République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique
Université des Sciences et de la Technologie
HOUARI BOUMEDIENE
B. P. 32, El-Alia, 16111 Bab-Ezzouar, ALGER
Téléphone : (213-2) 51 55 75 Fax : (213-2) 51 59 92
Télex : 64 343 USTA-DZ

جامعة هواري بومدين
للعلوم والتكنولوجيا
U S T H B

الجمهورية الجزائرية الديمقراطية الشعبية
وزارة التعليم العالي
والبحث العلمي
جامعة هواري بومدين
للعلوم والتكنولوجيا
ص.ب 32، العاليا، 16111، باب الزوار، الجزائر
الهاتف: 75 55 51 (2-213) الفاكس: 92 59 51 (2-213)
تلكس: 64 343 USTA-DZ

N° Ordre: 01/2008-M/IN

UNIVERSITE DES SCIENCES ET TECHNOLOGIES HOUARI BOUMEDIEN

USTHB/ALGER

FACULTE de Génie Eléctrique et Informatique

MEMOIRE
Présenté pour l'optention du diplôme de : MAGISTER
EN: Informatique
Spécialité: Programmation et systèmes

Présenté par:
SAIDI Selma

Optimization and Implementation of techniques for the Verification of Infinite Data Softwares

Soutenu le 11/03/2008, devant le jury composé de:

Mme M.Bouakala, Professeur, USTHB	Président
Mr M.Ahmed Nacer, Professeur,USTHB	Directeur de thèse
Mme Gherbi, Docteur, USTHB	Examinateur
Mr Boukraâ, Docteur, USTHB	Examinateur

Contents

Chapter 1

Introduction

The verification of software systems requires in general the consideration of infinite-state models. The sources of infinity in softwares are multiple. One of them is the manipulation of variables and data structures ranging over infinite domains (such as integers, reals, arrays, etc). Another source of infinity is the fact that the number of processes running in parallel in the system can be either a parameter (fixed but arbitrarily large), or it can be dynamically changing due to process creation.

For example, protocols for mobile phones like PGM (Pragmatic General Multicast) protocol (RFC 3208, [3]) suppose the transport of information from multiple sources (phones) to multiple receivers (phones) and the number of participants is not fixed. Moreover, a participant may enter dynamically in the spool of users. The protocol PGM specifies for each participant a number of counters to be stored in order to ensure reliable transmission and reception of data. The verification of such protocols is important because their behavior is very complex and the impact of a bug may produce important financial problems.

In the last ten years, several approaches have been proposed for the verification of infinite-state systems taking into account either the aspects related to infinite data domains, or the aspects related to unbounded network structures due to parametrization or dynamism. (While parametric systems are *static* networks of any (infinite) number of processes, dynamic systems may involve the change of the network by creation or termination of processes.) However, only few works addressed the verification problem taking into account both infinite data manipulation and parametric/dynamic network structures.

In [11] is proposed a generic framework for reasoning about parametrized and dynamic networks of concurrent processes which can manipulate (local and global) variables over infinite data domains. The framework is parametrized by a data domain (which may be a product of domains) and a first-order theory on it (e.g., Presburger arithmetic on natural numbers). It consists of (1) expressive models, called CPN, allowing to cover a wide class of systems, and (2) a logic, called CML, allowing to specify and to reason about the configurations of these models.

A positive result is that the satisfiability problem (i.e., checking that the formula have a model) is decidable for the fragment $\exists^*\forall^*$ of CML whenever the underlying color logic has a decidable satisfiability problem, e.g., Presburger arithmetics, the first-order logic of addition and multiplication over reals, etc.

Moreover, it is proved that the fragment $\exists^*\forall^*$ of CML is effectively closed under post and pre image computations (i.e., computation of immediate successors and immediate predecessors) for CPN's where all transition guards are also in $\exists^*\forall^*$.

These generic decidability and closure results can be applied in the verification of CPN models following different approaches such as pre-post condition (Hoare triples based) reasoning, bounded reachability analysis, and inductive invariant checking.

