literals.

Now consider the satisfiability of formula $H \wedge \varphi$. The definitional normal form for this last formula is $\delta(H \wedge \varphi) = \delta(H) \wedge \delta(\varphi) \wedge C_G^+(H \wedge \varphi) \wedge C_G^-(H \wedge \varphi)$. The polarity of H is positive in $H \wedge \varphi$: formula $H \wedge \varphi$ is satisfiable only when H is true, that is if L_H is true. Thus clauses $(L_H \vee \neg L_i)$ can safely be removed from the set of clauses. The formalisation of this simplification leads to p-definitional normal form.

3.4.2 p-definitional form

As for definitional normal form, a label L_G is introduced for every subformula G. But as opposed to definitional normal form, the label value is not perfectly defined with respect to the value given to the formula it represents. The p-definitional form [148] is a subset of the definitional form. The definitional form contains clauses which state that $L_G \equiv G_L$, where G_L is G where each direct subformula is replaced by its corresponding label. The p-definitional form contains only a subset of $L_G \equiv G_L$. In fact, it only contains clauses for $L_G \Rightarrow G_L$ if G only has positive polarity in the formula. If G only has negative polarity in the formula, only clauses for $G_L \Rightarrow L_G$ are in the p-definitional form. By taking subformulas polarities into account, it will be shown that the p-definitional form and the definitional form are equisatisfiable.

Definition 19. *If H is a ground formula, for any $G \in \Sigma(F)$, a label L_G (i.e., a new Boolean proposition) is introduced. The p-definitional form of H is the formula*

$$\delta_p(H) = \left(\bigwedge_{G \in \Sigma^+(H)} C_G^+ \right) \wedge \left(\bigwedge_{G \in \Sigma^-(H)} C_G^- \right)$$

where C_G^+ and C_G^- are given [22] *in Figure 3.3.*

Theorem 6. *A ground formula H is satisfiable if and only if $L_H \wedge \delta_p(H)$ is satisfiable, where L_H is the label corresponding to H. For any model of H there exists a model of $L_H \wedge \delta_p(H)$ (and conversely) differing only on the interpretation of the introduced labels.*

Proof. First, every clause in the conjunction $L_H \wedge \delta_p(H)$ belongs to the conjunction $L_H \wedge \delta(H)$. Using Theorem 6, every model of H can be extended in a model of $L_H \wedge \delta(H)$. This last model is trivially a model for $L_H \wedge \delta_p(H)$.

Notice that if $G \in \Sigma^+(H) \cap \Sigma^-(H)$, then every subformula of G also appears with both polarities in H. Assume $\mathcal{I} \Vdash L_H \wedge \delta_p(H)$. For every subformula G appearing with both polarities in H, $\mathcal{I} \Vdash L_G \equiv G$.

Using structural induction, for any subformula $G \in \Sigma^+(H)$ ($G \in \Sigma^-(H)$), one has $\mathcal{I} \Vdash L_G \Rightarrow G$ (respectively $\mathcal{I} \Vdash G \Rightarrow L_G$). For instance, in the case of an implication $A \Rightarrow B$ appearing only positively in H, A appears at least negatively in H, and B appears at least positively in H. The induction hypothesis states that $\mathcal{I} \Vdash A \Rightarrow L_A$ and $\mathcal{I} \Vdash L_B \Rightarrow B$. Also $\mathcal{I} \Vdash L_H \wedge \delta_p(H)$, and thus $\mathcal{I} \Vdash \neg L_{A \Rightarrow B} \vee \neg L_A \vee L_B$. Finally $\mathcal{I} \Vdash L_{A \Rightarrow B} \Rightarrow (A \Rightarrow B)$.

This proves the theorem as $\mathcal{I} \Vdash L_H \Rightarrow H$, and $\mathcal{I} \Vdash L_H$. □

Notice that Theorem 6 does not forbid the use of interpreted predicates: indeed, equisatisfiability of the formula and its p-definitional form is obtained by building an interpretation for the normal form from the interpretation for the original formula, and vice versa. Both interpretations only differ on the interpretation for the labels. Despite of this, p-definitional normal form is often avoided when dealing with formulas containing interpreted predicates. We would like to insist that there is no reason for this.

3.4.3 Improvements and advanced techniques

Both normal form transformations introduced here have linear time (and space) complexity with respect to the size of the input formula. It is generally understood that formulas are represented as syntactic trees. But the complexity remains linear if formulas are represented as directed acyclic graphs. In that case, several instances of the same subformula are represented by the same subgraph. For such subformulas, only one label is introduced for all instances. This avoids introducing too many labels having the same meaning.

Another straightforward small improvement is avoiding to introduce redundant labels. First, labels for atoms can be eliminated. Indeed any model of the normal form can be translated in a model giving the same truth value to the atom and to the label. Second, and for the same reason, labels for negations of formulas are not mandatory, as it is sufficient to replace them everywhere in the normal form by the negation of the label of the negated formula.

Last, a rather aesthetic modification consists in letting L_H be true in $L_H \wedge \delta(H)$, because no model makes L_H false. Generally this will not lighten the job for the automated prover too much, because provers do usually perform this step themselves.[24] In general, these small improvements do not improve the theoretical complexity of solving formulas, but they somewhat act as a preprocessing step to give all chances for the solver to succeed.

An ultimate enhancement of normal form transformations has been presented in [56]. This work presents a transformation algorithm which produces optimal normal forms. More precisely, those normal forms are optimal with respect to the number of clauses. It is an hybrid transformation between structural and standard transformation. However if it is optimal with respect to the number of clauses, it is not with respect to the total size. That is, clauses may get larger than in structural normal forms, so that the overall result is quadratic.

Finally, it should be noticed that there is *no such criterion* like "the smaller the normal form[25] the better it is". There is no decisive theoretical

[24] Indeed L_H is a unitary clause. This simplification corresponds to one Boolean constraint propagation step (see Chapter 6).

[25] Or "the smaller the number of clauses...", or "the smaller the number of labels...".

evidence or practical observation which has rewarded a transformation as the best in every domain.

3.5 If-then-else elimination

It is often convenient to use if-then-else inside terms and not only as a Boolean connective [21, 66, 147]. Indeed, formula

$$g\,(\text{if } c \text{ then } t_1 \text{ else } t_2) = t_3 \tag{3.5}$$

where t_1, t_2 and t_3 are terms and c is a Boolean formula, is compact and has an intuitive meaning. In this section, it is shown that these if-then-else terms can easily be eliminated.

A naive if-then-else elimination would result in formula size explosion. Indeed, formula (3.5) is logically equivalent to formula

$$\text{if } c \text{ then } g(t_1) = t_3 \text{ else } g(t_2) = t_3.$$

Now g appears twice. After complete expansion, part of the formula may be copied 2^n times, where n is the maximum number of nested if-then-else .

Just as for conjunctive normal forms, introducing new labels for sub-expressions can reduce complexity. Consider formula

$$(\text{if } c \text{ then } a = t_1 \text{ else } a = t_2) \wedge g(a) = t_3,$$

where a now refers to term (if c then t_1 else t_2). It is not logically equivalent to (3.5), but it is satisfiable if and only if (3.5) is. And its size is linear with respect to the size of (3.5).

Theorem 7. *Every quantifier-free formula*

$$H\,(\text{if } c \text{ then } t_1 \text{ else } t_2)$$

where t_1 and t_2 are terms is satisfiable if and only if

$$(\text{if } c \text{ then } a = t_1 \text{ else } a = t_2) \wedge H\,(a)$$

is, where symbol a does not appear in H (if c then t_1 else t_2). For any model of the first formula there exists a model of the second (and conversely) differing only on the interpretation of the new symbol a.

Proof. The condition is trivially sufficient. Indeed, let $\mathcal{I}$ be an interpretation for the second formula. If $\mathcal{I} \Vdash c$, as $\mathcal{I} \Vdash (\text{if } c \text{ then } a = t_1 \text{ else } a = t_2)$, we also have $\mathcal{I} \Vdash a = t_1$. Similarly, if $\mathcal{I} \Vdash \neg c$, then $\mathcal{I} \Vdash a = t_2$. In brief $\mathcal{I} \Vdash a = (\text{if } c \text{ then } t_1 \text{ else } t_2)$. From this last fact, as $\mathcal{I} \Vdash H(a)$ it can be deduced that $\mathcal{I} \Vdash H(\text{if } c \text{ then } t_1 \text{ else } t_2)$.

The condition is necessary. If $\mathcal{I}$ is a model of $H(\text{if } c \text{ then } t_1 \text{ else } t_2)$, and if $H(\text{if } c \text{ then } t_1 \text{ else } t_2)$ does not use the symbol a, a new interpretation $\mathcal{I}'$ can be built. $\mathcal{I}'$ is in all points similar to $\mathcal{I}$ except that $\mathcal{I}'[a] = t_1$ if $\mathcal{I} \Vdash c$ and $\mathcal{I}'[a] = t_2$ if $\mathcal{I} \Vdash \neg c$. It is thus easy to prove that $\mathcal{I}'$ is a model of the second formula. $\square$

A careful implementation of an algorithm to remove if-then-else terms systematically is linear with respect to the size of the directed acyclic graph representing formulas. Just as for conjunctive normal forms, this can involve the introduction of many new constants. In fact a new constant is introduced for every if-then-else node in the directed acyclic graph representing the formula.

Chapter 4

Formal systems

The initial steps in formal verification of systems are first to specify systems and their desired properties, and next to generate *verification conditions* (a.k.a. proof obligations) suitable for validity checking. We describe here a simple specification language for transition systems [98]. This simple language allows various levels of abstraction. In particular it contains a very expressive quantified assignment.[1] Examples of other modelling languages are UNITY [50] and SPL [124]. The advantage of transition systems is that they clearly state which actions are atomic, and how they are allowed to interleave.

The chosen language for describing properties, states, set of states and verification conditions is many-sorted first-order logic. Its expressive power is reasonably large and it permits some automated treatment. In the next chapter we will also show that an interesting fragment is decidable, which makes invariant validation decidable for many algorithms.

The generation of verification conditions from assertions and transitions is based on Hoare logic [107, 10, 11] and Dijkstra weakest precondition calculus [62]. The size of the generated formulas is an important issue: the weakest precondition may have a size exponential with respect to statement and postcondition sizes. The solution used in ESC/java [61] is to "passify" the statements [78]. The following verification condition generation method reaches the same good complexity results with respect to the size of generated formulas.

Generation of verification conditions is usually presented either using weakest precondition calculus (for example in the verification tool Why [72]) or using transition relations (for example in [3, 4]). Our presentation unifies both concepts, by defining the weakest precondition computation using

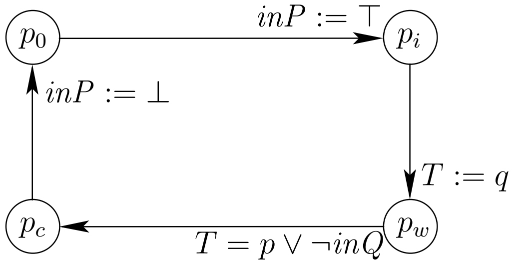
[1]A kind of quantified assignment is studied in [41], but it is less expressive than the general assignment introduced in Section 4.4.

transition relations.

4.1 Introducing formal systems: an example

Before studying the language and verification condition generation in depth, a little example is here discussed to get an intuitive idea about formal systems. The Peterson distributed algorithm [146] solves the mutual exclusion problem for two processes, p and q. Each process has a Boolean variable (respectively inP and inQ). A variable T is shared and takes its value in the set $\{p, q\}$. Each process has to be in one and only one of four locations (0, i, w, c). One of four Boolean variables (p_0, p_i, p_w, p_c for process p) is true, to reflect this location.[2] At a given moment, the values of the eight location variables constitute the *control state* of the distributed algorithm, whereas the *memory state* corresponds to the values of variables T, *inP* and *inQ*. The *state* of the system is the pair containing the control state and the memory state.

The set of transitions defines the way states change. Process p (and symmetrically for process q) changes according to the following graph:



The *transition* from p_0 to p_i is only active for states such that p is in location 0. It changes the location from 0 to i, that is p_0 becomes false whereas p_i becomes true. Another effect of this transition is to make variable *inP* true. The transition from p_w to p_c does not change the memory state, but it is active only if the state is such that T is p, or *inQ* is false. The condition on the memory state for the transition to be active is the *guard* of the transition.

Rather than the previous graphical representation of the transition set, the following compact notation (see Section 4.3) is preferred:

$$\begin{array}{l}
(p_0,\ inP := \top,\ p_i) \\
(p_i,\ T := q,\ p_w) \\
(p_w,\ T = p \vee \neg inQ \longrightarrow \textbf{skip},\ p_c) \\
(p_c,\ inP := \bot,\ p_0)
\end{array}$$

[2]Another modelling choice would bet to use a variable taking its value in the finite set of all possible locations. This is discussed in Section 4.5.

and symmetrically for process q. Actually, the system contains eight transitions: four for each process.

Initially, both processes p and q are in location 0, with variables inP and inQ false. An assertion describing the set of acceptable initial states is

$$\neg inP \wedge p_0 \wedge \neg p_i \wedge \neg p_w \wedge \neg p_c \wedge \neg inQ \wedge q_0 \wedge \neg q_i \wedge \neg q_w \wedge \neg q_c$$

It has as a consequence the following first-order assertion:

$$\begin{array}{rcl} I & = & \quad [p_w \Rightarrow (T = q \vee q_w)] \\ & & \wedge\ [q_w \Rightarrow (T = p \vee p_w)] \\ & & \wedge\ [p_c \Rightarrow (T = p \vee \neg inQ \vee q_i)] \\ & & \wedge\ [q_c \Rightarrow (T = q \vee \neg inP \vee p_i)] \\ & & \wedge\ [p_0 \equiv \neg inP] \\ & & \wedge\ [q_0 \equiv \neg inQ] \end{array}$$

Assertion I is furthermore preserved by every transition.[3] That is, without more knowledge of the transition system and its history, if the state makes I true before any system transition, it can be guaranteed that the state after the transition also makes I true. Assertion I is an *invariant* of the system, and it is true during any execution of the system. Verification generation (Sections 4.7 and 4.8) is used to build logical assertions to check for validity in order to guarantee I is an invariant of the system. As a corollary,[3] the mutual exclusion property is verified: assertion

$$\neg (p_c \wedge q_c)$$

is a logical consequence of the invariant.

4.2 Memory and states

In this section, we discuss data structures and their representation. This is necessary before any formal description of systems, as system semantics depends on state representation.

4.2.1 Memory

The *memory* of a system is a set of program variables. Variables (see for instance [183]) have a type defining the set of their possible values. Usual

[3] The location rules are implicitly used: one and only one of propositions p_0, p_i, p_w, p_c is true. Also constants p and q are assumed to be different.

simple types are Booleans, integers, strings,... whereas common *structured types* are structures (or records), arrays, sets, lists... A sequential program running on a given sequential machine can be modelled by its actual implementation, that is using a set of operations on a large one-dimensional array of Booleans (bits). But to capture only the very ideas of the algorithm, and push into the background all details like for example representations of integers,[4] systems and their properties would naturally be formulated in a language using abstract types (like integers, strings, arrays, lists, ...) without care of the way they are represented in memory. Verifying the model does not in that case mean the implementation is correct. It means the very idea of the algorithm is correct.[5]

Abstracting the data representation is one reason for using abstract and infinite types. Another reason is to model algorithms with a priori unknown parameters: for instance, parallel algorithms with a priori unknown number of symmetric processes, or a sliding window protocol with parameterized window size. In such a context, the necessity for arrays of parameterized size arises. Consider the simple parameterized Ticket mutual exclusion algorithm [42]: every process has a control variable ℓ which can takes a value in $\{\text{think}, \text{wait}, \text{critic}\}$, and a variable a to record the ticket number. To model this algorithm it is convenient to use one-dimensional arrays ℓ and a with an argument in the finite but parameterized set of processes.

In parameterized systems, the number of variables, i.e., the array sizes, are bounded by a parameter. It may also be worthwhile to consider unbounded arrays: putting a limit on a queue size may be unwanted in a prototype algorithm. Implementations will impose a limit, but it may be interesting to avoid this difficulty at least in a first study, just as avoiding integer overflow problems will make a first study of an algorithm easier.

A *memory state* is a function from memory variables to values in the set of their type. Actually, a *state* is a pair (memory state, control state[6]) where the control state describes for every process which transitions are possible and which are not. In other words it contains all information about program counters. This is not a fundamental concept as it can be modelled entirely into the memory state, by adding variables of appropriate types.[7] From now on, no distinction is made between state and memory state. Control will be studied in more details in Section 4.5.

[4]For a same system, these details may vary from one implementation to another.

[5]It still remains to prove that the concrete program is a correct implementation of the algorithm, according to the working conditions. For example, integer overflow should not happen.

[6]Also called *location state*.

[7]In sequential machines, the program counter is a CPU register.

4.2.2 Representing (sets of) states

Depending on the variable set, there may be an infinite number of different states. In that case, even if the number of states is countable, the sets of states are not. A formula language can only represent a countable number of state sets. The choice for such a language is thus crucial. Common languages are Higher-Order Logic (HOL), First-Order Logic (FOL), Quantifier-Free First-Order Logic (QF-FOL),... In that list, every language can express more than the following. But if the language is more expressive, it is also more difficult to handle. In fact there is no complete decision procedure to deal with whole HOL, there is a semi-complete decision procedure for FOL, and QF-FOL is decidable. The choice is thus a tradeoff between the expressive power, and the possibilities for automated treatment. State sets may be represented by other means, like regular languages (see for example [40]), automata (see for example [106, 185, 186]), or more classically BDDs [43]... which is one key to the successful treatment of the model checking problem for some systems.

In this work, assertions representing sets of states are first-order logic formulas. This allows to use quantifiers, and still permits some automated treatments of formulas. We will also show in Chapter 5 that an interesting fragment of quantified first-order language is decidable. Opting for a many-sorted language is a natural choice as many-sorted languages allow to deal with variables having different types (or sorts), as usually found in programs. The state of an algorithm having for sole variables a Boolean variable p, an integer variable i, and a string s is perfectly defined by an assertion like $\neg p \wedge i = 0 \wedge s =$ "". A set of states can be expressed as an (quantifier-free) assertion using proposition p and logical variables of different sort i and s.

The counterpart for program variables of simple type in logical assertions are naturally logical propositions and variables, depending on whether the program variables are Boolean or not. Record variables do not differ greatly: it suffices to consider every field as a simple variable. But translation of program variables in logic is not so clear for all structured typed variables. Consider the array type. A possible translation would assume arrays are considered as variables. In this context two functions are supplied to read and write an element at a given position in an array [172]. For example the term $\mathrm{write}(A, \mathrm{read}(A, j), i)$ is the array identical to A, but with the element on position i set equal to the one in position j. Assertion $\forall i, j\, (i < j \Rightarrow \mathrm{read}(A, i) < \mathrm{read}(A, j))$ expresses the fact that array A is sorted. The advantage of this way of modelling arrays is the ability to write equalities between arrays, and more generally to define predicates on arrays; for example, it is possible to define the predicate "sorted" such that

$$\forall A[\,\mathrm{sorted}(A) \equiv \forall i, j\, (i < j \Rightarrow \mathrm{read}(A, i) < \mathrm{read}(A, j))\,].$$

But the price to pay is the presence everywhere of those two (interpreted) functions "read" and "write". Handling many-dimensional arrays involve more complicated expressions to access the elements, either by introducing more general "read" and "write" functions, or by requiring several successive read applications on arrays.

A natural way to avoid "read" and "write" statements is by considering arrays as functions or predicates.[8] Now, assertion $\forall i, j\,(i < j \Rightarrow A[i] < A[j])$ is true if and only if function A represents a sorted array. The direct advantages are first the absence of interpreted handling functions and second a simplified access to elements in an array. As a counterpart, expressing the equality of two arrays A and B now requires quantifiers: $\forall i\,(A[i] = B[i])$. More generally there is no facility in first-order languages to easily define and handle predicates and functions on other predicates and functions.

Important data structures also include lists, queues, and sets. There is a similar choice as for arrays: they can be modelled using logic variables with accessor and constructor functions [143], or directly using functions and predicates. A common instance is a set of processes, which may be modelled as a Boolean one-dimensional array with a process type argument would be. In Chapter 5, the second option is chosen. Rather than introducing interpreted constructor and accessor functions for arrays, lists, or sets, we will prefer to model everything using uninterpreted predicates and functions. This will greatly simplify the discussion. However, in Chapter 8, some interesting results in the quantifier-free case for list and array theories will be given.

4.3 Transitions, guards

Basically a transition transforms a state to another in one atomic operation. The effect of a transition can be encapsulated in a binary relation between states, and can be be described by a formula just as states can be. Such formulas are usually referred to as *before-after predicates* [49] or *transition relations* [124]. But this way of describing transitions is obscure for the human mind. One prefers a representation in a form which has a direct meaning to the reader, like

$$(C \longrightarrow S)$$

where C is the *guard* (or condition), and S a *statement*. The guard C is an assertion representing the set of states the transition is allowed to change. A transition is *active* for the states which make true the condition. A transition

[8] Predicates stand for Boolean arrays, functions for other arrays.

can be active for all states. In that case the condition is the Boolean constant $\top$, and it may omitted; transitions (S) and $(\top \longrightarrow S)$ represent the same relation between states. The statement S describes the way those states may change. A transition active for a state s, may change this *starting state* s to an *ending state* s' according to the statement S. For example transition $(x > 0 \longrightarrow x := x - 1)$ is active for x strictly positive, and decreases the value of x by one. That is the transition only modifies states such that $x > 0$; if x is equal to value e in the starting state, the ending state will be such that $x = e - 1$.

Actually the difference between statement and transition is only superficial. Indeed both statement and transition represent relations between states. Transitions $(C \longrightarrow S)$ and (**if** C **then** S **else skip**) both involve a change in state according to the statement S, only if C is true in the starting state (**skip** is the do-nothing statement). The transition relation is the relation for the statement **if** C **then** S **else skip**.

Using intuitive common knowledge of statements one can observe two things. First the relation represented by a transition is not necessarily functional: consider the non deterministic transition $(x > 0 \longrightarrow x := ?)$ where ? represents a random generator of integer values. This transition may transform a starting state s such that $x > 0$ into any state s' such that x is any integer value. The relation may not even be an application: the ending state obtained by using transition $(x > 0 \longrightarrow x := x/y)$ from a starting state such that $y = 0$ is undefined.

4.4 Semantics of statement

Statements found in imperative programming languages are assignments, compound statements like sequence, conditional and loop statements… In this section, the semantic of some of those statements is specified. Next, it will be shown how verification conditions — ensuring a given assertion is preserved by a transition — can be generated automatically from the transitions, using the semantic specified in this section.

A statement changes the value of the program variables. It defines a binary relation between the values of variables before and after execution of the statement. For example, if x' refers to the value of variable x after execution of the statement $x := x + 1$, the effect of the execution of the statement is described by the relation $x' = x + 1$. In general if variables in the sets $\mathcal{M}$ and $\mathcal{M}'$ respectively refer to the values of before and after the execution of a statement S, the effect of S can be encapsulated in a relation $\delta[S](\mathcal{M}, \mathcal{M}')$. For instance, if the sets $\mathcal{M} = \{x, y\}$ and $\mathcal{M}' = \{x', y'\}$

respectively describe the memory states before and after execution of $S = (x := x * y; y := y - 1)$, the relation $\delta[S](\mathcal{M}, \mathcal{M}') = (x' = x * y \wedge y' = y - 1)$ specifies the semantic of statement S.

For the automatic generation of verification conditions, those relations δ have to be computed. This computation involves some syntactical treatments, and one of them is a generalized substitution. Assume $\mathcal{M}$ and $\mathcal{M}'$ are two bijective sets of program variables, for instance describing the memory state before and after execution of a statement. The bijection between $\mathcal{M}$ and $\mathcal{M}'$ defines two operators on variables, terms, and assertions: $\varphi[\mathcal{M} \to \mathcal{M}']$ ($\varphi[\mathcal{M}' \to \mathcal{M}]$) is variable, term, or assertion φ in which every variable in $\mathcal{M}$ (resp. $\mathcal{M}'$) has been replaced by its counterpart in $\mathcal{M}'$ (resp. $\mathcal{M}$). The operator $[\mathcal{M} \to \mathcal{M}']$ used in postfix notation is a substitution ranging not only on logical variables, but also on functions and predicates, as, for instance, functions and predicates can be used to model arrays. By extension, given a subset V of $\mathcal{M}$, $[V \to \mathcal{M}']$ substitutes every element in V by the corresponding element in $\mathcal{M}'$. Implementations of these extended substitutions can be linear in time with respect to the size of DAGs or trees representing formulas. Also, the number of DAG nodes is preserved.

Statements usually modify only a few program variables. The statement relation δ thus often states that values of the program variables are unmodified by the execution of the statement. For this role, it is convenient to define the assertion

$$\mathrm{pres}(V, \mathcal{M}, \mathcal{M}') = \bigwedge_{v \in V} \forall \vec{u}_v \; [\, (v[\mathcal{M} \to \mathcal{M}'])\, [\vec{u}_v] = v[\vec{u}_v]]$$

such that the length of list of variables $\vec{u}_v$, and the sort of its elements correspond to the arity and sorts of logical symbol v. Expression $(v[\mathcal{M} \to \mathcal{M}'])$ denotes the variable corresponding to $v \in \mathcal{M}$ in $\mathcal{M}'$. Following the common usage, the delimiters for the argument of a function or predicate representing an array program variable are square brackets rather than parentheses. Also, to agree with the syntax of many-sorted first-order language, logical equivalence ($\equiv$) is used rather than equality for comparing two elements of Boolean array. As an example if $\mathcal{M}$ is a set with an integer variable i, and a one-dimensional array of Boolean X, and if $\mathcal{M}' = \{i', X'\}$,

$$\mathrm{pres}(\mathcal{M}, \mathcal{M}, \mathcal{M}') = i' = i \wedge \forall u \, (X'[u] \equiv X[u])\,.$$

The simplest of all statements is the do-nothing or empty statement **skip**. Every variable is preserved; the transition relation associated with this statement is

$$\delta\,[\mathbf{skip}]\,(\mathcal{M}, \mathcal{M}') = \mathrm{pres}\,(\mathcal{M}, \mathcal{M}, \mathcal{M}')$$

Assertion $\delta\,[\mathbf{skip}]\,(\mathcal{M}, \mathcal{M}')$ is true if and only if variables in $\mathcal{M}$ and $\mathcal{M}'$ respectively define a starting and ending state of a **skip** statement. That is if every variable in $\mathcal{M}'$ is equal to its counterpart in $\mathcal{M}$. Such a relation is defined for every starting state. Assertion $\Omega\,[\mathbf{skip}]\,(\mathcal{M})$ is true if the **skip** statement is defined for state represented by variables in $\mathcal{M}$, that is:

$$\Omega\,[\mathbf{skip}]\,(\mathcal{M}) = \top$$

The do-nothing statement might not seem to be of great use, but it is a way to write a transition which only updates the control state (see Section 4.5), or to use the conditional statement in a restricted form.

The generalized assignment statement is written as

$$\forall \vec{u}_1\ v_1[\vec{u}_1] := E_1(\vec{u}_1) \parallel\ \ldots\ \forall \vec{u}_n\ v_n[\vec{u}_n] := E_n(\vec{u}_n) \tag{4.1}$$

where $v_1, \ldots v_n$ are different variables, and $\{v_1 \ldots v_n\}$ does not contain any quantified variable. The lists of quantified logic variables $\vec{u}_i$ have a length equal to the arity of program variables v_i. Every element in these lists has the sort of the corresponding argument (or index) of the array variables v_i. In particular, an assignment of a simple typed program variable is not quantified. For every i, $E_i(\vec{u}_i)$ is an expression of the same sort than $v_i[\vec{u}_i]$, or is a Boolean formula if $v_i[\vec{u}_i]$ is Boolean. Those expressions E_i might use if-then-else terms. Only functional expressions are allowed. For instance, random generators should not be used.

The generalized assignment is very expressive. For example, if X is a program variable of type array$[0..n]$ of integer, statement shift(X, a) shifts values in table X of a elements to greater indices. This can be modelled using the generalized assignment by

$$\forall u[X[u] := \text{if } -a \leq u \leq n-a \text{ then } X[u+a] \text{ else } 0]$$

where u is of type $\{0, 1, \ldots n\}$. The more common statement $X[e] := e'$ is simply expressed by

$$\forall u[X[u] := \text{ if } u = e \text{ then } e' \text{ else } X[u]]$$

if u does not appear in e and e'. The swapping statements, like $(a, b) := (b, a)$ or rather $(X[i], X[j]) := (X[j], X[i])$ are also easily written as a generalized assignment:

$$\forall u[X[u] := \text{ if } u = i \text{ then } X[j] \text{ else if } u = j \text{ then } X[i] \text{ else } X[u]]$$

Finally notice that statement $(X[i], X[j]) := (e_1, e_2)$ is ambiguous when $i = j$ and $e_1 \neq e_2$. It cannot be transcribed directly into a generalized assignment. Expression

$$\forall u[X[u] := \text{ if } u = i \text{ then } e_1 \text{ else if } u = j \text{ then } e_2 \text{ else } X[u]]$$

rather corresponds to the statement sequence $X[j] := e_2; X[i] := e_1$.

The transition relation of the generalized assignment (4.1) is

$$\delta\,[(4.1)]\,(\mathcal{M},\mathcal{M}') = \\ \operatorname{pres}(\mathcal{M}\setminus\{v_1,\dots v_n\},\mathcal{M},\mathcal{M}') \wedge \bigwedge_{1\le i\le n} \forall\vec{u}_i\ [\,(v_i[\mathcal{M}\to\mathcal{M}'])\,[\vec{u}_i] = E_i(\vec{u}_i)]$$

where quantified variables are not in $\mathcal{M}$ or $\mathcal{M}'$. Of course, if v_i represents a Boolean array, equivalence ($\equiv$) is used instead of equality in the corresponding expression. The statement is only defined if

$$\Omega\,[(4.1)]\,(\mathcal{M}) = \bigwedge_{1\le i\le n} \forall\vec{u}_i\ \Omega\,[E_i(\vec{u}_i)]\,(\mathcal{M})$$

where $\Omega\,[E_i(\vec{u}_i)]$ is true if and only if expression $E_i(\vec{u}_i)$ is computable, and has a value of type of $X[\vec{u}_i]$. For instance if x and y are both rational, then expression $\Omega[x := x/y]\,(\{x,y\})$ is true if and only if y is non null.

The remaining two statements studied in this section do not extend the expressiveness of transition systems, but they are convenient to use. They allow to express in one transition what would require several transitions without them. Furthermore, doing without the sequence statement may involve to complicate the system in order to group transitions in one atomic sequence of transitions.

The ending state s' obtained by applying the sequence statement $S_1; S_2$ to the starting state s is the state obtained by transforming state s (described by $\mathcal{M}$) to s'' (descr. by $\mathcal{M}''$) with S_1, then s'' to s' (descr. by $\mathcal{M}'$) with S_2:

$$\delta\,[S_1; S_2]\,(\mathcal{M},\mathcal{M}') = \exists\mathcal{M}''[\delta\,[S_1]\,(\mathcal{M},\mathcal{M}'') \wedge \delta\,[S_2]\,(\mathcal{M}'',\mathcal{M}')\,].$$

The present statement relation is prenexed by one existential quantifier for every program variable in $\mathcal{M}''$ ($\exists\mathcal{M}''$). This means there might be some quantifiers on predicates or functions. This assertion is not a first-order formula, because those quantifiers might be second order quantifiers. This problem will be addressed in Section 4.8.

The sequence statement $S_1; S_2$ is only defined if

$$\Omega\,[S_1; S_2]\,(\mathcal{M}) = \Omega\,[S_1]\,(\mathcal{M}) \wedge \forall\mathcal{M}''[\delta\,[S_1]\,(\mathcal{M},\mathcal{M}'') \Rightarrow \Omega\,[S_2]\,(\mathcal{M}'')\,].$$

This assertion also potentially contains second-order quantifiers.

Notice that the sequence operator is associative: statement $(S_1; S_2)\,; S_3$ has the same semantic that statement $S_1; (S_2; S_3)$. Statements like $S_1; S_2; S_3$ or more generally $S_1; \dots S_n$ have a well defined semantic:

$$\delta\,[S_1; \dots S_n]\,(\mathcal{M},\mathcal{M}') = \\ \exists\mathcal{M}_1 \dots \exists\mathcal{M}_{n-1}[\delta\,[S_1]\,(\mathcal{M},\mathcal{M}_1) \wedge \dots \\ \wedge\,\delta\,[S_i]\,(\mathcal{M}_{i-1},\mathcal{M}_i) \wedge \dots \wedge \delta\,[S_n]\,(\mathcal{M}_{n-1},\mathcal{M}')\,]$$

and

$$
\begin{array}{l}
\Omega\,[S_1; \ldots S_n]\,(\mathcal{M}) = \\
\quad \Omega\,[S_1]\,(\mathcal{M}) \wedge \\
\quad\; \forall \mathcal{M}_1 [\delta\,[S_1]\,(\mathcal{M}, \mathcal{M}_1) \Rightarrow \\
\quad\quad\quad [\Omega\,[S_2]\,(\mathcal{M}_1) \wedge \\
\quad\quad\quad\; \forall \mathcal{M}_2 [\delta\,[S_2]\,(\mathcal{M}_1, \mathcal{M}_2) \Rightarrow \\
\quad\quad\quad\quad\quad [\Omega\,[S_3]\,(\mathcal{M}_2) \wedge \ldots \\
\quad\quad\quad\quad\quad\; \forall \mathcal{M}_{n-1} [\delta\,[S_{n-1}]\,(\mathcal{M}_{n-2}, \mathcal{M}_{n-1}) \Rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \Omega\,[S_n]\,(\mathcal{M}_{n-1})\,] \ldots]]]]
\end{array}
$$

where $\mathcal{M}_1, \ldots \mathcal{M}_{n-1}$ are new disjoint sets of program variables.

Finally, an assertion expressing the relation for the conditional statement is

$$
\begin{array}{l}
\delta\,[\textbf{if } C \textbf{ then } S_1 \textbf{ else } S_2]\,(\mathcal{M}, \mathcal{M}') \\
\quad = \;\; [C\,(\mathcal{M}) \Rightarrow \delta\,[S_1]\,(\mathcal{M}, \mathcal{M}')\,] \wedge [\neg C\,(\mathcal{M}) \Rightarrow \delta\,[S_2]\,(\mathcal{M}, \mathcal{M}')\,] \\
\quad = \;\; [C\,(\mathcal{M}) \wedge \delta\,[S_1]\,(\mathcal{M}, \mathcal{M}')\,] \vee [\neg C\,(\mathcal{M}) \wedge \delta\,[S_2]\,(\mathcal{M}, \mathcal{M}')\,]
\end{array}
$$

whereas this statement is only defined when the following assertion is true:

$$
\begin{array}{l}
\Omega\,[\textbf{if } C \textbf{ then } S_1 \textbf{ else } S_2]\,(\mathcal{M}) = \\
\quad \Omega\,[C]\,(\mathcal{M}) \wedge [C\,(\mathcal{M}) \Rightarrow \Omega\,[S_1]\,(\mathcal{M})\,] \wedge [\neg C\,(\mathcal{M}) \Rightarrow \Omega\,[S_2]\,(\mathcal{M})\,]
\end{array}
$$

where $\Omega\,[C]\,(\mathcal{M})$ is true if and only if Boolean value for C is perfectly defined by values of variables in $\mathcal{M}$.

The restricted form without "else" part is often used. In that case the statement S_2 is implicitly the do-nothing statement **skip**. Also, grouping using **begin** and **end** words might be needed to avoid ambiguous notations.

4.5 Locations

It is often handy to have a specific notation for control information. For example if the *location* (or *label*) of a particular process has a value in $\{\text{think}, \text{wait}, \text{critic}\}$, a transition like

$$(\text{wait}, x = 1 \longrightarrow x := 0, \text{critic})$$

has a straight intuitive meaning: the process can go from location "wait" to "critic" only if $x = 1$ by changing this value to zero. Locations are some kind of variable: here "think", "wait" and "critic" are Boolean propositions and preceding transition naturally translates to

$$(\text{wait} \wedge x = 1 \longrightarrow x := 0; \text{wait} := \text{false}; \text{critic} := \text{true})$$

Implicitly, assertions

$$\text{think} \vee \text{wait} \vee \text{critic}$$

and

$$[\text{think} \Rightarrow (\neg\text{wait} \wedge \neg\text{critic})] \wedge [\text{wait} \Rightarrow (\neg\text{critic})]$$

are always true. The first expresses that the process has to be at least in one location, and the second states that the process is not ubiquitous. Generally, those assertions are referred to as the *label rules*.

Rather than associating propositions to locations, it is also possible to use a location variable for every process:

$$(\ell = \text{wait} \wedge x = 1 \longrightarrow x := 0; \ell = \text{critic})$$

Variable ℓ takes its value in $\{\text{think}, \text{wait}, \text{critic}\}$. The label rules now are simply expressed as

$$\ell = \text{think} \vee \ell = \text{wait} \vee \ell = \text{critic}$$

and

$$\text{think} \neq \text{wait} \wedge \text{think} \neq \text{critic} \wedge \text{wait} \neq \text{critic}.$$

Notice that the second label rule (in both translations) has a quadratic size with respect to the size of the label set.

Systems with a parameterized number of processes can be handled exactly the same way. Arrays are now used rather than simple variables. The argument is of process type. Finally, process meetings can be organized in transitions. For example, transition

$$(\text{wait}[i]\ \text{critic}[j],\ \mathbf{skip},\ \text{wait}[j]\ \text{critic}[i])$$

is only active if process i and j are respectively in location "wait" and "critic". Both processes exchange their locations. It is simply translated (using the second method) to

$$(\ell[i] = \text{wait} \wedge \ell[j] = \text{critic} \longrightarrow \ell[j] := \text{wait}; \ell[i] := \text{critic})$$

where ℓ is a function from the process set to $\{\text{think}, \text{wait}, \text{critic}\}$.

In process meetings, there may be several labels in the origin, and several labels in the end. But labels in the origin and in the end go by pairs, about the same process. This naturally leads us to briefly consider a borderline construct: forks. A fork is a process creation. It can be modelled using transition systems, by considering there is a pool of inactive processes. The fork transition awakes one of them. For instance:

$$(\ell_0[i], \neg\mathrm{active}[j] \longrightarrow \mathrm{active}[j], \ell_1[i]\ \ell_2[j])$$

It remains to ensure no inactive process will take part in a transition by adding to transition conditions an assertion true if and only if every participating process is active. If every transition has a label in its origin for all participating process, the same goal can be achieved simply by adding the "inactive" location to label rules.

4.6 Loops

Sequence and conditional statements are useful to express in one transition what would need several without them. For example transition

$$(C_1 \longrightarrow \textbf{if } C_2 \textbf{ then } S_1 \textbf{ else } S_2)$$

can be replaced by the two transitions

$$(C_1 \wedge C_2 \longrightarrow S_1) \qquad (C_1 \wedge \neg C_2 \longrightarrow S_2)\,.$$

Using the loop statement could also be convenient:

$$(\ell, \textbf{for } i := 0 \textbf{ to } j \textbf{ do } S, \ell')$$

Unfortunately, in general the relation associated with such transitions cannot be generated in an automated way. To guarantee an automated generation of verification conditions, loop statements should not be allowed in transitions. However, as for sequence and conditional statements, it is possible to express the same behaviour without these special statements. To circumvent the problem in previous transition, it suffices to rather use transitions:

$$\begin{array}{l}(\ell,\ \neg p \longrightarrow i := 0; p := \top,\ \ell_1)\\ (\ell_1,\ i > j \longrightarrow \textbf{skip}; p := \bot,\ \ell')\\ (\ell_1,\ i \leq j \longrightarrow S; i := i+1,\ \ell_1)\end{array}$$

where ℓ_1 is a new location, and p a new Boolean variable. It is assumed $\neg p$ is added in all conditions of other transitions in the system to ensure the "loop" action is not interleaved with other actions, in case of a parallel system. The price to pay for using this formulation rather than a loop statement in a single transition is the (slight) annoyance of having to introduce two new variables. But also and more significantly, any invariant will have to specify a subset of reachable states with process on location ℓ_1. This amounts to provide a loop invariant.