The theoretical results published in [11] have been implemented last year in a prototype using C++ and the Mona tool for the decision procedure for first order logic on integers. However, this implementation is not able to deal with simple examples because it implements naively the theoretical algorithms proposed. Indeed, it suffers from an explosion of the size of formulas generated for the invariance checking. Moreover, it can only consider models with integer data.

Our work consists in providing a new implementation such that:

- The logic of colors has to be as much generic as possible. The user may choose the domain of this logic, its operations and relations.
- The theoretical algorithms for invariance checking shall be implemented efficiently. For this, we have to identify some special cases of systems or invariants that allow efficient implementation and simplifications.
- The implementation has to benefit from the recent advances on satisfaction modulo theory (SMT) domain and reuse, as much as possible these results.

Although the accent is put on the implementation, an important part of this work is theoretical: understand verification techniques (e.g., invariant checking), understand the SMT tools and their limits, define formally the algorithms used to obtain an efficient implementation, etc. Moreover, the language used for the implementation, OCAML, belongs to an interesting family of languages, the functional languages, which is new for me.

This document is organized in two parts. The first part presents the context of the work. It contains two chapters as follows:

Chapter 2 provides a short introduction to the principles and techniques used for the software verification. It allows to introduce techniques like model-checking, proof, abstraction, and SMT. Two examples of implementation of these techniques are shortly described: the verification system WHY and the SMT solver YICES.

Chapter 3 is mainly an abstracted version of the work presented in [11]. It provides the main definitions of CPN and CML, the main theoretical results obtained, and an example of modeling with CPN, the Reader-Writer lock.

The second part presents our contribution in building a verification system for CPN. It contains three chapters as follows:

Chapter 4 shows how the language WHY is used to encode CPN models and the abstract syntax tree provided after the lexical and syntactical analysis of the WHY code. The Reader-Writer lock is encoded in WHY.

Chapter 5 presents the work done for the invariance checking. Several optimizations are presented theoretically and how they are implemented in practice. Experimental results are provided on the Reader-Writer lock example.

Chapter 6 shows how the satisfaction procedure presented in [11] is implemented for formulas generated from the invariance checking problem.

Chapter 7 concludes this work and gives some directions for future works.

Part I

The Context of the Work

Chapter 2

Software Verification

2.1 Introduction

"*The overall Resolve vision is that of a future in which no production software is considered properly engineered unless it has been fully specified, and fully verified as satisfying these specifications.*" [27]

Software verification is a broad and complex discipline of software engineering whose goal is to assure that a software fully satisfies all the expected requirements. Actually it is largely applied to software embedded in systems which are critical by their cost or their complexity (planes, launchers, cars, etc.). Recently, several big software editors are developing and using systems for verification of the produced software.

This chapter provides a short introduction to the principles and techniques actually used for the software verification. It introduces techniques like model-checking, proof, abstraction, and SMT. Two examples of implementation of these techniques are shortly described: the verification system WHY and the SMT solver YICES.

2.2 Definition of Formal Verification

For a hardware or software systems, formal verification is the act of proving or disproving the correctness of the system with respect to a certain formal specification or property, using formal methods of mathematics. Indeed, the process of testing the system cannot prove that the system does not contain any defects. Neither can it prove that it does have a certain property. Only the process of formal verification can prove that a system does not have a certain defect or does have a certain property. It is impossible to prove or test that a system has "no defect" since it is impossible to formally specify what "no defect" means. All that can be done is prove that a system does not have any of the defects that can be thought of, and has all of the properties that together make it functional and useful.

Formal verification can be used for systems such as cryptographic protocols, combinational circuits, digital circuits with internal memory, and software expressed as source code.

The verification of these systems is done by providing a formal proof on an abstract mathematical model of the system, the correspondence between the mathematical model and the nature of the system being otherwise known by construction or obtained by applying abstraction tools. Examples of mathematical objects often used to model systems are: finite state machines, labelled transition systems, Petri nets, timed automata, hybrid automata, process algebra, formal semantics of programming languages such as operational semantics, denotational semantics, axiomatic semantics and Hoare logic. The properties that shall be checked on these models may be expressed using different formalisms, e.g., first order formulas, temporal logic formulas, labelled transition systems, etc. Figure 2.1 resumes the components of the verification process.

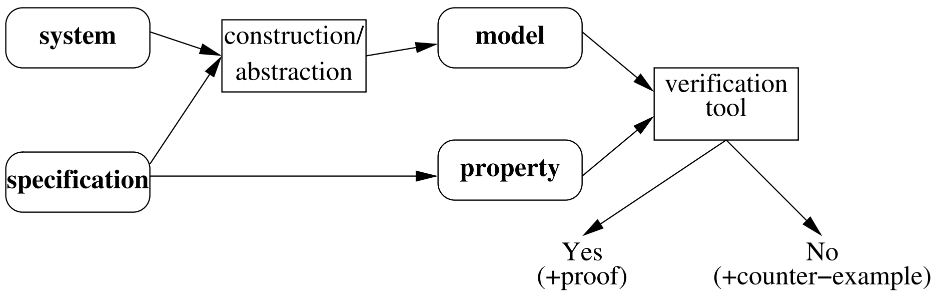

Figure 2.1: General approach for the verification process.

2.3 Approaches to Formal Verification

There are roughly two approaches to formal verification.

The first approach is called model-checking and has as inputs a (state-transition) model for the system under verification and a property about the reachable configurations of the model. The property may be either a logical property (given, e.g., in temporal logic) or another model built from the specification. Then, the model-checking consists of a systematically exhaustive exploration of the model while the property tested is true. Usually this consists of exploring all states and transitions in the model built from the intersection of the initial model with the property. This process is fully automatic. Since this exploration may be intensive time and memory consuming, smart techniques have been proposed to represent compactly sets of states (e.g., BDD) and to reduce the computing time of applying transitions. Moreover, sound abstraction techniques are usually applied to obtain results when the models are very big.

Model-checking has been first applied to finite state models, and then extended to infinite models where infinite sets of states can be effectively (finitely) represented. A successful approach to finitely represent infinite sets of states is the use of constraint systems. In such systems, the sets of states are represented using a set (conjunction) of constraints over variables belonging to an appropriate domain (Boolean for finite variables, integers or reals for infinite variables). Then, the model-checking can be reduced at checking the satisfaction of several systems of constrains. For this, SAT solvers can be used.

The second approach is the logical inference. The model and the property are both modelled by a logical system and a tool for mathematical reasoning is used. Such a tool is called a theorem prover. This process can be only partially automated and it is driven by the user's understanding of the system to validate.

2.4 SAT-based Formal Verification

As mentionned in the previous section, SAT solvers can be an useful tool for the model-checking techniques: explored sets of states can be represented as constraints and transitions between these sets can be computed by reduction to satisfaction of constraints.

Dramatic improvements in SAT solver technology over the last decade have accelerated the research in verification methods based on SAT solvers. Several powerful SAT-solvers have been developed [23, 25, 26, 31].

Verification methods based on these solvers have been shown to push the limit of verification in terms of both size of the systems dealt and efficiency, as reported in several academic and industrial case studies [6, 9, 10, 13]. Reversely, this has raised further the research activity in the area of SAT-solvers developement.

2.4.1 Boolean SAT Problem and Solvers

The Boolean satisfiability problem (SAT) asks whether a given propositional formula is satisfiable. It is of central importance in various areas of computer science, including theoretical computer science, algorithmics, artificial intelligence, hardware design and verification.

The input of the SAT problem is a Boolean expression written using only logical operators (AND, OR, NOT), variables, and parentheses. The problem can be reformulated in: given such an expression, is there some assignment of TRUE and FALSE values to the variables that will make the entire expression true? A formula of propositional logic is said to be satisfiable if logical values can be assigned to its variables in a way that makes the formula true [1]. The SAT problem is known to be NP-Complete [22]. However, in practice, there has been tremendous progress in SAT-solvers technology over the years, summarized in a recent survey [32]. Earlier work in the context of theorem proving is covered in [12].