4.7 Hoare triple, preconditions, invariant

By definition, the *Hoare triple* [107, 10]

$$\{P\}\, S\, \{Q\}$$

where P and Q are assertions (respectively called precondition and postcondition) using variables in $\mathcal{M}$ and S is a statement[9] about variables in $\mathcal{M}$ is a notation for the formula

$$\forall \mathcal{M}' \Big[\, [P \wedge \Omega\,[S]\,(\mathcal{M}) \wedge \delta\,[S]\,(\mathcal{M}, \mathcal{M}')] \Rightarrow Q[\mathcal{M} \rightarrow \mathcal{M}'] \Big]$$

Intuitively the Hoare triple is valid if any properly terminating execution of S starting in a state satisfying assertion P leads to a state satisfying assertion Q. A *weakest liberal precondition* [62] predicate transformer "wlp" is an operator such that $\{P\}\, S\, \{Q\}$ is valid if and only if $P \Rightarrow \mathrm{wlp}\,[S, Q]$ is, for any statement S and assertions P and Q. From the definition of Hoare triple, the operator

$$\begin{aligned}
\mathrm{wlp}\,[S, Q] \quad &=_{\mathrm{def}} \quad \forall \mathcal{M}' \Big[[\Omega\,[S]\,(\mathcal{M}) \wedge \delta\,[S]\,(\mathcal{M}, \mathcal{M}')\,] \Rightarrow Q[\mathcal{M} \rightarrow \mathcal{M}'] \Big] \\
&= \quad \neg\Omega\,[S]\,(\mathcal{M}) \vee \forall \mathcal{M}' \Big[\delta\,[S]\,(\mathcal{M}, \mathcal{M}') \Rightarrow Q[\mathcal{M} \rightarrow \mathcal{M}'] \Big]
\end{aligned}$$

is a weakest liberal precondition predicate transformer.

It is often more appropriate to consider the Hoare triples with a total correctness point of view [11]: the Hoare triple $\{P\}\, S\, \{Q\}$ is valid if every state satisfying assertion P leads *without failure or non-terminating execution* to a state satisfying assertion Q. Formally

$$[P \Rightarrow \Omega\,[S]\,(\mathcal{M})\,] \wedge \forall \mathcal{M}' \Big[\, [P \wedge \delta\,[S]\,(\mathcal{M}, \mathcal{M}')] \Rightarrow Q[\mathcal{M} \rightarrow \mathcal{M}'] \Big]$$

A *weakest precondition* [62] predicate transformer "wp" is an operator such that $\{P\}\, S\, \{Q\}$ is valid (with a total correctness point of view) if and only if $P \Rightarrow \mathrm{wp}\,[S, Q]$ is, for any statement S and assertions P and Q. From the previous definition, the operator

$$\mathrm{wp}\,[S, Q] =_{\mathrm{def}} \Omega\,[S]\,(\mathcal{M}) \wedge \forall \mathcal{M}' \Big[\delta\,[S]\,(\mathcal{M}, \mathcal{M}') \Rightarrow Q[\mathcal{M} \rightarrow \mathcal{M}'] \Big]$$

is a weakest precondition predicate transformer.

[9] The statement S may correspond to a transition. The definitions of Hoare triple and weakest (liberal) precondition directly extend to accept transitions in place of statements.

The difference between liberal and non-liberal weakest precondition predicate transformers is that non-liberal transformer requires the statement to be computable, in finite time:

$$\mathrm{wp}\,[S, Q] = \Omega\,[S]\,(\mathcal{M}) \wedge \mathrm{wlp}\,[S, Q]$$

whereas the liberal version accepts failing and non-terminating statements

$$\mathrm{wlp}\,[S, Q] = \neg\Omega\,[S]\,(\mathcal{M}) \vee \mathrm{wp}\,[S, Q]\,.$$

It is easily shown that our weakest (liberal) precondition operators verify all their usual properties [62].

Notice that if Q and S are an assertion and a statement on variables $\mathcal{M}$, then assertion $\mathrm{w(l)p}\,[S, Q]$ as expressed here will contain program variables which are not in $\mathcal{M}$. The multiplication of those different variables may be undesired for later automated treatment. Furthermore, some of those quantified program variables are translated to functions or predicates in all logical formulas representing relations and assertions about them: $\mathrm{w(l)p}\,[S, Q]$ may contain second-order quantifiers. Those two issues (introduced variables, and second-order quantifiers) are discussed in the following section.

An *invariant*[10] is an assertion preserved by every execution step of the system. In other words, assertion I is an invariant, and if the system state is such that I is true, the execution of any transition from this state leads[11] to a state which also makes the invariant I true. This also means that, if I is a logical consequence of the assertion describing the initial state, then I is true during every execution of the system, i.e., the set of all reachable states is a subset of the set of states represented by the invariant I. If an assertion P is a logical consequence of an invariant of the system, then P is an *invariance property*, i.e., it is true during every execution of the system. Using the definition of Hoare triples, assertion I is an invariant of a transition system if and only if each triple

$$\{I\}\,S\,\{I\}$$

is valid, where S is the statement corresponding to a transition of the system.

4.8 Hoare triples in practice

In the preceding section two operators corresponding to the predicate transformers "wlp" and "wp" were introduced. Assertions $\mathrm{w(l)p}\,[S, Q]$ can be

[10] Also referred as *inductive* invariant.
[11] Without failure or non-terminating execution, if total correctness is considered.

expressed simply in terms of Q and assertions δ and Ω associated to the statement S. Unfortunately, even when Q is a first-order logic formula, assertions $\mathrm{w(l)p}\,[S,Q]$ may contain second-order quantifiers. Those quantifiers potentially come from program variables modelled as predicates or functions, notably in δ and Ω assertions. Besides second-order quantifiers, assertions $\mathrm{w(l)p}\,[S,Q]$ may also contain many redundant variables, not appearing in Q, which may confuse a theorem prover for later validation. Those two problems must be solved for a practical use of the predicate transformers.

4.8.1 Eliminating redundant variables

Among the variables occurring in $\mathrm{w(l)p}\,[S,Q]$ but not in Q, many of them appear in equalities with both members being variables. Most of them stem from the "pres" operator. Basically such equalities duplicate variables. The number of such variables is in practice unacceptably large. Indeed, given a statement S, the number of variables in $\mathrm{w(l)p}\,[S,Q]$ would be the number of variables in Q multiplied by the number of assignment and **skip** statements in S. Luckily most of those variables can easily be eliminated.

Given a statement S, we define $\Sigma[S](\mathcal{M})$ as the set of all variables occurring as left member of an assignment in S. This set $\Sigma[S](\mathcal{M})$ can easily be computed for all statements studied in Section 4.4 (see Figure 4.1). The redundant variables occur because no attention is taken to the fact that most statements only change few program variables. By defining a new assertion δ' similar to δ but such that variables unmodified by the statement (that is, variables which are not in $\Sigma[S]$) are not taken into account, it is possible to circumvent the problem, and to avoid the extensive use of the "pres" operator in the δ formulas.

The new relation δ' given in Figure 4.2 does not contain variables which remain unmodified by the corresponding statement. The quantified variables ($\mathcal{M}_2$) in formula δ' associated to the sequence statement exactly correspond to those variables which are both modified by the first and the second statement of the sequence. Also observe that the only δ' formula containing "pres" operators correspond to the $\mathbf{if}-\mathbf{then}-\mathbf{else}$ statement. Those equalities are used to ensure transformed variables in the **then** and **else** branches are the same.

For any statement S containing only **skip**, generalized statements, sequences, and $\mathbf{if}-\mathbf{then}-\mathbf{else}$ statements, it is easily shown by structural induction on S that[12]

$$\delta[S](\mathcal{M},\mathcal{M}') \longleftrightarrow \delta'[S](\mathcal{M},\mathcal{M}') \wedge \mathrm{pres}(\mathcal{M}\setminus\Sigma[S](\mathcal{M}),\mathcal{M},\mathcal{M}')$$

[12] A summary of statement associated formulas is given in Appendix B.

Statement	$\Sigma\,(\mathcal{M})$
skip	$\emptyset$
$\forall\vec{u}_1\ v_1[\vec{u}_1] := E_1(\vec{u}_1) \parallel \ldots \forall\vec{u}_n\ v_n[\vec{u}_n] := E_n(\vec{u}_n)$	$\{v_1, \ldots v_n\}$
$S_1; S_2$	$\Sigma\,[S_1]\,(\mathcal{M}) \cup \Sigma\,[S_2]\,(\mathcal{M})$
if C **then** S_1 **else** S_2	$\Sigma\,[S_1]\,(\mathcal{M}) \cup \Sigma\,[S_2]\,(\mathcal{M})$

Figure 4.1: Variables modified by statements

Statement	$\delta'\,(\mathcal{M}, \mathcal{M}')$
skip	$\top$
$\forall\vec{u}_1\ v_1[\vec{u}_1] := E_1(\vec{u}_1) \parallel \ldots \forall\vec{u}_n\ v_n[\vec{u}_n] := E_n(\vec{u}_n)$	$\bigwedge_{1\le i\le n} \forall\vec{u}_i\ \ (v_i[\vec{u}_i])\,[\mathcal{M} \to \mathcal{M}'] = E_i(\vec{u}_i)$
$S_1; S_2$	$\exists\mathcal{M}_2[\delta'\,[S_1]\,(\mathcal{M}, \mathcal{M}_1) \wedge \delta'\,[S_2]\,(\mathcal{M}_1, \mathcal{M}')\,]$
if C **then** S_1 **else** S_2	$[C\,(\mathcal{M}) \wedge \delta'\,[S_1]\,(\mathcal{M}, \mathcal{M}') \wedge \mathrm{pres}(\mathcal{M}_3, \mathcal{M}, \mathcal{M}')]$ $\vee\,[\neg C\,(\mathcal{M}) \wedge \delta'\,[S_2]\,(\mathcal{M}, \mathcal{M}') \wedge \mathrm{pres}(\mathcal{M}_4, \mathcal{M}, \mathcal{M}')]$

with

$$
\begin{aligned}
\mathcal{M}_1 &= \mathcal{M}\,[\Sigma\,[S_1]\,(\mathcal{M}) \longrightarrow \mathcal{M}']\,[\Sigma\,[S_2]\,(\mathcal{M}') \longrightarrow \mathcal{M}'']\\
\mathcal{M}_2 &= \Sigma\,[S_1]\,(\mathcal{M}'') \cap \Sigma\,[S_2]\,(\mathcal{M}'')\\
\mathcal{M}_3 &= \Sigma\,[S_2]\,(\mathcal{M}) \setminus \Sigma\,[S_1]\,(\mathcal{M})\\
\mathcal{M}_4 &= \Sigma\,[S_1]\,(\mathcal{M}) \setminus \Sigma\,[S_2]\,(\mathcal{M})
\end{aligned}
$$

Figure 4.2: Variable sparing statement relations

This permits to easily rewrite Ω formulas in term of δ' formulas. Actually the only Ω formula using δ is the one associated to the sequence statement:

$$\begin{aligned}
&\Omega[S_1; S_2]\,(\mathcal{M}) \\
&\quad= \Omega\,[S_1]\,(\mathcal{M}) \wedge \forall \mathcal{M}''[\delta\,[S_1]\,(\mathcal{M}, \mathcal{M}'') \Rightarrow \Omega\,[S_2]\,(\mathcal{M}'')\,] \\
&\quad= \Omega\,[S_1]\,(\mathcal{M}) \wedge \forall \mathcal{M}_2[\delta'\,[S_1]\,(\mathcal{M}, \mathcal{M}_1) \Rightarrow \Omega\,[S_2]\,(\mathcal{M}_1)\,]
\end{aligned}$$

with

$$\begin{aligned}
\mathcal{M}_1 &= \mathcal{M}[\Sigma[S_1](\mathcal{M}) \longrightarrow \mathcal{M}'] \\
\mathcal{M}_2 &= \Sigma[S_1](\mathcal{M}')
\end{aligned}$$

that is, the quantified variables in $\Omega[S_1; S_2]\,(\mathcal{M})$ now in fact correspond to the variables used in S_1. The same translation can be applied to w(l)p formulas. For instance,

$$\mathrm{wp}\,[S, Q] = \Omega\,[S]\,(\mathcal{M}) \wedge \forall \mathcal{M}_2 \Big[\delta'\,[S]\,(\mathcal{M}, \mathcal{M}_1) \Rightarrow Q[\mathcal{M} \to \mathcal{M}_1]\Big] \tag{4.2}$$

where $\mathcal{M}_1$ and $\mathcal{M}_2$ are defined just as they were for Ω formulas.

Consider now the size of generated Σ, δ' and Ω formulas corresponding to a statement S. We assume $\Omega[C]$ and $\Omega[E_i(\vec{u}_i)]$ are linear in size with respect to the size of C and $E_i(\vec{u})_i$ respectively. If S is the general assignment or the **skip** assignment, the size of the resulting formulas is linearly related to the size of S. In general $\Sigma[S]$ is linear in size with respect to the size of the statement. Unfortunately this is not the case for the δ' and Ω formulas.

The δ' assertion associated to the conditional statement may contain some introduced equalities due to variables assigned in the **then** statement and not in the **else** statement or conversely. If S contains nested **if** − **then** − **else**, this may involve a quadratic size of the resulting δ' formula.

A similar behaviour occurs in Ω, with even more dramatic effects. If the operator grouping two statements into one (;) is considered to be left-associative, the Ω formula associated to $S_1; \ldots S_n$ may also be non linear in size with respect to n. Combined together, those two non linear behaviours involve the w(l)p expressions may have a size of $\mathcal{O}(|S|^4)$. However, if the depth of nested **if** − **then** − **else** is bounded by a constant, and if the sequence operator is assumed right-associative, then the linear behaviour is recovered.

4.8.2 Eliminating second-order quantifiers

The w(l)p, δ' and Ω assertions potentially contain (second-order) quantifiers on functions and predicates. They may constitute a problem for the automated validation of verification conditions. Fortunately, there is a simple validity preserving elimination method of those quantifiers in verification conditions, when considering Hoare triples with a total correctness point of view. In formula

$$P \Rightarrow \mathrm{wp}\,[S, Q]$$

the second order quantifiers stem from $\Omega[S]$, $\delta^{(\prime)}[S]$, or from the universal quantifiers introduced by the wp transformer. Those last quantifiers can be directly eliminated. Indeed assertion[13]

$$\begin{aligned} & P \Rightarrow [\Omega\,[S]\,(\mathcal{M}) \wedge \forall \mathcal{M}_2[\delta'\,[S]\,(\mathcal{M}, \mathcal{M}_1) \Rightarrow Q[\mathcal{M} \to \mathcal{M}_1]]] \\ \longleftrightarrow\;\; & \forall \mathcal{M}_2\Big[P \Rightarrow \Big(\Omega\,[S]\,(\mathcal{M}) \wedge [\delta'\,[S]\,(\mathcal{M}, \mathcal{M}_1) \Rightarrow Q[\mathcal{M} \to \mathcal{M}_1]]\Big)\Big] \end{aligned} \tag{4.3}$$

is valid if and only if

$$P \Rightarrow \Big(\Omega\,[S]\,(\mathcal{M}) \wedge [\delta'\,[S]\,(\mathcal{M}, \mathcal{M}_1) \Rightarrow Q[\mathcal{M} \to \mathcal{M}_1]]\Big)$$

is also valid. The same elimination method can be applied to all quantifiers $\forall \mathcal{M}$ and $\exists \mathcal{M}$ in $\Omega[S]$ and $\delta^{(\prime)}[S]$. Indeed, such universal quantifiers always appear with a positive polarity in (4.3), whereas existential quantifiers always appear with a negative polarity. As long as the set of quantified variables are disjoint, elimination (using second order Skolemization) of all those quantifiers is a validity preserving transformation.

Notice that verification conditions using the weakest *liberal* precondition cannot be freed from second-order quantifiers so easily. As the Ω formula is negated in the wlp assertion, all quantifiers in this subformula will have the polarity opposite to what would be required to eliminate them with a validity preserving transformation.

4.8.3 Eliminating introduced variables

The $\delta^{(\prime)}$ and Ω assertions, and thus the verification conditions as build in this chapter may involve many new quantified program variables. For instance, the δ' function associated to the simple statement $x := x + 1$ makes use of two logical variables, to denote the values of the program variable x, before and after execution of the statement: $x' = x + 1$. Also, some of those new "variables" may well be functions or predicates, and involve the presence of second-order quantifiers in verification conditions. We now show that it is possible to eliminate all those new symbols (and quantifiers) in verification conditions. Unfortunately this transformation is expensive.

[13] $\mathcal{M}_1$ and $\mathcal{M}_2$ are defined as in formula (4.2).

It is convenient here to use the new notation $\lambda(x_1, \ldots x_n).E(x_1, \ldots x_n)$ to denote the function whose value is $E(t_1, \ldots t_n)$ for any arguments $t_1, \ldots t_n$ of suitable sorts. As x_i is a logical variable (i.e., not a function nor a predicate) for every $i \in \{1, \ldots n\}$, allowing the use of this restricted class of lambda expressions as function does not augment the expressive power of many-sorted first-order logic.[14] Indeed,

$$[\lambda(x_1, \ldots x_n).E(x_1, \ldots x_n)]\,(t_1, \ldots t_n) = E(t_1, \ldots t_n). \tag{4.4}$$

The $\delta^{(\prime)}$ and Ω assertions associated to a statement S contain, among other things, formulas such as

$$\forall \vec{u}\ v'(\vec{u}) = E(\vec{u})$$

stemming from the $\delta^{(\prime)}$ relations associated to individual generalized assignments in S. Using the restricted form of lambda expression, those formulas can be rewritten as

$$v' = \lambda(\vec{u}).E(\vec{u}).$$

This means $\delta^{(\prime)}$ relations associated to generalized assignments can be written without quantifiers, with the help of lambda expressions. Also notice that in general the preceding equality sign denotes an equality between functions (or predicates). This is usually not allowed in first-order logic.

To remove all those second-order equalities, together with the second order quantifiers, and the introduced variables, it remains to notice that verification conditions can be rewritten to be such that, for every $\delta^{(\prime)}$ relation associated to a generalized assignment A, it appear in formulas of the form

$$\forall \mathcal{M}'[\delta^{(\prime)}\,[A]\,(\mathcal{M}, \mathcal{M}') \Rightarrow \varphi].$$

This is easily proved by structural induction. This transformation however already potentially involves an exponential formula growth, due to **if−then−else** statements.

As a final step it suffices to use an equality simplification method similar to the one presented in [141]. Formula

$$\forall \mathcal{M}'\Big[\Big(v_1' = \lambda(\vec{u}_1).E(\vec{u}_1)\ \wedge \ldots v_n' = \lambda(\vec{u}_n).E(\vec{u}_n)\Big) \Rightarrow \varphi\Big]$$

is logically equivalent to formula φ in which every v_i' ($i \in \{1, \ldots n\}$) has been substituted by the corresponding lambda expression $\lambda(\vec{u}_i).E(\vec{u}_i)$. Those lambda expressions can simply be eliminated using identity (4.4). The verification condition obtained by successively eliminating all introduced variables

[14] In [46] a quantifier-free language incorporating restricted lambda expression is studied.

and functions is a first-order assertion if the precondition, the postcondition and all statement conditions are first-order assertions, and if the expression in the right member of assignments are well behaved.[15]

To conclude, notice that renaming may be used to avoid size explosion of verification conditions, by introducing new variables, predicates and functions. In fact, renaming and elimination of introduced variables have two opposite goals. Renaming aims at keeping formula sizes small, at the cost of introducing symbols. In opposite, the method in this section aims at reducing the number of symbols.

4.9 Limits of first-order invariants

In most cases, using first-order logic as the language for verification conditions is a good choice, as it permits to verify the important safety properties (see Chapter 5 for examples). However, the first-order framework may not be suitable to catch crucial properties for some algorithms. The following algorithm taken from [121] is one of those. It rearranges — without using extra storage — the elements of an array $S_{\text{in}}[1], \ldots S_{\text{in}}[n]$ $(n \geq 1)$ according to a given permutation function f with its argument in $\{1, \ldots n\}$, such that $S_{\text{out}}[i] = S_{\text{in}}[f(i)]$ for every $i \in \{1, \ldots n\}$. Here is the code in Pascal style:

```
for k := 1 to n do
    begin
        l := f(k);
        while l < k do
            l := f(l);
        (S[k], S[ℓ]) := (S[ℓ], S[k]);
    end
```

A transition system corresponding to this code could be:

$$
\begin{aligned}
&(s_0, k := 1, s_1)\\
&(s_1, k > n \longrightarrow \mathbf{skip}, s_6)\\
&(s_1, k \leq n \longrightarrow \mathbf{skip}, s_2)\\
&(s_2, \ell := f(k), s_3)\\
&(s_3, \ell \geq k \longrightarrow \mathbf{skip}, s_4)\\
&(s_3, \ell < k \longrightarrow \ell := f(\ell), s_3)
\end{aligned}
$$

[15] For instance if they are first-order terms, or if-then-else expressions with first-order condition, and first-order terms for then and else-cases.

$$(s_4, (S[k], S[\ell]) := (S[\ell], S[k]), s_5)$$
$$(s_5, k := k+1, s_1)$$

where the final state is in location s_6. The initial state is perfectly described by the first-order assertion

$$s_0 \wedge \forall i\ S[i] = S_{\text{in}}[i].$$

The "sorted" array is computed using two loops: one loop (line 1) runs over all array indices successively, whereas a second loop (lines 3 to 5) computes the value ℓ such that $S[k]$ and $S[\ell]$ will be swapped. Observe that ℓ is strictly greater than k when swapping both array elements. In other words, the value assigned to $S[k]$ in line 6 will remain assigned to $S[k]$ until the end of the run, for every each execution of line 6. This involves that, in line 6, $S[\ell]$ should be $S_{\text{in}}[f(k)]$.

The formal verification task may be divided in two parts. One part ensures that the computation of ℓ is such that $S[\ell]$ is $S_{\text{in}}[f(k)]$. The other part states that the elements of array S are put in their right place, one by one, from position 1 to n. If the first task is fulfilled, the second is very simple, and can easily be expressed within first-order logic with order and the successor function. Formula

$$\begin{aligned} & s_0 \Rightarrow \forall i\ S[i] = S_{\text{in}}[i] \\ \wedge\ \ & (s_1 \vee s_2 \vee s_3 \vee s_4 \vee s_6) \Rightarrow \forall i\, [i < k \Rightarrow S[i] = S_{\text{in}}[f(i)]] \\ \wedge\ \ & s_5 \Rightarrow \forall i\, [i \leq k \Rightarrow S[i] = S_{\text{in}}[f(i)]] \end{aligned} \tag{4.5}$$

is preserved by every transition of the system, provided the two following assertions can be established:

$$s_4 \Rightarrow \ell \geq k,$$
$$s_4 \Rightarrow S[\ell] = S_{\text{in}}[f(k)].$$

The first assertion is in fact an inductive invariant of the system. Proving the second one is more difficult, and cannot immediately be done within first-order logic.

The computation of ℓ involves only lines 3 to 5, i.e., the loop entering from s_2 and exiting in s_4. In fact, the output of this loop will be the first element in the sequence $f(k)$, $f(f(k))$, $f(f(f(k)))$, $\ldots$ to be larger or equal to k. Given a permutation function f over $\{1, \ldots n\}$ and $k, \ell \in \{1, \ldots n\}$, let $W_{k,\ell}$ be the smallest set such that

- $f(\ell) \in W_{k,\ell}$;

- if $x \in W_{k,\ell}$ and $x < k$, then $f(x) \in W_{k,\ell}$.

If $\ell \geq k$, then $W_{k,\ell}$ contains one and only one element greater or equal to k. Let $f^{\star k}(\ell)$ be this element, for every ℓ and k such that $\ell \geq k$. If $\ell < k$, let $f^{\star k}(\ell) = \ell$. Notice that

1. for $k \in \{1, \ldots n\}$, function $f^{\star k}$ is defined and is a permutation function of $\{1, \ldots n\}$.
2. permutations functions $f^{\star(k+1)}$ and $f^{\star k}$ only differ (if they differ) by an elementary permutation of values in positions k and $f^{\star k}(k)$;
3. $f^{\star 1} = f$;
4. $f^{\star n}$ is the identity.

Proof. The last two points are direct consequences of the definition. Assuming $f^{\star k}$ is a permutation function,

- if $\ell < k$, $f^{\star(k+1)}(\ell) = f^{\star k}(\ell) = \ell$ (by definition);
- $f^{\star(k+1)}(k) = k$ (by definition);
- if $\ell > k$,
  - if $f^{\star k}(\ell) \neq k$ (in that case $f^{\star k}(\ell) > k$), then $f^{\star(k+1)}(\ell) = f^{\star k}(\ell)$ (as $W_{k,\ell} \subset W_{(k+1),\ell}$ and $f^{\star k}(\ell) \in W_{k,\ell}$, and $f^{\star k}(\ell) \geq (k+1)$);
  - if $f^{\star k}(\ell) = k$, then $f^{\star k}(k) \neq k$ as $f^{\star k}$ is a permutation function, and hence $f^{\star k}(k) > k$ (by definition). In that case, if $f^{\star k}(\ell) = k$, then $f^{\star(k+1)}(\ell) = f^{\star k}(k)$, as $W_{(k+1),\ell} = W_{k,\ell} \cup W_{k,k}$.

This permits to prove the remaining two points by induction on k. □

The value calculated by the loop in s_3 is actually $f^{\star k}(k)$, and the conjunction of Formula (4.5) with assertion

$$
\begin{aligned}
& (s_1 \vee s_2 \vee s_3 \vee s_4) \Rightarrow \forall i : S[f^{\star k}(i)] = S_{\text{input}}[f(i)] \\
\wedge \quad & s_3 \Rightarrow \ell \in W_{k,k} \\
\wedge \quad & s_4 \Rightarrow \ell = f^{\star k}(k) \\
\wedge \quad & s_5 \Rightarrow \forall i : S[f^{\star(k+1)}(i)] = S_{\text{input}}[f(i)]
\end{aligned}
$$

is an inductive invariant for the system. Unfortunately, definitions of functions $f^{\star k}$ and of sets $W_{k,k}$ are not easily caught by first-order assertions.

In [4], a protocol related to the IEEE 1394 standard is verified, using the invariant techniques of the B method [3]. The system describes a leader election protocol on a particular topology of the communication network: the network is represented by one bidirectional graph, such that there is a path from every node to every other, and such that every path from a node to itself is either the empty path, or contains two opposite edges between two nodes (the graph has an acyclic topology). The property to prove is that exactly one leader is elected, and a tree is computed with the leader on its root. The tree is a spanning tree, i.e., it contains all nodes.

A graph can be represented as a binary predicate on the sort of the nodes. However, it is not possible, in general, to express within first-order logic the fact that a relation represented by a binary predicate is acyclic, or represents a tree... In such a framework, it is impossible to exactly specify the initial conditions of the algorithm, as well as the properties to verify. The framework of the B method is a first-order logic *with set theory*. Graphs are not represented as binary predicates, but as variables. The properties of those graphs are described with the help of interpreted predicates and functions, and this allows to successfully describe the properties and verification conditions of the IEEE 1394 protocol.

Chapter 5

Invariant validation is (sometimes) decidable

Automatic verification of parameterized systems, and more precisely verification of systems with an unknown number of concurrent finite-state processes, has been shown to be undecidable [12]. Invariant validation is a subproblem of verification. A formula I is an invariant if and only if the system satisfies the property $\Box(I \Rightarrow \bigcirc I)$. Unfortunately this subproblem is also undecidable. Consider the Hoare triple $\{I\}\,\neg x \longrightarrow x := \text{true}\,\{I\}$, where $I = (\neg x \Rightarrow \varphi) \wedge (x \Rightarrow \psi)$, and x does not appear in φ and ψ. It is valid if and only if the formula $(I \wedge x) \Rightarrow \psi$ holds, that is, if and only if $\varphi \Rightarrow \psi$ is valid. Hoare triple validation and hence invariant validation is at least as difficult as the satisfiability problem for the language accepted for invariants. Full first-order logic being undecidable, verifying invariants written as first-order assertions is also undecidable in general.

Proving algorithms with the help of invariants reduces the verification of safety properties to the validity problem for logic formulas (see Chapter 4). It could happen that the formula to be proved belongs to a well known decidable class (for instance, Presburger arithmetic), but this is rarely the case because Boolean arrays (modelled by uninterpreted predicates) are often used in these algorithms, together with interpreted predicates.[1] In general, the use of complex data structures in algorithms makes difficult the application of decision procedures for well known quantified class of formulas.

Quantifier-free first-order logic satisfiability checking is decidable for a very wide class of formulas with non-interpreted and interpreted predicates and functions (see Chapters 6 to 8). Thus decidability can be reached if quantifier elimination can be done. We introduce here a simple quantifier

[1]Presburger arithmetic with (unary) uninterpreted predicates is undecidable [104].

elimination method for a large class of verification conditions. It is based on a variant of Herbrand's Theorem (cf. Theorem 2) which allows to have some kind of finite model property [68] even when some functions (interpreted or not) and interpreted predicates are used in formulas. This theorem is also preserving, so that interpreted predicates may be used provided some requirements are fulfilled. We then give criteria for verification conditions to benefit from this finite property. Those criteria allow to eliminate quantifiers in the proof by invariant of many reactive algorithms, and particularly for parameterized algorithms, leading to a powerful invariant validation procedure. It allows to reduce the invariant validation for a system with a parameterized number of processes to the invariant validation for a system with a known number of processes n_0 (Theorem 9). This method can be seen as an extension of the invariant validation procedure presented in [14, 149]: our approach does not restrict the use of functions and predicates to unary ones, and is not restricted to bounded variables. For instance, our method can deal with the Burns algorithm (see section 5.4) even though it uses a two dimensional Boolean array. Also notice that the presence of non-unary arrays makes difficult the use of regular model checking, by breaking the finite-state requirement for processes (see for instance [40, 39] for an overview of regular model checking techniques for parametric systems, and systems using integer or reals). More generally, it should also be mentioned that parameterized systems here do not mean systems with a parameterized number of *finite-state* processes: processes are not required to be finite state, as long as they verify the syntactic requirements of Theorem 9.

5.1 Herbrand and interpreted symbols

Using the classical Herbrand's Theorem, or the version for many-sorted logic with equality (Theorem 2) is problematic when dealing with interpreted predicates and functions. Indeed those theorems state the existence of an Herbrand model, but this Herbrand model may interpret the interpreted predicates and functions very differently from what is desired. The following theorem is very much inspired from Theorem 2. Just as the Herbrand's Theorem it allows to eliminate quantifiers, but as opposed to the Herbrand's Theorem, it preserves all but one domain and all predicates and functions without connection with this domain.

Theorem 8. *Given*

- *a closed formula S in* Skolem form *on the many-sorted language $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, r, d \rangle$;*

- *a sort $\tau \in \mathcal{S}$ such that there exists no function symbol in S of sort $\langle \tau_1, \ldots \tau_n, \tau \rangle$ with $n > 0$, $\tau_1, \ldots \tau_n \in \mathcal{S}$;*

the set H_τ is the set of constant symbols of sort τ ($H_\tau = \{c \in \mathcal{F} \mid d(c) = \tau\}$). If $\{c \in \mathcal{F} \mid d(c) = \tau\} = \emptyset$, then $H_\tau = \{a\}$, where a is an arbitrary new constant symbol such that $a \notin \mathcal{F} \cup \mathcal{P} \cup \mathcal{V}$.

For every model $\mathcal{I} = \langle D, I \rangle$ of S, there is a model $\mathcal{I}' = \langle D', I' \rangle$ such that

- *D'_τ is the quotient of the set H_τ by congruence $\leftrightarrow_=$ induced by $\mathcal{I}$, i.e., $a \leftrightarrow_= b$ is true if and only if $\mathcal{I}[a] = \mathcal{I}[b]$;*
- *$D'_{\tau'} = D_{\tau'}$ for every $\tau' \neq \tau$;*
- *$\mathcal{I}'[f] = \mathcal{I}[f]$ for every function symbol $f \in \mathcal{F}$ of sort $\langle \tau_1, \ldots \tau_n, \tau' \rangle$ such that $\tau_1 \neq \tau, \ldots \tau_n \neq \tau, \tau' \neq \tau$ ($n \geq 0$);*
- *$\mathcal{I}'[p] = \mathcal{I}[p]$ for every function symbol $p \in \mathcal{P}$ of sort $\langle \tau_1, \ldots \tau_n \rangle$ such that $\tau_1 \neq \tau, \ldots \tau_n \neq \tau$ ($n \geq 0$).*

Proof. The interpretation $\mathcal{I}'$ is built from $\mathcal{I}$:

- for every constant symbol c of sort τ in $\mathcal{F}$, $I'[c]$ is the class of c in D_τ;
- for every function symbol $f \in \mathcal{F}$ of sort $\langle \tau_1, \ldots \tau_n, \tau' \rangle$ ($n > 0$), and every $d'_1 \in D'_{\tau_1}, \ldots d'_n \in D'_{\tau_n}$, $I'[f](d'_1, \ldots d'_n) = I[f](d_1, \ldots d_n)$ where $d_i = d'_i$ if $\tau_i \neq \tau$. If $\tau_i = \tau$, $d_i = I(d''_i)$ where d''_i is any element of the class $d'_i \in D_\tau$;
- for every predicate symbol $p \in \mathcal{P}$ of sort $\langle \tau_1, \ldots \tau_n \rangle$, and every elements $d'_1 \in D'_{\tau_1}, \ldots d'_n \in D'_{\tau_n}$, $I'[p](d'_1, \ldots d'_n) = I[p](d_1, \ldots d_n)$ where $d_i = d'_i$ if $\tau_i \neq \tau$. If $\tau_i = \tau$, $d_i = I(d''_i)$ where d''_i is any element of the class $d'_i \in D_\tau$.

It remains to show that $\mathcal{I}'$ is a model of S. Formula S is of the form $\forall x_1 \ldots \forall x_n(\varphi)$. Thus for all elements $d'_1, \ldots d'_n$ such that d'_i belongs to $D'_{\tau'}$ if x_i is a variable of sort τ', the following equality hold:

$$\mathcal{I}'_{x_1/d'_1, \ldots x_n/d'_n}[\varphi] = \mathcal{I}_{x_1/d_1, \ldots x_n/d_n}[\varphi]$$

with $d_i = \mathcal{I}[d''_i]$ where d''_i is any element of the class $d'_i \in D'_\tau$ if x_i is of sort τ, $d_i = d'_i$ otherwise.

The interpretation $\mathcal{I}$ is a model of the formula S: this means that $\mathcal{I}_{x_1/d_1, \ldots x_n/d_n}[\varphi] = \top$ for all elements $d_1, \ldots d_n$ where d_i belongs to $D_{\tau'}$ if x_i is a variable of sort τ'. It follows that $\mathcal{I}'_{x_1/d'_1, \ldots x_n/d'_n}[\varphi] = \top$ for all elements $d'_1, \ldots d'_n$ such that d'_i belongs to $D'_{\tau'}$ if x_i is a variable of sort τ'. So $\mathcal{I}'$ is a model of S. □

Observe that the set H_τ does not become infinite in the presence of functions, as long as functions are well behaved towards this sort, i.e., if they do not have τ as domain. Also notice that its restriction to one-sorted first-order logic gives back the classical Herbrand's theorem (with equality), but restricted to the finite Herbrand universe case. Nevertheless this case is the most interesting one: having a finite domain means that quantifier elimination is possible.

Consider the simple unsatisfiable formula

$$\forall i \forall j\, [f\,(i) > g\,(j)] \;\wedge\; g(a) = 3 \;\wedge\; \exists i\, [f\,(i) < 4] \tag{5.1}$$

where "$<$" and "$>$" are interpreted as usual order predicates on $\mathbb{N} \times \mathbb{N}$. Variables i and j and constants a and b are of sort τ, differing from the sort τ' to which is associated the domain $\mathbb{N}$, whereas f and g are functions from τ to τ'. In this context, the preceding theorem states that formula (5.1) is satisfiable if and only if the formula

$$\begin{aligned} f\,(a) > g\,(a) \;\wedge\; f\,(a) > g\,(b) \;\wedge\; f\,(b) > g\,(a) \\ \wedge\, f\,(b) > g\,(b) \;\wedge\; g\,(a) = 3 \;\wedge\; f\,(b) < 4 \end{aligned}$$

is, using Skolemization (see Section 3.3). This last formula belongs to the decidable class of quantifier-free first-order logic with linear arithmetic on $\mathbb{N}$ and uninterpreted function symbols (see Chapter 8).

Corollary 2. *A τ-universally quantified formula $\forall x\, \varphi(x)$ verifying every condition of Theorem 8 is satisfiable if and only if the finite conjunction $\bigwedge_{c \in H_\tau} \varphi(c)$ is satisfiable.*

5.2 Interpreted predicates and functions

A formula containing interpreted predicates and functions is satisfiable if and only if it has a model in a restricted subset of all interpretations, that is the set where interpretations associate a fixed domain to given sorts and a fixed meaning to those interpreted predicates and functions. In Theorem 8, both interpretations $\mathcal{I}$ and $\mathcal{I}'$ associate the same domain to every sort but τ, and give the same meaning to every predicate and function, provided none of their arguments is of sort τ. In other words, Theorem 8 is compatible with the use of interpreted predicates and functions provided none of their arguments is of sort τ. For instance, in the preceding example (i, j and a are of sort τ) the arguments of the order predicates ($f(i)$, $g(j)$, $\ldots$) are not of the sort τ. Using Theorem 8, interpretation $\mathcal{I}$ and $\mathcal{I}'$ are such that

$\mathcal{I}[<] = \mathcal{I}'[<]$ and $\mathcal{I}[>] = \mathcal{I}'[>]$. This allows to eliminate the quantifiers on the sort τ in presence of interpreted predicates with no argument of sort τ.

It is also possible to use order predicates on the sort of quantified variables. Let φ be a formula with order predicates ("$\leq$", ...) on sort τ, and ψ be the conjunction of the axioms of total order theory,

$$\begin{aligned}\psi \quad = \quad & \forall x\,(x \leq x) \\ & \wedge\ \forall x \forall y\,((x \leq y \wedge y \leq x) \Rightarrow x = y) \\ & \wedge\ \forall x \forall y \forall z\,((x \leq y \wedge y \leq z) \Rightarrow x \leq z) \\ & \wedge\ \forall x \forall y\,(x \leq y \vee y \leq x)\end{aligned}$$

with variables x, y, z of sort τ. An interpretation is a model of $\psi \wedge \varphi$ if and only if it is a model of φ interpreting "$\leq$" as the usual order predicate on D_τ. It is possible and simple to put $\psi \wedge \varphi$ in Skolem form without introducing new Skolem functions: it is guaranteed by Standard Skolemization (see Section 3.3). The conditions of Theorem 8 are verified for $\psi \wedge \varphi$ if they are verified for φ. Theorem 8 can be applied also if some comparisons are made between terms of the sort of quantified variables.[2]

5.3 Quantifier elimination in invariant validation

In order to verify that the assertion H is an invariant of the transition system $\mathcal{S}$, one has to validate the Hoare triple $\{H\}\,\sigma\,\{H\}$ for each transition σ of the system $\mathcal{S}$. This is first reduced to first-order logic proving, using Dijkstra weakest precondition (wp) operator (see Section 4.7): the Hoare triple $\{H\}\,\sigma\,\{H\}$ is valid if and only if the formula $H \Rightarrow \mathrm{wp}[\sigma, H]$ can be proved. Weakest precondition calculus is easy, provided transitions do not contain full loops in their statement part (see Section 4.8).

In general, the invariant is a conjunction ($H = \bigwedge_{k \in K} h_k$) of relatively small assertions h_k. In parameterized systems, these assertions are often quantified over the (parameterized) set of processes. In order to avoid the appearance of Skolem functions when verification conditions are put in Skolem form, an assumption is made about these quantified assertions: they can be put both in prenex form $\exists^\star\forall^\star$ (called *hypothesis form* in the following, because this will be the allowed form in the antecedent of formulas of the form

[2] As in [14], "$+1$" and "$\oplus 1$" functions can sometimes be eliminated without introducing new Skolem functions, by noticing that $h = i + 1 \longleftrightarrow i < h \wedge \forall j\,(j \leq i \vee h \leq j)$ and $h = i \oplus 1 \longleftrightarrow [i < h \wedge \forall j\,(j \leq i \vee h \leq j)] \vee [h < i \wedge \forall j\,(h \leq j \vee j \leq i)]$.

$A \Rightarrow B$) *and* in prenex form $\forall^\star\exists^\star$ (called *conclusion form* in the following, because this will be the allowed form in the conclusion of formulas of the form $A \Rightarrow B$).

In practice, two particular cases of such formulas are met frequently:

- formulas in prenex form containing one type of quantifier;
- formulas containing only monadic predicates (and no equality).[3]

There is also an assumption for guards: guards must be formulas in hypothesis form. Guards met in practice fulfil this assumption as they are quantifier-free formulas or singly quantified formulas.

Taking the preceding conditions on quantifiers into account, proving the formula $H \Rightarrow \mathrm{wp}[\sigma, H]$ (with $H = \bigwedge_{k\in K} h_k$) reduces to proving a set of formulas (called verification conditions) of the form

$$(h_1 \wedge \ldots h_k \wedge G) \Rightarrow C_j$$

where G is the guard of σ. All formulas $h_1 \ldots h_k, G$ are in hypothesis form. There is one verification condition for each h_k ($k \in K$). Formula C_j comes from hypothesis h_j: $C_j \equiv \mathrm{wp}\,[A, h_j]$, where A is the statement part of σ. It is assumed for simplicity that A does not contains conditional statements.[4] The formula C_j can be put in conclusion form, as h_j can be put in conclusion form, and there is a weakest precondition operator preserving the quantifier order of formulas: using the definition of the weakest precondition operator introduced in Section 4.7, and the second-order quantifier eliminating method presented in Section 4.8, those formulas are Boolean combinations containing universally quantified formulas stemming from generalized statements (see Section 4.4) and formula h_j occurring with a positive polarity with some functions, predicates, and constants substituted.