Most SAT solvers use a Conjunctive Normal Form (CNF) representation of the Boolean formula. In CNF, the formula is represented as a conjunction of clauses, each clause is a disjunction of literals, and a literal is a variable or its negation. Note that in order for the formula to be satisfied, each clause must also be satisfied, i.e., evaluate to true. There exist polynomial algorithms to transform an arbitrary propositional formula into a satisfiability equivalent CNF formula that is satisfiable if and only if the original formula is satisfiable.

The modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [14, 15], which performs takes as input a formula in CNF and performs a branching search with backtracking for valuations of variables. The DPLL algorithm is sound and complete, i.e., it finds a solution if and only if the formula is satisfiable.

In the following, we summarize the main features of modern DPLL-based SAT solvers. The basic skeleton of DPLL-based SAT solvers is shown in the listing below, adapted from the GRASP work [25]:

```
sat-solve() {
 if preprocess() = CONFLICT
 then return UNSAT;

 while TRUE do
   if not decide-next-branch()
   then return SAT;
```

```
    while deduce() = CONFLICT do
      blevel = analyze-conflict();
      if blevel = 0
      then return UNSAT;
      backtrack (blevel);
    done;
  done;
```

The initial step consists of some preprocessing, during which it may be discovered that the formula is unsatisfiable. The outer loop starts by choosing an unassigned variable, and a value to assign to it (decide-next-branch). If no such variable exists, a solution has been found. Otherwise, the variable assignments deducible from this decision are made (using deduce), through a procedure called Boolean Constraint Propagation (BCP). It typically consists of iterative application of the unit clause rule, which is invoked whenever a clause becomes a unit clause, i.e., all but one of its literals are false and the remaining literal is unassigned. According to the rule, the last unassigned literal is implied to be true this avoids the search path where the last literal is also false, since such a path cannot lead to a solution. A conflict occurs when a variable is implied to be true as well as false. If no conflict is discovered during BCP, then the outer loop is repeated, by choosing the next variable for making a decision. However, if a conflict does occur, backtracking is performed within an inner loop in order to undo some decisions and their implications. If all decisions need to be undone (i.e., the backtracking level blevel is 0), the formula is declared unsatisfiable since the entire search space has been exhausted.

The original DPLL algorithm used chronological backtracking, i.e., it would backtrack up to the most recent decision, for which the other value of the variable had not been tried. However, modern SAT solvers use conflict analysis techniques (shown as (analyze-conflict) in the algorithm), to analyze the reasons for a conflict. Conflict analysis is used to perform conflict-driven learning and conflict-driven backtracking. Conflict-driven learning consists of adding conflict clauses to the formula, in order to avoid the same conflict in the future. Conflict-driven backtracking allows nonchronological backtracking, i.e., up to the closest decision which caused the conflict. These techniques greatly improve the performance of the SAT solver on structured problems, i.e., problems where the conjunctions belonging to the formula have a symmetric form.

Examples of Boolean SAT solvers are: GRASP [25], GSAT [30], relsat [8], WALKSAT [29], etc.

2.4.2 SAT Modulo Theories Problem and Solvers

An extension of the Boolean SAT-solvers that has gained significant popularity since 2003 is satisfiability modulo theories (SMT). It enrich CNF formulas by allowing that some of binary variables are replaced by predicates over a suitable set of non-binary variables. These predicates are classified according to the theory the non-binary variables belong to. For instance, linear inequalities over real variables are evaluated using the rules of the theory of linear real arithmetic, whereas predicates involving uninterpreted terms and function symbols are evaluated using the rules of the theory of uninterpreted functions with equality. Such extensions typically remain NP-complete, but very efficient solvers are now available that can handle many such kinds of constraints [1].

Then, the SMT problem is a decision problem for logical formulas over a signature (i.e., a domain, a set of functions and a set of relations) which have a particular meaning given by a theory. For example, the signature $(\mathbb{N}, \{+\}, \{\leq, <\})$ represents natural numbers with addition and comparison relation. The theory corresponding to this signature defines the "+" function and the comparison relations by logical formulas, called axioms. One of them is the neutrality of 0 for addition, i.e., $\forall x.\ x + 0 = x$. Most SMT solvers support only quantifier free fragments of their logics.

Early attempts for solving SMT instances involved translating them to Boolean SAT instances (e.g., a 32-bit integer variable would be encoded by 32 bit variables with appropriate weights and word-level operations such as "plus" would be replaced by lower-level logic operations on the bits) and passing this formula to a Boolean SAT solver. This approach has its merits: by pre-processing the SMT formula into an equivalent Boolean SAT formula we can use existing Boolean SAT solvers "as-is" and leverage their performance and capacity improvements over time. On the other hand, the loss of the high-level semantics of the underlying theories means that the Boolean SAT solver has to work a lot harder than necessary to discover "obvious" facts (such as x + y = y + x for integer addition.)

This observation led to the development of a number of SMT solvers that tightly integrate the Boolean reasoning of a DPLL-style search with theory-specific solvers that handle conjunctions (ANDs) of predicates from a given theory [1]. Because the problem of checking the validity of a ground formula φ in a theory T is equivalent to checking that no interpretation of T satisfies $\neg\varphi$, the literature on the subject more often speaks in terms of satisfiability in T.

The use of SMT solvers in formal methods is not new. It was championed in the early 1980s by Greg Nelson and Derek Oppen at Stanford University, by Robert Shostak at SRI, and by Robert Boyer and J. Moore at the University of Texas at Austin. Building on this work, several SMT solvers have been developed in academia and industry with continually increasing scope and performance. Some of them have or are being integrated into: interactive theorem provers for high-order logic (such as HOL and Isabelle); extended static checkers (such as CAsCaDE, Boogie, and ESC/-Java 2); verification systems (such as ACL2, Caduceus, SAL, UCLID and WHY); formal CASE environments (such as KeY); model checkers (such as BLAST, MAGIC and SLAM); certifying compilers (such as Touchstone); unit test generators (such as CUTE and MUTT). In industry, there are currently SMT-related projects at Cadence Berkeley Labs, Intel Strategic CAD Labs, Microsoft Research, and NEC Labs, just to name some.

In the remaining of this chapter, we present two of the tools above that we used in this work. First, we present the SMT solver Yices and then the verification system WHY.

2.4.3 The Yices Tool

Yices is an efficient SMT solver developed at SRI (USA) that decides the satisfiability of arbitrary formulas containing uninterpreted function symbols with equality, linear real and integer arithmetic, quantifiers, etc. Yices has its own input language but also supports the common language SMT-LIB defined to be the common language for several SMT provers. Yices is available at [5]

Yices is implemented in C++, it uses the Nelson-Oppen method for combining decision procedures. Yices is based on a generalized search engine which supports different kinds of case splits and constraint propagation rules. Yices tracks which atoms are relevant/irrelevant for the satisfiability of the whole formula, this feature is specially useful for handling expensive theories (e.g., arrays), and to control the instantiation of quantified formulas. Every deduction step in Yices is associated with a proof object. The proofs of unsatisfiability produced by Yices are composed by a sequence of lemmas and a main theorem [16].

2.4.4 Verification System WHY

The verification tool WHY [4] was developed by Jean-christophe Filliâtre at the university of Paris Sud.

Basically, the WHY tool takes annotated programs written in a very simple imperative programming language of its own, produces verification conditions and sends them to existing provers (proof assistants such as Coq, PVS, etc. or automatic provers such as Simplify, CVC Lite, etc.) as illustrated below [19]:

$$\begin{tikzcd}
& \text{annotated programs} \arrow[d] & \\
& \text{WHY} \arrow[d] & \\
& \text{verification conditions = first-order formulas} \arrow[d] & \\
& \text{WHY} \arrow[dl] \arrow[dr] & \\
\text{Interactive provers} \arrow[d] & & \text{Automatic Provers} \arrow[d] \\
\text{(Coq, PVS, Isabelle/HOL, etc.)} & & \text{(Simplify, Yices, Ergo, CVC3, etc.)}
\end{tikzcd}$$

Indeed, the general approach is to generate Verification Conditions (VC): logical formulas whose validity implies the soundness of the code with respect to the given specifications. These VCs must be discharged by any theorem prover. Additionally, VCs are generated to guarantee the absence of run-time errors: null pointer dereferencing, out-of-bounds array access, etc [20].