The last requirement concerns functions: we require that no function used in the invariant H, or in the transition system $\mathcal{S}$ has the process set as domain. This may seem rather restrictive, but as reactive algorithms mainly use Boolean arrays (modelled by predicates, not functions), this requirement remains acceptable in practice.

Under those conditions, Theorem 8 can be used to eliminate the quantifiers:

[3]Indeed no quantifier of the miniscope form of a monadic formula (see Section 3.3.2) is in the scope of another quantifier. A monadic formula is logically equivalent to a Boolean combination of prenex forms with one quantifier. So every monadic formula can be put in both hypothesis and conclusion forms.

[4]This does not restrict the language, but simplifies the requirements.

Theorem 9. *If H is a conjunctive formula, and Σ is a transition system with a parameterized number n of processes, where*

- *all quantified variables in H and in the guard of the transitions of Σ range over the set of processes;*
- *every conjunct in H can be put both in hypothesis form ($\exists^\star\forall^\star$) and in conclusion form ($\forall^\star\exists^\star$);*
- *every transition guard can be put in hypothesis form;*
- *no interpreted predicate other than equality and order is used on the set of processes, neither in H nor in Σ;*
- *no function has the process set as domain, neither in H nor in Σ;*

then H is an invariant of Σ if and only if H is an invariant of the system Σ' with at most n_0 processes, where n_0 is the sum of

- *the number of existential quantifiers in H when put in hypothesis form;*
- *the maximum number of existential quantifiers in guards of transitions in Σ;*
- *the maximum number of universal quantifiers in the conjuncts of H, when put in conclusion form;*
- *the number of constants in H;*
- *the maximum number of processes taking part in a transition.*[5]

Proof. From the theorem conditions, every verification condition is of the form

$$(h_1 \wedge \dots h_k \wedge G) \Rightarrow C$$

where formulas $h_1, \dots h_k, G$ can be put in hypothesis form, and C is can be put in conclusion form. It is valid if and only if

$$h_1 \wedge \dots h_k \wedge G \wedge \neg C \tag{5.2}$$

is unsatisfiable. Applying Standard Skolemization to this last formula will introduce new Skolem constants but no new functions, if $h_1, \dots h_k, G$ and $\neg C$ are put in hypothesis form. Notice that $\neg C$ can be put in hypothesis form if and only if C can be put in conclusion form. A Skolem constant will be introduced for each strong quantifier in the formula. Let p be the number of such constants. It is bounded by the sum of

[5] usually at most two.

- the number of existential quantifiers in H $(h_1 \wedge \ldots h_k)$ when put in hypothesis form;
- the maximum number of existential quantifiers in guards (G) of transitions in Σ;
- the maximum number of universal quantifiers in the conjuncts (h_k from which C is computed) of H, when put in conclusion form.

Thanks to Theorem 4, the obtained formula S is satisfiable if and only if the formula (5.2) is also satisfiable. Using Theorem 8, S is satisfiable if and only if there is a model with a finite process set, which contains all process constants used in S. So the cardinality of this process domain is at most n_0, where n_0 is the sum of

- p;
- the number of constants coming from H in φ;
- the maximum number of constants coming from the transitions through G and C, which is the maximum number of processes involved at the same time in a transition.

□

Comment. The satisfiability problem for the Schönfinkel-Bernays class [30], that is, the class of function-free first-order formulas of the form

$$\exists x_1 \ldots \exists x_p \forall y_1 \ldots \forall y_q \, \varphi(x_1, \ldots x_p, y_1, \ldots y_q),$$

has first been shown to be decidable by Bernays and Schönfinkel without equality [30] and by Ramsey with equality [153, 64]. Theorem 8 extends this decidable class to allow the use of some functions (interpreted or not) and some interpreted predicates.

Corollary 3. *When conditions of Theorem 9 are verified, checking if Σ preserves the invariant H is reduced to a quantifier-free first-order logic satisfiability checking problem, with possibly some interpreted predicates and functions.*

Checking the satisfiability for quantifier-free first-order logic formulas is the topic of Chapters 6 to 8. It will be shown (in Chapter 8) that the quantifier-free satisfiability problem is decidable for formulas with interpreted predicates and functions from numerous common useful theories. In particular, the many-sorted quantifier-free satisfiability problem is decidable when domains from some sorts are associated with $\mathbb{N}$, $\mathbb{Z}$, or $\mathbb{R}$ as long as formulas use only linear arithmetic, and uninterpreted predicates and functions. When Theorem 9 applies, and when the quantifier-free satisfiability problem is decidable, the invariant validation problem is decidable. This is the case for numerous algorithms.

5.4 Parameterized Burns algorithm

In this well-known simple example, variables of only one sort are used. Theorem 8 thus reduces to the Herbrand's theorem (with equality, without functions). This simple example allows to clearly exhibit the underlying fact which enables quantifier elimination: a finite Herbrand universe.

Burns algorithm [109], [119, p. 294] guarantees exclusive access to a critical section for a set of n identical processes. Each process p can be in one of six different location states (i.e., $s_0 \dots s_5$). A rule expresses the trivial property that each process is in one and only one state at each time: one and only one variable in $s_0[p], \dots,\ s_5[p]$ is true (for each p). A process p being in s_5 (i.e., $s_5[p]$ is true) is in the critical section.

Twelve transitions are allowed, between the six states:

$$
\begin{array}{l}
(\mathrm{s}_0[p],\ \mathrm{flag}[p]{:=}\mathrm{false},\ \mathrm{s}_1[p]) \\
(\mathrm{s}_1[p], \neg S[p,q]\ \wedge\ q<p\ \wedge\ \mathrm{flag}[q]\ \rightarrow\ \forall q : S[p,q]{:=}\mathrm{false},\ \mathrm{s}_0[p]) \\
(\mathrm{s}_1[p], \neg S[p,q]\ \wedge\ q<p\ \wedge\ \neg\mathrm{flag}[q]\ \rightarrow\ S[p,q]{:=}\mathrm{true},\ \mathrm{s}_1[p]) \\
(\mathrm{s}_1[p], \forall q(q<p \Rightarrow S[p,q])\ \rightarrow\ \forall q : S[p,q]{:=}\mathrm{false},\ \mathrm{s}_2[p]) \\
(\mathrm{s}_2[p],\ \mathrm{flag}[p]{:=}\mathrm{true},\ \mathrm{s}_3[p]) \\
(\mathrm{s}_3[p], \neg S[p,q]\ \wedge\ q<p\ \wedge\ \mathrm{flag}[q]\ \rightarrow\ \forall q : S[p,q]{:=}\mathrm{false},\ \mathrm{s}_0[p]) \\
(\mathrm{s}_3[p], \neg S[p,q]\ \wedge\ q<p\ \wedge\ \neg\mathrm{flag}[q]\ \rightarrow\ S[p,q]{:=}\mathrm{true},\ \mathrm{s}_3[p]) \\
(\mathrm{s}_3[p], \forall q(q<p \Rightarrow S[p,q])\ \rightarrow\ \forall q : S[p,q]{:=}\mathrm{false},\ \mathrm{s}_4[p]) \\
(\mathrm{s}_4[p], \neg S[p,q]\ \wedge\ p<q\ \wedge\ \mathrm{flag}[q]\ \rightarrow\ \forall q : S[p,q]{:=}\mathrm{false},\ \mathrm{s}_4[p]) \\
(\mathrm{s}_4[p], \neg S[p,q]\ \wedge\ p<q\ \wedge\ \neg\mathrm{flag}[q]\ \rightarrow\ S[p,q]{:=}\mathrm{true},\ \mathrm{s}_4[p]) \\
(\mathrm{s}_4[p], \forall q(p<q \Rightarrow S[p,q])\ \rightarrow\ \forall q : S[p,q]{:=}\mathrm{false},\ \mathrm{s}_5[p]) \\
(\mathrm{s}_5[p],\ \mathrm{flag}[p]{:=}\mathrm{false},\ \mathrm{s}_0[p])
\end{array}
$$

Mutual exclusion is obtained using two waiting rooms (s_3 and s_4). The first one ensures that when a process p has reached s_4, any other process q with $q < p$ and flag$[q]$ = true (trying to get access to the critical section, or in the critical section) has gone through the transition $s_2 \rightarrow s_3$ *after* p. The second waiting room guarantees that this process q (with $q < p$) will be blocked in s_4 at least until p resets flag[p]. Only the highest process (the one with the highest identifier) will thus get access to the critical section.[6]

The algorithm uses one single-writer shared register per process: flag[p] is set by process p when it wants to access the critical section. Each process p also uses a local array variable $S[p]$. This variable is used in three loops (s_1, s_3, s_4). In the loops for process p the value of the flag[q] variable of the other processes q is checked (processes q such that $q < p$ or $q > p$). $S[p]$ is used to keep track of processes already checked and those which still have to be checked. The algorithm makes also extensive use of a total order relation between processes.

Formula $H =_{\text{def}} \forall p\ H_1(p) \wedge \forall p \forall q\, [H_2(p,q) \wedge H_3(p,q)]$, with

$$
\begin{aligned}
H_1(p) &=_{\text{def}} \neg\text{flag}[p] \Rightarrow (\text{s}_0[p] \vee \text{s}_1[p] \vee \text{s}_2[p]) \\
H_2(p,q) &=_{\text{def}} \text{s}_2[p] \Rightarrow \neg S[p,q] \\
H_3(p,q) &=_{\text{def}} [q < p\ \wedge\ \text{flag}[q]\ \wedge\ (\text{s}_5[p] \vee \text{s}_4[p] \vee (\text{s}_3[p] \wedge S[p,q]))\,] \\
&\qquad \Rightarrow [\neg\text{s}_5[q] \wedge \neg(\text{s}_4[q] \wedge S[q,p])]
\end{aligned}
$$

is an invariant which entails[7] the mutual exclusion property:

$$\forall p \forall q[p \neq q \Rightarrow (\neg s_5[p] \vee \neg s_5[q])].$$

Every condition is verified for Theorem 9 to be used. Indeed:

- no function (at all) is used;
- every guard is in hypothesis form. In fact, every guard is at most once quantified;
- the invariant is a conjunction of formulas which are in both hypothesis and conclusion form, as they are universally quantified;
- the only interpreted predicates are equality and order; objects compared belong to a finite, but parameterized, domain: the set of processes.

From Theorem 9, if H is an invariant of this algorithm for $n_0 = 4$ *processes*, then H will be an invariant of this algorithm *for any number of processes.*

Let's see how this work for a given verification condition: if H is an invariant, it is preserved by every transition, and in particular, by transition $\sigma_{1\to 2}$

[6]The access to critical section will be easier for processes with high identifiers. This algorithm does not guarantee high-level-fairness.

[7]together with the rule which expresses the fact that each process is in one and only one state at a time.

from s_1 to s_2. The Hoare triple $\{H\}\,\sigma_{1\to 2}\,\{H\}$ must be provable, and so must be the three Hoare triples $\{H\}\,\sigma_{1\to 2}\,\{\forall p\ H_1(p)\}$, $\{H\}\,\sigma_{1\to 2}\,\{\forall p \forall q\ H_2(p,q)\}$ and $\{H\}\,\sigma_{1\to 2}\,\{\forall p \forall q\ H_3(p,q)\}$. In particular, from $\{H\}\,\sigma_{1\to 2}\,\{\forall p \forall q H_2(p,q)\}$ comes the verification condition

$$\varphi =_{\text{def}} (h_1 \wedge h_2 \wedge h_3 \wedge g_1 \wedge g_2 \wedge l_1 \wedge l_2 \wedge l_3 \wedge l_4 \wedge l_5) \Rightarrow C$$

with

- $h_1 =_{\text{def}} \forall p\ H_1(p)$
- $h_2 =_{\text{def}} \forall p \forall q\ H_2(p,q)$
- $h_3 =_{\text{def}} \forall p \forall q\ H_3(p,q)$
- $g_1 =_{\text{def}} s_1[p]$
- $g_2 =_{\text{def}} \forall q[q < p \Rightarrow S[p,q]]$
- $l_1 =_{\text{def}} \forall p[s_0[p] \Rightarrow \neg(s_1[p] \vee s_2[p] \vee s_3[p] \vee s_4[p] \vee s_5[p])]$
- $l_2 =_{\text{def}} \forall p[s_1[p] \Rightarrow \neg(s_2[p] \vee s_3[p] \vee s_4[p] \vee s_5[p])]$
- $l_3 =_{\text{def}} \forall p[s_2[p] \Rightarrow \neg(s_3[p] \vee s_4[p] \vee s_5[p])]$
- $l_4 =_{\text{def}} \forall p[s_3[p] \Rightarrow \neg(s_4[p] \vee s_5[p])]$
- $l_5 =_{\text{def}} \forall p[s_4[p] \Rightarrow \neg s_5[p]]$
- $C =_{\text{def}} \forall s \forall r[(s \neq p \Rightarrow s_2[s]) \Rightarrow \neg(s \neq p\ \wedge\ S[s,r])]$

Hypotheses $h_{1,2,3}$ come from the invariant, $g_{1,2}$ from the transition guards. Formulas $l_{1,\ldots 5}$ state that each process is in one and only one state. The conclusion C is the result of applying the weakest precondition operator, i.e.,

$$C \equiv \text{wp}\,[\forall q : S[p,q]{:=}\text{false}; \text{s}_1[p] := \text{false}; \text{s}_2[p] := \text{true}; \forall p \forall q\ H_2(p,q)]$$

Every formula from h_1 to l_5 is in hypothesis form, and C is in conclusion form. The Herbrand universe for the negation of this verification condition contains four elements (p, q, and the new constants coming from the Skolemization of C). Every universal quantifier in the hypotheses will then give rise to four instances, for a total of 61 hypotheses.[8]

[8]Each formula h_1, g_2, $l_{1\ldots 5}$ generates four instances, whereas formulas h_2 and h_3 generate 16 instances. The 61st hypothesis is g_1.

5.5 Generalized Railroad Crossing

The Generalized Railroad Crossing benchmark [105, 35] uses predicates and functions from arithmetic. It gives a general idea of what Theorem 8 allows to deal with.

A controller operates on a gate of a railroad crossing protecting N parallel railroad tracks. The gate must be down whenever a train takes the intersection, so that the intersecting road is closed. Each of the N trains can be in three different regions: in the intersection (I), in the section preceding the intersection (P), or anywhere else (not_here). The array variable "trains" records the position of each train: trains[i] can be one of the three values $I, P, \mathrm{not\_here}$. The gate can be in four states: the value of variable "gate" can be $\mathrm{down}, \mathrm{up}, \mathrm{going\_down}$ or $\mathrm{going\_up}$, with obvious meanings. The system should verify the safety property, which expresses the fact that the gate must be down when any of the N trains is passing the intersection:

$$\forall i\,(\mathrm{trains}[i] = I \Rightarrow \mathrm{gate} = \mathrm{down})\,.$$

The gate takes some time to go from the state "up" to "down". This time must not exceed "gateRiseTime". Similarly the time to go from the "down" to the "up" states must not exceed "gateDownTime". Trains getting in P would take a minimum time "minTimeToI" and a maximum time "maxTimeToI" to get to the intersection. It is the controller job to know when to lower the gate, and when to raise it. Initially, the gate is up, and no train is either in the intersection or in the section preceding the intersection.

The system transitions are given on Figure 5.1. The first three transitions model the position changes of the train i. The two following ones express the controller decision to lower or raise the gate. The next two mean that the gate reaches the up or down states. The last one models the time flow.

Only two transition guards are not quantifier-free. But they can easily be put in prenex form with a single quantifier. Functions are used (trains, firstEnter, lastEnter, schedTime, +) but they do not range over the process set. All requirements are thus verified for Theorem 9 to be used, as long as the invariants to be checked also verify the requirements about quantifiers.

Figure 5.2 shows several invariance properties of the system. Together with the safety property, they give an invariant for the system. As the safety property is one conjunct of the invariant, it is trivially entailed by the invariant. In order to validate the invariant, it is necessary to take into account the constraints on constants (Figure 5.3) as well as the progress axioms[9] (Figure 5.4). They are supplementary hypotheses to be put in the

[9] For example, progress axiom P_1 states that the train does not stay indefinitely in section P before going in I.

verification conditions.

In the whole proof, only two properties (or guards) are existentially quantified, properties are at most once quantified, and at most one train takes part in a transition. From Theorem 9, if the invariant (which guarantees that the algorithm is safe) is preserved *for four trains*, the algorithm will be safe *for any number of trains*.

$$
\begin{array}{l}
(\mathrm{trains}[i] = \mathrm{not\_here} \;\longrightarrow\; \mathrm{begin} \\
\qquad\qquad\qquad\qquad\quad \mathrm{trains}[i] := P; \\
\qquad\qquad\qquad\qquad\quad \mathrm{firstEnter}[i] := T + \mathrm{minTimeToI}; \\
\qquad\qquad\qquad\qquad\quad \mathrm{lastEnter}[i] := T + \mathrm{maxTimeToI}; \\
\qquad\qquad\qquad\qquad\quad \mathrm{schedTime}[i] := T + \mathrm{conMinI}; \\
\qquad\qquad\qquad\qquad\quad \mathrm{trainHere}[i] := \mathrm{true} \\
\qquad\qquad\qquad\qquad \mathrm{end})
\end{array}
$$

$$(\mathrm{trains}[i] = P \;\wedge\; T \geq \mathrm{firstEnter}[i] \;\longrightarrow\; \mathrm{trains}[i] := I)$$

$$(\mathrm{trains}[i] = I \;\longrightarrow\; \mathrm{begin\ trains}[i] := \mathrm{not\_here};\; \mathrm{trainHere}[i] := \mathrm{false\ end})$$

$$
\begin{array}{l}
(\;(\mathrm{gate} = \mathrm{up} \;\vee\; \mathrm{gate} = \mathrm{going\_up}) \;\wedge\; \mathrm{gstatus} = \mathrm{up} \\
\;\wedge\, \exists i\, (\mathrm{trainHere}[i] \;\wedge\; \mathrm{schedTime}[i] \leq T + \gamma_{\mathrm{down}} + \beta) \\
\;\longrightarrow\; \mathrm{begin} \\
\qquad\qquad \mathrm{gate} = \mathrm{going\_down}; \\
\qquad\qquad \mathrm{lastDown} := T + \mathrm{gateDownTime}; \\
\qquad\qquad \mathrm{gstatus} := \mathrm{down} \\
\qquad\quad \mathrm{end})
\end{array}
$$

$$
\begin{array}{l}
(\;(\mathrm{gate} = \mathrm{down} \;\vee\; \mathrm{gate} = \mathrm{going\_down}) \;\wedge\; \mathrm{gstatus} = \mathrm{down} \\
\;\wedge\, \forall i\, (\mathrm{trainHere}[i] \Rightarrow \mathrm{schedTime}[i] > T + \gamma_{\mathrm{down}} + \gamma_{\mathrm{up}} + \mathrm{carPassingTime}) \\
\;\longrightarrow\; \mathrm{begin} \\
\qquad\qquad \mathrm{gate} := \mathrm{going\_up}; \\
\qquad\qquad \mathrm{lastUp} := T + \mathrm{gateRiseTime}; \\
\qquad\qquad \mathrm{gstatus} := \mathrm{up} \\
\qquad\quad \mathrm{end}\;)
\end{array}
$$

$$(\mathrm{gate} = \mathrm{going\_up} \;\longrightarrow\; \mathrm{gate} := \mathrm{up})$$

$$(\mathrm{gate} = \mathrm{going\_down} \;\longrightarrow\; \mathrm{gate} := \mathrm{down})$$

$$(T := T + \varepsilon)$$

Figure 5.1: The transitions modelling the General Railroad Crossing system

$$T_1 =_{\text{def}} \forall i\,(T < \text{firstEnter}[i] \Rightarrow \text{trains}[i] \neq I)$$

$$\begin{aligned} T_2 \;\; =_{\text{def}} \;\; & \forall i\,(\text{trains}[i] = P \Rightarrow \\ & \quad (\text{firstEnter}[i] \leq T + \text{minTimeToI} \;\wedge\; T \leq \text{lastEnter}[i] \\ & \quad \wedge \text{lastEnter}[i] - \text{firstEnter}[i] = \text{maxTimeToI} - \text{minTimeToI})) \end{aligned}$$

$$C_1 =_{\text{def}} \text{gstatus} = \text{up} \Rightarrow \forall i\,(\text{trainHere}[i] \Rightarrow T < \text{schedTime}[i] - \gamma_{\text{down}})$$

$$GC_1 =_{\text{def}} \text{gstatus} = \text{down} \equiv (\text{gate} = \text{goingDown} \;\vee\; \text{gate} = \text{down})$$

$$GC_2 =_{\text{def}} \text{gstatus} = \text{down} \Rightarrow \text{lastDown} \leq T + \text{gateDownTime}$$

$$GC_3 =_{\text{def}} \text{gstatus} = \text{up} \Rightarrow \forall i\,(\text{trainHere}[i] \Rightarrow \text{lastDown} < \text{schedTime}[i])$$

$$TC_1 =_{\text{def}} \forall i\,(\text{trainHere}[i] \equiv \text{trains}[i] \neq \text{notHere})$$

$$TC_2 =_{\text{def}} \forall i\,(\text{trainHere}[i] \Rightarrow \text{schedTime}[i] < \text{firstEnter}[i])$$

Figure 5.2: Invariance properties

$$AC_1 := \gamma_{\text{down}} < \text{conMinI}$$

$$ACT_1 := \text{conMinI} < \text{minTimeToI}$$

$$AGC_1 := \text{gateDownTime} < \gamma_{\text{down}}$$

$$AGC_2 := \text{gateRiseTime} < \gamma_{\text{up}}$$

Figure 5.3: Constraints on constants

$$G_1 =_{\mathrm{def}} \mathrm{gate} = \mathrm{goingDown} \Rightarrow T \leq \mathrm{lastDown}$$

$$G_2 =_{\mathrm{def}} \mathrm{gate} = \mathrm{goingUp} \Rightarrow T \leq \mathrm{lastUp}$$

$$P_1 =_{\mathrm{def}} \forall i\,(\mathrm{trains}[i] = P \Rightarrow T \leq \mathrm{lastEnter}[i])$$

$$P_2 =_{\mathrm{def}} \mathrm{gstatus} = \mathrm{up} \Rightarrow \forall i\,(\mathrm{trainHere}[i] \Rightarrow T < \mathrm{schedTime}[i] - \gamma_{\mathrm{down}})$$

$$\begin{aligned} P_3 =_{\mathrm{def}} \mathrm{gstatus} = \mathrm{down} \Rightarrow \exists i\,(&\mathrm{trainHere}[i] \wedge \\ &\mathrm{schedTime}[i] \leq T + \gamma_{\mathrm{up}} + \mathrm{carPassingTime} + \gamma_{\mathrm{down}}) \end{aligned}$$

Figure 5.4: Progress axioms

Chapter 6

Propositional logic and beyond

This chapter starts the discussion about the formula proving process, and investigates the decision problem for Boolean formula satisfiability (SAT for short). Although it is certainly one of the oldest problem of Logic, it has seen some important advances these last years.[1] New satisfiability solvers (SAT-solvers for short) like BerkMin [91], Chaff [133],... have improved by orders of magnitude the size of the formulas which can be decided automatically. They have begun to progressively take importance in fields where Binary Decision Diagrams (BDDs) [43] were traditionally used.

In the following, both methods (BDDs, and SAT-solvers) are briefly reviewed, with their advantages and drawbacks. More specifically our concerns are their integration within a framework to decide more expressive logics, that is quantifier-free first-order logics. In that context, the propositional methods are used to find models of a Boolean abstraction of the quantifier-free formula to prove. This abstraction is refined until either a model for the original formula is found, or the formula is proved unsatisfiable. This separates the proving work in two parts: the propositional method takes over the Boolean part, whereas the prover for first-order atoms only deals with conjunctive sets of atoms. Our attention will be focused on the quality of the information given by each module, which is important for the efficiency of the association.

6.1 Preliminaries

Propositional languages are first-order languages without variables (and thus without quantifiers) and functions, and with only nullary predicates. The

[1]A symptom and a cause for this renewed interest in the Boolean satisfiability problem is an annual contest between solvers.

sorts have no meaning in such restrictions, and a propositional language is perfectly defined by a set of propositions $\mathcal{P}$. A literal is either a proposition or the negation of a proposition. Boolean formulas are quantifier-free.[2] An interpretation in a propositional language $\mathcal{P}$ assigns a Boolean value to every proposition of $\mathcal{P}$; it defines a set M of literals such that $p \in M$ or $\neg p \in M$ (but not both) for every proposition of $\mathcal{P}$, depending of whether or not the interpretation assigns $\top$ or $\bot$ to the proposition. Conversely a set M of literals such that $p \in M$ or $\neg p \in M$ (but not both) for every proposition of $\mathcal{P}$ perfectly defines an interpretation in $\mathcal{P}$. In this chapter, interpretations refer either to functions from $\mathcal{P}$ to $\{\top, \bot\}$ or to the set of literals, depending on the context.

A *partial interpretation* M of a propositional language $\mathcal{P}$ is a set of literals such that $p \notin M$ or $\neg p \notin M$ (or both) for every proposition p of $\mathcal{P}$. Given a formula φ, a partial interpretation M in its language is a *partial model* if every interpretation M' such that $M \subset M'$ is a model of φ. Every interpretation M' *extending* a partial model $M \subset M'$ of a formula φ is a model for φ.

To simplify the presentation, a clause c is a disjunctive set of literals, such that either $p \notin c$, or $\neg p \notin c$. This excludes valid clauses and clauses with literal repetitions. A (partial) interpretation M for a set of clauses C is a (partial) model for C if and only if $c \cap M \neq \emptyset$ for every clause $c \in C$.

6.2 Boolean methods

The methods for deciding satisfiability of Boolean formulas are numerous. Two of those methods are widely and successfully used: Binary Decision Diagrams and SAT-solvers based on the ideas of Davis, Putnam, Logemann and Loveland [55].

6.2.1 BDDs

Binary Decision Diagrams (BDDs) [43] are used as a powerful technique for representing very large propositional logic formulas. BDDs are nested if-then-else formulas represented as directed acyclic graphs. Every BDD has one root and two leaves[3] labelled by "true" and "false". Each node has a high child ("then" case) and a low child ("else" case). A BDD with proposition p labelling its root corresponds to formula "if p then A else B",

[2] The quantifiers of Quantified Boolean Logic are quantifiers over propositions. In some sense, they are second-order quantifiers.

[3] Except when BDDs are reduced to one single leaf, in the special cases of unsatisfiable and valid formulas respectively reduced to the "false" and "true" leaves.

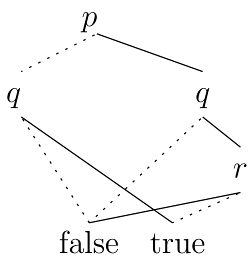


Figure 6.1: Propositional

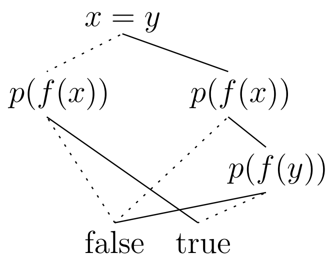


Figure 6.2: Unsat. path

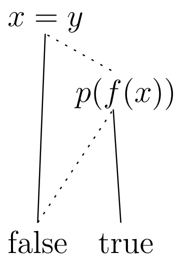


Figure 6.3: Reduced

where A and B are formulas corresponding to the high and low children respectively. For instance the BDD on Figure 6.1 represents the formula "if p then (if q then $\neg r$) else q", or in disjunctive normal form $(p \wedge q \wedge \neg r) \vee (\neg p \wedge q)$. Plain lines correspond to the high child ("then" case), whereas dotted lines correspond to the low child ("else" case).

Generally, BDDs are *ordered* and *reduced* [43]. A total ordering on propositions is given, and the depths of their occurrences in the BDD respect that order. Furthermore, two nodes with the same labels, high and low children should be merged. Finally, each node with a unique high and low child is replaced by this child. Reduced and ordered BDDs provide a canonical representation for Boolean formulas, and hence, a decision procedure for propositional logic. The BDD in Figure 6.1 is in fact a reduced and ordered BDD. The reduced ordered BDD corresponding to a formula requires exponential space in the worst case. The choice of an ordering is critical for the BDD size.

Reduced and ordered BDDs supply a compact representation of *all* models of a formula. Each path to the leaf "true" corresponds to a partial model of the formula. Those partial models might be very much smaller than the full models of the formula. One partial model provides a more valuable information than a full model, as it corresponds to many full models.

6.2.2 SAT-solvers

Most modern SAT-solvers are based on the Davis Putnam Logemann Loveland (DPLL for short) algorithm [55], enhanced by conflict analysis and non-chronological backtracking, as well as preprocessing, good branching heuristics, and efficient data structures (see [191] for references). These solvers operate on a set of clauses. Given a Boolean formula, an equisatisfiable set of clauses can be efficiently obtained using the techniques presented in Section 3.4. This translation to conjunctive normal form is required.

A set Φ of clauses is satisfiable if and only if it has a model. Given a proposition p used in Φ this model makes either p true, or $\neg p$ true. The general idea of the DPLL algorithm is to build a model by incrementally choosing a value for every Boolean variable used in the formula. So it is like searching for a model in a binary tree. Every node is labelled by a set Λ of literals. If Λ contains every Boolean proposition in Φ, then the node is a leaf. Otherwise, its high and low children are labelled by $\Lambda \cup \{p\}$ and $\Lambda \cup \{\neg p\}$ where p is a Boolean variable not occurring in Λ. The depth of this tree is equal to the number of Boolean variables in Φ. Each leaf corresponds to an interpretation, whereas the internal nodes correspond to partial interpretations.

If a partial interpretation makes Φ false (i.e., if it falsifies every literal in a clause in Φ), then there is no need to search for a model in the corresponding subtree. Furthermore if a partial interpretation falsifies every literal in a clause but one (this clause is called *unitary*), then every model in its subtree should make this last literal true. Applying this last remark is commonly called the Boolean constraint propagation (bcp).

Data structures like counter-based approaches, head/tail lists or watched literals are used [118] in order to quickly find falsified and unitary clauses. For instance, in a simple counter-based approach, two counters are associated with each clause. One of those counters is used to keep the number of unassigned literals. It is initially equal to the number of literals in the clause. For a unitary clause, it should be equal to one, and for a false clause, it is zero. The other counter keeps the number of positively assigned literals. Whenever this counter is greater than zero, the clause is true. If the counter of unassigned literals is equal to one, and if the counter of positively assigned literals is zero, then the clause is unitary. So the unassigned literal should be assigned to true. Every time a variable is assigned, the counters of all clauses containing this variable are updated. Modern solvers generally use more sophisticated and efficient methods, like the watched literal technique, to implement Boolean constraint propagation.

Conflict analysis (or learning) and non-chronological backtracking [128, 190] are largely responsible for the success of current SAT-solvers. Whenever a conflict is found (a clause in the conjunctive set is given the value "false"), a new (conflict) clause is added to the set of initial clauses. This conflict clause summarises the reasons for the conflict. A straightforward way to resolve the conflict is to backtrack to the last decision, and switch the value assigned for the variable (let's call it p) at that time. So after backtracking, the value of p will be assigned in such a way that the same conflict does not happen again. In one word, non-chronological backtracking is just one step further: backtrack to the point where the value to be assigned to p is already known from the values of the other assigned variables. If the DPLL method can be viewed as a restricted version of resolution [26], conflict analysis (also called learning) gives back to DPLL some of the power of full resolution [127] by augmenting the set of clauses. Among the different heuristics for conflict analysis, the First Unique Implication Point heuristic [190] (FUIP for short) appears to be a good learning heuristic choice.

Besides data structures and conflict analysis, the last important point for efficiency is the decision heuristics: when no Boolean constraint propagation can be done, and when there remain some unassigned propositions, one proposition has to be chosen and given a Boolean value. Different heuristics exist [126] to make this decision. A delicate point about it concerns the redundant propositions: given a formula, its (p-)definitional normal form contains many introduced propositions. The value of those propositions can be deduced from the values of all the original propositions. It is not clear, neither theoretically nor experimentally whether or not those dependent propositions should be chosen as decision variables [88]. If they can be chosen, the search tree is wider, but conflicts might also be found more quickly. For instance, consider a definitional normal form for formula $\varphi \wedge \neg\varphi$, with p as a label for φ: if it is allowed to take p as a decision proposition, then unsatisfiability may be deduced directly, whereas preventing taking p as a decision proposition might induce building and examining every model of φ.

Finally we should mention here the DPLL methods for non-clausal formulas [89, 144]. However it appears that those methods are equivalent to clausal DPLL methods on definitional normal form, with efficient detection of new unitary clauses, and preventing choosing introduced labels for decision variables.

6.2.3 BDDs versus SAT-solvers

First notice that the purposes of BDDs and SAT-solvers are not the same. BDDs are mainly a canonical form for Boolean formulas. Checking if two BDDs represent the same formula is immediate; as a consequence BDDs are the predilection tool to efficiently do fix-point computation. Used as decision procedures for Boolean formulas, they directly provide a complete set[4] of partial models. SAT-solvers are fast tools to decide satisfiability of a set of clauses. When the set is satisfiable, they provide *one full* model for it. Those facts should be kept in mind when comparing SAT-solvers and BDDs as decision procedures. In [181] it appears BDD and DPLL both have their preferred class of problems. However those results hold for the "clas-

[4]Every model of the formula extends one partial model given by the BDD.

sical" DPLL method without the modern learning techniques. More recent works [33, 182] state that SAT-solvers are better at checking large formulas issued in the fields of symbolic model checking, and formal verification. Our own opinion based on the comparative performances of the tool haRVey used with the BDD and SAT-solver machineries clearly exhibits the fact that recent SAT-solvers like Chaff [133] easily and quickly deal with *propositional* formulas much too large for common BDD packages.

However the problem must be considered as a body. BDDs provide partial models whereas a SAT-solver returns one whole model, i.e., every proposition used in the formula is assigned a truth value. As a consequence the SAT-solvers provide a less valuable information to the modules they are interfaced with. This leads to an increase of work for those modules. This increase is not necessarily compensated by the time spared using SAT-solvers instead of BDDs: the minimalization algorithm in Section 6.4 is one technique to refine the information issued from the SAT-solver.

6.3 Beyond propositional logic

Assume that the formula φ is a Boolean combination of first-order atoms $\{a_1, \dots a_n\}$ and $\mathcal{I}$ is a model of φ. The model $\mathcal{I}$ perfectly defines a model $\mathcal{I}'$ (a Boolean abstraction of $\mathcal{I}$) of the Boolean abstraction φ' of φ, that is formula φ where every atom a_i has been substituted by proposition p_i everywhere: $\mathcal{I}'$ assigns to p_i the same value $\mathcal{I}$ gives to a_i. Conversely, let $\mathcal{T}$ be a set of interpretations (defined for instance by a first-order theory) in the language of φ. If the Boolean abstraction of one of those interpretations $\mathcal{I}$ is a model of φ', then $\mathcal{I}$ is a model of the original formula φ.

This defines a strategy to find a model in $\mathcal{T}$ for φ. First choose a model $\mathcal{I}'$ of the Boolean abstraction φ'. This model defines a set of true literals on propositions $\{p_1, \dots p_n\}$, and so a set of literals L on the corresponding atoms $\{a_1, \dots a_n\}$. If L has a model in $\mathcal{T}$, then φ has a model in $\mathcal{T}$. If all Boolean models of φ' have been checked without success, then formula φ is unsatisfiable. The problem is thus divided in two parts: first a pure Boolean model search, and second, a search of a model in $\mathcal{T}$ for the corresponding *conjunctive set of literals*. The Boolean structure is left to the Boolean satisfiability checker.

An important problem is inherent to this strategy: the number of models might be (and is in practice) very large. For instance if M is a partial model of φ' with only propositions $\{p_1, \dots p_k\}$, then there are 2^{n-k} full models on $\{p_1, \dots p_n\}$ corresponding to this single partial model. Here the advantage of BDDs over SAT-solvers is clear. BDDs provide partial models of the formula they represent, rather than giving full models. Section 6.4 will provide a way to transform full models from SAT-solvers to partial models. But this is not enough to bring the difficulty at an acceptable level. The number of paths to the true leaf in a BDD, i.e., the number of partial models encapsulated in the BDD, may be (and is in practice) exponential with respect to the size of the BDD. In general, the number of minimal partial models remains exponential with respect to the size of the formula.

The solution is to refine the Boolean abstraction rather than eliminating (partial) models one by one. If a set L of first-order literals on $\{a_1, \ldots a_n\}$ corresponding to a (partial or full) model of a Boolean abstraction is unsatisfiable, only a small subset of L is responsible for unsatisfiability. The Boolean abstraction can be refined by adding a clause which guarantees that this small subset will not correspond anymore to a Boolean model of the abstraction.

In summary, two abilities are important for using a Boolean method with richer logics. First, it is absolutely essential for the method to incrementally accept new "facts" about the literals in the abstraction: it is necessary to introduce new clauses conjunctively to the original formula, without requiring a whole new computation to check for satisfiability of the abstraction, and for possibly providing another model. Second, whenever possible, the methods should provide partial models rather than full models. Partial models contain less literals than full models, so the search for an interpretation in $\mathcal{T}$ satisfying all literals is made easier. Also, refining the Boolean abstraction by rejecting one of its partial model suppress many full models in one shot. From this point of view, an optimal partial model is *minimal*. Indeed, if L and L' are sets of first-order literals corresponding to partial models of the Boolean abstraction φ' of an unsatisfiable formula φ, then both L and L' should be unsatisfiable. If L' is a proper subset of L, then any refining clause used to eliminate the propositional models of φ' corresponding to L' will also eliminate the propositional models corresponding to L. It can be argued that optimal partial models should not only be minimal, but minimum, i.e., they should contain the minimum number of literals. However there is no reason for a minimum model to be better than a minimal one. In particular, if two minimal models have no literal in common (let's say $\{a, b\}$ and $\{c\}$), both of those partial models must be considered, since adding a refining clause (for instance $\neg c$) for one of those partial models does not remove the other partial model.

The required ability for provers for sets of first-order literals is most importantly to provide a small unsatisfiable subset of the input whenever it is unsatisfiable. The optimal solution is a *minimal* unsatisfiable subset: every literal is necessary to keep it unsatisfiable. Assume L is an unsatisfiable set of literals corresponding to a model of the Boolean abstraction of a formula. If L' is a proper unsatisfiable subset of L, then the refining clause corresponding to L' will also eliminate the propositional model corresponding to L. As previously, there is no reason to prefer the minimum unsatisfiable subset over other minimal unsatisfiable subsets. For instance, the formula

$$(a \wedge b \wedge c \wedge d) \vee (a \wedge b \wedge e)$$

has the set $\{a, b, c, d\}$ for minimal model. Assume both sets corresponding to $\{a, b\}$ and $\{c\}$ are unsatisfiable. Refining the abstraction by the clause $\neg a \vee \neg b$ makes the formula unsatisfiable, whereas adding the clause $\neg c$ corresponding to the *minimum* model does not.

Two (partial) models successively enumerated by the Boolean method have often many literals in common. The prover for the set of literals should also be incremental and backtrackable: i.e., after having found unsatisfiability of a list M of literals it should be able to efficiently decide unsatisfiability of a list M' of literals which differs from M by only few literals in tail, without redoing the whole computation.

6.3.1 BDDs beyond propositional logic

In [79] we presented a BDD-based decision procedure for combinations of theories relying on the remarks given at the beginning of this section. Notice that BDDs, when used with FOL atoms, are not a canonical representation for formulas anymore. For instance, the BDD on Figure 6.1 is an abstraction of the BDD with FOL atoms of Figure 6.2. The set of literals corresponding to the rightmost path to true is unsatisfiable. The constraint $\neg (x = y \wedge p(f(x)) \wedge \neg p(f(y)))$ can be added to the BDD, to reduce it to the one in Figure 6.3. The elimination of all unsatisfiable paths to the true leaf guarantees that the representation is normal: a formula is unsatisfiable if and only if it is reduced to the false leaf. However it is not a canonical representation anymore: for instance the BDDs representing the formulas $x = y \wedge p(x)$ and $x = y \wedge p(y)$ are different although they are logically equivalent.

There were several other attempts to extend the expressive power of BDDs towards quantifier-free first-order logic with equality. In [44, 45, 47], methods are given to efficiently translate logic formulas containing equalities and uninterpreted predicates and functions to propositional formulas represented by BDDs. In [90] uninterpreted functions and equalities are allowed to be used in BDDs at the cost of losing canonicity, by encoding them into Boolean variables. In [103] this method is refined by having a better integration of the decision diagrams with this new type of variables. In return the ability to deal with uninterpreted functions is lost. It also implies modifications of BDD handling functions. In a similar way, Difference Decision Diagrams are introduced in [131]. They allow to efficiently represent and manipulate a Boolean logic over inequalities of the form $x - y \leq c$. Some work on BDD based decision procedures for quantified first-order logic (without equality) has also been done [94, 92, 150].