This tool ressembles many others. However, it relies on a technology and on some design choices which are less common. It differs from other systems in that it accepts several languages as input (currently C and ML, and Java with the help of the companion tool Krakatoa) and outputs conditions for severals existing provers (currently Coq, PVS, HOL Light, Mizar, simplify and haRVey). It also provides a geat safety through some de Bruijn criterion: once the obligations are established, a proof that the program satisfies its specification is built and type-checked automatically [17]. In particular there is a unique, stand-alone verification condition generator called Why, which is able to output VCs in the syntax of many provers, both automatic and interactive ones [20].

The Why tool implements a programming language designed for the verification of sequential programs. This is an intermediate language to which existing programming languages can be compiled and from which verification conditions can be computed [18].

Indeed, Why is not limited to one input language. Instead, it provides its own internal language -let us call it WL from now on- is a small ML-like language with imperative features(references and arrays), exceptions and annotations. In the ML tradition, WL merges expressions, statements, local variables and functions into a single syntactic class, which eases symbolic manipulations and limits the number of cases to consider when computing weakest preconditions or verification conditions. Similarly, exceptions are used to deal with abrupttermінations such as return, break or continue when translating C or Java programs, and thus there is no need to implement special rules for these constructs; the rules for exceptions are giving the excepted verification conditions [17]. WHY currently interprets C programs, and Java programs with the help of the companion tool Krakatoa [24]. WHY also provides an input syntax for its internal language WL, which is very close to a subset of Objective Caml [2].

In the following, we present a trivial example to illustrate the mechanism of WHY.

A Trivial Example Here is a small example of WHY input code:

```
logic min: int, int -> int

parameter r: int ref

let f (n:int) = r := min !r n {r <= r@}
```

This code declares a function symbol min and gives its arity. Whatever the status of this function is on the prover side (primitive, user-defined, axiomatized, etc.), it simply needs to be declared in order to be used in the following of the code. The next line declares a parameter, that is a value that is not defined but simply *assumed* to exist i.e. to belong to the environment. Here the parameter has name r and is an integer reference (WHY's concrete syntax is very close to Ocaml's syntax). The third line defines a function f taking a integer n as argument (the type has to be given since there is no type inference in WHY) and assigning to r the value of min !r n. The function f has no precondition and a postcondition expressing that the final value of r is smaller than its initial value. The current value of a reference x is directly denoted by x within annotations (not !x) and within postconditions x@ is the notation for the value of x in the prestate(i.e. at the precondition point).

Let us assume the three lines code above to be in file test.why. Trying an automatic decision procedure is an easy as running WHY with a command line option. For instance, to use Yices , we type in

```
why --yices test.why
```

A yices input file test_why.ys is produced.

2.5 Conclusion

We presented shortly in this chapter concepts involved in the software verification, especially verification methods based on SAT solvers as they have recently emerged as a promising solution.

We conclude by giving further references for lecture to the reader.

A basic presentation of the SAT and SMT solvers is given in the book of Bradley and Manna (Springer 2007). This book also presents the Nelson Oppen decision procedure and other decision procedures (e.g., on arrays).

[28] is a very rich overview of the SMT techniques and their application to formal verification.

Chapter 3

Constrainted Petri Nets and their Verification

3.1 Introduction

This chapter introduces the main definitions and results proposed in [11].

The models in proposed in [11] to model unbounded dynamic networks of infinite state systems are called Constrained Petri Nets (CPN for short). They are based on *(place/transition) Petri nets* where tokens are colored by data values. Intuitively, tokens represent different occurrences of processes, and places are associated with control locations and contain tokens corresponding to processes which are at a same control location. Since processes can manipulate local variables, each token (process occurrence) has several colors corresponding to the values of these variables. Then, configurations of CPN models are *markings* where each place contains a set of colored tokens, and transitions modify the markings as usual by removing tokens from some places and creating new ones in some other places. Transitions are guarded by constraints on the colors of tokens before and after firing the transition.

In the papers is shown that CPNs allow to model various aspects such as unbounded dynamic creation of processes, manipulation of local and global variables over unbounded domains such as integers, synchronization, communication through shared variables, locks, etc.

Concerning verification, a logic is proposed for specifying configurations of CPN, logic called Colored Markings Logic (CML for short). It is a first order logic over tokens and their colors. It allows to reason about the presence of tokens in places, and also about the relations between the colors of these tokens. The logic CML is parametrized by a first order logic over the color domain allowing to express constraints on tokens. For example, it allows to reason about the presence of tokens in places, and about the colors of these tokens. For instance, it is possible to express in CML that *there exists a token x in the place p and a token y in the place q such that the colors of x and y satisfy some constraint $\phi_1(x, y)$, and for every token z in the place r, the colors of x and r satisfy some other constraint $\phi_2(x, z)$.*

Bouajjani et all. show that the CML is decidable for finite color domains (such as booleans), but as soon as is added an infinite domain color (e.g., naturals) with the usual ordering relation (and without any arithmetical operations), CML becomes undecidable for a fragment where universal quantifiers over tokens precedes existential ones (i.e., the fragment $\forall^*\exists^*$ of the logic).

These results have been used in [11] to deal with the parametric verification of a Reader-Writer lock with an arbitrarily large number of processes. This case study was introduced in [21] where the authors provide a correction proof for the case of one reader and one writer.

3.2 Colored Marking Logic (CML)

The CML logic is built on the notions of colors, tokens, coloring symbols and places, that are defined in the following.

Let $\mathbb{C}$ be an infinite *domain of values* over which are defined functions (constant or or) in Ω and relations in Ξ. Intuitively, the domain $\mathbb{C}$ is the domain of data used in the software system and Ω and Ξ are operators on these data.

Let $\mathbb{T}$ be an enumerable set of *tokens*. Intuitively, tokens represent occurrences of (parallel) processes. We assume that tokens may have colors corresponding for instance to data values attached to the corresponding processes.

Colors are associated with tokens through *coloring functions*. Let Γ be a finite set of *token coloring symbols*. Each element in Γ is interpreted as a mapping from $\mathbb{T}$ (the set of tokens) to $\mathbb{C}$ (the set of colors). Then, let a valuation of the token coloring symbols be a mapping in $[\Gamma \to (\mathbb{N} \to \mathbb{C})]$.

Tokens can be located at *places*. Let $\mathbb{P}$ be a finite set of such places. A *marking* is a mapping in $[\mathbb{T} \to \mathbb{P} \cup \{\perp\}]$ which associates with each token in $\mathbb{T}$ the unique place where it is located if it is defined, or $\perp$ otherwise. A *colored marking* is a pair $\langle M, \mu \rangle$ where M is a marking and μ is a valuation of the token coloring symbols.

Let T be set of *token variables* (which take values in $\mathbb{T}$) and let C be set of *color variables* (which take values in $\mathbb{C}$), and assume that $T \cap C = \emptyset$. The set of CML terms (called *token color terms*) is given by the grammar:

$$t ::= z \mid \gamma(x) \mid \omega(t_1, \dots, t_n)$$

where $z \in C$, $\gamma \in \Gamma$, $x \in T$, and $\omega \in \Omega$. Then, the set of CML formulas is given by:

$$\varphi ::= x = y \mid p(x) \mid \xi(t_1, \dots, t_m) \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists z.\ \varphi \mid \exists x.\ \varphi$$

where $x, y \in T$, $z \in C$, $p \in \mathbb{P} \cup \{\bot\}$, $\xi \in \Xi$, and $t_1, \dots, t_m$ are token color terms. Boolean connectives such as conjunction ($\wedge$) and implication ($\Rightarrow$), and universal quantification ($\forall$) can be defined in terms of $\neg$, $\vee$, and $\exists$.