6.3.2 SAT-solvers beyond propositional logic

The classical DPLL algorithm is the basis for many provers for more expressive logics (e.g. STeP [36], Math-SAT [15],...). But it is only recently that provers for more expressive logics fully benefit from the strength of state-of-the-art SAT-solvers. The incorporation of Chaff into CVC [171] (the successor of SVC [22]) resulted in a substantial gain of performance [23]. Recently ICS [73] also incorporated a SAT-solver. SAT provers are also used to solve the reduction to propositional logic of problems from more expressive logics [46, 170, 169].

In Section 6.2.2, we saw that the DPLL algorithm decides the satisfiability problem of Boolean formulas by searching for a model in a tree. Each node of the search tree corresponds to a partial interpretation which makes true every element of a set Λ of literals. In the propositional case, Λ should not contain a pair of complementary literals, since no interpretation makes complementary literals true. When atoms are not propositions but first-order atoms, refining the Boolean abstraction ensures that Λ will eventually be satisfiable, or the refined abstraction will be found unsatisfiable. Whenever a full model is found, the decision procedure for conjunctions of first-order atoms either refines the abstraction or agrees with the model, as explained in the general scheme at the beginning of the section. This is known as *lazy notification* [23]: the decision procedure on top of the Boolean method is called only after a full model for the abstraction is found.

The particularities of DPLL algorithms make also possible *eager notification* [23, 76]: the decision procedure on top of the Boolean method is notified regularly of the building of Λ. In other words, a full model does not have to be generated for the Boolean method to be informed that the set Λ is unsatisfiable. Searching in the tree involves two sort of moves: going down corresponding to adding a literal in Λ, or going up corresponding to removing the last entered literal. The decision procedure for literals should efficiently implement those two actions: adding and removing a literal to a satisfiable set in a LIFO style, i.e., the procedure should be incremental and backtrackable.

Boolean constraint propagation can also be enhanced by using the de-

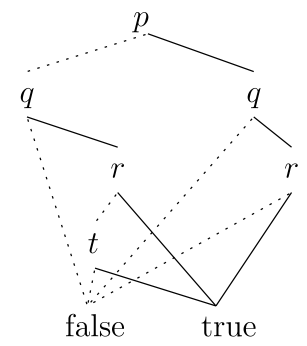


Figure 6.4: BDD with non minimal branch

cision procedure for conjunction of literals. Indeed, constraint propagation should also occur if Λ is such that $\Lambda \models l$ [173], where l is a literal used in the formula which is unassigned in Λ. There is a practical difficulty for doing this: for the decision procedure to be able to provide such an information, it should be aware of every atom in the set of clauses. This would dramatically increase the time taken at each variable assignment. However, some heuristic implementation of this idea has been implemented with promising results [86].

6.4 Model minimalization

Section 6.3 sketched the importance of providing partial models rather than full models to the prover for first-order literals. The optimal partial model is *minimal*, i.e., every literal in the model is necessary for the partial interpretation to remain a partial model of the formula.

In general, paths to the "true" leaf in BDDs do not correspond to minimal models. Consider as an example the BDD in Figure 6.4. It represents the formula $(q \wedge r) \vee (\neg p \wedge q \wedge \neg r \wedge t)$, where the proposition order is alphabetical. Two paths correspond to the minimal model $\{q, r\}$. However, given a formula φ and a minimal model M for φ, there is always an order on propositions such that there is a path in the BDD from the root to the "true" leaf labelled with all and only those literals in M. Indeed, any order which puts all propositions in M before the others is suitable.

Given a formula φ and a model M, computing a minimal model which is a subset of M is a co-NP-hard problem. Indeed, it has a larger complexity than the validity problem: if φ is valid, then its only minimal partial model is the empty set.

Finding a model for a set of clauses is NP-complete. The problem we are interested in here is different: given a set of clauses C and a model M, the problem is to find a minimal partial model of C. We now state that, for conjunctive normal forms, this problem is linear with respect to the size of the set of clauses, as opposed to the corresponding problem for general Boolean combinations.

A partial model can be built by iteratively removing the unnecessary literals in the model M. Let $M'_0 = \emptyset$ and $M_0 = M$. Then M_{i+1} and M'_{i+1} are defined from M_i and M'_i as follows: $M_{i+1} = M_i \setminus \{l_i\}$ where l_i is a literal chosen[5] in M_i. If $(M_i \cup M'_i) \setminus \{l_i\}$ is a partial model for C, then $M'_{i+1} = M'_i$. Otherwise $M'_{i+1} = M'_i \cup \{l_i\}$. Then $M_{|M|} = \emptyset$, $M'_{|M|}$ is a minimal partial model for C.

Theorem 10. *Given a satisfiable set of clauses C and a model M, $M'_{|M|}$ (computed as above) is a minimal partial model for C*

Proof. Assume $M'_{|M|} \setminus \{l_i\}$ is a partial model, included in $M'_{|M|}$. For any j, $\left(M_{j+1} \cup M'_{j+1}\right) \subset \left(M_j \cup M'_j\right)$. This also means that $M'_{|M|} \setminus \{l_i\} \subset (M_i \cup M'_i) \setminus \{l_i\}$. So $(M_i \cup M'_i) \setminus \{l_i\}$ should also be a partial model. In that case $l_i \notin \left(M_{i+1} \cup M'_{i+1}\right)$, and hence $l_i \notin M'_{|M|}$. □

Notice that if N is a partial model, then $N \setminus \{l\}$ is also a partial model if and only if $c \cap N \setminus \{l\} \neq \emptyset$ for every clause $c \in C$.

Usually, the set of clauses is indexed by clause, and literals in each clause can be enumerated in constant time. A set of clause can also be viewed as a subset of $L \times C$ where L is the set of literals (i.e., $L = V \times \{\top, \bot\}$ where V is the set of variables) and C is a set of identifiers for clauses. The set of clauses can thus be indexed by literal instead of by clause. Each literal is associated to the list of clauses in which it appears. Building this index can be done in linear time with respect to the size of the set of clauses (more precisely, with respect to the size of the subset of $L \times C$ representing the set of clauses). This is a common indexing in the field of SAT-solvers. This indexing allows to compute the partial model $M'_{|M|}$ in linear time with respect to the size of the set of clauses.

Theorem 11. *A minimal partial model $M'_{|M|}$ for a set of clauses can be computed in linear time with respect to the size of the set of clauses.*

Proof. Suppose there is a counter associated with every clause and that this counter can be read and modified in constant time. The counter for each clause c will be maintained such that at every step i, its value will be $n_{c,i} =$

[5] The successive choices of literals will determine which partial model will be built.

$|c \cap (M_i \cup M'_i)|$. The value of $n_{c,0}$ is $|c \cap M|$. Computing $n_{c,0}$ for all clauses is done in linear time with respect to the set of clauses. Now $(M_i \cup M'_i) \setminus \{l_i\}$ is a partial model if and only if $n_{c,i} > 1$ for every clause containing l_i. If $(M_i \cup M'_i) \setminus \{l_i\}$ is a partial model, then for every clause c containing l_i, $n_{c,i+1} = n_{c,i} - 1$. Computing M_{i+1}, M'_{i+1}, and $n_{c,i+1}$ for each clause c just involves examining twice all clauses containing l_i.

In brief, computing $M'_{|M|}$ will involve examining twice all literals, and every clause in the associated list. □

Theorem 12. *Let M be a minimal partial model for the (p-)definitional normal form of a propositional formula φ. Let M' be the partial interpretation M restricted to propositions in φ, that is M' does not assign values to introduced labels in the (p-)definitional form. Then M' is a partial model for formula φ.*

Proof. Assume M' is not a partial model for φ; there is a full extension M_1 (for φ) of M' which makes φ false. Notice that $M_2 = M_1 \cup (M \setminus M')$ is an extension of M_1 for the set of clauses, because $M \setminus M'$ contains only literals on labels of the (p-)definitional form whereas M_1 does not contain any of such literals. The set M_2 thus does not contain two complementary literals. Also $M' \subset M_1$, and so $M \subset M_1 \cup (M \setminus M')$. The partial interpretation M_2 is an extension of M, and is thus a partial model for the (p-)definitional form. Any model for the (p-)definitional form is a model for φ. M_2 makes true the formula φ, but a restriction of M_2 to propositions in φ makes it false. □

In practice, to minimalize the number of propositions in the partial model for formula φ in the preceding theorem, one should prefer a minimal partial model for the (p-)definitional form with the less literals corresponding to literals in φ. In other words, in the procedure of computing the minimal partial model for the set of clauses (see proof of Theorem 11), the literals in φ should correspond to literals l_i with the smallest i.

Notice that a minimal partial model for the (p-)definitional normal form of a propositional formula φ does not, in general, correspond to a *minimal* partial model for φ. Consider the formula $\varphi = (a \wedge b) \vee c$, and its (ameliorated) p-definitional form:

$$\{L \vee c, \neg L \vee a, \neg L \vee b\}.$$

The set $\{a, b, c\}$ is a minimal partial model for the p-definitional form, and a model for φ, but not a minimal partial model for φ.

Also, a minimal partial model M for φ does not always extend to a minimal partial model M' for the (p-)definitional form, such that M is M'

restricted to propositions in φ. Consider again the formula $\varphi = (a \wedge b) \vee c$ and its definitional form

$$\{L \vee c, L \vee \neg a \vee \neg b, \neg L \vee a, \neg L \vee b\}$$

$\{c\}$ is a minimal model for φ, but neither $\{c, L\}$ nor $\{c, \neg L\}$ is a minimal model for the set of clauses.

Those two examples state that minimal models for the formula and for the (p-)definitional normal form are not strongly related. To compute a minimal model for the original formula, other techniques have to be used [157]. However, the complexity of those techniques is high. If the decision procedure for sets of first-order literals is efficient, then it is in practice better to accept the trade-off of an imperfect partial model rather than computing a minimal model at any (high) price.

6.5 Experimental results

We conclude this discussion on propositional methods beyond propositional logic with some experimental results. The haRVey prover has been run on various benchmarks, first using BDDs as the propositional method, and then using a SAT solver (zChaff). Using the SAT solver, experiments have been done with and without the minimalization method presented in the previous section. The columns (each containing 3 subcolumns) in Figure 6.5 refer respectively to the total validation times, to the times taken by the decision procedure on top of the propositional method, to the numbers of calls to this external decision procedure, to the total numbers of literals given to the decision procedure, and to the total numbers of literals in refining clauses obtained from the decision procedure. The first subcolumns are related to haRVey with BDD, the second ones to haRVey with SAT without the minimalization method from Section 6.4, and the last ones to haRVey with SAT with minimalization. Times are given in seconds, and they refer to cpu time on a Pentium 4 at 2.60 GHz, with 1 Gbyte of memory. The execution time limit for validating one verification condition was set to 50 seconds.

Clearly, using SAT without minimalization leads to deceiving results, compared to what is obtained using BDDs: execution times are longer, but mostly *the prover fails to validate some proof obligations* within the time limit (one “delete” formula out of three, and the “succ-1-0” verification condition). The increase of run time is mainly due to the increase of time taken by the external prover, i.e. by the prover for sets of literals. The external prover is called more often and the amount of literals given to it is much larger with SAT than with BDDs. The SAT solver provides full models, and the external prover gets large sets of literals. As a result, the external prover time increases, and the refining clauses are not always useful. As a last consequence, the number of calls to the external prover is larger, and in two cases, the appropriate refining clauses are not generated in the time limit.

Using minimalization partially solves those problems: the number of literals given to the external prover decreases, the refining clauses get more precise, and the number of calls to the external prover also decreases. Also the ratio propositional time over external time decreases dramatically. But mostly, using this method, the prover is able to handle many verification conditions that were not successfully handled by the BDD version, or by the naïve SAT version.

Benchmark	Time (s)			External time (s)			External calls			Literals in			Literals out		
Burns-eq (40)	38.5	7.59	5.62	1.11	3.17	2.33	95	264	199	4035	17191	10542	224	718	523
delete (3)	9.39	(46.38)	3.16	4.85	(4.86)	1.82	54	81	61	1177	2577	1394	214	613	487
domino6x4x-0	1.87	2.31	1.59	0.91	1.22	0.8	41	49	38	4327	5537	3424	123	147	114
domino6x6x-0	7.03	6.94	4.57	3.13	3.82	2.47	78	94	68	14357	18236	10609	234	282	204
find (6)	23.26	15.16	13.55	13.48	9.38	7.98	229	285	321	8049	10287	7259	902	2299	2432
ready (4)	0.19	0.36	0.35	0.09	0.13	0.15	5	11	11	66	182	131	4	28	29
remove-2-0	0.45	0.68	0.57	0.2	0.32	0.27	11	15	14	482	1125	727	11	33	20
remove-all-3-0	0.18	0.28	0.21	0.07	0.13	0.08	6	8	6	144	328	174	6	24	6
rotate-2-0	1.1	0.73	0.56	0.23	0.39	0.28	14	17	15	490	1156	597	14	31	19
select-store (4)	3.15	5.37	3.13	1.84	3.11	1.72	62	87	85	1219	2447	1599	218	591	584
set (5)	0.16	0.24	0.24	0.06	0.08	0.07	5	7	6	10	23	16	7	15	11
succ-1-0	0.08	(-)	0.51	0.05	(-)	0.33	3	39	10	51	2964	484	2	193	42
succ-2-0	0.15	2.57	1.15	0.07	1.27	0.57	6	74	39	108	5624	1888	6	355	169
TransP (6)	1.32	3.31	2.84	0.64	1.63	1.37	58	136	124	928	3516	2245	139	460	419
TOTAL	86.83	(91.92)	38.05	26.73	(29.51)	20.24	667	1167	997	35443	71193	41089	2104	5789	5059

Figure 6.5: haRVey prover's performances, with BDDs, with SAT, with SAT with minimalization (Section 6.4).

Chapter 7

Congruence closure

Basically *congruence closure* algorithms decide satisfiability of a set of ground first-order logic literals containing only equalities with uninterpreted predicates and functions. More precisely congruence closure constructs term equivalence classes corresponding to a set of equalities. Two terms belong to the same class if and only if they are equal according to the set of equalities; that is the equality of those two terms is a logical consequence of the conjunction of all equalities in the set. A set of literals may be unsatisfiable for two reasons. First if it contains a pair of complementary literals of the same predicate such that the corresponding arguments are in the same congruence class as in $\{a = b, p(a), \neg p(b)\}$. Second if there is an inequality with both terms in the same class: for instance the set $\{a = b, f(a) \neq f(b)\}$ is unsatisfiable. To simplify the presentation, this contradiction finding will be temporarily ignored to concentrate on the construction of the congruence closure.

Many implementations of congruence closure exist, but three are commonly referred to: Nelson-Oppen [136], Downey, Sethi and Tarjan (DST for short) [63] and Shostak [166, 167, 168]. All three algorithms have the same goal, i.e., regrouping terms in congruence classes according to a set of equalities. They differ by their strategies, described by a unified abstract model in [19, 20], and by their data structures.

Congruence closure algorithms may be used[1] in association with propositional decision procedures to lift propositional methods beyond propositional logics (see Chapter 6). For this task procedures are used very intensively. Their efficiency is thus crucial. All three algorithms are polynomial; Shostak and Nelson-Oppen are quadratic in the worst case [136, 111], but the DST

[1]Actually, several other decision procedures are usually combined (see Chapter 8) with the decision procedure for uninterpreted predicates and functions. The combination is then used in association with propositional methods.

algorithm has been proven to be $\mathcal{O}(n\log(n))$, as long as enter and query operations on a hash table are assumed to be constant in time. Without such an assumption, other data structures allow DST to reach a worst-case time complexity of $\mathcal{O}\left(n\left[\log(n)\right]^2\right)$ [63, 137].

Nelson-Oppen is the simplest of all algorithms. This simplicity is also its principal advantage over both others. In our opinion, Shostak's algorithm success — it is probably the most frequently used one — is due to three facts. First it is quicker than Nelson-Oppen (but DST is in a sense even quicker). Second [54], it accepts new terms and equalities "on-the-fly". Congruence closure algorithms regroup terms in congruence classes; for Shostak's algorithm, the set of terms is not determined *a priori*, new terms can be introduced not only before computing the congruence classes, but during the process. Furthermore equalities are handled one by one, and do not have to all be given to the algorithm before the computation. Equalities may also be added during the process. These facilities are essential in an eager evaluation context (see Chapter 6, page 87). Third reason for Shostak's algorithm success — and undoubtedly the most important one — in its original presentation Shostak's algorithm is not only a congruence closure algorithm, but also a combination scheme: it allows to decide conjunctions of literals containing equalities, uninterpreted predicates and functions, *and* interpreted functions[2] from one theory verifying some requirements. Combination of decision procedures on disjoint languages will be the topic of the next chapter; but let us mention already that the importance of this third argument is diminishing (see page 120).

We already mentioned the importance of algorithm efficiency and of the ability to handle terms and equalities "on-the-fly" (that is being *incremental*). Usually congruence closure is used as a mean to distinguish equal terms according to a set of equalities, but is also used as part of decision procedure for sets of literals without interpreted predicates and functions, i.e., inequalities and uninterpreted predicates should also be handled in some way. Finally, a decision procedure for sets of literals should not only decide if a set of literals is satisfiable or not, but also be proof producing, that is, efficiently compute the hopefully small subset of literals necessary to prove unsatisfiability. This last capability is of crucial importance, as much as its efficiency when used in combination with propositional solvers (see Chapter 6).

The new algorithm presented in this chapter implements all those capabilities. It inherits the good complexity of Downey, Sethi and Tarjan. It is incremental (or "on-the-fly"). It does not only compute congruence closure,

[2]Interpreted predicates are not immediately handled in Shostak's algorithm (see for example [24] for more information).

but is a full decision procedure for sets of literals with uninterpreted predicates and functions.[3] This decision procedure is proof-producing, thanks to the method we presented in [79].

The final algorithm will be obtained by successive transformations starting with a very simple abstract model of Nelson-Oppen algorithm.

7.1 A simple abstract model

First let us state the problem precisely. Given a finite set of terms $\mathcal{T}$ and a finite set of equalities E, we want to compute the congruence closure of $\mathcal{T}$ corresponding to E. That is, we want to compute a partition $\mathcal{C}$ of $\mathcal{T}$ such that two terms $u, v \in \mathcal{T}$ are in the same class of $\mathcal{C}$ if and only if $E \models u = v$. It is assumed that every term used in E or $\mathcal{T}$ belongs to $\mathcal{T}$. In other words the finite set of terms $\mathcal{T}$ contains every term (and subterm) of E and $\mathcal{T}$.

A simple congruence closure algorithm can be described as the non-deterministic application of the simple rules[4] in Figure 7.1. The initial state is $\langle \mathcal{C}_\mathcal{T}, E\rangle$, where the set $\mathcal{C}_\mathcal{T}$ is a partition of $\mathcal{T}$, such that every term is in its own class. Rules are applied to construct the congruence closure, which is always terminates. The final state is $\langle \mathcal{C}, \emptyset\rangle$ and $\mathcal{C}$ is the congruence closure of E. That is two (sub)terms u and v from T are in the same class of $\mathcal{C}$ if and only if $E \models u = v$.

Property 1 (termination). *Every derivation starting from state $\langle \mathcal{C}_\mathcal{T}, E\rangle$ using the rules in Figure 7.1 is terminating. The final state $\langle \mathcal{C}, E'\rangle$ is such that $E' = \emptyset$.*

Proof. The second part of the theorem is straightforward: if E' is not empty, then either the Deletion rule or the Merge rule can be applied. Indeed every term used in E' belongs to one class of $\mathcal{C}$.

The partition $\mathcal{C}_\mathcal{T}$ in the initial state is a partition of the finite set of terms $\mathcal{T}$. Every application of the Merge rule decreases the partition class number. There can be at most $|\mathcal{T}|$ applications of the Merge rule.

The number of applications of the Deduction rule is strictly smaller than $|\mathcal{T}|^2$. Indeed, there is at most $|\mathcal{T}|^2$ different equalities to generate from $|\mathcal{T}|$ terms. Every generated equality is either removed using the Deletion rule (when both terms are in the same class) or the Merge rule altogether merging classes containing the two terms of the equality. Conditions for applying the

[3] Combination with decision procedures for interpreted predicates and functions will be introduced in the next chapter.

[4] Rules Deletion, Merge, and Deduction respectively correspond in [20] to Deletion, Orientation, and NODeduction.

Deletion:

$$\frac{\langle \{A\} \cup \mathcal{C}, E \cup \{x = y\} \rangle}{\langle \{A\} \cup \mathcal{C}, E \rangle}$$

if $\{x, y\} \subseteq A$.

Merge:

$$\frac{\langle \{A, B\} \cup \mathcal{C}, E \cup \{x = y\} \rangle}{\langle \{A \cup B\} \cup \mathcal{C}, E \rangle}$$

if $x \in A$ and $y \in B$, with $A \neq B$.

Deduction:

$$\frac{\langle \{A, B\} \cup \mathcal{C}, E \rangle}{\langle \{A, B\} \cup \mathcal{C}, E \cup \{f(x_1, \ldots x_p) = f(y_1, \ldots y_p)\} \rangle}$$

if $f(x_1, \ldots x_p) \in A$, $f(y_1, \ldots y_p) \in B$, $A \neq B$, and for each i, there exists a set $D_i \in \{A, B\} \cup \mathcal{C}$ such that $\{x_i, y_i\} \subseteq D_i$. It is also assumed that equality $f(x_1, \ldots x_p) = f(y_1, \ldots y_p)$ does not belong to E.

Figure 7.1: Simple rules for congruence closure

Deduction rule imply that the same equality will never be generated twice by the Deduction rule.

The Deduction rule is the only equality producing rule. As the number of its applications is bounded by $|\mathcal{T}|^2$, there can be at most $|\mathcal{T}|^2 + |E|$ applications of the Deletion rule. □

Remark: It can already be noticed that the problem is polynomial. Testing if a rule can be applied is a polynomial problem, and the number of rule applications is polynomial.

Theorem 13 (correctness). *Let $\langle \mathcal{C}, \emptyset \rangle$ be the final state of a derivation starting from state $\langle \mathcal{C}_\mathcal{T}, E \rangle$. For all terms u, v in $\mathcal{T}$, $E \models u = v$ if and only if u and v belong to the same class of $\mathcal{C}$.*

Proof. Given any partition $\mathcal{C}$ of the set of terms $\mathcal{T}$, let relation $\leftrightarrow_\mathcal{C}$ on $\mathcal{T}$ be such that $u \leftrightarrow_\mathcal{C} v$ if and only if u and v belongs to the same class of $\mathcal{C}$. Given any finite set E of equalities on $\mathcal{T}$ let $\leftrightarrow_E$ be the relation on $\mathcal{T}$ such that $u \leftrightarrow_E v$ if and only if $E \models u = v$. Let relation $\leftrightarrow_{E,\mathcal{C}}$ on $\mathcal{T}$ be the symmetric and transitive[5] closure of the union of relations $\leftrightarrow_E$ and $\leftrightarrow_\mathcal{C}$.

[5]Relation $\leftrightarrow_{E,\mathcal{C}}$ is trivially reflexive, as relation $\leftrightarrow_E$ is reflexive.

Consider the states $\langle \mathcal{C}, E \rangle$ of any derivation using the Deletion, Merge and Deduction rules. Consider also for each state the set of equalities

$$\begin{aligned} E' \ =_{\text{def}} \ \{ & f(x_1, \ldots x_p) = f(y_1, \ldots y_p) \mid \\ & A, B \in \mathcal{C}, A \neq B, f(x_1, \ldots x_p) \in A, f(y_1, \ldots y_p) \in B, \\ & \forall i : \exists D_i \in \mathcal{C} : \{x_i, y_i\} \subseteq D_i \}. \end{aligned}$$

Informally E' is the set of all equalities which may potentially be generated by the Deduction rule at the given state.

Remember that it is assumed that every term used in E or $\mathcal{T}$ belongs to $\mathcal{T}$. In that case, the relation $\leftrightarrow_{E \cup E', \mathcal{C}}$ on $\mathcal{T}$ is kept identical by one application of any of the three rules. The relation $\leftrightarrow_{E \cup E', \mathcal{C}_\mathcal{T}}$ corresponding to the initial state $\langle \mathcal{C}_\mathcal{T}, E \rangle$ is $\leftrightarrow_E$, as $\mathcal{C}_\mathcal{T}$ contains a different class for every term in $\mathcal{T}$. In the final state, E is empty (Theorem 1), and E' too since the Deduction rule cannot be applied anymore. The relation $\leftrightarrow_{\emptyset, \mathcal{C}}$ corresponding to the final state is just relation $\leftrightarrow_\mathcal{C}$. □

Remark: Relation $\leftrightarrow_\mathcal{C}$ is trivially reflexive, symmetric, and transitive. Every subterm of every term of $\mathcal{T}$ belongs to $\mathcal{T}$, and thus belongs to a class of $\mathcal{C}$. If $\langle \mathcal{C}, \emptyset \rangle$ is a terminating state, the Deduction rule cannot be applied. It follows that for each term $f(x_1, \ldots x_p), f(y_1, \ldots y_p) \in \mathcal{T}$, if $x_i \leftrightarrow_\mathcal{C} y_i$ for every i, then $f(x_1, \ldots x_p) \leftrightarrow_\mathcal{C} f(y_1, \ldots y_p)$, that is, the relation $\leftrightarrow_\mathcal{C}$ is congruent on $\mathcal{T}$.

Theorem 13 establishes that the final state is only dependent on the initial state. In other words, there may be several derivations starting from a given state, but they all terminate in the same state. The set of rules of Figure 7.1 is confluent, thus convergent.

Observe that the application of the Deduction rule always involve the later application of the Deletion or Merge rule. Using this fact, the Deduction rule could be replaced by the rule:

$$\frac{\langle \{A, B\} \cup \mathcal{C}, E \rangle}{\langle \{A \cup B\} \cup \mathcal{C}, E \rangle}$$

if $f(x_1, \ldots x_p) \in A$, $f(y_1, \ldots y_p) \in B$, $A \neq B$, and for each i, there exists a set $D_i \in \{A, B\} \cup \mathcal{C}$ such that $\{x_i, y_i\} \subseteq D_i$. However with such a rule the task of identifying classes which will eventually be merged is constantly interrupted by the class merges. This rule removes freedom on task order. Identifying classes to be merged is not easy: forcing immediate Merge after Deduction would make the Deduction task even more difficult and put unnecessary constraints in the design of an efficient algorithm.

Different implementations may differ in rule application order, and in the use of different data structures. In the next section, we will introduce a very simple algorithm based on Deletion, Merge, and Deduction rules. Its strategy is simple. In the initial state every term is in its own class so the Deduction rule cannot be applied. The Deduction rule may be activated only by a Merge rule application. Systematic application of the Deduction rule after each Merge rule application gives the following algorithm.

7.2 A simple concrete algorithm

Rules in previous section can be used to implement a really simple congruence closure algorithm, the Nelson-Oppen algorithm. This translation to pseudo-code will be the basis for all transformations in the rest of the chapter. From this naïve and inefficient concrete version of congruence closure, an efficient version will be derived, with all the desiderata listed in the introduction of this chapter.

The goal is to construct the congruence closure of a set of terms $\mathcal{T}$ corresponding to a set of equalities E. We assume that terms in $\mathcal{T}$ and E are represented by DAGs with maximal sharing: two terms cannot be syntactically equal and have different representations. Using notations from [136], given a node u representing $f\,(x_1, \dots x_p)$, $\lambda\,(u)$ is its topmost symbol f, $\delta\,(u)$ is its arity p,[6] and $u\,[i]$ is the representation of x_i. Unlike in the original presentation we assume that all arities are bounded. Larger verification conditions generally do not mean predicates and functions with a larger arity. The number of arguments of program variables can be assumed to be upper-bounded by a constant.

States of the previous abstract algorithm are pairs $\langle \mathcal{C}, E\rangle$, where $\mathcal{C}$ is a partition of terms, and E is a set of equalities. Every term belongs to one and only one class of $\mathcal{C}$. Similarly, in this concrete version, each term $u \in \mathcal{T}$ has a congruence class record (CCR for short) $\textsc{Find}(u)$ associated with it. This CCR can be shared: $\textsc{Find}(u) = \textsc{Find}(v)$ means the CCR associated to u and v is the same, and u and v are said to be in the same class. In the abstract version, this corresponds to u and v being in the class of $\mathcal{C}$ in state $\langle \mathcal{C}, E\rangle$. Initially every term is in its own class, and $\textsc{Find}(u) \neq \textsc{Find}(v)$ for every two different terms u and v. Accessing to a CCR (a call to $\textsc{Find}(u)$) and modifying the CCR associated to a term ($\textsc{Find}(u) := x$) are assumed to be constant time operations. This can be implemented by associating a CCR pointer to every DAG node. CCRs contain two fields: TERM and PRED, to store the terms in the class represented by the CCR, and the set of terms having an argument in the class represented by the CCR respectively. Initially the following properties hold:

[6] If u represents a constant, then $\delta\,(u) = 0$.

```
procedure MERGE(a, b);
    var COMBINE : set of TERM × TERM;
    begin
        COMBINE := {(a, b)};
        repeat
            (u, v) := element of COMBINE;
            COMBINE := COMBINE \ {(u, v)};
            if FIND(u) ≠ FIND(v) then
                COMBINE := COMBINE ∪ UNION(FIND(u), FIND(v));
        until COMBINE = ∅;
    end
```

Figure 7.2: Procedure MERGE (Nelson-Oppen)

Property 2. *For all terms* $u, v \in \mathcal{T}$, $u \in \mathrm{FIND}(v).\mathrm{TERM}$ *if and only if* $\mathrm{FIND}(u) = \mathrm{FIND}(v)$. *In particular, for each term* $u \in \mathcal{T}$, $u \in \mathrm{FIND}(u).\mathrm{TERM}$.

Property 3. *For all terms* $u, v \in \mathcal{T}$, $u \in \mathrm{FIND}(v).\mathrm{PRED}$ *if and only if there exists* $i \in \{1, \dots \delta(u)\}$ *such that* $\mathrm{FIND}(u[i]) = \mathrm{FIND}(v)$.

It will next be shown that those properties also hold nearly always.

Function SIG (for signature) is defined to be such that $\mathrm{SIG}(u) = \mathrm{SIG}(v)$ if and only if u and v are such that $\lambda(u) = \lambda(v)$, $\delta(u) = \delta(v)$, and for each $i \in \{1, \dots \delta(u)\}$, $\mathrm{FIND}(u[i]) = \mathrm{FIND}(v[i])$. In other words, $\mathrm{SIG}(u) = \mathrm{SIG}(v)$ if and only if u and v represent two terms $f(x_1, \dots x_p)$ and $f(y_1, \dots y_p)$, with x_i and y_i in the same class for each i. Concretely, a call to $\mathrm{SIG}(f(x_1, \dots x_p))$ could return $(f, \mathrm{FIND}(x_1), \dots \mathrm{FIND}(x_p))$. Function SIG is used to detect possible applications of the Deduction rule. Also notice that, assuming that arities are bounded by a constant and that computing $\mathrm{FIND}(t)$ for any term t is done in constant time, then testing if $\mathrm{SIG}(u) = \mathrm{SIG}(v)$ may be done in constant time.

In order to compute the congruence closure of a set of equalities E, function MERGE (Figure 7.2) is repeatedly called with u and v as arguments for each equality $u = v$ in E. Such executions implement derivations using rules in Figure 7.1.

Theorem 14 (correctness). *Let* E *be a set of equalities,* $\mathcal{T}$ *be a set of terms, let* $\mathrm{FIND}(u) \neq \mathrm{FIND}(v)$ *for all distinct terms* u *and* v *in* $\mathcal{T}$*, and let Property 2 and Property 3 hold initially. If* MERGE *is called with every*

function UNION(a, b);
    var DEDUCT : **set of** TERM $\times$ TERM;
    begin
        for each $x \in b$.TERMS **do** FIND$(x) := a$;
        a.TERMS := a.TERMS $\cup$ b.TERMS;
        DEDUCT := $\emptyset$;
        for each $x \in a$.PRED$, y \in b$.PRED **do**
            if FIND$(x) \neq$ FIND$(y) \wedge$ SIG$(x) =$ SIG(y) **then**
                DEDUCT := DEDUCT $\cup \{(x, y)\}$;
        a.PRED := a.PRED $\cup$ b.PRED;
        return DEDUCT;
    end

Figure 7.3: Function UNION (Nelson-Oppen)

equality in E, then $E \models u = v$ with $u, v \in \mathcal{T}$ if and only if u and v belong to the same class.

Proof. For any state of an execution of the Nelson-Oppen algorithm, let

- $E' \subset E$ be the subset of equalities which have not yet been processed by function MERGE;
- $\mathcal{C}$ be the partition of terms in $\mathcal{T}$ such that u and v are in the same class if and only if FIND$(u) =$ FIND(v).

Consider the sequence of abstract states

$$\langle \mathcal{C}, E' \cup \{u = v\} \cup \textsc{combine} \rangle$$

corresponding to every concrete state (that is, Nelson-Oppen algorithm state) of an execution such that the program counter is at line 7 of function MERGE. At line 8, the set COMBINE is augmented by new equalities to process, very much as new equalities are found by the deduction rule. Those abstract states correspond to states of derivation using rules in Figure 7.1. Indeed initially E' is $E \setminus \{u = v\}$, COMBINE is empty, and $\mathcal{C}$ is a partition of $\mathcal{T}$ with every term in its own class.

If the condition at line 7 of function MERGE is false, equality $u = v$ is discarded. This corresponds to an application of the Deletion rule. If the condition is true, function UNION (Figure 7.3) implements the Merge rule, followed by a systematic application of the Deduction rule. New equalities may be added to COMBINE, by the Deduction rule. Notice that the properties 2 and 3 are preserved by function UNION.

The algorithm stops when COMBINE and E' are empty. In the final state, no more Deduction rule can be applied, as it is systematically applied after each merge. The final state of the Nelson-Oppen algorithm corresponds to a final state of the previous abstract model. □

Operations on sets are at the core of this algorithm. Before studying complexity it is necessary to describe how sets and their handling functions are implemented. Actually lists are used instead of sets. There may be multiple occurrences of identical elements in lists, but they do not matter for the complexity results. Circular linked lists are used, so that it is possible to merge two lists, and add, remove, extract the first element, or remove the first element in constant time. However testing for membership in a list is linear with respect to the size of the list.

Theorem 15 (complexity). *The congruence closure of a set of terms $\mathcal{T}$ corresponding to a set E of equalities is computed in a time $\mathcal{O}\left(n^2 + |E|\right)$, where n is the total number of nodes in the representations of terms in $\mathcal{T}$.*

Proof. The total number of calls to function UNION is at most n, just like the total number of applications of the Merge rule. Indeed they both decrease the number of classes in the term partition. The term partition contains at most n classes (one for each term), and the number of classes never increases.

Operations at lines 4, 5, 9, and 10 in function UNION take constant time. Elements in a list can be enumerated in constant time, so that line 3 is at most linear in time with respect to n. The total time taken by executions of lines 3-5, 9-10 is quadratic with respect to n.

The total number of inspected predecessor pairs (x, y) at line 6 of UNION is $\mathcal{O}(n^2)$. Indeed there are no more than n^2 predecessor pairs, and every pair may be investigated only when two of their direct subterms are merged. As we assumed[7] that the direct subterm number is bounded by a constant k, there can be no more than $2k-1$ mergings between those direct subterms. The number of executions at lines 7 and 8 is bounded by $(2k-1)n^2$. Finally the total number of equalities added to DEDUCT, and hence to COMBINE, is bounded by $(2k-1)n^2$. □

[7] This assumption is not necessary to prove that the algorithm is quadratic. But this simplifies the discussion, and it is reasonable.

Remark: Condition FIND(x) $\neq$ FIND(y) at line 7 in function UNION is not required neither for correctness nor for keeping the algorithm quadratic in time. It has been added, or rather left, to keep the symmetry with the abstract model. Furthermore, this test is cheap in practice whereas adding and removing elements to a list are quite expensive in time (even if those operations take constant time). The global effect of this test is to decrease execution time in practice.

7.3 Being efficient

In the preceding section two reasons for inefficiency were pointed out, both in the function UNION (Figure 7.3). First, at line 3 changing the associated CCR for all terms in b.TERMS potentially implies a quadratic number of operations with respect to the size of the input set. Second, at lines 6 to 8, predecessors are inspected by pairs, and there may be a quadratic number of such pairs. Two simple transformations of the previous algorithm will remove those two sources of inefficiency.

First notice that calling UNION(u, v) or UNION(v, u) does not involve exactly the same operations, but has the same effect, that is merging classes represented by the CCRs u and v. One of those calls may involve less operations at line 3. If the set u.TERMS is smaller than v.TERMS, UNION(u, v) may involve less computation. In order to systematically choose the less expensive execution, one needs to efficiently compare the set sizes. We assume that for each set field[8] the number of elements is implicitly stored in an integer variable; the number of elements in such a set can be obtained in constant time. In other words, the size of two sets can be compared in constant time, and the operation that follows takes constant time:[9]

2 bis: **if** $|a.\text{TERMS}| < |b.\text{TERMS}|$ **then** $(a, b) := (b, a)$;

Adding this swap at the beginning of function UNION decreases the amount of operations at line 3, from quadratic to $\mathcal{O}(n \log(n))$. This is proved later.

The second change is not so straightforward. The idea is to suppress the double **for each** loop at line 6. The task of lines 6 to 8 is to implement the Deduction rule, i.e., to find classes that should be merged as a consequence of the class merge for a and b. In fact two classes should be merged if they each contain a term with the same signature. Merging classes potentially

[8] Sets are actually implemented with lists. Lists may contain repeated elements, without negative consequences on the good behaviour of the algorithm.

[9] Only the differences between two successive versions are given. For a full version of the final algorithm, see in Appendix C.

changes the signature for predecessors. If $\textsc{sig}(f(x_1, \ldots x_p))$ is defined to be $(f, \textsc{Find}(x_1), \ldots \textsc{Find}(x_p))$, signatures for predecessors of terms in class b are changed by class updates at line 3. These signature changes may potentially give the same signature to two terms in different classes. The code at lines 6 to 8 ensures that such circumstances are detected, and that such classes are eventually merged. This is checked by inspecting pairs of terms. But it could also be checked by ensuring that every signature either corresponds to a unique class, or will eventually correspond to a unique class due to future UNIONs. That is basically the idea of this last change.

Predecessor terms are not compared by pairs anymore, but rather signatures are kept in a table. Every time a signature changes it is compared with those in the table to ensure that either no other term of a different class also has this signature, or a new equality will eventually be added to COMBINE about those two distinct classes. Formally the new table used to keep a track of all signatures and classes containing them is an association list containing pairs $(\textsc{sig}(x), \textsc{Find}(x))$ for every signature. But as classes may disappear by merging, it is better to rather store pairs $(\textsc{sig}(x), x)$, one for each signature; it saves the time of updating those pairs $(\textsc{sig}(x), x)$ whenever classes for x are changed.

Three functions allow to access this association list:

- $\textsc{enter}(x)$ adds an element $(\textsc{sig}(x), x)$ in the list;
- $\textsc{delete}(x)$ removes the element corresponding to $\textsc{sig}(x)$;
- $\textsc{query}(x)$ returns the symbol $\emptyset$ (by convention) if no element corresponds to $\textsc{sig}(x)$. Otherwise, a pair $(\textsc{sig}(x), y)$ belongs to the association list, and y is returned.

Those operations should be implemented efficiently. A solution is to use one hash table, indexed by signature. In that case, all three operations can be assumed to be constant time.[10] In the initial state, the signature table contains one entry for every term with non-null arity:

Property 4. *For every term* $f(x_1, \ldots x_p) \in \mathcal{T}$*,* $\textsc{query}(f(x_1, \ldots x_p)) \neq \emptyset$*,*

Lines 6 to 8 can be replaced by the following ones:

6:     **for each** $x \in b.\textsc{pred}$ **do**
7:         **if** $\textsc{query}(x) = \emptyset$ **then**
8:             $\textsc{enter}(x)$;
9:         **else if** $\textsc{Find}(x) \neq \textsc{Find}(\textsc{query}(x))$ **then**
10:             $\textsc{deduct} := \textsc{deduct} \cup (x, \textsc{query}(x))$;

[10] Other data structures for signature table are briefly discussed in [63, 137].

Notice that there is no need to clean the table by removing the old signatures, that is signatures $(f, a_1, \dots a_p)$ where for some i, a_i is a disappeared class (that is a b class at line 3 of function UNION). Those elements remain inactive for the rest of the execution.[11]

Finally, as a last change, it is possible to reiterate the trick used for class updates, i.e., updating the smallest list, but for signature updates. If signature $\textsc{sig}(f(x_1, \dots x_p))$ is defined to be $(f, \textsc{Find}(x_1), \dots \textsc{Find}(x_p))$, every predecessor of the disappearing class will have to be updated. It is more efficient to define $\textsc{sig}(f(x_1, \dots x_p))$ to be $(f, \textsc{Find}(x_1).\textsc{s}, \dots \textsc{Find}(x_p).\textsc{s})$, where S is a new CCR field. It is assumed that it can be accessed and modified in constant time. Also, it is required for those fields to be unique, initially:

Property 5. *For all terms $u, v \in \mathcal{T}$, $\textsc{Find}(u) = \textsc{Find}(v)$ if and only if $\textsc{Find}(u).\textsc{s} = \textsc{Find}(v).\textsc{s}$*

and in fact, this property will be preserved by the execution. As a consequence defining $\textsc{sig}(f(x_1, \dots x_p))$ to be $(f, \textsc{Find}(x_1).\textsc{s}, \dots \textsc{Find}(x_p).\textsc{s})$ agrees with its previous and more abstract definition: $\textsc{sig}(u) = \textsc{sig}(v)$ if and only if u and v are such that $\lambda(u) = \lambda(v)$, $\delta(u) = \delta(v)$, and for every $i \in \{1, \dots \delta(u)\}$, $\textsc{Find}(u[i]) = \textsc{Find}(v[i])$. But it is now possible to preserve the signatures in the larger predecessor list[12], and just update those in the smaller one. This is done by swapping the s fields for both merged classes if the disappearing class is also the one to have the larger predecessor list.

```
5bis:   if |b.PRED| > |a.PRED| then
5ter:       (a.S, b.S, a.PRED, b.PRED) := (b.S, a.S, b.PRED, a.PRED);
```

The whole efficient UNION function is given in Appendix C (Figure C.1). We now adapt proofs for correctness and complexity.

Theorem 16 (correctness). *Theorem 14 also holds with function* UNION *of Figure C.1.*

Proof. No fundamental change is induced by the class swap at the beginning of function UNION. We only need to prove that the second change does preserve correctness. As it modifies the Deduction rule, it suffices to verify

[11] If old entries are not cleaned, the algorithm is $\mathcal{O}(n \ \log(n))$ in space. In practice, this is not important, because space is not a limitation. Cleaning the old entries will just take time without advantages, as long as the hash table is sufficiently large to avoid frequent collisions.

[12] Actually some terms may belong to both predecessor lists. Their signatures will not be preserved anyway. One should rather say here "it will not be necessary to recompute the signatures in the larger predecessor list."

that this new version still correctly implements systematic application of the Deduction rule. That means, all deductions are done and no false equality is generated. This last fact is trivial, as $\text{SIG}(x) = \text{SIG}(\text{QUERY}(x))$ if $\text{QUERY}(x) \neq \emptyset$ by definitions of SIG and QUERY.

Notice that Property 5 is preserved by every execution step. Also Property 4 is preserved by a call to UNION, and it is true for every term in $\mathcal{T} \setminus b.\text{PRED}$ just before inspecting predecessors in $b.\text{PRED}$.

Let $\leftrightarrow_{\text{eqs}}$ be the smallest reflexive, symmetric and transitive relation between terms of $\mathcal{T}$, such that $u \leftrightarrow_{\text{eqs}} v$ if $(u, v) \in \text{COMBINE} \cup \text{DEDUCT}$,[13] or if u and v belong to the same class. The following property is true initially:

Property 6. *For every term $f(x_1, \ldots x_p) \in \mathcal{T}$, $\text{QUERY}(f(x_1, \ldots x_p)) = y$ if and only if $f(x_1, \ldots x_p) \leftrightarrow_{\text{eqs}} y$.*

and if it is true before a call to UNION, it remains true afterwards. Indeed, as the property holds before the call to UNION, it should be true for every term in $\mathcal{T} \setminus b.\text{PRED}$ just before inspecting predecessors in $b.\text{PRED}$.

Finally notice that in the final state (COMBINE is empty), Property 6 means that the Deduction rule cannot be applied anymore. □

Remark: Nelson-Oppen algorithm implements the Deduction rule more systematically than this efficient version. Indeed Nelson-Oppen generates all deducible equalities whereas this version does not.

Theorem 17 (complexity). *If the operations on the signature table are assumed to be constant time, the congruence closure of a set of terms $\mathcal{T}$ corresponding to a set E of equalities is computed in time $\mathcal{O}(n \log(n) + |E|)$, where n is the total number of nodes in the representations of terms in $\mathcal{T}$.*

Proof. After congruence closure computation, every term in any class a has seen its class updated a certain amount of time. Thanks to line 2bis, every time the class for one of the terms is updated, the class will just be merged with another class containing at least the same number of terms. That is, at most half of the terms in $a.\text{TERM}$ have been updated by the last execution of line 3 in function UNION. Let f be the function such that the number of class updates for a list of p terms is at most $f(p)$. We have

$$f(p) = \text{SUP}\left\{ f(p-a) + f(a) + a \mid a \in \mathbb{N} \text{ and } 0 < a \leq \frac{p}{2} \right\}$$

with $f(1) = 0$. Then[14] $f(p) < \mathcal{O}(p \log(p))$. Every term appears in at most one class, and the total number of terms is n so the total number of operations

[13] It is assumed $\text{DEDUCT} = \emptyset$ outside function UNION.

[14] Another way to come to the same conclusion is to notice that every element is updated at most $\log_2(p)$ times, because the list size at least doubles every time the element is updated.

is bounded by

$$\sum_{c \in \mathcal{C}} \mathcal{O}\left[|c| \log\left(|c|\right)\right] < \mathcal{O}\left[n \log\left(n\right)\right]$$

where $\mathcal{C}$ is the final congruence closure. This inequality is a consequence of the concavity of function $p \log(p)$.

The number of constant time operations involved by predecessor list inspections is also $\mathcal{O}[n \log(n)]$ for exactly the same reasons. One only needs to notice that every predecessor of arity k appears k times in predecessor lists (one for every argument). This remains true during the whole execution. Conclusion is obtained using the same fact that only the smallest list is updated. □

Remark: Introduction of the s field for CCR and updating the smallest predecessor lists are not required to guarantee $\mathcal{O}(n \log(n))$ time complexity. This complexity is preserved even without lines 5bis and 5ter, as long as the arity is assumed to be bounded. This last condition is not necessary for the algorithm presented here.

Nothing has been said yet about how to assign a unique value to the CCR s field. Initially every term is in its own class, so a simple solution is to assign to s the DAG identifier of this element. In that case $\textsc{Find}(u).\textsc{s} \in \textsc{Find}(u).\textsc{term}$ is always true.

7.4 Being incremental

Having an efficient algorithm is important, but it is also important to have an algorithm which is "on-the-fly", that is accepting literals one by one — particularly when used eagerly with a SAT solver (see Chapter 6). The previous algorithm is somewhat incremental, as building the congruence closure of the set of equalities $\{u_1 = v_1, \ldots u_n = v_n\}$ is done by successive calls to $\textsc{Merge}(u_i, v_i)$ with every $i \in \{1, \ldots n\}$. But in one aspect, it is not incremental: the finite set $\mathcal{T}$ of terms in the congruence closure is defined *a priori* and is static. However this set of terms might not be known *a priori*. In a such context it is necessary to be able to include terms in the congruence closure during the process of its computation.

We now assume that $\mathcal{T}$ is initially empty. No DAG has an associated CCR; by convention $\textsc{Find}(u) = \emptyset$ for every term. A call to procedure $\textsc{Add\_node}(u)$ (Figure 7.4) adds a term in $\mathcal{T}$. That is, if the congruence closure has been built for set E on $\mathcal{T}$, and terms $\{t_1, \ldots t_n\}$ are added to $\mathcal{T}$, the congruence closure on $\{T\} \cup \{t_1, \ldots t_n\}$ is immediately obtained. Further equalities from set E' can be given to $\textsc{Merge}$ to compute the congruence

```
procedure Add_node(u);
begin
    if Find(u) ≠ ∅ then return;
    for i := 1 to δ(u) do Add_node(u[i]);
    Find(u) := newclass;
    Find(u).s := u;
    Find(u).term := {u};
    Find(u).pred := ∅;
    if δ(u) > 0 then
        if query(u) then
            Union(Find(u), Find(query(u)));
        else
            begin
                enter(u);
                for i := 1 to δ(u) do
                    Find(u[i]).pred := Find(u[i]).pred ∪ {u};
            end
end
```

Figure 7.4: Procedure Add_node

closure of $E \cup E'$ on $\mathcal{T} \cup \{t_1, \ldots t_n\}$. Adding terms in this way preserves all properties established above, including correctness and complexity. Furthermore, we have the following result:

Theorem 18. *A call to function* ADD_NODE *with a representation of the term u as argument is linear with respect to the number of nodes in u such that* FIND$(u) = \emptyset$.

Proof. Lines 4 to 16 in function ADD_NODE are executed in constant time. This is trivially true for each line except the call to function UNION. If UNION is called, then u is such that FIND(u).TERM $= \{u\}$, and FIND(u).PRED $= \emptyset$. It is easily proved that in such cases the calls to UNION are constant in time. □

This result clearly establishes that terms should be added to $\mathcal{T}$ as late as possible, because they do not contribute to the execution time before their introduction, and their introduction is cheap.

7.5 Decision procedure

The previous versions of the algorithm construct the congruence closure but do not handle inequalities, and predicates. Testing if a congruence closure is compatible with the set of inequalities and predicates is supposed to be done afterwards: first the congruence closure is computed, then every inequality is inspected to ensure both terms are not in the same class, and finally predicates are checked for complementary pairs. Congruence closure is $\mathcal{O}(n \log(n))$ in time, and the second operation is linear. The third one requires to be able to efficiently find pairs of complementary literals. As before, this can be done in linear time using a hash table indexed by signatures and storing polarities, as long as adding and testing for membership are assumed to be constant time operations in the hash table.

So there is no complexity problem here. But such a decision procedure assumes that all equalities are treated before the other literals. This is not compatible with an "on-the-fly" or incremental approach. But with little changes, an $\mathcal{O}(n \log(n))$ algorithm can be built, not only for computing congruence closures, but also for checking satisfiability of sets of literals with (in)equalities, and uninterpreted predicates.

First assume that there are no predicate in the set of literals, but only equalities and negations of inequalities. Two new functions are now provided for taking literal ℓ into consideration for satisfiability check. The functions ADD_EQUALITY and ADD_INEQUALITY (Figure 7.5) provide this facility if literal ℓ is respectively an equality, or the negation of an equality. These functions do not assume that terms used in literals are all in the set $\mathcal{T}$ of terms u such that $\textsc{Find}(u) \neq \emptyset$. In fact terms which are not already in $\mathcal{T}$ are put in $\mathcal{T}$ as soon as the functions receive the literals containing them.

Both functions return either $\top$ or $\bot$ depending on satisfiability of the set of already given literals. A new global variable $\textsc{status}$ is introduced. Initially it is $\top$, and becomes $\bot$ whenever the set $\mathcal{L}$ of introduced literals becomes unsatisfiable. The following property holds initially:

Property 7. *Variable* $\textsc{status}$ *is* $\top$ *if and only if the set of processed literals* $\mathcal{L}$ *is satisfiable. It is* $\bot$ *otherwise.*

Indeed the empty set is satisfiable. It will be shown that this property also holds after every literal processing.

CCRs now have a new field $\textsc{ineq}$. This field is used to keep track of all inequalities relevant to the class. This field is initially empty, and line

7bis: $\textsc{Find}(u).\textsc{ineq} = \emptyset;$

is added to the code of function $\textsc{Add\_node}$ (Figure 7.4) to reflect this. Property:

Property 8. *Let* $u \neq v \in \textsc{Find}(a).\textsc{ineq}$. *Then* u *and* v *are in the same class if and only if* $\textsc{status} = \bot$.

is trivially true initially. But it is also true (nearly) every time, if the function $\textsc{Union}$ is modified in the appropriate way, adding a test

if $|b.\textsc{ineq}| > |a.\textsc{ineq}|$ **then** $(a, b) := (b, a);$
for each $(u \neq v) \in b.\textsc{ineq}$ **do**
    if $\{\textsc{Find}(u), \textsc{Find}(v)\} = \{a, b\}$ **then** $\textsc{status} = \bot;$

right at the beginning of function $\textsc{Union}$, and the updating line

$a.\textsc{ineq} := a.\textsc{ineq} \cup b.\textsc{ineq};$

after class update, just as for merging predecessor lists. Now correctness is easily verified:

Theorem 19 (correctness). *Property 7 is true after every successive call to functions* $\textsc{Add\_equality}$ *and* $\textsc{Add\_inequality}$ *(Figure 7.5).*

```
function Add_equality(ℓ);
    var u, v : Term;
    begin
        let u, v such that u = v is ℓ;
        Add_node(u);
        Add_node(v);
        Merge (u, v);
        return status;
    end
```

```
function Add_inequality(ℓ);
    var u, v : Term;
    begin
        let u, v such that u ≠ v is ℓ;
        Add_node(u);
        Add_node(v);
        if Find(u) = Find(v) then status = ⊥;
        Find(u).ineq := Find(u).ineq ∪ {ℓ};
        Find(v).ineq := Find(v).ineq ∪ {ℓ};
        return status;
    end
```

Figure 7.5: Procedures Add_(in)equality

Proof. Property 8 is trivially preserved by function UNION, and is hence true every time while not executing UNION. Also, it is easy to show that if $\mathcal{L}$ is the set of processed literals, if $u \neq v \in \mathcal{L}$, then $u \neq v \in \text{FIND}(u).\text{INEQ}$ and $u \neq v \in \text{FIND}(v).\text{INEQ}$.

Hence if STATUS $= \top$ after all literals in $\mathcal{L}$ have been processed, it is possible to find an interpretation which satisfies every literal. It is just necessary to ensure that every pair of classes a and b containing terms t_a and t_b such that $t_a \neq t_b \in \mathcal{L}$ are not associated to the same domain element. □

Theorem 20. *Every sequence of calls to functions* ADD_EQUALITY *and* ADD_INEQUALITY *with literals in $\mathcal{L}$ is $\mathcal{O}(n \log(n) + |\mathcal{L}|)$ in time, where n is the number of distinct nodes in the list of processed literals $\mathcal{L}$.*

Proof. Consider inequality lists. Only the smallest list is inspected for inconsistency before merging lists. This treatment is similar to the one used for class updates and predecessor signature updates. In the same way the number of constant time operations induced by this new check is $\mathcal{O}(n \log(n))$. □

Now some words about predicates. Actually predicates can be considered as functions into the set $\{\top, \bot\}$. Treatment of literal $p(t_1, \ldots t_n)$ is done by adding equality $p(t_1, \ldots t_n) = \top$, whereas literal $\neg p(t_1, \ldots t_n)$ translates to equality $p(t_1, \ldots t_n) = \bot$. Inequality $\top \neq \bot$ should be provided to the decision procedure before any other literal. This simple direct adaptation to accept predicates preserves correctness and complexity results. But it might produce very large classes of terms, that is classes for $\top$ and $\bot$.

It is important to ensure this is not a cause of inefficiency in practice. First remember adding a term to the set of handled term is linear with respect to the number of its nodes which are not yet in this set of handled term. This part of the work is not problematic. Inefficiency could come from the call to MERGE. But once again, it is constant-time, and is not affected by class size: only one inequality is relevant for those classes, and predicates do not have predecessors. Calls to UNION with predicate classes are constant in time and do not involve more calls to UNION.

7.6 Being proof producing

Once a set of literals has been found unsatisfiable, it is important for the collaboration with propositional solvers (see Chapter 6) to provide a finer answer than just satisfiable or unsatisfiable, that is, to provide a small unsatisfiable subset. If this subset is minimal[15] the returned information is

[15] That is if no literal can be removed keeping the set unsatisfiable.

optimal (see Section 6.3 for more detailed arguments). The basic idea is to remember the cause of each class union. This information will then be collected after an inconsistency has been detected. We used this idea with Nelson-Oppen algorithm in [79]. A similar idea has also been previously used in [134].

Every DAG a is assumed to be associated with a set $a.\text{EQ}$ of equalities, or rather a set of triples (b, c, E) where b and c are terms, and E is a possibly empty set of equalities. Initially, this set of triples is empty. The line

3bis: $u.\text{EQ} = \emptyset;$

is added to function ADD_NODE (Figure 7.4). Every time two classes are merged, a new element is added to the set of triples of one element in each class. One argument E (a set of equalities) is added to function MERGE, and statement

8: $(u.\text{EQ}, v.\text{EQ}, E) := (u.\text{EQ} \cup \{(u, v, E)\}, v.\text{EQ} \cup \{(u, v, E)\}, \emptyset);$

is executed every time UNION is called. Function ADD_NODE is modified in a similar way, whereas in ADD_EQUALITY the call to MERGE is added $\{\ell\}$ as argument. The whole algorithm is given in Appendix C.

Let R_l be the set of triples $R_l =_{\text{def}} \{(u, v, E) \mid \exists t : (u, v, E) \in t.\text{EQ}\}$, and R be the relation $R =_{\text{def}} \{(u, v) \mid \exists E : (u, v, E) \in R_l\}$. A path from a_1 to a_n in R (or R_l) is a sequence of nodes $a_1, \ldots a_n$ such that for every $i \in \{1, \ldots n-1\}$, either $(a_i, a_{i+1}) \in R$ or $(a_{i+1}, a_i) \in R$. A non-looping path is a path $a_1, \ldots a_n$ which does not contains twice (or more) the same node. The following property holds every time outside function MERGE

Property 9. *Distinct terms u and v are in the same class if and only if there is a unique non-looping path in R (or R_l) from u to v.*

Proof. Indeed, it is true initially as R is empty, and every time there is a class MERGE, elements are added in R to keep the property true. □

Property 10. *If $(u, v, E) \in t.\text{EQ}$, then t is either u or v, and $(u, v, E) \in u.\text{EQ}$ if and only if $(u, v, E) \in v.\text{EQ}$. If $(u, v, E) \in u.\text{EQ}$ is such that $E \neq \emptyset$, then E is $\{u = v\}$. If $(u, v, \emptyset) \in u.\text{EQ}$, then $u = f(t_1, \ldots t_n)$ and $v = f(t'_1, \ldots t'_n)$ with t_i and t'_i is the same class for every i.*

Proof. It is easily proved by induction. □

Notice that whenever E is non empty in $(u, v, E) \in t.\text{EQ}$, then $E \models u = v$. In fact empty sets of equalities in triples correspond also to sets of equalities implying equality of the two terms. But the computation of those sets are left for later evaluation. This is a kind of lazy evaluation.

The *set of labels* of a non-looping path $a_1, \dots a_n$ in a set of triples R_l is the equality set

$$\bigcup \{E \mid (a_i, a_{i+1}, E) \in R_l \text{ or } (a_{i+1}, a_i, E) \in R_l \text{ with } 0 < i < n\}$$

The set of *computed triples* R'_l associated to R_l is the smallest set such that:

- if $(u, v, E) \in R_l$ with $E \neq \emptyset$, then $(u, v, E) \in R'_l$;
- if $(f(t_1, \dots t_n), f(t'_1, \dots t'_n), \emptyset) \in R_l$, then $(f(t_1, \dots t_n), f(t'_1, \dots t'_n), E')$ belongs to R'_l if there exists a path $a_{1,i}, \dots a_{n_i,i}$ in R'_l from t_i to t'_i for every i, and E' is the union of the set of labels for every i.

Property 11. *Let R_l be the set of triples $R_l =_{\text{def}} \{(u, v, E) \mid \exists t : (u, v, E) \in t.\text{EQ}\}$, and R'_l the set of computed triples associated to R_l. For every element $(u, v, E) \in R_l$, there is an element $(u, v, E') \in R'_l$, such that $E' \models u = v$.*

Proof. This is easily proved by induction. □

Used in conjunction with Property 9 it shows that for every two terms u and v in the same class, the set of labels E' associated to the path between u and v is such that $E' \models u = v$. But how much useful is this computation? The set containing all equalities implies the equality of any two terms in the same class. But the goal here is to obtain a small set of equalities, that is a set which is nearly minimal: a set of equalities E such that $E \models u = v$ is minimal (with respect to $u = v$) if for any proper subset E' of E, $E' \not\models u = v$. Intuitively, a minimal set does not contain unnecessary equalities.

In general the computed set of equalities is not minimal. Indeed consider an execution where the equalities $f(a) = f(b), f(b) = a, a = b$ are successively given to the congruence closure algorithm. The computed set for $f(a) = b$ will contain all three equalities but only $f(b) = a$ and $a = b$ are necessary to imply $f(a) = b$. However, there is a case where the computed set can be proved minimal.

Property 12. *Let R_l be the set of triples $R_l =_{\text{def}} \{(u, v, E) \mid \exists t : (u, v, E) \in t.\text{EQ}\}$, and R'_l the set of computed triples associated to R_l. If $a_1, \dots a_n$ is a path in $R_l \cap R'_l$ from u to v, then the set E of labels of $a_1, \dots a_n$ in $R_l \cap R'_l$ is such that $E \models u = v$, and E is minimal.*

Proof. This is easily proved by induction. □

Notice this trivially holds for every two terms in the same class in the function-free case. More generally the computed set of equalities is often minimal in practice. When it is not, it contains very few unnecessary equalities.

Chapter 8

Rich ground logics

The preceding chapter introduced a decision procedure for finite sets of ground literals containing only equalities with uninterpreted predicates and functions. Besides equality, other interpreted predicates and functions may be required, like order predicates, addition for reals, rationals or integers, place predicates... Quantifier-free first-order logic with equality is decidable; but some quantifier-free formula sets with interpreted predicates and functions are not decidable: as an example Hilbert's tenth problem has been proved undecidable [129]. Fortunately, common fragments are decidable: linear arithmetic on reals, rationals, or integers, first-order theories of arrays with extensionality, lists, ...

The problem addressed here is the *combination* of decision procedures. The set of literals

$$\{x \leq y,\ y \leq x + f(x),\ P(h(x) - h(y)),\ \neg P(0),\ f(x) = 0\}$$

contains some linear arithmetic, uninterpreted functions f, h and one uninterpreted predicate P. Clearly, it is unsatisfiable: $f(x) = 0$ implies $y \leq x$, thus x should be equal to y using $x \leq y$, and finally $P(0)$ should be both true and false. Linear arithmetic is decidable, and congruence closure implements a decision procedure for a logic of quantifier-free formulas with equalities and uninterpreted predicates and functions. The topic of this chapter is to design a practical decision procedure for (finite sets of ground literals from) the union of two disjoint *quantifier-free* decidable languages with equality, from both individual decision procedures.

8.1 The state of the art

Usually a preliminary step to study the decision problem for restricted languages of logic with interpreted functions or predicates is to build a *the-*

ory. The theory is a set of formulas or *axioms* representing the constraints on interpretations. Those formulas encapsulate the meaning of interpreted predicates and functions. When considering satisfiability of formulas with interpreted functions, only models of the theory are examined.

Whether or not the built set of formulas perfectly describes the interpreted predicates and functions is a delicate problem. Given a decision procedure for a decidable fragment of logic, let's say, linear arithmetic on $\mathbb{N}$, it is possible to build a theory for it: the set of all valid formulas of increasing size is suitable. A formula is valid (unsatisfiable) in $\mathbb{N}$ if and only if it (respectively its negation) belongs to the formulas in the theory. An arbitrary formula is satisfiable in a model of the theory if and only if the formula has a model with $\mathbb{N}$ as domain, and interpreting symbols $+$, 0, 1, ... as expected. The same theory, augmented by two axioms for multiplication, is incomplete as well as any theory containing those axioms (see for example [74]). This essentially means there exist formulas which are satisfied by a model of the theory, although they are expected to be unsatisfiable: those formulas have no model with $\mathbb{N}$ as domain and $+$, $\times$, 0, 1,... interpreted as usual, i.e., they have no *standard model*. Adding axioms to the theory does not correct the problem. The set of models of such a theory is larger than the set of interpretations one would like to consider using this theory.

When dealing with a combination of decision procedures, predicates and functions are added to a decidable fragment. One concern is whether or not it remains decidable, but another is whether or not satisfiability within the theory really means there is a model within our criteria. If φ is a satisfiable formula within a combination scheme for quantifier-free logic adding uninterpreted predicates and functions to the decidable fragment of linear arithmetic, is φ satisfied by an interpretation giving the expected meaning to $+$, 0, 1, ...?

Two approaches are possible: extending the concept of theory to allow to better select the "models" of the theory, or reasoning within the first-order framework and prove that every satisfiable formula has a model satisfying the criteria. The latter might be tedious with regard of the developments for arithmetic in quantified logic [74]. Following the approach of [85], an *I-theory* in a many-sorted language $\mathcal{L}$ is an arbitrary set of interpretations on $\mathcal{L}$. The I-theory corresponding to a first-order theory T in $\mathcal{L}$ is the set of models of all formulas (axioms) in T. Given an I-theory $\mathcal{T}$ in a language $\mathcal{L}$, a formula φ of $\mathcal{L}$ is $\mathcal{T}$-satisfiable ($\mathcal{T}$-unsatisfiable, $\mathcal{T}$-valid) if one (respectively none, all) interpretations in $\mathcal{T}$ makes φ true. Notation $\psi \models_{\mathcal{T}} \varphi$ (respectively $\models_{\mathcal{T}} \varphi$) means $\psi \Rightarrow \varphi$ (resp. φ) is $\mathcal{T}$-valid. Given a first-order theory (that is, a set of axioms), the corresponding I-theory is just the set of all models of those axioms.

The I-theory of linear arithmetic on naturals has already been mentioned in this section. The language contains a countable set of variables, two constants 0 and 1, two functions $+$ and $-$, and the predicate $\leq$.[1] The interpretations in the I-theory are such that the domain for the only sort τ is interpreted as the set of naturals $\mathbb{N}$, the operations $+$ and $-$ are associated to the usual addition and subtraction operation in $\mathbb{N}$, and predicate $\leq$ is interpreted as the order in $\mathbb{N}$. It is convenient to add the constants $2, 3, \ldots$ and the predicates $<, \geq, >$. Another common theory is the theory of equality with uninterpreted predicates and functions. It is usually referred to as the "empty theory" as it is the first-order theory with equality with no axioms.

In the introductory example at the beginning of this chapter, it was noticed that it was possible to reason about a conjunction of literals with uninterpreted functions and predicates, and arithmetic by considering on one side the "arithmetical part" and on the other, the "uninterpreted part". This is linked to the fact that both languages are disjoint: formally signatures for languages $\mathcal{L}_1 = \langle \mathcal{S}_1, \mathcal{V}_1, \mathcal{F}_1, \mathcal{P}_1, r_1, d_1 \rangle$ and $\mathcal{L}_2 = \langle \mathcal{S}_2, \mathcal{V}_2, \mathcal{F}_2, \mathcal{P}_2, r_2, d_2 \rangle$ are *disjoint* when $\mathcal{F}_1 \cap \mathcal{F}_2 = \mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$, i.e., if both languages have no predicate or function in common. The only common symbols are the equality and variables. Theories are disjoint if their languages are disjoint.

Two frameworks exist to combine decision procedures for disjoint theories: Shostak's [167, 168] and Nelson-Oppen [135] combination scheme.[2] The Nelson-Oppen framework has been formalized and proved correct in [174]. Completeness of the different versions of Shostak's combination scheme has always been difficult to prove [54, 162, 85]. It appears a satisfactory description and proof of Shostak's combination scheme comes with the idea that the Shostak's framework is a special case of the Nelson-Oppen framework [178, 155].

The Nelson-Oppen scheme is usually presented as a combination scheme for (unsorted) first-order theories. In this context, the theories to be combined using the Nelson-Oppen framework are required to have disjoint signatures and be *stably infinite*. A theory $\mathcal{T}$ for language $\mathcal{L}$ is stably infinite if for any $\mathcal{T}$-satisfiable formula φ in $\mathcal{L}$, there is a model of φ in $\mathcal{T}$ whose domain is infinite. In other words, given a stably infinite first-order theory, any satisfiable formula has a model with cardinality $\aleph_0$, thanks to Löwenheim-Skolem Theorem (see for example [71]).

[1] Our framework is many-sorted logic with equality. Symbol $=$ is interpreted as the identity.

[2] Preceding chapter introduced the Nelson-Oppen congruence closure and mentioned Shostak's congruence closure. Congruence closure is not the point of this chapter, but combination schemes. Another confusing fact is that Shostak's combination scheme is an extension of congruence closure.

Shostak's framework combines the empty theory with one (non necessarily first-order) I-theory $\mathcal{T}$ having some properties: it should be *convex*, it should not contain predicates, it should have a *solver* and a *canonizer*. An I-theory $\mathcal{T}$ for language $\mathcal{L}$ is convex if for any finite conjunction Γ of equalities of $\mathcal{L}$, then

$$\Gamma \models_{\mathcal{T}} \bigvee_{1 \leq i \leq n} x_i = y_i \quad \text{if and only if} \quad \Gamma \models_{\mathcal{T}} x_j = y_j \quad \text{for some } j \in [1..n]$$

for some $i \in \{1, \ldots n\}$, where $x_1, y_1, \ldots y_n, x_n$ are variables of $\mathcal{L}$. For instance, linear arithmetic is convex for reals and rationals[3], but not for integers.[4] Formal definitions for canonizer and solver can be found in [114]. Intuitively, a canonizer is a transformation operator of terms of the language to a canonical form, and thus reduces semantically equal terms to syntactically identical ones. For instance a canonizer for linear arithmetic rewrites every sum into a canonical form using associativity and commutativity properties, and a predefined order on variables and constants. A solver returns a general solution (see [114] for more information) for a finite satisfiable set of literals in the language, or $\perp$ if the set is unsatisfiable. For instance a solver for a set of linear arithmetic equations expresses the value of some variables as a function of the other variables such that every equation is satisfied.

Examples of Shostak theories are linear arithmetic on reals or rational and list theory. The empty theory and the theory of arrays [172] are not Shostak theories. The empty theory has no solver [125], and the theory of arrays with or without extensionality is not convex. Interpreted predicates are not allowed, and thus order arithmetic predicates cannot be integrated within the Shostak's framework. The strong requirements on Shostak theories induced the need to accommodate Shostak's framework with the Nelson-Oppen scheme [24]. Furthermore, if Shostak's method was originally based [114] on an efficient decision procedure for the union of the empty theory and one Shostak theory, and the claim that the disjoint union of two (and therefore any finite number of) Shostak theories is a Shostak theory, it appeared combining solvers was not always possible [165]. In [114] it is even proved that canonizers can be combined, but there exists no solver for combination of theories of practical interest.

However, what cannot be done in Shostak's framework can (often) be done in the Nelson-Oppen framework; if $\mathcal{T}$ is a convex first-order theory,

[3] The set of solutions of a conjunction of literals defines a convex polyhedron in the space of variables. A disjunction of equalities defines a union of hyperplanes. If a convex polyhedron is included in a union of hyperplanes, it is totally included in one of them.

[4] Indeed the set $\{x > 0, x < 3, y = 1, z = 2\}$ entails $x = y \vee x = z$ but does neither entail $x = y$ nor $x = z$.

then $\mathcal{T} \cup \{\exists x \exists y\ x \neq y\}$ is stably infinite [21, 125]. Also, a refinement of the Nelson-Oppen framework establishes that every (i.e., not necessarily stably infinite) first-order theory is combinable with the empty theory [176]. It becomes natural to present Shostak's combination as an refinement of the Nelson-Oppen combination [155], confirming the remark from [53]:

> We hypothesise that in a modular implementation there is no advantage in allowing the core module to have access to Shostak module primitives (canonizer and solver); the same efficiency can be achieved with a plain Nelson-Oppen core that communicates with Shostak theory modules through generic theory module interfaces, while canonizer and solver are used only to implement those interfaces.

The Nelson-Oppen combination scheme currently has several weaknesses. First, non first-order theories are ignored. Second, the stably infinite requirement prevents combining theories with only finite domains. Third, its usual presentation is unsorted. Those three weaknesses are important in a verification context. For instance, when checking verification conditions for systems with a parameterized number of processes, the process sort should be associated with a domain of any finite size. Such a theory is not first-order, and is not stably infinite. Finally, verification conditions are naturally expressed using many-sorted logic.

Solutions for each of the last two weaknesses have been proposed. Any single first-order theory can be combined with a *shiny* [176] theory: this relaxes the stably infinite requirement. However requirements on shiny theories are strong: one useful instance of shiny theories is the empty theory. In [177] the Nelson-Oppen scheme is adapted to many-sorted first-order logic.[5] However, the stably infinite requirement is still present and only first-order theories are discussed. Finally it should be mentioned here that some work has also been done on combining non-disjoint theories [160, 175, 188].

The following combination framework is a unified answer to the previously mentioned weaknesses. It is based on a many-sorted logic, and theories are neither assumed to be first-order, nor stably infinite.

[5] In this work, they present a combination scheme in an *order-sorted* context, i.e., sorts can be subsorts of other sorts. In the following we assume disjoint sorts.

8.2 Combining decision procedures

8.2.1 Preliminaries

A first-order theory is a set of axioms, which defines a set of interpretations, i.e., the models of the first-order theory. A natural extension is to consider I-theories: an I-theory is an arbitrary set of interpretations in a given many-sorted language. The I-theory corresponding to a first-order theory is the set of the models of the first-order theory. An I-theory may leave some predicates and functions uninterpreted. A predicate p of sort $\langle \tau_1, \ldots \tau_n \rangle$ (resp. a function f of sort $\langle \tau_1, \ldots \tau_n, \tau \rangle$) is uninterpreted in an I-theory $\mathcal{T}$ if for every interpretation $\mathcal{I} = \langle D, I \rangle \in \mathcal{T}$, and for every predicate q (resp. function g) of suitable sort there is an interpretation $\mathcal{I}' \in \mathcal{T}$ such that $\mathcal{I}'$ is the same as $\mathcal{I}$ except that $\mathcal{I}'[p] = q$ (resp. $\mathcal{I}'[f] = g$). It is assumed that variables are always left uninterpreted in any I-theory, with a meaning similar to uninterpreted constants. Given an I-theory $\mathcal{T}$, a formula φ is $\mathcal{T}$-satisfiable if it has a model in $\mathcal{T}$.

Two I-theories are disjoint if their languages are disjoint. Disjoint languages are languages with disjoint sets of functions (and constants) and predicates (though the equality symbol is shared). Here is presented a framework to combine disjoint I-theories: a decision procedure for the union of two disjoint I-theories is build on the basis of decision procedures for each I-theory.

Formulas including predicates and functions of two disjoint I-theories are written in the union of the disjoint languages.

Definition 20 (union of disjoint languages). *The union of two disjoint languages $\mathcal{L}_1 = \langle \mathcal{S}_1, \mathcal{V}_1, \mathcal{F}_1, \mathcal{P}_1, r_1, d_1 \rangle$ and $\mathcal{L}_2 = \langle \mathcal{S}_2, \mathcal{V}_2, \mathcal{F}_2, \mathcal{P}_2, r_2, d_2 \rangle$ is the language $\mathcal{L} = \langle \mathcal{S}_1 \cup \mathcal{S}_2, \mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{F}_1 \cup \mathcal{F}_2, \mathcal{P}_1 \cup \mathcal{P}_2, r, d \rangle$, where r is the function equal to r_1 when its argument is in the domain of r_1, otherwise it is equal to r_2. Function d is defined similarly.*

An I-theory $\mathcal{T}$ in the union $\mathcal{L}$ of disjoint languages $\mathcal{L}_1$ and $\mathcal{L}_2$ can be seen as the union of two disjoint I-theories: the restrictions $\mathcal{T}_i$ of $\mathcal{T}$ to each language $\mathcal{L}_i$, i.e., the set of all restrictions[6] to $\mathcal{L}_i$ of interpretations in $\mathcal{T}$. However, it is more convenient to define the union of I-theories in a constructive way:

Definition 21 (union of I-theories). *The union $\mathcal{T}$ of I-theories $\mathcal{T}_1$ and $\mathcal{T}_2$ in disjoint languages $\mathcal{L}_1$ and $\mathcal{L}_2$ is the set of all interpretations $\mathcal{I}$ such that interpretations $\mathcal{I}_i = \langle D_i, I_i \rangle \in \mathcal{T}_i$ exist for $i = 1, 2$, with $\mathcal{I}_i$ being the restriction to $\mathcal{L}_i$ of $\mathcal{I}$.*

[6]See Chapter 2, page 13.

In this definition the restriction to $\mathcal{L}_1$ of $\mathcal{T}$ is not necessarily $\mathcal{T}_1$. Indeed, if for a common sort τ in $\mathcal{L}_1$ and $\mathcal{L}_2$, an interpretation $\mathcal{I}_1$ from $\mathcal{T}_1$ assigns to τ a domain which is never assigned by any interpretation in $\mathcal{T}_2$, then no interpretation in $\mathcal{T}$ will have $\mathcal{I}_1$ as a restriction.

Notice that if $\mathcal{T}_1$ and $\mathcal{T}_2$ are the set of models of first-order theories T_1 and T_2, the union of I-theories $\mathcal{T}_1$ and $\mathcal{T}_2$ is the set of models of the (first-order) theory corresponding to the set of axioms $T_1 \cup T_2$. Our definitions for I-theory and union of I-theories extend the classical definitions of theory and union of theories but are not in contradiction with them.

As the combining method we propose applies to the general notion of I-theories introduced earlier, it is necessary to encapsulate the "combinable property" in a new notion. Combining two decision procedures for disjoint I-theories will require that, for each common sort, at least one of the I-theories is *flexible*:

Definition 22 (flexible theory). *An I-theory $\mathcal{T}$ in a many-sorted language $\mathcal{L}$ is* flexible *on a sort τ if either τ is not a sort of $\mathcal{L}$, or if for any interpretation $\mathcal{I} = \langle D, I \rangle$ in $\mathcal{T}$, any interpretation $\mathcal{I}' = \langle D', I' \rangle$ such that*

- *the sets D_τ and D'_τ have the same cardinality, and $D_{\tau'} = D'_{\tau'}$ for any $\tau' \neq \tau$. Function b defines a bijection from set D_τ to D'_τ, and is the identity on $D_{\tau'}$ for any $\tau' \neq \tau$;*
- *for any variable x, $\mathcal{I}'[x] = b(\mathcal{I}[x])$;*
- *for any function $f \in \mathcal{F}$, $\mathcal{I}'[f](b(d_1), \dots b(d_n)) = b(\mathcal{I}[f](d_1, \dots d_n))$;*
- *for any predicate $p \in \mathcal{P}$, $\mathcal{I}'[p](b(d_1), \dots b(d_n)) \equiv \mathcal{I}[p](d_1, \dots d_n)$;*

also belongs to $\mathcal{T}$.

Intuitively it means that terms are assigned elements in the domain modulo a permutation. It allows to extend the use of the classic combination scheme to I-theories which do not necessarily correspond to first-order theories. Since only one I-theory is needed to be flexible, the requirement is directly satisfied if one of the two I-theories corresponds to a first-order theory:

Theorem 21. *If I-theory $\mathcal{T}$ is the set of models of a first-order theory in a given language $\mathcal{L}$, then $\mathcal{T}$ is flexible on every sort.*

Proof. Given any first-order formula φ, it is straightforward to prove by structural induction that $\mathcal{I}$ is a model for φ if and only if $\mathcal{I}'$ is also a model for φ, where interpretations $\mathcal{I}$ and $\mathcal{I}'$ are related to each other as in Definition 22. $\square$

When combining more than two I-theories, it may seem the "flexible requirement" will impose all but one I-theory in the combination to correspond to first-order theories. Fortunately that is not the case. The following theorem shows that no restriction occurs if any two individual non-flexible I-theories have different sorts:

Theorem 22. *Let $\mathcal{T}$ be the union of the disjoint I-theories $\mathcal{T}_1$ and $\mathcal{T}_2$. If $\mathcal{T}_1$ and $\mathcal{T}_2$ are flexible on τ, then $\mathcal{T}$ is flexible on τ.*

8.2.2 Separation

The set of literals

$$\{x \leq y,\, y \leq x + f(x),\, P(h(x) - h(y)),\, \neg P(0),\, f(x) = 0\}$$

is unsatisfiable, and so is $L_1 \cup L_2$, with

$$\begin{array}{rcl} L_1 & = & \{x \leq y,\, y \leq x + v_1,\, v_1 = 0,\, v_2 = v_3 - v_4,\, v_5 = 0\} \\ L_2 & = & \{P(v_2),\, \neg P(v_5),\, v_1 = f(x),\, v_3 = h(x),\, v_4 = h(y)\}. \end{array}$$

But every literal in L_i $(i = 1, 2)$ contains functions (and constants) and predicates from only one I-theory. Every literal in L_1 is in the language of arithmetic, whereas every literal in L_2 is in the language of uninterpreted predicates and functions. The shared symbols are variables only (and equality). The sets L_1 and L_2 are built from L simply by introducing new variables. The only shared symbols are variable symbols (and equality). This set of literals directly supplies a *separation* for the set of literals in our introductory example:

Definition 23 (separation). *Let $\mathcal{L}$ be the union of disjoint many-sorted first-order languages $\mathcal{L}_1$ and $\mathcal{L}_2$. Let L be a finite set of literals in $\mathcal{L}$. Let $\mathcal{T}$ be a theory in $\mathcal{L}$. The pair (L_1, L_2) is a* separation *of L according to $\mathcal{L}_1$ and $\mathcal{L}_2$ if*

- *L is $\mathcal{T}$-satisfiable if and only if $L_1 \cup L_2$ is $\mathcal{T}$-satisfiable;*
- *L_i is a finite set of literals in language $\mathcal{L}_i$, for $i = 1, 2$. The only shared symbols are variable symbols.*

In fact, given two languages and a set of literals in their union, a separation can always be built, using variable abstraction. The following theorem states that any term in a set of literals can be replaced by a new variable, provided that the definition for this new variable is added to the set of literals:

Theorem 23 (variable abstraction). *Let $\mathcal{T}$ be an I-theory in language $\mathcal{L}$. Assume that variable x is unused in L. The set of literals $L[t/x] \cup \{x = t\}$ is $\mathcal{T}$-satisfiable if and only if L is $\mathcal{T}$-satisfiable, where $L[t/x]$ denotes the set of literals L where every term t has been replaced by x.*

Proof. Any $\mathcal{T}$-model for $L[t/x] \cup \{x = t\}$ is a $\mathcal{T}$-model for L. Any $\mathcal{T}$-model for L directly extends in a $\mathcal{T}$-model for $L[t/x] \cup \{x = t\}$, if x is unused in L. □

This leads to the *separation existence* theorem:

Theorem 24 (separation existence). *Let $\mathcal{L}$ be the union of disjoint many-sorted first-order languages $\mathcal{L}_1$ and $\mathcal{L}_2$. Let L be a finite set of literals in $\mathcal{L}$. There always exists a* separation *of L according to $\mathcal{L}_1$ and $\mathcal{L}_2$.*

Proof. The separation is obtained by a finite number of successive applications of variable abstraction. □

8.2.3 Cooperation of decision procedures

Separating a finite set of literals according to disjoint languages may introduce variables. The only shared symbols between both parts of a separation are variable symbols. An arrangement on the set of shared variable symbols is defined as follow:

Definition 24 (arrangement). *Let $\mathcal{C}$ be a partition of the finite set of variables $\{x_1, \dots x_n\}$ of a many-sorted first-order language. The* arrangement *of variables $\{x_1, \dots x_n\}$ induced by $\mathcal{C}$ is the set of all equalities between variables of the same sort in the same class of $\mathcal{C}$, and of all inequalities between two variables of the same sort in two distinct classes of $\mathcal{C}$.*

For instance, the arrangement of the three variables of the same sort x_1, x_2, x_3 induced by partition $\{\{x_1, x_2\}, \{x_3\}\}$ is $\{x_1 = x_2, x_1 \neq x_3, x_2 \neq x_3\}$. In fact the last inequality is not essential: the set of literals $\{x_1 = x_2, x_1 \neq x_3\}$ is logically equivalent to the preceding arrangement.

The following theorem provides the abstract basis for the combination procedure. With further requirements on the I-theories, it will allow to combine two decision procedures for disjoint languages into one for the union of languages:

Theorem 25 (combination). *Let the I-theory $\mathcal{T}$ in the language $\mathcal{L}$ be the union of the I-theories $\mathcal{T}_1$ and $\mathcal{T}_2$ in disjoint languages $\mathcal{L}_1$ and $\mathcal{L}_2$. For each sort, assume that either $\mathcal{T}_1$ or $\mathcal{T}_2$ is flexible. Let L be a set of literals on $\mathcal{L}$,* *and let (L_1, L_2) be a separation of L according to $\mathcal{L}_1$ and $\mathcal{L}_2$. Let V be the set of common variables in L_1 and L_2.*

Assume[7] *that for every domain τ in $\mathcal{L}_i$ and every term t of sort τ used in L_{2-i} there is an equality $x = t \in L_{2-i}$ where x is a variable. The set of all such variables is V'.*

The set of literals L is $\mathcal{T}$-satisfiable if and only if there exists an arrangement $\mathcal{A}$ of $V \cup V'$ and $\mathcal{T}_i$-models $\mathcal{I}_i$ of $\mathcal{A} \cup L_i$ for $i = 1, 2$ such that, for any common sort τ in $\mathcal{L}_1$ and $\mathcal{L}_2$, the domains associated to τ in $\mathcal{I}_1$ and $\mathcal{I}_2$ have the same cardinality. In that case $\mathcal{A} \cup L_1 \cup L_2$ is $\mathcal{T}$-satisfiable.

Proof. The condition is necessary. Let $\mathcal{I} \in \mathcal{T}$ be a model of L. By definition of separation, there is a model $\mathcal{I}' \in \mathcal{T}$ of $L_1 \cup L_2$. This interpretation $\mathcal{I}'$ perfectly defines an arrangement $\mathcal{A}$ of $V \cup V'$, and $\mathcal{I}'$ is a model of $\mathcal{A} \cup L_1 \cup L_2$, and so of $\mathcal{A} \cup L_i$.

Assume now that the interpretation $\mathcal{I}_i \in \mathcal{T}_i$ makes true $\mathcal{A} \cup L_i$ for $i = 1, 2$. The interpretation $\mathcal{I}_i$ defines a partition $\mathcal{C}_i$ of all terms used in $\mathcal{A} \cup L_i$ such that two terms a and b are in the same class if and only if $\mathcal{I}_i \Vdash a = b$. Consider $c_1 \in \mathcal{C}_1$ and $c_2 \in \mathcal{C}_2$, then $c_1 \cap c_2 \subset V \cup V'$. But $\mathcal{A}$ perfectly defines which elements of $V \cup V'$ are in the same class. It is thus easy to make the partitions $\mathcal{C}_1$ and $\mathcal{C}_2$ coincide on classes containing a same element. Let $\mathcal{C}$ be a partition of terms in partitions $\mathcal{C}_1$ and $\mathcal{C}_2$ such that

- if $c_1 \in \mathcal{C}_1$ and $c_2 \in \mathcal{C}_2$, then $c_1 \cup c_2 \in \mathcal{C}$ if and only if $c_1 \cap c_2$ is non empty;
- for every $c_i \in \mathcal{C}_i$ there exists $c \in \mathcal{C}$ such that $c_i \subset c$. In other words, every class of $\mathcal{C}_i$ is fully contained in one class of $\mathcal{C}$;
- if $c_i \neq c_i' \in \mathcal{C}_i$, and if $c, c' \in \mathcal{C}$ are such that $c_i \subset c$ and $c_i' \subset c'$, then $c \neq c'$.

Partition $\mathcal{C}$ will allow to build an interpretation $\mathcal{I}$ from $\mathcal{I}_1$ and $\mathcal{I}_2$ such that $\mathcal{I} \Vdash \mathcal{A} \cup L_1 \cup L_2$.

First, let's define the domains for $\mathcal{I}$. For every sort τ in language $\mathcal{L}$, at least $\mathcal{T}_1$ or $\mathcal{T}_2$ is flexible. It can thus be assumed that $\mathcal{I}_1$ and $\mathcal{I}_2$ assign the same domain to every common sort. $\mathcal{I}$ naturally assigns this domain for every common sort, whereas it assigns the domain assigned by $\mathcal{I}_i$ for every sort which is only in language $\mathcal{L}_i$.

Second, as for every sort τ in language $\mathcal{L}$, at least $\mathcal{T}_1$ or $\mathcal{T}_2$ is flexible, $\mathcal{I}_1$ and $\mathcal{I}_2$ can be assumed to associate the same element to every variable in $V \cap V'$.

[7] Every separation can be transformed into a separation verifying this condition.

Every class c of $\mathcal{C}$ contains elements of the same sort. If this is a common sort to both I-theories, then $c \cap (V \cup V') \neq \emptyset$, and for every term t in c, $\mathcal{I}[t] = \mathcal{I}_1[t] = \mathcal{I}_2[t]$. If it is not, then every term $t \in c$ belongs to one language $\mathcal{L}_i$, and $\mathcal{I}[t] = \mathcal{I}_i[t]$. □

In Theorem 25, the set of variables on which the arrangement is built is larger than the set of *shared* variables, usually used in the Nelson-Oppen method. This is necessary for the combination with non-stably infinite I-theories, for instance, I-theories having only finite domains. A similar solution was also used in [189], to combine the theory of sets with a theory for the elements, even if this last theory is not stably infinite.

8.2.4 Handling cardinality constraints

The classical way of solving the cardinality constraints like those in Theorem 25 is to consider only *stably infinite* theories. In a many-sorted context, it means both theories should be stably infinite with respect to the set of common sorts.

Definition 25 (stably infinite I-theory). *An I-theory $\mathcal{T}$ in a many-sorted language $\mathcal{L}$ is* stably infinite with respect to a set of sorts S *if every $\mathcal{T}$-satisfiable ground set of literals L in $\mathcal{L}$ has a $\mathcal{T}$-model assigning a domain of cardinality $\aleph_0$ for every sort $\tau \in S$ used in $\mathcal{L}$.*

Many useful I-theories are stably infinite: the I-theory of linear arithmetic on integer, I-theories corresponding to the first-order theories of arrays [172] or lists [142, 143], etc. Also observe that, when talking about I-theories which are sets of models of first-order theories, the Löwenheim-Skolem Theorem[8] allows to relax the condition on domain cardinality in this definition: the cardinalities should be greater or equal to $\aleph_0$. But this remains a very strong requirement. Notably it prevents to combine I-theories with only finite domains.

We will rather adopt a pragmatic view, and study two I-theories in particular. *Compatible I-theories* are I-theories requiring no "cardinality clause" in Theorem 25:

Definition 26 (compatible I-theories). *Two disjoint I-theories $\mathcal{T}_1$ and $\mathcal{T}_2$ in many-sorted languages $\mathcal{L}_1$ and $\mathcal{L}_2$ are* compatible *if for every separation (L_1, L_2) and arrangement $\mathcal{A}$ such that*

- *V is the set of common symbols in L_1 and L_2;*

[8]A formulation of Löwenheim-Skolem Theorem for many-sorted first-order logic can be found in [177].

- *for every sort τ in $\mathcal{L}_i$ and every term t of sort τ used in L_{2-i} there is an equality $x = t \in L_{2-i}$ where x is a variable in V';*
- *$\mathcal{A}$ is an arrangement of $V \cup V'$;*
- *$\mathcal{A} \cup L_i$ is $\mathcal{T}_i$-satisfiable for $i = 1, 2$,*

then there are $\mathcal{T}_i$-models $\mathcal{I}_i$ of $\mathcal{A} \cup L_i$ for $i = 1, 2$ such that $\mathcal{I}_1$ and $\mathcal{I}_2$ assign sets with the same cardinality for every common sort.

If two I-theories are stably infinite with respect to the set of common sorts, then they are compatible. But this is not a necessary condition: many I-theories are not stably infinite, but are compatible with some other I-theories.

Assume that $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible disjoint I-theories such that, for each sort, at least one of them is flexible. Then there is a satisfiability decision procedure for sets of literals in the language of their union if there are satisfiability decision procedures for sets of literals in the language of each I-theory. Indeed, $L_1 \cup L_2$ is unsatisfiable in the union if, for every arrangement $\mathcal{A}$, $\mathcal{A} \cup L_i$ is unsatisfiable for $i = 1$ or $i = 2$. Theoretically, the decision procedure can be, first, to compute the separation, and second, to check every arrangement of the suitable set of variables together with each part of the separation. The original set is unsatisfiable if and only if no arrangement has been found to be satisfiable with both parts of the separation. If the decision procedures for each $\mathcal{T}_i$ are NP-time, then the decision procedure for the union is NP-time: indeed it suffices to choose non deterministically an arrangement $\mathcal{A}$, and check satisfiability of both formulas $\mathcal{A} \cup L_i$ in the respective interpretation sets. In [151] it is shown that the decision problem for conjunctive sets for separation logic (a subset of linear arithmetic in $\mathbb{N}$) with uninterpreted functions is NP-complete whereas both decision problems for separation logic and for uninterpreted functions are polynomial. The combination of decision procedures may thus have a worse complexity than the component decision procedures.

8.2.5 Avoiding arrangement enumeration

Given a set of variables V, there are as many arrangements as partitions for V (at least in a one-sorted logic). The number of partitions, also known as Bell numbers (see for example [125]), grows faster than exponentially [112] with respect to the size of $|V|$. In practice, checking every arrangement is not feasible, even for a small number of variables.[9] Hopefully, another way to

[9] There are more than four million arrangements of 12 variables.

achieve the same goal (that is make both procedures work together) exists: it is based on the exchange of disjunctions of equalities about common symbols:

Theorem 26 (residue). *Assume that the conditions for Theorem 25 are met. Assume that $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible. Assume that Δ is a set of equalities on $V \cup V'$.*

If $\Delta \cup L_1 \cup L_2$ is $\mathcal{T}$-unsatisfiable, then either $\Delta \cup L_i$ is $\mathcal{T}_i$-unsatisfiable for $i = 1$ or $i = 2$, or there exists a disjunction of equalities δ on $V \cup V'$ (called the residue*), such that $\Delta \not\models \delta$ for $i = 1, 2$, and $\Delta \cup L_i \models_{\mathcal{T}_i} \delta$ for $i = 1$ or $i = 2$.*

Proof. Assume that Δ is $\mathcal{T}_i$-satisfiable for $i = 1, 2$. The set of equalities Δ defines a relation $\leftrightarrow$ on $V \cup V'$ such that $x \leftrightarrow x'$ if and only if $x = x' \in \Delta$. This relation also defines a partition $\mathcal{C}$ of $V \cup V'$ such that x and x' are in the same class of $\mathcal{C}$ if and only if $x \leftrightarrow^\star x'$, where $\leftrightarrow^\star$ is the reflexive, symmetric and transitive closure of relation $\leftrightarrow$.

The set γ is the set of all inequalities $x \neq y$ for every variables $x \in c_1$ and $y \in c_2$ where c_1, c_2 are any two disjoint classes of $\mathcal{C}$. The arrangement $\mathcal{A}$ given by $\mathcal{C}$ is logically equivalent to $\Delta \cup \gamma$.

From Theorem 25 and using the fact that $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible, if $\mathcal{A} \cup L_i$ is $\mathcal{T}_i$-satisfiable for $i = 1, 2$, then $\mathcal{A} \cup L_1 \cup L_2$ is $\mathcal{T}$-satisfiable and $\Delta \cup L_1 \cup L_2$ too. It can be assumed $\mathcal{A} \cup L_i$ is $\mathcal{T}_i$-unsatisfiable for $i = 1$ or $i = 2$.

If $\mathcal{A} \cup L_i$ is $\mathcal{T}_i$-unsatisfiable, then

$$\Delta \cup L_i \models_{\mathcal{T}_i} x_1 = y_1 \vee \ldots x_k = y_k$$

if $\gamma = \{x_1 \neq y_1, \ldots x_k \neq y_k\}$. By construction, $\delta =_{\mathrm{def}} (x_1 = y_1 \vee \ldots x_k = y_k)$ is suitable. □

As an example of application of this theorem, consider again the separation

$$\begin{array}{rcl} L_1 & = & \{x \leq y,\ y \leq x + v_1,\ v_1 = 0,\ v_2 = v_3 - v_4,\ v_5 = 0\} \\ L_2 & = & \{P(v_2),\ \neg P(v_5),\ v_1 = f(x),\ v_3 = h(x),\ v_4 = h(y)\} \end{array}$$

From L_1, it can be deduced in $\mathcal{T}_1$ that $x = y$. Then $L_2 \cup \{x = y\} \models v_3 = v_4$, $L_1 \cup \{v_3 = v_4\} \models_{\mathcal{T}_1} v_2 = v_5$ and finally $L_2 \cup \{v_2 = v_5\}$ is unsatisfiable. The set L is thus unsatisfiable. Notice that the exchanged disjunctions of equalities reduce here to single equalities. In fact, the theories generating only single equalities and no disjunctions are, by definition, the convex theories.

As a second example, assume that x, y, z are messages which can either be M_1 or M_2 (but no other). It is natural to consider the I-theory $\mathcal{T}_1$ containing only interpretations on the domain $\{M_1, M_2\}$. Assume that $\mathcal{T}_2$ is the uninterpreted I-theory with interpreted predicates P, Q and R. The set of literals

$$L = \{P(x), \neg P(y), Q(x), \neg Q(z), R(y), \neg R(z)\}$$

is unsatisfiable in the union of $\mathcal{T}_1$ and $\mathcal{T}_2$. Here, the set V' in conditions of Theorem 26 is not empty: it is $\{x, y, z\}$. A separation of L according to the languages of $\mathcal{T}_1$ and $\mathcal{T}_2$ is $(\emptyset, L)$. L is $\mathcal{T}_2$-satisfiable. And so it is for the empty set in $\mathcal{T}_1$. However, $x = y \vee x = z \vee y = z$ is a logical consequence of $\mathcal{T}_1$, and examining every of the three cases leads to conclude unsatisfiability.

From this example, two things must be noticed. First, the set V' in Theorems 25 and 26 is important. Those theorems break if elements are omitted in V'. Second, decision procedures do not always interact by exchanging simple equalities. Some I-theories are not convex, and notably those which admit only interpretations with finite domain of fixed size. If a disjunction $x_1 = y_1 \vee \ldots x_k = y_k$ is generated by $\Delta \cup L_1$ in $\mathcal{T}_1$, it remains to examine the k pairs of literals

$$(\Delta \cup \{x_i = y_i\} \cup L_1, \Delta \cup \{x_i = y_i\} \cup L_2)$$

with $i \in \{1, \ldots k\}$ for satisfiability. The presence of a disjunction introduces a case split, but it is still possible to give sets of literals to the decision procedures and no sets of disjunctions of literals. There is no need for the decision procedures to be able to deal with disjunctions of literals.

Theorem 26 ensures that, for any unsatisfiable set, there exists a sequence of equality (or disjunction of equalities) exchanges that lead to unsatisfiability of one part or another of the separation. Splitting is used when disjunction of equalities are generated. Convex I-theories do not require splitting. If two polynomial time convex theories are combined, the combination is also polynomial time [142]. In general, the combination of (non convex) polynomial time theories might be NP-complete [151].

8.3 Some important (I-)theories

Combining decision procedures for a finite number of I-theories not only requires those I-theories to have a decision procedure. Some I-theories in the combination should be flexible on some sorts in such a way that for any two I-theories in the combination, at least one is flexible for each (common) sort. This requirement is not very restricting, as I-theories corresponding to first-order theories are flexible on every sort (Theorem 21). The requirements on domain cardinality are more restricting: I-theories to be combined should be compatible, but at this point, there is no criterion to easily find compatible I-theories. Unfortunately, there is indeed no simple criterion to find if I-theories are compatible. However, some I-theories have nice properties regarding compatibility.

Besides compatibility with other I-theories, other properties are important for combining I-theories. Convexity has already been mentioned: a convex I-theory will only produce single equalities and will not require splittings. Two other practical concerns are the ability for the decision procedures to produce (disjunction of) equalities, and their ability to accept new literals (i.e., new equalities generated by other I-theories) on the fly.

8.3.1 The uninterpreted I-theory

The empty first-order theory in a language $\mathcal{L}$ defines the I-theory in which every constant, predicate, and function is uninterpreted, and no constraints are put on the domains. This I-theory will be referred to as the uninterpreted I-theory. It is certainly the simplest one, but it is nonetheless important as it allows uninterpreted predicates and functions to be added to the language of a decidable I-theory.

The uninterpreted I-theory is decidable: congruence closure (see Chapter 7) provides an incremental decision procedure. It is also convex. Otherwise, there would exist a set of literals L having a disjunction $x_1 = y_1 \vee \ldots x_n = y_n$ of equalities between variables as a logical consequence, but none of them independently. It can be assumed without loss of generality[10] that $x_1 = y_1 \vee \ldots x_n = y_n$ is a minimal conjunction implied by L, that is, no smaller disjunctive set of the same equalities is implied by L. Now the set of literals $L \cup \{x_3 \neq y_3, \ldots x_n \neq y_n\}$ is satisfiable, and does not imply $x_1 = y_1$. Adding $x_2 \neq y_2$ to this last set keeps it satisfiable, and makes $x_1 = y_1$ true. However, adding an inequality to a satisfiable set of literals either makes it unsatisfiable, or does not involve new class merges in the congruence closure (see Chapter 7).

Besides those nice properties, the uninterpreted I-theory has an optimal behaviour in a combination framework. Classically, only its stable infiniteness is exploited as all theories considered are assumed to be stably infinite. However, it was also noticed in [176] that it has the required properties to be combined not only with stably infinite theories but also with every first-order theory. Here, we show that it can be combined with *any* I-theory, corresponding or not to a first-order theory. The deep reason for this is encapsulated in this lemma:

[10] A minimal disjunction can always be built "decrementally" by suppressing one element at a time.

Lemma 2. *Let $\mathcal{T}$ be the uninterpreted I-theory in a many-sorted language $\mathcal{L}$, and L be a set of literals on $\mathcal{L}$. If $\mathcal{I} = \langle D, I\rangle \in \mathcal{T}$ is a model of L, then there is a model $\mathcal{I}' = \langle D', I'\rangle \in \mathcal{T}$ of L such that, for any sort τ and any cardinality $\kappa \geq |D_\tau|$, $|D'_\tau| = \kappa$ and $|D'_{\tau'}| = |D_{\tau'}|$ for every $\tau' \neq \tau$.*

Proof. Let Δ be a set such that $|\Delta \cup D_\tau| = \kappa$, and such that $\Delta \cap D_{\tau'} = \emptyset$ for every sort τ' of language $\mathcal{L}$. The interpretation $\mathcal{I}' = \langle D', I'\rangle$ is built from $\mathcal{I}$:

- $D'_\tau = D_\tau \cup \Delta$, and $D'_{\tau'} = D_{\tau'}$ for every sort $\tau' \neq \tau$;
- for any constant a, $I'[a] = I[a]$;
- for any variable x, $I'[x] = I[x]$;
- for any function f, and any elements in the domains of appropriate sort $d_1, \dots d_n$, $I'[f](d_1, \dots d_n) = I'[f](d_1, \dots d_n)$ if $\{d_1, \dots d_n\} \cap \Delta = \emptyset$. Otherwise $I'[f](d_1, \dots d_n)$ is any element in the domain of appropriate sort;
- for any predicate p, and any elements in the domains of appropriate sort $d_1, \dots d_n$, $I'[p](d_1, \dots d_n) = I'[p](d_1, \dots d_n)$ if $\{d_1, \dots d_n\} \cap \Delta = \emptyset$. Otherwise $I'[p](d_1, \dots d_n)$ is any value in $\{\top, \bot\}$.

The interpretation $\mathcal{I}'$ gives the same value to every term, atom, and literal in L than the model $\mathcal{I}$ of L. □

A corollary of the preceding Lemma is the stable infiniteness of the uninterpreted I-theory:

Theorem 27. *The uninterpreted I-theory in a language $\mathcal{L}$ is stably infinite on every set of sorts of $\mathcal{L}$.*

Proof. Assume that L is a satisfiable finite set of literals in $\mathcal{L}$. There is an interpretation $\mathcal{I} = \langle D, I\rangle$ of L such that D_τ is finite for every sort τ of $\mathcal{L}$. Using Lemma 2 on each sort successively, it is possible to build from $\mathcal{I}$ a model $\mathcal{I}'$ assigning to every sort a domain of cardinality $\aleph_0$. □

As a consequence, the uninterpreted I-theory is compatible with every stably infinite I-theory. But a much stronger result can be deduced from Lemma 2:

Theorem 28. *The uninterpreted I-theory in a language $\mathcal{L}$ is compatible with every I-theory.*

Proof. Assume that $\mathcal{T}_1$ is the uninterpreted I-theory in language $\mathcal{L}_1$ and $\mathcal{T}_2$ is any I-theory in language $\mathcal{L}_2$ disjoint from $\mathcal{L}_1$. Let (L_1, L_2) and $\mathcal{A}$ be a separation and an arrangement verifying the conditions of Definition 26.

For every common sort τ of $\mathcal{L}_1$ and $\mathcal{L}_2$, the arrangement $\mathcal{A}$ defines a partition of variables of sort τ: two variables of sort τ are in the same class if and only if they are assigned the same value by a model of $\mathcal{A}$. This partition contains a given number k_τ of classes. The models $\mathcal{I}_i$ of $\mathcal{A} \cup L_i$ assign to each common sort τ a domain $D_{i,\tau}$ such that $|D_{i,\tau}| \geq k_\tau$. Furthermore, if $\mathcal{A} \cup L_1$ is satisfiable, then it has a model $\mathcal{I}_1$ which assigns to each common sort τ a domain $D_{1,\tau}$ such that $|D_{1,\tau}| = k_\tau$.

Using Lemma 2 on each common sort successively, it is possible to build from $\mathcal{I}_1$ a model $\mathcal{I}_1'$ assigning to every sort a domain of the same cardinality as interpretation $\mathcal{I}_2$. $\square$

Given any decidable I-theory $\mathcal{T}$ in language $\mathcal{L}$, adding any finite number of uninterpreted functions, predicates, or constants keeps the I-theory decidable, for sets of literals in the resulting language. Remember that this results holds for Boolean combination of atoms, i.e., quantifier-free logic. With quantifiers, the behaviour is completely different: Presburger theory is decidable. However, Presburger with only one monadic uninterpreted predicate is undecidable [104]. The empty theory itself is undecidable for quantified formulas.

On a practical side, the algorithm presented in Chapter 7 deciding conjunction of literals containing only equalities, and uninterpreted predicates and functions can be made equality producing: the process of building the congruence closure is achieved by successively merging classes, and each time two classes are merged, a new equality could be produced and propagated to the other decision procedures. Also it is incremental, and it thus may accept equations generated by other I-theories with only incremental work.

8.3.2 The theory of arrays

The first-order theory of arrays is one of the most popular theories used in combination schemes. A survey of results and history about it can be found in [125, 172]. The theory contains "read-over-write" axioms:

$$\forall a \forall i \forall e \, [\mathrm{read}(\mathrm{write}(a, i, e), i) = e]$$

$$\forall a \forall i \forall j \forall e \, [i \neq j \Rightarrow \mathrm{read}(\mathrm{write}(a, i, e), j) = \mathrm{read}(a, j)]$$

and an extensionality axiom:

$$\forall a \forall b \, [\forall i \, \mathrm{read}(a, i) = \mathrm{read}(b, i) \Rightarrow a = b] \, .$$

Sometimes the theory without the extensionality axiom is considered [172].

In a many-sorted framework, the three previous axioms do not suffice to perfectly define the first-order theory. There are obviously three possible sorts: indices (i, j), arrays (a, b), and elements (e). But some sorts may be merged. It is natural and conservative to consider that the sort of arrays is disjoint from the sorts of elements and indices.[11] However, in numerous practical cases, it may be interesting to merge sorts for indices and elements. Two theories should be thus considered: the two-sorted theory, and the three-sorted theory.

None of both two and three-sorted theories is convex. The following logical consequence provides a clear counterexample:

$$\mathrm{read}(a, i) = e \wedge a = \mathrm{write}(b, j, e) \wedge e' = \mathrm{read}(b, i) \models e = e' \vee i = j$$

The disjunction produced here contains one equality on the sort of elements, and one on the sort of indices. But there are sets of literals producing disjunctions with only equalities between elements, or indices, or arrays, or a mix of all types of equalities.

The I-theory corresponding to the first-order theory of arrays is not compatible with every I-theory like the uninterpreted I-theory is. However, very positive results hold about the compatibility with the I-theory of arrays, thanks to the following lemma:

Lemma 3. *Let $\mathcal{T}$ be a two-sorted or three-sorted I-theory of arrays in a language $\mathcal{L}$ and L be a set of literals on $\mathcal{L}$. If $\mathcal{I} = \langle D, I \rangle \in \mathcal{T}$ is a model of L, then there is a model $\mathcal{I}' = \langle D', I' \rangle \in \mathcal{T}$ of L such that, for sort τ of values (or indices) and any cardinality $\kappa \geq |D_\tau|$, $|D'_\tau| = \kappa$. If $\mathcal{T}$ is a three-sorted I-theory and τ' is the sort of indices (resp. values), then $|D'_{\tau'}| = |D_{\tau'}|$.*

Proof. This is proved in a similar way as for the uninterpreted I-theory (Lemma 2). First assume that $\mathcal{T}$ is the three-sorted I-theory and τ is the sort of values; τ' is the sort of indices and σ the sort of arrays. Let Δ be a set such that $|\Delta \cup D_\tau| = \kappa$, and such that $\Delta \cap D_{\tau'} = \Delta \cap D_\sigma = \emptyset$. The interpretation $\mathcal{I}' = \langle D', I' \rangle$ is built from interpretation $\mathcal{I}$:

- $D'_\tau = D_\tau \cup \Delta$, $D'_{\tau'} = D_{\tau'}$;
- for any constant a, $I'[a] = I[a]$;
- for any variable x, $I'[x] = I[x]$;

[11] Nevertheless elements in a theory of arrays can be arrays. But their language should be disjoint. For instance, read_1, write_1 and read_2, write_2.

- if $v \in D_\tau$, $i \in D_{\tau'}$ and $a \in D_\sigma$, then $I'[\text{read}](a, i) = I[\text{read}](a, i)$ and $I'[\text{write}](a, i, v) = I[\text{write}](a, i, v)$.

This suffices to make $\mathcal{I}'$ a model of L, but it is not yet perfectly defined, and not yet a model of $\mathcal{T}$. For every element a in D_σ, there is an element a' in $D'_\sigma \setminus D_\sigma$ corresponding to an array similar to a but such that a finite number of elements have been replaced by elements of Δ. More formally:

$$\{\text{write}(a, i, v), \text{write}(\text{write}(a, i, v), i', v'), \ldots\} \in D'_\sigma \setminus D_\sigma$$

for $\{v, v', \ldots\} \subset D'_\tau \setminus D_\tau$ and $\{i, i', \ldots\} \subset D_{\tau'}$. Special care must be taken to verify the extensionality and read-over-write axioms. Functions "read" and "write" are extended in the natural way.

The two-sorted case, and the three-sorted case when τ is the sort of indices are similar. Also notice that, if $\kappa \leq \aleph_0$, $|D_\tau| \leq \aleph_0$, $|D_{\tau'}| \leq \aleph_0$, and $|D_\sigma| \leq \aleph_0$, then $|D'_\sigma| \leq \aleph_0$. □

The stable-infiniteness is a direct consequence of preceding lemma:

Theorem 29. *The two-sorted (three-sorted) I-theory of arrays in language $\mathcal{L}$ is stably infinite on every set of sorts.*

Proof. Assume that L is a satisfiable finite set of literals in $\mathcal{L}$. There is an interpretation $\mathcal{I} = \langle D, I \rangle$ of L such that D_τ is finite for every sort τ of $\mathcal{L}$. Using Lemma 3 on the sort of indices and values (successively on the sorts of indices and values in the case of the three-sorted theory), it is possible to build from $\mathcal{I}$ a model $\mathcal{I}'$ assigning to the sort(s) of indices and values a domain of cardinality $\aleph_0$. Taking into account the remark at the end of the proof of Lemma 3, the sort of arrays is assigned a set of cardinality at most $\aleph_0$. Given an array a and an index i, $\text{write}(a, i, v) \neq \text{write}(a, i, v')$ if and only if $v \neq v'$. The sort of arrays is also assigned a set of cardinality $\aleph_0$. □

It is often required to use the I-theory of arrays with a finite domain for elements or indices. Its stable-infiniteness is of no help in those cases. To guarantee that it is still possible to combine the I-theory of arrays with non-stably infinite theories, it is further required that no inequality is used on the array sort. This can be achieved by replacing, in the separation part in the language of arrays, every inequality $t_1 \neq t_2$ where t_1 and t_2 are array terms by $\text{read}(t_1, i) \neq \text{read}(t_2, i)$ where i is a new index variable. As a consequence, new variables may have to be introduced in the set V' to verify conditions of Theorem 25.

Theorem 30. *The two-sorted and three-sorted I-theories of arrays are compatible with any I-theory in a language $\mathcal{L}$, if the sort of arrays is not a sort of $\mathcal{L}$, as long as no inequality is used on the array sort.*

Proof. Assume that $\mathcal{T}_1$ is the two-sorted I-theory of arrays in language $\mathcal{L}_1$ and $\mathcal{T}_2$ is any I-theory in language $\mathcal{L}_2$ disjoint from $\mathcal{L}_1$. Let (L_1, L_2) and $\mathcal{A}$ be a separation and an arrangement verifying the conditions of Definition 26.

For every common sort τ of $\mathcal{L}_1$ and $\mathcal{L}_2$, the arrangement $\mathcal{A}$ defines a partition of variables of sort τ: two variables of sort τ are in the same class if and only if they are assigned the same value by a model of $\mathcal{A}$. This partition contains a given number k_τ of classes. The models $\mathcal{I}_i$ of $\mathcal{A} \cup L_i$ assign to each common sort τ a domain $D_{i,\tau}$ such that $|D_{i,\tau}| \geq k_\tau$. As opposed to the empty theory, it is not direct to guarantee that, if $\mathcal{A} \cup L_1$ is satisfiable, then it has a model $\mathcal{I}_1$ (in the array I-theory) which assigns to each common sort τ a domain $D_{1,\tau}$ such that $|D_{1,\tau}| = k_\tau$.

Consider the two-sorted theory. Every term of the common sort τ used in L_1 is equal to a term used in $\mathcal{A}$. Assume that $d \in D_{1,\tau}$ is associated to no term used in $\mathcal{A}$. Let $\mathcal{I}'_1$ be the same as $\mathcal{I}_1$ except that its domain for sort τ, $D'_{1,\tau}$ is $D_{1,\tau} \setminus \{d\}$, and $\mathcal{I}'_1[\mathrm{read}(a,i)] = \mathcal{I}_1[\mathrm{read}(a,i)]$ if $\mathcal{I}_1[\mathrm{read}(a,i)] \neq d$, or d' otherwise, where d' is an element of $D'_{1,\tau}$ chosen once and for all. The interpretation $\mathcal{I}'_1$ is still a model of every literal in L_1, and of both "read-over-write" axioms, as those axioms are only universally quantified. If the formula $\forall i\, \mathrm{read}(a,i) = \mathrm{read}(b,i) \Rightarrow a = b$ is made false by $\mathcal{I}'_1$ for elements a and b of the array domain in $\mathcal{I}'_1$, then $\mathrm{read}(a,i) = \mathrm{read}(b,i)$ for every i. It is possible to build $\mathcal{I}''_1$ from $\mathcal{I}'_1$ such that only one representative of all such elements is kept. Interpretation $\mathcal{I}''_1$ still makes true both "read-over-write" axioms, and also the extensionality axiom. Assuming no inequality is used on the array sort in L_1, $\mathcal{I}''_1$ is also a model of every literal in L_1.

The same argument can be applied in the three-sorted case, for both elements and indices. Thus, if $\mathcal{A} \cup L_1$ is $\mathcal{T}_1$-satisfiable, it has a model $\mathcal{I}_1 \in \mathcal{T}_1$ which assigns to each common sort τ a domain $D_{1,\tau}$ such that $|D_{1,\tau}| = k_\tau$. Using Lemma 3 it is thus possible to build a model $\mathcal{I}_1$ assigning to every common sort a domain of the same cardinality as interpretation $\mathcal{I}_2$. $\square$

Notice that the previous theorem does not imply that the I-theory of arrays is not compatible if the array sort is a common sort. It only states that if the array sort is a common sort, then the other I-theory must have a nice property to guarantee they are compatible. For instance, the array sort can be the sort of elements for another array I-theory. Also, the I-theory of arrays is compatible with the uninterpreted I-theory, even if both languages contain the sort of arrays.

The satisfiability problem for sets of literals in the language of a theory of arrays is NP-complete [172].[12] In [172] the occurrences of "write"

[12] The satisfiability problem for first-order formulas with quantifiers in the language of the theory of arrays is undecidable [125], as for the empty theory.

are eliminated and every atom read$(a, \cdot)$ is translated to an application of an uninterpreted function associated to a, and finally congruence closure is used. This decision procedure is equality producing. In [13] it is proven that the superposition calculus is a satisfiability procedure for the (one-sorted) theory of arrays. The approach of [13] is such that equality propagation is unnecessary, unlike in the classical Nelson-Oppen framework.

8.3.3 The theory of lists

Another popular theory is the first-order theory of lists. In fact, there have been several axiomatisations for the theory of lists [142], with their own weaknesses. Notably, the associated decision problem for set of literals in the language of three of those is NP-complete for one of them [136], quadratic [136] for another, and linear for the last one [143]. Since each of those three theories imperfectly models the reality of lists at a similar level, the most popular one is the one having the most efficient decision procedure [125]. Its axioms are, one construction axiom

$$\forall x\,[\,\mathrm{cons}(\mathrm{car}(x), \mathrm{cdr}(x)) = x\,]$$

two selection axioms

$$\begin{array}{c}\forall x \forall y\,[\,\mathrm{car}(\mathrm{cons}(x, y)) = x\,]\\ \forall x \forall y\,[\,\mathrm{cdr}(\mathrm{cons}(x, y)) = y\,]\end{array}$$

and finally, an infinite number of acyclicity axioms:

$$\begin{array}{c}\forall x\,[\,\mathrm{car}(x) \neq x\,]\\ \forall x\,[\,\mathrm{cdr}(x) \neq x\,]\\ \forall x\,[\,\mathrm{car}(\mathrm{cdr}(x)) \neq x\,]\\ \vdots\end{array}$$

Those last axioms imply an infinite cardinality of the domain of lists. This theory is convex [143].

Like in the previous example, one might want to consider two different theories: the theory with one single sort, and the two-sorted theory with one sort for elements and one sort for lists. In that last case, the acyclicity axioms mixing “car” and “cdr” functions are omitted. Elements may be lists, but from another theory.

Both I-theories corresponding to the one and two sorted theories of lists are flexible on every sort, thanks to Theorem 21. The Löwenheim-Skolem Theorem also guarantees they are stably infinite.

As a final remark about lists, a satisfiability procedure for lists with linear arithmetic on integer including the "length" function[13] has been given in [187].

8.3.4 Linear arithmetic

The I-theories of linear arithmetic on reals, rationals or integers are built on the language with function $+$ and predicate $<$, with the domain respectively interpreted as $\mathbb{R}$, $\mathbb{Q}$ or $\mathbb{Z}$. Multiplication of a variable by an integer or rational constant is also admitted as it reduces (after the multiplication of the constraint by a chosen integer in the case of rational coefficients) to the sum of this variable a certain amount of times with itself. These I-theories are not flexible. However, they can be combined with stably infinite first-order theories, thanks to the Löwenheim-Skolem-Tarski Theorem [71].[14]

There has been a great deal of research for methods to deal with linear constraints on $\mathbb{R}$, $\mathbb{Q}$ or $\mathbb{Z}$ [37]. If the integration of the convex I-theories (i.e., linear arithmetic with order on $\mathbb{R}$ and $\mathbb{Q}$, and linear arithmetic without order on $\mathbb{Z}$) is quite simple [73, 171], the practical integration of the non-convex I-theory of linear arithmetic with order on $\mathbb{Z}$ into a Nelson-Oppen framework is still problematic. The few works on the subject [28, 29] remain unclear about how to generate efficiently the disjunctions of equalities. This is a cause of incompleteness for the integration of this I-theory into a combination framework. We believe that automata-based decision procedures [116, 184, 186, 38] could fill this gap as they not only provide the answer to the satisfiability problem, but also give a representation of all solutions of a set of constraints. The procedure in [38] also decides linear arithmetic for mixed integers and reals.

8.3.5 Other useful I-theories

In a verification context, proof obligations contain terms corresponding to processes, messages, labels, etc. those sets being essentially finite. For instance, two processes could collaborate by exchanging messages, every message being in the set $\{M_1, \dots M_n\}$. In that case, the domain for the message sort should also be $\{M_1, \dots M_n\}$. A classical solution for this is to augment verification conditions with formulas $M_i \neq M_j$ $(i, j \in \{1, \dots n\}, i \neq j)$ and $\forall x\, (x = M_1 \vee \dots x = M_n)$, where x is in the message sort. A simpler solution

[13] The theories of lists with length function and linear arithmetic on integer are not disjoint: the axiom $\forall x, y\, (\text{length}(\text{cons}(x, y)) = 1 + \text{length}(y))$ contains functions from both theories.

[14] The theorem should be adapted to many-sorted first-order logic.

consists in using the Nelson-Oppen framework to add a decision procedure for this. Notice however that an I-theory assigning a domain of a finite given size to a sort is essentially non-convex: for instance, given a set $\{x_1, \ldots x_{n+1}\}$ of message variables, then at least two of those variables are equal. The size of the produced clauses is quadratic with respect to n.

Control information (see Section 4.5) is another concept for which finite sets of elements might be used. As previously, a decision procedure for terms corresponding to control information may be used, for instance to ensure that every term has a value in the set of three elements $\{\text{think}, \text{wait}, \text{critic}\}$. Again, this decision procedure might generate large disjunctions of equalities. There is an alternative: if predicates (for instance $p_1, \ldots p_n$) with a process as argument are used to model locations, then the decision procedure ensuring that every process is in one location:

$$\forall x\, (p_1[x] \vee \ldots p_n[x])$$

and that no process is ubiquitous:

$$\begin{gathered}\forall x\, (p_1[x] \Rightarrow \neg\, (p_2[x] \vee \ldots p_n[x]))\\ \vdots \\ \forall x\, (p_{n-1}[x] \Rightarrow \neg p_n[x])\end{gathered}$$

is convex, and does not even generate equations. This solution is thus preferable.

8.3.6 User-defined theories

In [13], it is shown that the superposition calculus [138] provides decision procedures for sets of literals containing predicates and functions from various first-order theories, and notably for the empty theory and first-order theories of lists, encryption, arrays and finite sets. Furthermore, the superposition calculus also provides in a direct manner a decision procedure for the combination of such theories. Comforted by those very positive aspects, the haRVey prover [67] simply uses the superposition calculus as the decision procedure for first-order literals on top of the propositional method, and does not implement a Nelson-Oppen scheme to combine decision procedures. In a way, the haRVey prover accepts user-defined theories. The disadvantage of this approach is that it is not easily extended to deal with, for instance, linear arithmetic, since linear arithmetic itself is not handled easily by the superposition calculus. An ideal solution would be to combine, within a Nelson-Oppen framework, decision procedures based on the superposition calculus with other (I-)theories. Unfortunately, at the present time, there are some practical problems for this, since there is no implementation of the superposition calculus that is equality producing, and that accepts literals incrementally.

8.4 Proof minimalization

It has been shown in Chapter 6 that the decision procedures for sets of literals should have some properties, and notably be proof producing in order to nicely integrate with Boolean methods. The decision procedure for uninterpreted functions and predicates of Chapter 7 is proof producing. If we want the combination scheme to be proof producing, every decision procedure in the scheme should be proof producing.

Notice that a proof that a set of literals L implies, let's say, equality of x and y can simply be the valid formula $L \Rightarrow x = y$. However L may (and will in practice) contain many useless literals. Each of those literals may prevent the proving process to take advantage of the fact that some literals in L imply $x = y$. Any efficient method helping in eliminating those unnecessary literals in the proof is thus interesting. The following method applies to any incremental decision procedure for any theory. It has been presented independently in [79] and [57].

Let $H = \{h_1, \ldots h_n\}$ be the unsatisfiable set of literals in the language of a given decidable theory. If the decision procedure is incremental, then it is possible to extract a minimal unsatisfiable set out of H. Let $H_0 = \emptyset$. For all $i \geq 0$, let k_i be the smallest j such that $H_i \cup \{h_r \,|\, 1 \leq r \leq j\}$ is unsatisfiable. An incremental decision procedure directly provides this information; all literals in H_i are first given to the decision procedure, then $h_1, \ldots h_n$ are added one by one until the set of all added constraints is found unsatisfiable. The last added constraint is thus h_{k_i}. Let H_{i+1} be $H_i \cup \{h_{k_i}\}$.

Theorem 31. *For all i,*

- *H_i contains i elements;*
- *$H_i \cup \{h_r : 1 \leq r \leq k_i\}$ is unsatisfiable;*
- *$H_{i+1} \cup \{h_r \,|\, 1 \leq r < k_i\}$ is unsatisfiable;*
- *for every $h \in H_{i+1}$, the set $(H_{i+1} \setminus \{h\}) \cup \{h_r \,|\, 1 \leq r < k_i\}$ is satisfiable.*
- *the sequence $k_0, k_1, \ldots$ is strictly decreasing.*

Proof. If $H = \{h_1, \ldots h_n\}$ is unsatisfiable, the theorem is true for $i = 0$. Indeed from the definition of k_0, $\{h_r : 1 \leq r \leq k_0\}$ is unsatisfiable. As $H_1 = \{h_{k_0}\}$, the set $H_1 \cup \{h_r : 1 \leq r < k_0\}$ is also unsatisfiable. Finally, $\{h_r : 1 \leq r < k_0\}$ is satisfiable, otherwise k_0 is not the smallest j such that $\{h_r : 1 \leq r \leq j\}$ is unsatisfiable.

By induction, the theorem is also true for $i > 0$. The inductive hypothesis states that $H_i \cup \{h_r : 1 \leq r < k_{i-1}\}$ is unsatisfiable, so k_i exists and is strictly smaller than k_{i-1}. The definition of k_i states that $H_i \cup \{h_r : 1 \leq r \leq k_i\}$ is unsatisfiable. The sets $H_i \cup \{h_r : 1 \leq r \leq k_i\}$ and $H_{i+1} \cup \{h_r : 1 \leq r < k_i\}$ are equal.

Finally, given $h \in H_{i+1}$, either $h \in H_i$ or $h = h_{k_i}$. If $h \in H_i$, the set $H_{i+1} \setminus \{h\} \cup \{h_r : 1 \leq r < k_i\}$ is satisfiable as it is included in $H_i \setminus \{h\} \cup \{h_r : 1 \leq r < k_{i-1}\}$ which is satisfiable. If $h = h_{k_i}$, then $H_{i+1} \setminus \{h\} = H_i$, and $H_{i+1} \setminus \{h\} \cup \{h_r : 1 \leq r < k_i\}$ is satisfiable from the definition of k_i. □

Corollary 4. *Let H_f be the last element of the sequence of sets H_i. H_f is a minimal unsatisfiable set and it contains f elements.*

The practical success of this procedure is due to the fact that minimal unsatisfiable sets often contain very few literals. If m is the maximum number of those necessary literals in a set H, and if t is the time to decide if H is unsatisfiable, then the time to get a minimal set from H is $\mathcal{O}(m \times t)$ in the worst case.[15] As m is small, this remains acceptable, and anyway, an approximation H_k with $k < f$ suffices.

[15] The procedure is order-sensitive. The worst case corresponds to all necessary constraints being added late.

Chapter 9

Instantiation procedure

In Chapter 8 we presented a method to build a decision procedure for sets of literals containing functions and predicates from various disjoint decidable theories. This method, used with techniques presented in Chapter 6 and Chapter 7, allows to design a decision procedure for *ground* formulas in a language containing interpreted functions and predicates from various disjoint theories, as well as uninterpreted functions and predicates. In this chapter, we investigate *instantiation*, i.e., how quantified formulas can be freed form quantifiers and be refuted (or proved) with the help of the decision procedure for ground formulas.

Instantiation lifts in a direct manner the applicability of the previous methods for quantifier-free formulas to quantified formulas. It permits to handle quantified formulas containing interpreted predicated and functions from various theories, whereas other theorem proving methods (for example, resolution [18]) do not easily accept interpreted predicates and functions: even equality requires specialized methods [138]. Instantiation is also a natural tool to use in the context of verification of parameterized systems: for instance, intuitive proofs of concurrent algorithms with a finite but parameterized set of processes usually consider the properties verified by a small number of well chosen instances [101, 102]. When formally verifying proof obligations, this means that the number of required instances of the quantified formulas is usually small. The adequacy of instantiation in verification condition validation was already noticed for STeP [36]. In [65], a generic method for first-order equational logic (superposition) is used as an instantiation method. More generally, there is a renewed interest in instantiation as an alternative to resolution [87, 108].

After introducing the necessary notions (Section 9.1), we examine the completeness of instantiation when using interpreted functions and predicates. The method introduced in the previous section provides complete decision procedures for rich quantifier-free languages, for instance, the language of linear arithmetic with uninterpreted functions. However, it is well known that the corresponding quantified language is undecidable [104]. At best, a refutation (or proof) method for such a quantified language would be complete, that is, it would eventually refute any formula that is false (respectively true) for every interpretation giving the intended meaning to the interpreted symbols. In Section 9.2, we show that an instantiation based method cannot provide a complete refutation (or proof) method for the language of quantified formulas with linear arithmetic and uninterpreted functions, despite the fact that the quantifier-free problem is decidable. In other words, considering only a finite number of instances of a quantified formula is a cause of incompleteness of instantiation-based refutation methods, in presence of interpreted predicates and functions.

Unlike in Chapter 8, there are strong theoretical limits for handling quantified formulas in a non first-order I-theory $\mathcal{T}$. The pragmatic decision taken here is to study the quantified formulas in a first-order theory $\mathcal{T}^{\mathrm{FOL}}$. Those theories may imperfectly describe the interpreted predicates, for example when dealing with arithmetic and uninterpreted functions: every formula of the first-order theory $\mathcal{T}^{\mathrm{FOL}}$ is a truth of $\mathcal{T}$, but some models of $\mathcal{T}^{\mathrm{FOL}}$ may not belong to $\mathcal{T}$.[1] This problem reflects the incompleteness of the instantiation method. However, if the method succeeds in refuting a formula, then the formula is unsatisfiable in $\mathcal{T}$. Furthermore, working within a first-order theory is sufficient to prove the verification conditions met in practice.

In [100], a refutation method based on selection of hypotheses is presented for the propositional framework. The heuristic method presented in Section 9.4 for ground logic, and in Section 9.5 for quantified logic, tends to provide a similar method in a first-order framework. It has been implemented in the tool CAVEAT, and gives good results when considering proof obligations from the verification of various algorithms.

9.1 Preliminaries

Given a many-sorted language $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, r, d \rangle$, a substitution[2] is a mapping from variables in $\mathcal{V}$ onto terms of $\mathcal{L}$. The term $x\sigma$ associated to the variable x by the substitution σ is a τ-term if x is of sort τ. The *domain* of σ is defined as the set of variables modified by the substitution, i.e.,

$$\mathrm{Dom}(\sigma) = \{x \mid x \in \mathcal{V} \text{ and } x\sigma \neq x\}$$

[1] In [51], a set of interpretations is an *elementary class* if there exists a first-order theory T such that the set of interpretations is exactly the models of T.

[2] A general discussion about substitutions and unification can be found in [16].

whereas its *range*[3] is the set of all terms that are images of variables in the domain of the substitution:

$$\mathrm{Ran}(\sigma) = \{x\sigma \mid x \in \mathrm{Dom}(\sigma)\}\,.$$

A substitution extends naturally to a transformation of terms and quantifier-free formulas. Given a term (or formula) φ, the term (formula) $\varphi\sigma$ is obtained from φ by simultaneously replacing every variable in φ present in the domain of σ by the corresponding term in the range of σ.

The *composition* $\sigma\theta$ of two substitutions σ and θ is the substitution such that, for each term t, $t\,(\sigma\theta) = (t\sigma)\,\theta$. The composition of two substitutions is a substitution, and there is no ambiguity to note $t\,(\sigma\theta) = (t\sigma)\,\theta = t\sigma\theta$. A substitution σ is *idempotent* if $\sigma\sigma = \sigma$; a substitution is idempotent if and only if no term in its range contains variables in its domain. If a substitution σ has a finite domain $\{x_1, \dots, x_n\}$, it can be denoted by $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$, with $t_i = x_i\sigma$. If σ is furthermore idempotent, it is equal to the substitution $\{x_1 \mapsto t_1\} \dots \{x_n \mapsto t_n\}$, also noted $[x_1 \mapsto t_1; \dots x_n \mapsto t_n]$; it is equal to any[4] composition of the n substitutions having a single variable as domain.

Every term in the range of a *ground substitution* is a ground term. Ground substitutions are trivially idempotent; a ground substitution with $\{x_1, \dots x_n\}$ as domain may thus be written as a composition of the n substitutions with $\{x_i\}$ as domain ($i \in \{1, \dots n\}$).

Assume that φ is a closed Skolem formula (in a refutation context) in a many-sorted language $\mathcal{L}$. As long as two quantifiers in φ do not bind the same variable, the (weak) quantifiers in φ all have a universal semantic, and they can simply be omitted: all variables are implicitly universally quantified. If φ' is formula φ where every quantifier has been removed, an *instance* of φ is a formula $\varphi'\sigma$ where σ is a substitution with a domain included in the set of variables in φ'. Formula $\varphi'\sigma$ is a *ground instance* if it is a ground formula, i.e., if σ is a ground substitution with a domain including all variables in φ.

9.2 Limits of instantiation-based methods

A set of first-order axioms corresponding to the theory of natural numbers with addition are the four axioms for addition:

- $\forall x\, [x + 1 \neq 0]$

[3] Also called codomain.

[4] Indeed the order of substitutions $\{x_i \mapsto t_i\}$ does not matter.

- $\forall x \forall y\, [x + 1 = y + 1 \Rightarrow x = y]$
- $\forall x\, [x + 0 = x]$
- $\forall x \forall y\, [(x + y) + 1 = x + (y + 1)]$

and the induction axiom schema $[P(0) \wedge \forall x\, [P(x) \Rightarrow P(x + 1)]] \Rightarrow \forall x P(x)$. This set of axioms is complete (see for example [74]): every closed first-order formula using only the function $+$, and the constants 0 and 1 is either a logical consequence of this set of axioms, or its negation is. As every axiom is also a truth of linear arithmetic on $\mathbb{N}$, a closed formula using only function $+$, and constants 0 and 1 is a truth of linear arithmetic on $\mathbb{N}$ if and only if it is a logical consequence of the set of axioms.

The first-order theory obtained by augmenting the previous set of axioms with two axioms for multiplication:

- $\forall x\, [x.0 = 0]$
- $\forall x \forall y\, [x.(y + 1) = x.y + y]$,

known as the theory of Peano arithmetic, is essentially incomplete. Gödel's incompleteness Theorem (see for example [74]) states that no consistent first-order theory containing this set of axioms is complete. Given such a theory, there always exists a closed first-order formula using only addition and multiplication, and constants 0 and 1, which is neither valid nor unsatisfiable. In other words, no first-order theory containing those axioms is sufficient to prove every truth of arithmetic expressed as a first-order formula.

Let $\mathcal{T}_{\mathrm{P}}^{\mathrm{FOL}}$ be the first-order theory containing all previous axioms for addition and the induction axiom schema. Assume that $\mathcal{T}_{\mathrm{P}}$ is the I-theory (in the sense introduced in Section 8.2) containing the interpretation on $\mathbb{N}$ assigning the usual meaning to $+$, 0, and 1. From the completeness of $\mathcal{T}_{\mathrm{P}}^{\mathrm{FOL}}$, a closed formula using only function $+$, and constants 0 and 1 is unsatisfiable in $\mathcal{T}_{\mathrm{P}}^{\mathrm{FOL}}$ if and only if it is unsatisfiable in $\mathcal{T}_{\mathrm{P}}$. A refutation method for formulas in $\mathcal{T}_{\mathrm{P}}^{\mathrm{FOL}}$ can be built by first Skolemizing formulas augmented by the axioms, using instantiation based on the Herbrand's Theorem (Theorem 2 and Corollary 1), and then using a quantifier-free decision procedure for first-order logic with uninterpreted functions (see Chapters 6 and 7).

The four axioms for addition in $\mathcal{T}_{\mathrm{P}}^{\mathrm{FOL}}$ do not contribute to the Herbrand sets: they are already in Skolem form, and thus do not involve the appearance of Skolem functions. Using $\mathcal{T}_{\mathrm{P}}$, a refutation method can be built by first adding to the formula the necessary induction axioms, Skolemizing, using Herbrand-based like instantiation, and finally studying the set of quantifier-free instances in $\mathcal{T}_{\mathrm{P}}$ (see Chapters 6 and 7).

One question arises: is this method still complete for studying — in the I-theory $\mathcal{T}_{\mathrm{P}}$ — quantified formulas in a language including, for instance, linear arithmetic and uninterpreted predicates? Unfortunately, the answer is negative. Consider the formula φ built in a language using $+$, 0, 1, and an uninterpreted binary function m:

$$\varphi = \forall x\,[m(x,0) = 0] \wedge \forall x \forall y\,[m(x, y+1) = m(x,y) + y] \wedge \varphi.$$

Any model in $\mathcal{T}_{\mathrm{P}}$ of φ assigns to m the multiplication function on $\mathbb{N}$. It has been shown in Chapter 8 that there is a complete method for dealing in $\mathcal{T}_{\mathrm{P}}$ with quantifier-free formulas in the combination of languages of linear arithmetic and uninterpreted functions. If the preceding instantiation-based refutation method were complete for quantified formulas in a language containing $+$, 0, 1 and uninterpreted functions, then this would contradict Gödel's incompleteness Theorem. The instantiation-based refutation method is essentially incomplete when used on formulas containing interpreted functions of linear arithmetic *and* uninterpreted functions. Explicitly augmenting those formulas with the required induction axioms does not solve the problem.

9.3 Instantiation for verification conditions

The verification conditions generated when validating invariants (see Chapter 4) have a particular structure: in practice, the verification condition associated to the Hoare triple $\{I\}\,\tau\,\{I\}$ is (see section 4.7) of the form $(I \wedge A) \Rightarrow I'$, where I' is obtained by simple syntactical transformations from I, and A is a set of small assertions encapsulating the meaning of transition τ into the verification condition. Invariant candidates are often conjunctions of properties which must be preserved by the system. Formulas I and I' are thus conjunctions of small formulas. This means that the proof obligations are essentially of the form $H \Rightarrow (C_1 \wedge \dots C_n)$, where H is is a (large) conjunctive set of quantified formulas called hypotheses and $C_1, \dots C_n$ are quantified formulas called conclusions. It is useful to split the proving task of $H \Rightarrow (C_1 \wedge \dots C_n)$ into n simpler proving tasks of formulas $H \Rightarrow C_i$ with $i \in \{1, \dots n\}$, that is into n refutation tasks of formulas $H \wedge \neg C_i$ with $i \in \{1, \dots n\}$.

In Section 9.2, it was noticed that a refutation method using instantiation is essentially incomplete for formulas mixing uninterpreted and interpreted functions and predicates, notably from linear arithmetic on $\mathbb{N}$. The pragmatic decision taken here is to study the quantified verification conditions within a first-order theory for describing the interpreted predicates and functions. This also means that Theorem 2 can be used. A complete refutation method for a set $\{h_1, \dots h_n, \neg C\}$ can simply be to generate one by one the instances of the quantified formulas $h_1, \dots h_n, \neg C$; the Herbrand sets are computed from functions used in $h_1, \dots h_n, C$, and in the first-order theory, all formulas being assumed to be in Skolem form (see Section 3.3). Theorem 2 together with the compactness theorem ensures that, if $\{h_1, \dots h_n, \neg C\}$ is unsatisfiable, then the process will eventually[5] generate a finite unsatisfiable set of instances. Practically, the ground decision procedure used with the instantiation method should be incremental: it must be possible to add, with only incremental work, new instances to the set of instances checked for satisfiability.

When proving formula $H \Rightarrow C$, it is reasonable to assume that H is satisfiable. Indeed, H is the conjunction of the invariant, transition guards, and formulas defining some symbols in C with respect to the symbols in the invariant. Unsatisfiability of H essentially means the transition is useless. It may also mean that the invariant in H is unsatisfiable, i.e., this invariant is useless and it is not a logical consequence of the initial conditions. Under the assumption that H is satisfiable, a strategy to find instances can be built by first guessing ground instances for C, and then use models of C as an information to find the instances of formula in H which may lead to the refutation of the set $H \cup \{\neg C\}$.

9.4 Hypothesis selection for ground logic

The goal of this chapter is to solve a practical problem. From a theoretical point of view, instantiation provides a complete refutation method for studying first-order formulas with a first-order theory describing some interpreted predicates and functions. In practice, the number of instances might be infinite, or very large. We propose a simple method to guide the instantiation process, that is, to select the interesting instances. In this section, a preliminary study for ground formulas is presented. This is motivated by the fact that a finite set of Skolemized quantified formulas can be seen as a compact notation for a large or countably infinite set of ground formulas. The results in this section will be lifted to logic with quantifiers in the next section.

Satisfiability checkers for ground formulas, with uninterpreted and interpreted predicates and functions usually take a finite set S of ground formulas as input. Checkers decide if this set of clauses is satisfiable or not. When the set S is satisfiable, checkers usually return a satisfiable subset L of all literals used in the set of formulas. Any model for this set of literals is also a model for the set of ground formulas ($L \models_{\mathcal{T}} S$, where $\mathcal{T}$ is the theory describing the

[5] It is required for the instantiation method to eventually generate every instance.

interpreted predicates and functions in L and S). Essentially, this set is the set of literals labelling a satisfiable branch in the Davis-Putnam search tree (see Chapter 6). This set L may be used to guide the instantiation process. If S' is the set of required instances such that $S \cup S'$ is unsatisfiable in $\mathcal{T}$, then $S' \cup L$ is unsatisfiable.

To simplify the discussion we will now assume that the considered formulas are clauses. Let $\overline{\ell}$ denote the literal which is complementary to ℓ. The following simple theorem is the basis for the formula selection heuristic:

Theorem 32. *If*

- *S is a $\mathcal{T}$-satisfiable set of ground clauses;*
- *L is a set of ground literals such that $L \models_{\mathcal{T}} S$;*
- *h is a ground clause;*
- *there is a literal $\ell \in h$ such that $L \not\models_{\mathcal{T}} \overline{\ell}$,*

then $L \cup \{\ell\}$ and $S \cup \{h\}$ are $\mathcal{T}$-satisfiable. Furthermore $L \cup \{\ell\} \models_{\mathcal{T}} S \cup \{h\}$.

Proof. The proof is straightforward: as $L \not\models_{\mathcal{T}} \overline{\ell}$, $L \cup \{\ell\}$ is $\mathcal{T}$-satisfiable. $L \cup \{\ell\} \models_{\mathcal{T}} S \cup \{h\}$ follows immediately from $\ell \models_{\mathcal{T}} h$. □

It can also be formulated in another way:

Corollary 5. *If*

- *S is a $\mathcal{T}$-satisfiable set of ground clauses;*
- *L is a set of ground literals such that $L \models_{\mathcal{T}} S$;*
- *h is a ground clause such that $S \cup \{h\}$ is $\mathcal{T}$-unsatisfiable,*

then $L \models_{\mathcal{T}} \overline{\ell}$ for every $\ell \in h$.

Notice that the only knowledge used to select a clause to augment S is one class of models of S. Nothing guarantees that this class contains all models. This can be shown with a simple propositional example: the set of literals $L = \{p, r\}$ makes the set of clauses $S = \{p \vee q, r \vee t\}$ true. Every literal of the clause $h = \neg p \vee \neg r$ is falsified by L, however adding h to S does not make the set unsatisfiable.

Corollary 5 provides a straightforward method to easily detect the final clause to add to the set S to make it unsatisfiable. Adding one clause does not always suffice. Assume $S \cup \{h_1, \ldots h_n\}$ is $\mathcal{T}$-unsatisfiable, and $\ell_1 \in h_1, \ldots, \ell_n \in h_n$, then $L \models_{\mathcal{T}} \neg(\ell_1 \wedge \ldots \ell_n)$. The number of those formulas

$\ell_1 \wedge \ldots \ell_n$ might be very large: for instance, if each clause h_i contains 2 literals, then there are 2^n possibilities for $\neg(\ell_1 \wedge \ldots \ell_n)$. This motivates the need for selecting clauses such that several literals are falsified by L.

Finally, it is important to notice that obstinately selecting the clause with a minimum number of unfalsified literals leads to an incomplete strategy. For instance, in a set containing the infinite set of clauses $p(a) \vee \neg p(f(a))$, $p(f(a)) \vee \neg p(f(f(a)))$, ... selecting every such clause and only those ones when trying to refute $\neg p(a) \wedge q(a)$ will not help. In order to ensure completeness, the selection method must verify a fairness condition: every instance must eventually be generated.

9.5 Quantified hypotheses

In the previous section, a simple method was presented to guide the refutation process i.e., selecting the most promising clauses from a countably infinite unsatisfiable set of ground instances to add to a satisfiable set of ground formulas in order to make it unsatisfiable. When dealing with quantified logic, the set of ground clauses (from the previous section) corresponds to the set of all instances of a finite set of Skolemized clauses. Indeed, from Herbrand's Theorem (Theorem 2) and the Compactness Theorem, if H is a finite set of Skolemized clauses, then H is unsatisfiable if and only if there exist formulas (not necessarily all distinct) $h_1, \ldots, h_n \in H$ and ground substitutions $\sigma_1, \ldots \sigma_n$ such that $\{h_1\sigma_1, \ldots, h_n\sigma_n\}$ is unsatisfiable. This allows to reduce the unsatisfiability problem for quantified formulas to the problem for ground logic. The price to pay is entirely in the choice of picking the right formulas and using the right substitutions.

Corollary 5 can easily be transposed to quantified clauses; if

- S is a $\mathcal{T}$-satisfiable set of ground clauses;
- L is a set of ground literals such that $L \models_{\mathcal{T}} S$;
- there is single ground instance h' of a satisfiable Skolem clause h such that $S \cup \{h'\}$ is unsatisfiable,

then there exists a ground substitution σ such that $L \models_{\mathcal{T}} \overline{\ell}\sigma$ for each $\ell \in h$. The existence and construction of the substitution σ will be discussed in Section 9.6, but first notice that several instances of the same quantified formula may be needed in order to refute the set of formulas. For example, assume $S = L = \{p(f(a)), \neg p(f(f(f(a))))\}$, and $h = \forall x[p(f(x)) \Rightarrow p(f(f(x)))]$. The set $h \cup L$ is unsatisfiable, but two instances of h are required to refute it by instantiation: the substitution is $[x \mapsto a]$ for one instance, and $[x \mapsto f(a)]$ for the other.

Adding instances of only one formula does not always suffice to make the set unsatisfiable. If $S \cup \{h_1, \dots h_n\}$ is $\mathcal{T}$-unsatisfiable, then there exists substitutions $\sigma_1, \dots \sigma_m$ with $m \geq n$ such that, for any literals $\ell_1 \in h'_1\sigma_1, \dots, \ell_m \in h'_m\sigma_m$ ($h'_i \in \{h_1, \dots h_n\}$ for each $i \in \{1, \dots m\}$), then $L \models_{\mathcal{T}} \neg(\ell_1 \wedge \dots \ell_n)$. Just as for ground formulas, the number of those combinations might be very large, and this motivates to choose instances such that many literals are falsified by L. Notice also that, if a necessary instance $h'_1\sigma_1$ contains a literal $p(t_1, \dots t_n)$ (respectively $\neg p(t_1, \dots t_n)$) where p is an uninterpreted literal unused in L, then another necessary instance contains a literal $\neg p(t'_1, \dots t'_n)$ (resp. $p(t'_1, \dots t'_n)$).

The instantiation task involves, first, to pick a Skolem formula in the set of "hypotheses", and second, to find ground substitutions to falsify most literals in the context expressed by the set of ground literals L.

9.6 Finding ground substitutions

The hypothesis selection heuristic relies on the problem of finding ground substitutions σ such that

$$L \models_{\mathcal{T}} \ell_1\sigma \wedge \dots \ell_n\sigma, \tag{9.1}$$

where L is a set of ground literals, $\mathcal{T}$ is a theory, and $\ell_1, \dots \ell_n$ are literals. Equivalently, the problem is to find a ground substitution σ such that $L \models_{\mathcal{T}} \ell_1\sigma$, $\dots L \models_{\mathcal{T}} \ell_n\sigma$ hold simultaneously. This can be done by solving each problem $L \models_{\mathcal{T}} \ell_i\sigma_i$, and building a global solution from the substitutions σ_i that coincide on shared variables. Problem (9.1) is thus solved by solving n problems

$$L \models_{\mathcal{T}} \ell\sigma. \tag{9.2}$$

When $\mathcal{T}$ is the empty theory, problem (9.2) can be recast as a (non-simultaneous) rigid E-unification problem (see for instance [58, 83, 113]). The goal of the problem of rigid E-unification is to find a substitution σ such that

$$E\sigma \models t\sigma = t'\sigma$$

where E is a set of equations, and t and t' are terms. The set E may use variables. This problem is NP-complete.

To see that problem (9.2) is indeed a instance of rigid E-unification (when $\mathcal{T}$ is the empty theory), some preliminary transformations are necessary. First, σ is a solution of $L \models \ell\sigma$ if and only if $L \cup \{\bar{\ell}\sigma\}$ is unsatisfiable. The literals in $L \cup \{\bar{\ell}\sigma\}$ may contain predicates. For each p predicate, a new function f_p is introduced. Literals $p(t_1, \dots t_n)$ are replaced by $f_p(t_1, \dots t_n) = a_\top$, whereas literals $\neg p(t_1, \dots t_n)$ are replaced by $f_p(t_1, \dots t_n) = a_\perp$, where $a_\top$ and $a_\perp$ are two new constants. Also $a_\top \neq a_\perp$ is added to the set. Then, each inequality $x \neq y$ (except $a_\top \neq b_\top$) is replaced by the two equalities $f_{\neq}(x) = a_\top$ and $f_{\neq}(x) = a_\perp$ where $f_{\neq}$ is a newly introduced function for each inequality. Now, the only inequality in the obtained set is $a_\top \neq a_\perp$, no predicate is used at all, and it is straightforward to translate the problem into a rigid E-unification problem. The transformation is polynomial, thus the problem is NP. Unfortunately, our problem is also NP-hard.

First, observe that finding substitutions such that

$$L \models_{\mathcal{T}} (t_1 = t'_1)\sigma \wedge \dots (t_n = t'_n)\sigma \tag{9.3}$$

is equivalent to finding substitutions such that

$$L \models_{\mathcal{T}} f(t_1, \dots t_n)\sigma = f(t'_1, \dots t'_n)\sigma$$

where f is a new n-ary function symbol. It is possible to encode the NP-complete 3-SAT problem (satisfiability of a set of clauses with 3 literals) into instances of problem (9.3). Assume that C is a set of 3-clauses containing propositions from the set P. The set of equalities $E = \{t_i = t'_i \,|\, i = 1 \dots n\}$ and of literals L are built using:

- constants t and f;
- a unary function v;
- a ternary function c for each clause in C;
- a variable p for each proposition in P.

The set of ground literals L contains $t \neq f$, $v(t) = t$ and $v(f) = t$, and, for each clause C, the seven literals encoding the models of this clause; for instance, if the clause is $p_1 \vee \neg p_2 \vee p_3$, and if c is the function associated to it, then L contains $c(t, t, t) = t$, $c(t, t, f) = t$, $c(t, f, t) = t$, $c(t, f, f) = t$, $c(f, f, t) = t$, $c(f, f, f) = t$ and $c(f, t, t) = t$. The set of non-ground equations E contains equation $v(p) = t$ for each variable p associated to a proposition in P. Furthermore, for each clause $l_1 \vee l_2 \vee l_3$ in C associated to the function c, $c(p_1, p_2, p_3) = t$ also belongs to the set E, where p_1, p_2 and p_3 are the variables associated to propositions in literals l_1, l_2 and l_3. Then there exists a substitution σ such that $L \models \wedge_{\ell \in E} \ell\sigma$ if and only if C is satisfiable.

Notice that, given a set of ground literals L, a literal ℓ and a ground substitution σ such that $L \models_{\mathcal{T}} \ell\sigma$, any substitution

$$\theta = \{x_1 \mapsto t_1, \dots x_n \mapsto t_n\}$$

such that $\{x_1, \dots x_n\} = \mathrm{Dom}(\sigma)$ and $L \models_{\mathcal{T}} t_i = x_i\sigma$ for all $i \in \{1, \dots n\}$ is also such that $L \models_{\mathcal{T}} \ell\theta$. In particular, for the empty theory, a solution σ can be seen as a mapping from variables in ℓ to congruence classes containing ground terms of $L \cup \{\ell\}$.

Finding a ground substitution such that $L \models \ell\sigma$ is equivalent to finding a ground substitution such that $L \cup \{\neg\ell\sigma\}$ is unsatisfiable. Since L does not contain variables, this also amounts to finding a set of equation E such that $L \cup E \cup \{\ell\}$ is unsatisfiable, with $E = \{x_1 = t_1, \dots x_n = t_n\}$ where $x_1, \dots x_n$ are variables from ℓ ($x_i \neq x_j$ if $i \neq j$) and $t_1, \dots t_n$ are ground terms. More generally, for every idempotent substitution σ, if $L\sigma = L$, then $L \models \ell\sigma$ if and only if $L \cup \{x_1 = x_1\sigma, \dots x_n = x_n\sigma, \neg\ell\}$ is unsatisfiable, as $x_i\sigma$ does not contain[6] any x_j (with $i, j \in \{1, \dots n\}$). The relation between congruence closure and rigid E-unification has been studied in [179].

At the present time, the selection heuristic used in Caveat does not use a complete solution for solving problem (9.1). First, $\mathcal{T}$ is assumed to be the empty theory, and second, the substitution is built using only literals ℓ_i of the form $p(x_1, \dots x_n)$ or $\neg p(x_1, \dots x_n)$ where p is a predicate symbol[7] and $x_1, \dots x_n$ are variables. In that case, finding a substitution such that[8]

$$L \models p(x_1, \dots x_n)\sigma$$

is straightforward, as σ is a solution if and only if there is a literal $p(t_1, \dots t_n)$ in L such that, for each $i \in \{1, \dots n\}$, $x_i\sigma$ is in the same class than t_i in the congruence closure built with L. If a substitution is seen as a mapping of variables to classes of congruence closure, the number of solutions is upper-bounded by the numbers of literals in L. For the general problem stated in (9.1), where $\mathcal{T}$ is the empty theory, solutions for each (negation of) predicates are found, and global solutions are compiled with substitutions that coincide on shared variables. The heuristic is effective, since many literals used in our context are in the required form. But it is also highly incomplete. To circumvent this problem, instances are guessed by associating ground terms to variables, with a preference for terms of small depth.

Figure 9.1 collects some experimental results from Caveat, for the verification of various systems (a formal description of those systems can be found

[6]Except maybe when $x_i\sigma = x_i$.

[7]This predicate is uninterpreted, with the assertion that $\mathcal{T}$ is the empty theory.

[8]The case of the negated predicate is similar.

System	time (s)	n	i
Bakery	2.74	49	423
Burns	2.15	40	498
Dijkstra ME	11.86	182	3965
Fisher	2.06	42	196
Railroad Crossing	42.39	221	288
Ricart-Agrawala	12.23	112	2155
Szymanski	5.28	48	1318

Figure 9.1: Caveat on some systems.

in [35, 105, 119, 159]). The first column gives the name of the system being verified, the second the execution time (in seconds, on a Pentium M at 1.50 GHz with 1 Gbyte of memory), n is the total number of generated validation conditions, and i the total number of produced quantifier-free instances. The performance of the selection heuristic can be measured by the number of instances per validation condition (i/n).

Chapter 10

Conclusion

In this thesis, we have presented several techniques for system verification with invariants. In Chapter 4, the syntax and semantics of transition systems have been introduced, as well as the verification condition generation method. We then noticed (Chapter 5) that, for important classes of formulas and algorithms, invariant validation is a decidable problem. In particular, the decision procedure has been successfully used to prove all verification conditions for a parameterized railroad crossing system [105] used as benchmark for STeP, whereas STeP itself requires interactive verification for some of them [35]. It is not required for processes in the parameterized system to be finite-state. Our decision procedure is essentially a quantifier elimination method, which is suitable even when some interpreted predicates and functions are used in verification conditions. It relies on methods to prove sets of ground formulas, containing uninterpreted and interpreted predicates and functions.

The rest of the thesis was concerned with automated proving techniques for the validation of verification conditions. In Chapter 6, we briefly surveyed major propositional methods, and discussed their use within a framework to decide quantifier-free languages. In particular, we sketched the importance for propositional provers to provide near-minimal partial models, and for methods dealing with sets of first-order literals to be proof-producing, i.e., to provide near-minimal unsatisfiable subsets of literals. Congruence closure, studied in Chapter 7, allows to decide the satisfiability of sets of literals with equalities, and uninterpreted functions and predicates. Our algorithm is proof-producing, incremental, and efficient. The combination of decision procedures (Chapter 8) is a scheme allowing several decision procedures for sets of literals on specific disjoint languages (that is, with interpreted predicates and functions) to collaborate, in order to provide a decision procedure for the union of languages. Our framework extends the classical one in several aspects: it is a many-sorted framework to deal with theories non necessarily stably infinite and first-order. Methods from Chapters 6 to 8 are the basis for automated validation of quantifier-free formulas in rich languages containing uninterpreted and interpreted predicates and functions. Instantiation (Chapter 9) relies on those techniques to build a proof method for quantified formulas.

The techniques presented in this thesis have been implemented into the invariant validation tool CAVEAT, and successfully applied to the verification of many parameterized algorithms at several levels of abstraction, from coarse to fine-grained versions. Examples of such algorithms are the Bakery, Burns, Dijkstra, Fisher, Ricart-Agrawala, and Szymanski mutual exclusion algorithms, a railroad crossing algorithm, the Stenning protocol,... (see [35, 97, 105, 119, 159] for formal descriptions of those algorithms). Techniques from Chapters 6 and 7 have also been implemented inside the HARVEY tool [67], a prover for ground first-order formulas, with very good results for benchmarks issued from various verification efforts, including formulas from the UCLID tool [115].

Throughout the proving process with invariants, a number of choices in techniques have to be made. Those choices essentially depend on the problem instances, and on the techniques used in the whole verification process. New techniques, for instance in propositional SAT-checkers (see Chapter 6), may well change from one extreme to another the choices considered as the best for verification condition generation. We now present a retrospective of the main places where some choice exists and explain the chosen option. It is important to clearly state those choices, as they often represent parameters for fine tuning of the methods.

First, in Chapter 4, it was noticed that structured variables could be modelled into verification conditions either by directly using predicates and functions, or by introducing interpreted predicates and functions for constructing and accessing those structured variables. This occurs notably for modelling arrays, lists, sets,... Whereas validation of quantifier-free formulas containing those constructor and accessor interpreted functions is decidable in most cases (see Chapter 8), we found that those interpreted functions were a real problem for handling quantifiers. They complicate the Herbrand sets (see Definition 10), and possibly hide Skolem functions in the theory. In particular, the results about decidability of invariant validation for parameterized systems (see Chapter 5) deeply rely on the absence of functions (not predicates) dealing with quantified variables. In Chapter 9, it is noticed that, when dealing with interpreted predicates and functions, the completeness of instantiation-based refutation methods may be lost. As the constructor and accessor functions for structured data are usually described with first-order theories, completeness is not the real issue here. However, as it has been observed when introducing equality in automated theorem proving, the intensive use of interpreted predicates and functions without providing provers with appropriate tools to deal with those predicates and functions might lead to strong limitations of the automated reasoning tools. Taking those difficulties into account, the option chosen in Caveat is to avoid the introduction of interpreted functions to handle structured data.

During the verification condition generation, another important issue (see Section 4.8) is whether it is preferable to have less functions and predicates at the cost of producing larger formulas, or to keep formulas as small as possible at the cost of introducing many functions and predicates which may confuse the theorem prover. The (extreme) choice made in Caveat is to avoid symbol duplication, at the cost of generating larger formulas. This is motivated by the fact that, in our context, transitions generally have a short statement part, and thus formula explosion is under control. However it remains to investigate if a trade-off solution would not be more efficient.

In Chapter 3, several formula transformations were presented. Skolemization, allowing to remove the quantifiers having an existential semantic, may be applied in different ways, but generally the strategy is clear, as Andrews' Skolemization is theoretically better than prenex and structural Skolemization. There exist more complicated strategies, but with a prohibitive cost. The case of conjunctive normal form transformations is not exactly the same: both p-definitional and definitional normal form transformations introduce many new propositions (or labels). A transformation that introduces labels sparingly may make the proving job easier. However, it appears that recent SAT-provers (see Chapter 6) do cope well with those introduced labels. Also, there is no substantial difference between proving times for p-definitional and definitional normal forms.

A much more important issue is the elimination of if-then-else terms (see Section 3.5). It appears that the collaboration of provers for propositional and for first-order literals in haRVey fails at proving benchmarks extensively using such terms. More precisely, the satisfiability check for first-order literals is not a problem, but the quality of information returned by the prover to refine the Boolean abstraction (see Chapter 6) does not, in some cases, sufficiently refine the abstraction for the cooperation of both provers to be successful. For instance, unsatisfiable sets of literals like $t_1 = 1 + t_2, t_2 = t_3, \ldots t_{n-1} = t_n, t_n = t_1$ often occur as models for abstraction when proving the UCLID benchmarks.[1] The refining clause corresponding to this set is too particular to greatly refine the abstraction, and as a symptom, the first-

[1]Terms t_i ($i \in \{1, \ldots n\}$) are introduced by if-then-else elimination.

order prover is very much solicited. Solutions for this problem may involve more careful treatments of if-then-else terms, but also new ways of refining Boolean abstractions.

Among the various preprocessing steps for formulas, we did not investigate renaming inside quantified formulas,[2] i.e., replacing repeated subformulas with new predicates defined to be equal to those sub-formulas. This omission is justified by the fact that introducing new predicates may have disastrous effects on the instantiation method (Chapter 9), by requiring several instances to refute simple facts proved with only one instance when no renaming is used. However, this extreme position may be counterproductive in the case of formulas containing many times the same large subformula.

In order to cooperate successfully with the provers for first-order atoms, solvers for propositional logic should provide near-minimal models (see Chapter 6). Propositional methods are designed to provide models, but not minimal models. It has already been mentioned that extracting a minimal model from a model of a formula is a difficult problem. In Section 6.4, we presented a simple linear method to extract minimal models for sets of clauses. As it is linear, its cost is bearable within the cooperation of provers. Of course, it only provides a minimal model for the set of clauses of the structural normal form of the formula, but not a minimal model for the formula itself. Nevertheless, this method has very positive effects on proving times. A question thus arises: to what extent is it productive to use expensive methods to provide minimal models of the Boolean abstraction of formulas?

A similar question concerns the other module in the collaboration: to what extent is it useful to use expensive methods to refine the proofs provided by decision procedures for sets of literals (see Chapters 7 and 8)? The method proposed in Section 7.6 provides approaching solutions, the method from Section 8.4 may be exact if required. The first is, in practice, very efficient, but the second may be very expensive. In Caveat and haRVey, the approaching solutions supplied by the proof-producing congruence closure algorithm appeared to be of very good quality, so that there is no need for more expensive methods. For arithmetic atoms (in Caveat), it appeared that using the method in Section 8.4 to provide exact minimal models was valuable despite its high cost.

The preceding choices may be seen as open questions, because most of them deeply rely on the problem instances and on the global design of the validation procedure. Beside those choices, several future works are also motivated by results of this thesis.

In Chapter 5, a quantified class of formulas accepting quantifier elimina-

[2] It has been mentioned when studying structural normal forms in Section 3.4.

tion is identified. In Chapters 6 to 8, the ground fragment is shown to be decidable when formulas contain interpreted predicates and functions from various theories. The results of this thesis deeply rely on these two decidable classes of formulas. In Chapter 6, it is noticed that BDDs with first-order atoms as nodes do not provide a *canonical* form for quantifier-free formulas. However, it is possible to check if two ground formulas are logically equivalent. Thus, a canonical form exists, at least in theory: given any order on the lexicon of the language of formulas, the canonical form corresponding to formula φ may be, for instance, the smallest formula (according to the lexical order) in the class which is equivalent to φ. Usable canonical forms exist for propositional logic (BDDs), and also for linear arithmetic (with quantifiers) on $\mathbb{Z}$ and $\mathbb{R}$ [38]. A practical canonical form for the ground formulas accepted by the techniques of Chapter 8, and for the quantified class of formulas used in Chapter 5 would be useful, notably for fix-point computation and invariant generation.

Results in Chapter 5 state that formulas of identified form are invariants for some parameterized system if and only if they are invariant of the system for a fixed value of the parameter. This makes the invariant validation problem decidable, for identified classes of formulas and parameterized systems. One important step of the method of invisible invariants [14, 149] consists of transforming computed invariants for some finite number of the parameter into invariant candidates for an unknown parameter.[3] This process is heuristic, and is only justified *a posteriori*, when the invariant candidates are validated. A direction for future research could be to investigate if this heuristic process cannot be turned into a theoretically founded one. In other words, is it possible to have an automatic invariant generation procedure for systems and formulas considered in Chapter 5?

The combination scheme presented in Chapter 8 requires for individual decision procedures to be able to generate disjunctions of equalities. In CAVEAT, the module used for linear arithmetic is the automata-based decision procedure LASH [116]. Its integration within the combination scheme is incomplete, as it does not generate equalities and disjunctions of equalities. This incompleteness does not have much consequence for the studied algorithms, but makes the whole decision procedure incomplete. Finding algorithms to efficiently generate the disjunctions of equalities associated to a set of arithmetic constraints is a major issue for the use of automata-based techniques into combinations of provers for ground logic. Another issue of equal importance is to augment the number of variables handled by those

[3]These invariant candidates are then validated using results similar to those in Chapter 5.

techniques.

A weakness of the sorted combination framework presented in Chapter 8 is that it does not allow to deal, for instance, with verification conditions about systems using lists of elements which can either be numbers or lists of numbers. Future improvements of the combination scheme should be done in a less rigid many-sorted framework in order to handle such cases.

Instantiation (see Chapter 9), as presently used in CAVEAT, only generates ground instances, i.e., variable-free formulas. Delaying instantiation might provide a better strategy to find the right substitutions. Authorizing and handling non-ground instantiation however implies new mechanisms to possibly instantiate later the preserved variables. More generally there is much place in the instantiation method for other efficient and successful heuristics. Restricting the formula and substitution selection without losing completeness is a major issue for future enhancement of the method.

The two challenges for the future of automated theorem proving in verification are naturally to be able to prove larger verification conditions, but also to extend the language of handled interpreted functions and predicates. This was the motivation for combination schemes as presented in Chapter 8.

Extending the language handled by the prover is one solution to deal with algorithms relying on properties non easily expressed in pure first-order logic (see Section 4.9). Another solution would be to use *abstraction* (see [48, 77, 96] for examples of the use of abstraction in verification with invariants) to reduce the verification conditions to more restrictive languages. Invariance properties requiring an expressive language would first be verified using interactive theorem proving, then abstracted, and finally the automatic prover would be used on the abstracted formula. In [102], predicate abstraction was used successfully to eliminate some quantifiers in verification conditions. First investigations to extend this method to handle an algorithm for distributed garbage collector [132] using properties non easily expressed in first-order logic were promising: it appears that the needs for very expressive languages are very localized, whereas first-order reasoning is sufficient to tackle the major part of the proof. Finding methods, using abstraction, to combine interactive theorem proving for very expressive languages with automated theorem proving is one of our goal for very near future research.

Bibliography

[1] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parametrized system verification. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145. Springer-Verlag, July 1999.

[2] Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Logic In Computer Science (LICS)*, pages 313–321, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society.

[3] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[4] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of IEEE 1394 tree identify protocol. *Formal Aspects of Computing*, 14(3):215–227, December 2002.

[5] Husam Abu-Haimed, Sergey Berezin, and David L. Dill. Strengthening invariants by symbolic consistency testing. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, July 2003.

[6] Wilhelm Ackermann. *Solvable Cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[7] Peter B. Andrews. Refutations by Matings. *Journal of the ACM*, 28(2):193–214, April 1981.

[8] Peter B. Andrews. Classical type theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 15, pages 965–1007. Elsevier Science B.V., 2001.

[9] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A theorem-proving system for classical type theory. *Journal of Automated Reasoning*, 16(3):321–353, June 1996.

[10] Krzysztof R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[11] Krzysztof R. Apt. Ten years of Hoare's logic: A survey — Part II. *Theoretical Computer Science*, 28:83–109, 1984.

[12] Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, May 1986.

[13] Alessandro Armando, Silvio Ranise, and Michael Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140–164, 2003.

[14] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore Zuck. Parameterized verification with automatically computed inductive assertions. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science*, pages 221–234. Springer-Verlag, July 2001.

[15] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In Andrei Voronkov, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 195–210. Springer-Verlag, July 27-30 2002.

[16] Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 447–533. Elsevier Science B.V., 2001.

[17] Matthias Baaz, Uwe Egly, and Alexander Leitsch. Normal form transformations. In John Alan Robinson and Andrei Voronkov, editors,

Handbook of Automated Reasoning, volume I, chapter 5, pages 273–333. Elsevier Science B.V., 2001.

[18] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science B.V., 2001.

[19] Leo Bachmair and Ashish Tiwari. Abstract congruence closure and specializations. In David McAllester, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 64–78, Pittsburgh, PA, June 2000. Springer-Verlag.

[20] Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 2(31):129–168, January 2003.

[21] Clark W. Barrett. *Checking validity of quantifier-free formulas in combinations of first-order theories*. PhD thesis, Stanford University, 2003.

[22] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity checking for combinations of theories with equality. In Mandayam K. Srivas and Albert John Camilleri, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996.

[23] Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer-Verlag, 2002.

[24] Clark W. Barrett, David L. Dill, and Aaron Stump. A generalization of Shostak's method for combining decision procedures. In Alessandro Armando, editor, *Frontiers of Combining Systems (FroCoS)*, volume 2309 of *Lecture Notes in Computer Science*, pages 132–146. Springer-Verlag, April 2002.

[25] Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Verification of Parameterized Protocols. *Journal of Universal Computer Science*, 7(2):141–158, February 2001.

[26] Paul Beame, Richard Karp, Toniann Pitassi, and Michael Saks. The efficiency of resolution and Davis-Putnam procedures. *SIAM Journal on Computing*, 31(4):1048–1075, August 2002.

[27] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335, New Brunswick, NJ, USA, 1996. Springer-Verlag.

[28] Sergey Berezin, Vijay Ganesh, and David L. Dill. An online proof-producing decision procedure for mixed-integer linear arithmetic. Unpublished manuscript, 2002.

[29] Sergey Berezin, Vijay Ganesh, and David L. Dill. An online proof-producing decision procedure for mixed-integer linear arithmetic. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 521–536. Springer, 2003.

[30] Paul Bernays and Moses Schönfinkel. Zum Entscheidungsproblem der mathematischen Logik. *Math. Annalen*, 99:342–372, 1928.

[31] Wolfgang Bibel. An approach to a systematic theorem proving procedure in first-order logic. *Computing*, 12(1):43–55, 1974.

[32] Wolfgang Bibel and Elmar Eder. Methods and calculi for deduction. In Dov M. Gabbay, Christopher John Hogger, and John Alan Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1: Logical Foundations, pages 67–182. Oxford University Press, 1992.

[33] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC)*, pages 317–320. ACM press, 1999.

[34] Nikolaj S. Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[35] Nikolaj S. Bjørner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Deductive verification of real-time systems using STeP. *Theoretical Computer Science*, 253, 2001.

[36] Nikolaj S. Bjørner, Mark E. Stickel, and Tomás E. Uribe. A practical integration of first-order reasoning and decision procedures. In William McCune, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 1249 of *Lecture Notes in Computer Science*, pages 101–115. Springer-Verlag, 1997.

[37] Alexander Bockmayr and V. Weispfenning. Solving numerical constraints. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 12, pages 751–842. Elsevier Science B.V., 2001.

[38] Bernard Boigelot, Sébastien Jodogne, and Pierre Wolper. On the use of weak automata for deciding linear arithmetic with integer and real variables. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 2083 of *Lecture Notes in Computer Science*, pages 611–625. Springer, June 2001.

[39] Bernard Boigelot, Axel Legay, and Pierre Wolper. Omega-regular model checking. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, pages 561–575. Springer-Verlag, 2004.

[40] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer-Verlag, July 2000.

[41] Luc Bougé, Yann Le Guyadec, Gil Utard, and Bernard Virot. On the expressivity of a weakest precondition calculus for a simple data-parallel programming language. In Bruno Buchberger and Jens Volkert, editors, *Proc. Conference on Vector and Parallel Processing (CONPAR)*, volume 854 of *Lecture Notes in Computer Science*, pages 100–111, Linz, Austria, September 1994. Springer-Verlag.

[42] Marco Bozzano and Giorgio Delzanno. Beyond parameterized verification. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2280 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2002.

[43] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[44] Randal E. Bryant, Steven M. German, and Miroslav N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*, pages 470–482. Springer, 1999.

[45] Randal E. Bryant, Steven M. German, and Miroslav N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):93–134, January 2001.

[46] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2002.

[47] Randal E. Bryant and Miroslav N. Velev. Boolean satisfiability with transitivity constraints. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 85–98. Springer, 2000.

[48] Dominique Cansell, Dominique Méry, and Stephan Merz. Predicate diagrams for the verification of reactive systems. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Integrated Formal Methods (IFM)*, volume 1945 of *Lecture Notes in Computer Science*, pages 380–397, Dagstuhl, Germany, November 2000. Springer-Verlag.

[49] Dominique Cansell and Dominique Méry. Foundation of the B method. *Computing and Informatics*, 22:1–31, 2003.

[50] K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison-Wesley, Reading, Massachusetts, 1988.

[51] Chen-Chung Chang and H. Jerome Keisler. *Model theory*, volume 73 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam-London, 1973.

[52] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[53] Sylvain Conchon and Sava Krstić. Strategies for combining decision procedures. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 537–553. Springer-Verlag, April 2003.

[54] David Cyrluk, Patrick Lincoln, and Natarajan Shankar. On Shostak's decision procedure for combinations of theories. In Michael A. McRobbie and John K. Slaney, editors, *Proc. Conference on Automated Deduction (CADE)*, volume 1104 of *Lecture Notes in Computer Science*, pages 463–477, New Brunswick, NJ, 1996. Springer-Verlag.

[55] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[56] Thierry Boy de la Tour. Minimizing the number of clauses by renaming. In Mark E. Stickel, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 558–572, Kaiserslautern, FRG, July 1990. Springer-Verlag.

[57] Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. Theory and Applications of Satisfiability Testing (SAT), Cincinnati, USA, May 2002.

[58] Anatoli Degtyarev and Andrei Voronkov. Equality reasoning in sequent-based calculi. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 10, pages 611–706. Elsevier Science B.V., 2001.

[59] Nachum Dershowitz. A taste of rewrite systems. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 199–228, 1993.

[60] Nachum Dershowitz and David A. Plaisted. Rewriting. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 9, pages 535–610. Elsevier Science B.V., 2001.

[61] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, December 1998.

[62] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[63] Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, October 1980.

[64] Burton Dreben and Warren D. Goldfarb. *The Decision Problem: Solvable Classes of Quantificational Formulas*. Addison-Wesley, Reading, Massachusetts, 1979.

[65] David Déharbe, Abdessamad Imine, and Silvio Ranise. Abstraction-driven verification of array programs. Technical report, INRIA, 2004.

[66] David Déharbe and Silvio Ranise. BDD-driven first-order satisfiability procedures. Technical Report 4630, INRIA, November 2002.

[67] David Déharbe and Silvio Ranise. Light-weight theorem proving for debugging and verifying units of code. In *Software Engineering and Formal Methods (SEFM)*, pages 220–228. IEEE Computer Society, September 2003.

[68] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite Model Theory*. Perspectives in Mathematical Logic. Springer-Verlag, Berlin, 1995.

[69] Elmar Eder. An implementation of a theorem prover based on the connection method. In Wolfgang Bibel and B. Petkoff, editors, *Artificial Intelligence—Methodology Systems Application (AIMSA)*, pages 121–128. North-Holland, September 1984.

[70] Uwe Egly. On the value of antiprenexing. In Frank Pfenning, editor, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 822 of *Lecture Notes in Artificial Intelligence*, pages 69–83, Berlin, July 1994. Springer.

[71] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, Inc., Orlando, Florida, 1972.

[72] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[73] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and Natarajan Shankar. ICS: integrated canonizer and solver. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*

(CAV), volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer-Verlag, 2001.

[74] Alec Fisher. *Formal Number Theory and Computability.* Oxford University Press, Oxford, 1982.

[75] Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, Berlin, 1990.

[76] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 355–367. Springer-Verlag, July 2003.

[77] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Principles of Programming Languages (POPL)*, pages 191–202, Portland, Oregon, January 16–18, 2002. ACM press.

[78] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In Cindy Norris and Jr. James B. Fenwick, editors, *Principles of Programming Languages (POPL)*, volume 36 (3) of *ACM SIGPLAN Notices*, pages 193–205, January 17–19 2001.

[79] Pascal Fontaine and E. Pascal Gribomont. Using BDDs with combinations of theories. In Matthias Baaz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2514 of *Lecture Notes in Computer Science*, pages 190–201. Springer-Verlag, 2002.

[80] Pascal Fontaine and E. Pascal Gribomont. Decidability of invariant validation for parameterized systems. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 97–112. Springer-Verlag, 2003.

[81] Pascal Fontaine and E. Pascal Gribomont. Combining non-stably infinite, non-first order theories. In Cesare Tinelli and Silvio Ranise, editors, *Proceedings of the IJCAR Workshop on Pragmatics of Decision Procedures in Automated Deduction, PDPAR*, July 2004. to appear.

[82] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*, volume 5 of *Computer Science and Technology Series*. Harper & Row, New York, 1986.

[83] Jean H. Gallier, Paliath Narendran, David A. Plaisted, and Wayne Snyder. Rigid E-unification: NP-completeness and applications to theorem proving. *Information and Computation*, 87(1-2):129–195, 1990.

[84] Jean H. Gallier, Paliath Narendran, Stan Raatz, and Wayne Snyder. Theorem proving using equational matings and rigid E–unification. *Journal of the ACM*, 39(2):377–429, April 1992.

[85] Harald Ganzinger. Shostak light. In Andrei Voronkov, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 332–346. Springer-Verlag, July 27-30 2002.

[86] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer, 2004. (To appear).

[87] Harald Ganzinger and Konstantin Korovin. New directions in instantiation-based theorem proving. In *Logic In Computer Science (LICS)*, pages 55–64. IEEE Computer Society, 2003.

[88] Enrico Giunchiglia, Marco Maratea, and Armando Tacchella. Dependent and independent variables in propositional satisfiability. In Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni, editors, *Proc. European Conference Logics in Artificial Intelligence (JELIA)*, volume 2424 of *Lecture Notes in Computer Science*, pages 296–307. Springer, 2002.

[89] Enrico Giunchiglia and Roberto Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In Evelina Lamma and Paola Mello, editors, *Advances in Artificial Intelligence, 6th Congress of the Italian Association for Artificial Intelligence (AI*IA)*, volume 1792 of *Lecture Notes in Computer Science*, pages 84–94. Springer, 1999.

[90] Anuj Goel, Khurram Sajid, Hai Zhou, Adnan Aziz, and Vigyan Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided*

Verification (CAV), volume 1427 of *Lecture Notes in Computer Science*, pages 244–255. Springer-Verlag, 1998.

[91] Evgueni Goldberg and Yakov Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE)*, pages 142–149. IEEE Computer Society, March 2002.

[92] Jean Goubault. Proving with BDDs and control of information. In Alan Bundy, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 814 of *Lecture Notes in Computer Science*, pages 499–513. Springer-Verlag, 1994.

[93] Jean Goubault. A BDD-based simplification and Skolemization procedure. *Journal of the IGPL*, 3(6):827–855, 1995.

[94] Jean Goubault and Joachim Posegga. BDDs and automated deduction. In Zbigniew W. Raś and Maria Zemankova, editors, *Methodologies for Intelligent Systems*, volume 869 of *Lecture Notes in Artificial Intelligence*, pages 541–550, Berlin, October 1994. Springer.

[95] Susanne Graf and Hassen Saïdi. Verifying invariants using theorem proving. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 196–207. Springer Verlag, 1996.

[96] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer Verlag, 1997.

[97] E. Pascal Gribomont. Development of concurrent systems by incremental transformations. In Neil D. Jones, editor, *European Symposium on Programming (ESOP)*, volume 432 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 1990.

[98] E. Pascal Gribomont. A programming logic for formal concurrent systems. In Jos C. M. Baeten and Jan Willem Klop, editors, *Theories of Concurrency: Unification and Extension (CONCUR)*, volume 458 of *Lecture Notes in Computer Science*, pages 298–313, Amsterdam, The Netherlands, 27–30August 1990. Springer-Verlag.

[99] E. Pascal Gribomont. Concurrency without toil: a systematic method for parallel program design. *Science of Computer Programming*, 21(1):1–56, August 1993.

[100] E. Pascal Gribomont. Simplification of Boolean verification conditions. *Theoretical Computer Science*, 239(1):165–185, May 2000.

[101] E. Pascal Gribomont and Didier Rossetto. CAVEAT : technique and tool for Computer Aided VErification And Transformation. In Pierre Wolper, editor, *Computer Aided Verification (CAV)*, volume 939 of *Lecture Notes in Computer Science*, pages 70–83, Liege, Belgium, July 1995. Springer-Verlag.

[102] E. Pascal Gribomont and Guy Zenner. Automated verification of Szymanski's algorithm. In Bernhard Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1384 of *Lecture Notes in Computer Science*, pages 424–438. Springer-Verlag, 1998.

[103] Jan Friso Groote and Jaco van de Pol. Equational binary decision diagrams. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Computer Science*, pages 161–178. Springer-Verlag, 2000.

[104] Joseph Y. Halpern. Presburger arithmetic with unary predicates is Π^1_1 complete. *The Journal of Symbolic Logic*, 56(2):637–642, June 1991.

[105] Constance L. Heitmeyer and Nancy A. Lynch. The generalized railroad crossing — a case study in formal verification of real-time systems. In *IEEE Real-Time Systems Symposium*, pages 120–131, San Juan, Puerto Rico, December 1994.

[106] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. MONA: Monadic second-order logic in practice. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110. Springer, 1995.

[107] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[108] John N. Hooker, Gabriela Rago, Vijay Chandru, and Anjul Shrivastava. Partial instantiation methods for inference in first order logic. *Journal of Automated Reasoning*, 28(5):371–396, May 2002.

[109] Henrik E. Jensen and Nancy A. Lynch. A proof of Burns N-process mutual exclusion algorithm using abstraction. In Bernhard Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1384 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, March 1998.

[110] Robert B. Jones and David L. Dill. Efficient validity checking for processor verification. In *IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 2–6, San Jose, California, USA, 1995. IEEE Computer Society.

[111] Deepak Kapur. Shostak's congruence closure as completion. In H. Comon, editor, *Rewriting Techniques and Applications (RTA)*, volume 1232 of *Lecture Notes in Computer Science*, pages 23–37. Springer-Verlag, 1997.

[112] Martin Klazar. Counting even and odd partitions. *American Mathematical Monthly*, 110(6):527–532, 2003.

[113] Dexter Kozen. Positive first-order logic is NP-Complete. *IBM Journal of Research and Development*, 25(4):327–332, July 1981.

[114] Sava Krstić and Sylvain Conchon. Canonization for disjoint unions of theories. In Franz Baader, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 2741 of *Lecture Notes in Computer Science*, pages 197–211. Springer-Verlag, July 2003.

[115] Shuvendu K. Lahiri, Sanjit A. Seshia, and Randal E. Bryant. Modeling and verification of out-of-order microprocessors using UCLID. In Mark Aagaard and John W. O'Leary, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, volume 2517 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2002.

[116] The Liège Automata-based Symbolic Handler (LASH). Available at `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.

[117] Jeremy R. Levitt. *Formal Verification Techniques for Digital Systems.* PhD thesis, Stanford University, December 1998.

[118] Inês Lynce and João P. Marques-Silva. Efficient data structures for backtrack search SAT solvers. Theory and Applications of Satisfiability Testing (SAT), Cincinnati, USA, May 2002.

[119] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CS, 1996.

[120] Monika Maidl. A unifying model checking approach for safety properties of parameterized systems. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 2001.

[121] Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, New York, 1974.

[122] Zohar Manna, Anuchit Anuchitanukul, Nikolaj S. Bjørner, Anca Browne, Edward Chang, Michael Colon, Luca de Alfaro, Harish Devarajan, Henny B. Sipma, and Tomas Uribe. STeP: The stanford temporal prover. Technical Report CS-TR-94-1518, Stanford University, Computer Science Department, June 1994.

[123] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In Cynthia Dwork, editor, *ACM Symposium on Principles of Distributed Computing*, pages 377–408, Québec City, Québec, Canada, August 1990. ACM press.

[124] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[125] Zohar Manna and Calogero G. Zarba. Combining decision procedures. In *Formal Methods at the Cross Roads: From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 381–422. Springer, 2003.

[126] João P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In Pedro Barahona and José J. Alferes, editors, *Proc. Portuguese Conference on Progress in Artificial Intelligence (EPIA)*, volume 1695 of *Lecture Notes in Artificial Intelligence*, pages 62–74, Berlin, September 21–24 1999. Springer.

[127] João P. Marques-Silva. Algebraic Simplification Techniques for Propositional Satisfiability. In Rina Dechter, editor, *Principles and Practice of Constraint Programming (CP)*, volume 1894 of *Lecture Notes in Computer Science*, pages 537–542. Springer, September 2000.

[128] João P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *IEEE International Conference on*

Computer-Aided Design (ICCAD), pages 220–227, San Jose, California, USA, November 10–14 1996. IEEE Computer Society.

[129] Yu. V. Matijasevič. Diophantine representation of recursively enumerable predicates. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 171–177, Amsterdam, 1971. North-Holland.

[130] P. G. Neumann (Moderator). The RISK digest: Forum on risks to the public in computers and related systems. ACM Committee on Computers and Public Policy, `http://risks.org`.

[131] Jesper B. Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. Difference decision diagrams. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic (CSL)*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 1999.

[132] Luc Moreau and Jean Duprat. A Construction of Distributed Reference Counting. Technical Report RR1999-18, Ecole Normale Supérieure, Lyon, March 1999.

[133] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference (DAC)*, pages 530–535. ACM press, June 2001.

[134] George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[135] Greg Nelson and Derek C. Oppen. Simplifications by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[136] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.

[137] Robert Nieuwenhuis and Albert Oliveras. Congruence closure with integer offsets. In Moshe Y. Vardi and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2850 of *Lecture Notes in Computer Science*, pages 78–90, Almaty, Kazakhstan, September 2003. Springer.

[138] Robert Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science B.V., 2001.

[139] Andreas Nonnengart. Strong Skolemization. Research Report MPI-I-96-2-010., 1996.

[140] Andreas Nonnengart, Georg Rock, and Christoph Weidenbach. On generating small clause normal forms. In Hélène Kirchner and Claude Kirchner, editors, *Proc. Conference on Automated Deduction (CADE)*, volume 1421 of *Lecture Notes in Computer Science*, pages 397–411. Springer-Verlag, 1998.

[141] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier Science B.V., 2001.

[142] Derek C. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, 12(3):291–302, November 1980.

[143] Derek C. Oppen. Reasoning about recursively defined data structures. *Journal of the ACM*, 27(3):403–411, July 1980.

[144] Jens Otten. On the advantage of a non-clausal Davis-Putnam procedure. Technical Report AIDA-97-1, TH Darmstadt, Intellektik, 1997.

[145] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga Springs, NY, June 1992. Springer.

[146] James L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[147] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Programming Language Design and Implementation (PLDI)*, volume 23 (7) of *ACM SIGPLAN Notices*, pages 199–208, New York, June 1988. ACM press.

[148] David A. Plaisted and Steven Greenbaum. A structure preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.

[149] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2001.

[150] Joachim Posegga and Peter H. Schmitt. Automated deduction with Shannon graphs. *Journal of Logic and Computation*, 5(6):697–729, December 1995.

[151] V. Pratt. Two easy theories whose combination is hard, 1977.

[152] I. V. Ramakrishnan, R. Sekar, and Andrei Voronkov. Term indexing. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1964. Elsevier Science B.V., 2001.

[153] Frank P. Ramsey. On a Problem of Formal Logic. *Proceedings of the London Mathematical Society*, 30:264–286, 1930.

[154] Silvio Ranise and David Déharbe. Applying light-weight theorem proving to debugging and verifying pointer programs. In Ingo Dahn and Laurent Vigneron, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.

[155] Silvio Ranise, Christophe Ringeissen, and Duc-Khanh Tran. Extended canonizers for combining Nelson-Oppen and Shostak theories, 2004.

[156] Silvio Ranise and Cesare Tinelli. The SMT-LIB format : an initial proposal, July 2003.

[157] Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*, pages 31–45. Springer-Verlag, 2004.

[158] Alexandre Riazanov and Andrei Voronkov. The design and implementation of Vampire. *AI Communications*, 15(2):91–110, 2002.

[159] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.

[160] Christophe Ringeissen. Cooperation of decision procedures for the satisfiability problem. In F. Baader and Klaus U. Schulz, editors, *Frontiers of Combining Systems (FroCoS)*, Applied Logic, pages 121–140. Kluwer Academic Publishers, March 1996.

[161] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science B.V., 2001.

[162] Harald Rueß and Natarajan Shankar. Deconstructing shostak. In *Logic In Computer Science (LICS)*, pages 19–28, Los Alamitos, CA, 2001. IEEE Computer Society.

[163] Stephan Schulz. System Abstract: E 0.61. In R. Goré, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, number 2083 in Lecture Notes in Artificial Intelligence, pages 370–375. Springer, 2001.

[164] Natarajan Shankar. Verification of Real-Time Systems Using PVS. In Costas Courcoubetis, editor, *Computer Aided Verification (CAV)*, volume 697 of *Lecture Notes in Computer Science*, pages 280–291, Elounda, Greece, June 1993. Springer-Verlag.

[165] Natarajan Shankar and Harald Rueß. Combining Shostak theories. In Sophie Tison, editor, *Rewriting Techniques and Applications (RTA)*, volume 2378 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, July 22-24 2002.

[166] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[167] Robert E. Shostak. Deciding combinations of theories. In Donald W. Loveland, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 138 of *Lecture Notes in Computer Science*, pages 209–222. Springer, 1982.

[168] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[169] Ofer Strichman. On solving Presburger and linear arithmetic with SAT. In Mark Aagaard and John W. O'Leary, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, volume 2517 of *Lecture Notes in Computer Science*, pages 160–170. Springer, November 2002.

[170] Ofer Strichman, Sanjit A. Seshia, and Randal E. Bryant. Deciding separation formulas with SAT. In Ed Brinksma and Kim G. Larsen, editors, *Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 265–279. Springer, July 2002.

[171] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: a cooperating validity checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer, 2002.

[172] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *Logic In Computer Science (LICS)*, pages 29–37. IEEE Computer Society, June 2001.

[173] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni, editors, *Proc. European Conference Logics in Artificial Intelligence (JELIA)*, volume 2424 of *Lecture Notes in Artificial Intelligence*, pages 308–319. Springer-Verlag, 2002.

[174] Cesare Tinelli and Mehdi T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In F. Baader and Klaus U. Schulz, editors, *Frontiers of Combining Systems (FroCoS)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, March 1996.

[175] Cesare Tinelli and Christophe Ringeissen. Unions of non-disjoint theories and combinations of satisfiability procedures. *Theoretical Computer Science*, 290(1):291–353, January 2003.

[176] Cesare Tinelli and Calogero G. Zarba. Combining non-stably infinite theories. In Ingo Dahn and Laurent Vigneron, editors, *Workshop on First Order Theorem Proving, (FTP)*, volume 86 of *Electronic Notes in Theoretical Computer Science*, Valencia, Spain, 2003. Elsevier Science Publishers.

[177] Cesare Tinelli and Calogero G. Zarba. Combining decision procedures for theories in sorted logics. Technical Report 04-01, Department of Computer Science, The University of Iowa, February 2004.

[178] Ashish Tiwari. *Decision procedures in automated deduction*. PhD thesis, Department of Computer Science, State University of New York at Stony Brook, 2000.

[179] Ashish Tiwari, Leo Bachmair, and Harald Rueß. Rigid E-unification revisited. In David McAllester, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 220–234, Pittsburgh, PA, June 2000. Springer-Verlag.

[180] Ashish Tiwari, Harald Rueß, Hassen Saïdi, and Natarajan Shankar. A technique for invariant generation. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 113–127, Genova, Italy, April 2001. Springer-Verlag.

[181] Tomás E. Uribe and Mark E. Stickel. Ordered binary decision diagrams and the Davis-Putnam procedure. In Jean-Pierre Jouannaud, editor, *Constraints in Computational Logics (CCL)*, volume 845 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 1994.

[182] Miroslav N. Velev and Randal E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW. In *Proceedings of the 38th Conference on Design Automation Conference 2001*, pages 226–231, New York, NY, USA, 2001. ACM press.

[183] Niklaus Wirth. *Systematic Programming: An Introduction*. Prentice-Hall, 1973.

[184] Pierre Wolper and Bernard Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proc. Static Analysis Symposium (SAS)*, volume 983 of *Lecture Notes in Computer Science*, pages 21–32, Glasgow, September 1995. Springer-Verlag.

[185] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97, Vancouver, July 1998. Springer-Verlag.

[186] Pierre Wolper and Bernard Boigelot. On the construction of automata from linear arithmetic constraints. In Susanne Graf and Michael I. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, March 2000. Springer-Verlag.

[187] Calogero G. Zarba. Combining lists with integers. In R. Goré, A. Leitsch, and T. Nipkov, editors, *International Joint Conference on*

Automated Reasoning, (IJCAR), Short Papers, Technical Report DII 11/01, pages 170–179, Siena, Italy, 2001.

[188] Calogero G. Zarba. Combining non-disjoint theories. In R. Goré, A. Leitsch, and T. Nipkov, editors, *International Joint Conference on Automated Reasoning (IJCAR), Short Papers*, Technical Report DII 11/01, pages 180–189, Siena, Italy, 2001. Springer.

[189] Calogero G. Zarba. Combining sets with elements. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 762–782. Springer, 2004.

[190] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 279–285, San Jose, California, USA, 2001. IEEE Computer Society.

[191] Lintao Zhang and Sharad Malik. The quest for efficient Boolean satisfiability solvers. In Andrei Voronkov, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 295–313. Springer-Verlag, 2002.

Appendix A

Some simplification rules

$$
\begin{aligned}
\neg\neg A &\rightarrow A \\
\neg A \Rightarrow \neg B &\rightarrow B \Rightarrow A \\
\neg A \equiv \neg B &\rightarrow A \equiv B \\
\text{if } \neg C \text{ then } A \text{ else } B &\rightarrow \text{if } C \text{ then } B \text{ else } A
\end{aligned}
$$

Figure A.1: Negation eliminating rules

$$
\begin{aligned}
t = t &\rightarrow \top & \qquad t \leq t &\rightarrow \top \\
t \neq t &\rightarrow \bot & \qquad t < t &\rightarrow \bot
\end{aligned}
$$

Figure A.2: Some simple rules for predicates

$$
\begin{aligned}
\text{if } \top \text{ then } t_1 \text{ else } t_2 &\rightarrow t_1 \\
\text{if } \bot \text{ then } t_1 \text{ else } t_2 &\rightarrow t_2 \\
\text{if } C \text{ then } t \text{ else } t &\rightarrow t
\end{aligned}
$$

Figure A.3: If-then-else term simplification rules

$$
\begin{aligned}
A_1 \wedge A_1 \wedge \ldots \wedge A_n &\rightarrow A_1 \wedge \ldots \wedge A_n \\
A_1 \wedge \neg A_1 \wedge \ldots \wedge A_n &\rightarrow \bot \\
A_1 \vee A_1 \vee \ldots \vee A_n &\rightarrow A_1 \vee \ldots \vee A_n \\
A_1 \vee \neg A_1 \vee \ldots \vee A_n &\rightarrow \top
\end{aligned}
$$

$$
\begin{aligned}
A \Rightarrow A &\rightarrow \top \\
A \Rightarrow \neg A &\rightarrow \neg A \\
\neg A \Rightarrow A &\rightarrow A \\
A \equiv A &\rightarrow \top \\
A \equiv \neg A &\rightarrow \bot
\end{aligned}
$$

$$
\begin{aligned}
\text{if } A \text{ then } A \text{ else } B &\rightarrow A \vee B \\
\text{if } A \text{ then } \neg A \text{ else } B &\rightarrow \neg A \wedge B \\
\text{if } A \text{ then } B \text{ else } A &\rightarrow A \wedge B \\
\text{if } A \text{ then } B \text{ else } \neg A &\rightarrow \neg A \vee B
\end{aligned}
$$

$$
\begin{aligned}
\text{if } C \text{ then } A \text{ else } A &\rightarrow A \\
\text{if } C \text{ then } A \text{ else } \neg A &\rightarrow C \equiv A \\
\text{if } C \text{ then } \neg A \text{ else } A &\rightarrow C \equiv \neg A
\end{aligned}
$$

Figure A.4: Repeated subformulas

Appendix B

Summary of formulas associated to statements

Statement	$\delta\,(\mathcal{M}, \mathcal{M}')$
skip	$\mathrm{pres}\,(\mathcal{M}, \mathcal{M}, \mathcal{M}')$
$\forall \vec{u_1}\ v_1[\vec{u_1}] := E_1(\vec{u_1}) \parallel \ldots$ $\forall \vec{u_n}\ v_n[\vec{u_n}] := E_n(\vec{u_n})$	$\mathrm{pres}\,(\mathcal{M} \setminus \{v_1, \ldots v_n\}, \mathcal{M}, \mathcal{M}')$ $\wedge \bigwedge_{1 \le i \le n} \forall \vec{u_i}\ \ (v_i[\vec{u_i}])\,[\mathcal{M} \to \mathcal{M}'] = E_i(\vec{u_i})$
$S_1; S_2$	$\exists \mathcal{M}''[\delta\,[S_1]\,(\mathcal{M}, \mathcal{M}'') \wedge \delta\,[S_2]\,(\mathcal{M}'', \mathcal{M}')\,]$
if C **then** S_1 **else** S_2	$[C\,(\mathcal{M}) \Rightarrow \delta\,[S_1]\,(\mathcal{M}, \mathcal{M}')\,]$ $\wedge\,[\neg C\,(\mathcal{M}) \Rightarrow \delta\,[S_2]\,(\mathcal{M}, \mathcal{M}')\,]$ $[C\,(\mathcal{M}) \wedge \delta\,[S_1]\,(\mathcal{M}, \mathcal{M}')\,]$ $\vee\,[\neg C\,(\mathcal{M}) \wedge \delta\,[S_2]\,(\mathcal{M}, \mathcal{M}')\,]$

Statement	$\Omega\,(\mathcal{M})$
skip	$\top$
$\forall \vec{u_1}\ v_1[\vec{u_1}] := E_1(\vec{u_1}) \parallel \ldots$ $\forall \vec{u_n}\ v_n[\vec{u_n}] := E_n(\vec{u_n})$	$\bigwedge_{1 \le i \le n} \forall \vec{u_i}\ \Omega\,[E_i(\vec{u_i})]\,(\mathcal{M})$
$S_1; S_2$	$\Omega\,[S_1]\,(\mathcal{M}) \wedge \forall \mathcal{M}''[\delta\,[S_1]\,(\mathcal{M}, \mathcal{M}'') \Rightarrow \Omega\,[S_2]\,(\mathcal{M}'')\,]$
if C **then** S_1 **else** S_2	$\Omega\,[C]\,(\mathcal{M}) \wedge [C\,(\mathcal{M}) \Rightarrow \Omega\,[S_1]\,(\mathcal{M})\,]$ $\wedge\,[\neg C\,(\mathcal{M}) \Rightarrow \Omega\,[S_2]\,(\mathcal{M})\,]$

Appendix C

Congruence closure procedures

```
function UNION(a, b);
    var DEDUCT : set of TERM × TERM;
    begin
        if |a.TERMS| < |b.TERMS| then (a, b) := (b, a);
        for each x ∈ b.TERMS do FIND(x) := a;
        a.TERMS := a.TERMS ∪ b.TERMS;
        DEDUCT := ∅;
        if |b.PRED| > |a.PRED| then
            (a.S, b.S, a.PRED, b.PRED) := (b.S, a.S, b.PRED, a.PRED);
        for each x ∈ b.PRED do
            if QUERY(x) = ∅ then
                ENTER(x);
            else if FIND(x) ≠ FIND(QUERY(x)) then
                DEDUCT := DEDUCT ∪ (x, QUERY(x));
        a.PRED := a.PRED ∪ b.PRED;
        return DEDUCT;
    end
```

Figure C.1: Efficient function UNION

function ADD_EQUALITY(ℓ);
  var u, v : TERM;
  begin
    let u, v **such that** $u = v$ **is** ℓ;
    ADD_NODE(u);
    ADD_NODE(v);
    MERGE ($u, v, \{\ell\}$);
    return STATUS;
  end

function ADD_INEQUALITY(ℓ);
  var u, v : TERM;
  begin
    let u, v **such that** $u \neq v$ **is** ℓ;
    ADD_NODE(u);
    ADD_NODE(v);
    if FIND(u) = FIND(v) **then** STATUS = $\perp$;
    FIND(u).INEQ := FIND(u).INEQ $\cup \{\ell\}$;
    FIND(v).INEQ := FIND(v).INEQ $\cup \{\ell\}$;
    return STATUS;
  end

Figure C.2: Final procedures ADD_(IN)EQUALITY

```
procedure ADD_NODE(u);
begin
    if FIND(u) ≠ ∅ then return;
    for i := 1 to δ(u) do ADD_NODE(u[i]);
    u.EQ = ∅;
    FIND(u) := NEWCLASS;
    FIND(u).S := u;
    FIND(u).TERM := {u};
    FIND(u).PRED := ∅;
    FIND(u).INEQ = ∅;
    if δ(u) > 0 then
        if QUERY(u) then
            begin
                UNION(FIND(u), FIND(QUERY(u)));
                (u.EQ, QUERY(u).EQ) :=
                        ({(u, v, ∅)}, QUERY(u).EQ ∪ {(u, v, ∅)});
            end
        else
            begin
                ENTER(u);
                for i := 1 to δ(u) do
                    FIND(u[i]).PRED := FIND(u[i]).PRED ∪ {u};
            end
end
```

Figure C.3: Final procedure ADD_NODE

```
procedure MERGE(a, b, E);
var COMBINE : set of TERM × TERM;
begin
    COMBINE := {(a, b)};
    repeat
        (u, v) := element of COMBINE;
        COMBINE := COMBINE \ {(u, v)};
        if FIND(u) ≠ FIND(v) then
            begin
                COMBINE := COMBINE ∪ UNION(FIND(u), FIND(v));
                (u.EQ, v.EQ, E) :=
                        (u.EQ ∪ {(u, v, E)}, v.EQ ∪ {(u, v, E)}, ∅);
            end
    until COMBINE = ∅;
end
```

Figure C.4: Final procedure MERGE

```
function Union(a, b);
    var deduct : set of Term × Term;
    begin
        if |b.ineq| > |a.ineq| then (a, b) := (b, a);
        for each (u ≠ v) ∈ b.ineq do
            if {Find(u), Find(v)} = {a, b} then status = ⊥;
        if |a.terms| < |b.terms| then (a, b) := (b, a);
        for each x ∈ b.terms do Find(x) := a;
        a.terms := a.terms ∪ b.terms;
        deduct := ∅;
        if |b.pred| > |a.pred| then
            (a.s, b.s, a.pred, b.pred) := (b.s, a.s, b.pred, a.pred);
        for each x ∈ b.pred do
            if query(x) = ∅ then
                enter(x);
            else if Find(x) ≠ Find(query(x)) then
                deduct := deduct ∪ (x, query(x));
        a.pred := a.pred ∪ b.pred;
        a.ineq := a.ineq ∪ b.ineq;
        return deduct;
    end
```

Figure C.5: Final function Union