The notation $\exists x \in p.\ \varphi$ (resp. $\forall x \in p.\ \varphi$) is used as an abbreviation of the formula $\exists x.\ p(x) \wedge \varphi$ (resp. $\forall x.\ p(x) \Rightarrow \varphi$).

The notions of free/bound occurrences of variables in formulas and the notions of closed/open formulas are defined as usual in first-order logics [7]. In the sequel, we assume w.l.o.g. that in every formula, each variable is quantified at most once.

The semantics of CML formulas is defined formally in [11].

3.2.1 Syntactical Forms and Fragments

Prenex Normal Form

A formula is in *prenex normal form* (PNF) if it is of the form

$$Q_1 y_1 Q_2 y_2 \dots Q_m y_m.\ \varphi$$

where (1) $Q_1, \dots, Q_m$ are (existential or universal) quantifiers, (2) $y_1, \dots, y_m$ are variables in $T \cup C$, and φ is a quantifier-free formula. It can be proved that for every formula φ in CML, there exists an equivalent formula φ' in prenex normal form.

Quantifier Alternation Hierarchy

Let consider two families $\{\Sigma_n\}_{n \geq 0}$ and $\{\Pi_n\}_{n \geq 0}$ of fragments of CML defined according to the alternation depth of existential and universal quantifiers in their PNF:

- Let $\Sigma_0 = \Pi_0$ be the set of formulas in PNF where all quantified variables are in C,

- For $n \geq 0$, let Σ_{n+1} (resp. Π_{n+1}) be the set of formulas $Qy_1 \ldots y_m.\ \varphi$ in PNF where $y_1, \ldots, y_m \in T \cup C$, Q is the existential (resp. universal) quantifier $\exists$ (resp. $\forall$), and φ is a formula in Π_n (resp. Σ_n).

It is easy to see that, for every $n \geq 0$, Σ_n and Π_n are closed under conjunction and disjunction, and that the negation of a Σ_n formula is a Π_n formula and vice versa. For every $n \geq 0$, let $B(\Sigma_n)$ denote the set of all boolean combinations of Σ_n formulas. Clearly, $B(\Sigma_n)$ subsumes both Σ_n and Π_n, and is included in both Σ_{n+1} and Π_{n+1}.

Special Form

The set of formulas in special form is given by the grammar:

$$\varphi ::= x = y \mid \xi(t_1, \ldots, t_n) \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists z.\ \varphi \mid \exists x \in p.\ \varphi$$

where $x, y \in T$, $z \in C$, $p \in \mathbb{P} \cup \{\bot\}$, $\xi \in \Xi$, and $t_1, \ldots, t_n$ are token color terms.

3.2.2 Satisfiability Problem for CML

Bouajjani and all investigate the decidability of the satisfiability problem of the logic $\mathsf{CML}(L)$, assuming that the underlying color logic L has a decidable satisfiability problem.

Let us mention that in the case of a finite color domain, for instance for the domain of booleans with equality and usual operations, the logic CML is decidable.

In [11] is proved that the satisfiability problem for formula in the Σ_2 fragment of CML is decidable. The proof is giving a procedure to test satisfiability which is discussed in details in Section 6.

Moreover, this result can be extended to Σ_1 and Π_1 fragments.

3.3 Modeling System: Constrained Petri Nets (CPN)

Let T be a set of token vaiables and C be a set of color variables such that $T \cap C \neq \emptyset$. A constraint Petri Net (CPN) is a tuple $S = (\mathbb{P}, L, \Gamma, \Delta)$ where $\mathbb{P}$ is a finite set of places, $L = (\mathbb{C}, \Omega, \Xi)$ is a colored tokens logic, Γ is a finite set of token coloring symbols, and Δ is a finie set of transitions of the form:

$$\overrightarrow{x} \in \overrightarrow{p} \ \hookrightarrow\ \overrightarrow{y} \in \overrightarrow{q}\ :\ \varphi(\overrightarrow{x}, \overrightarrow{y})$$

where $\overrightarrow{x} = (x_1, \ldots, x_n) \in T^n$, $\overrightarrow{y} = (y_1, \ldots, y_m) \in T^m$, $\overrightarrow{p} = (p_1, \ldots, p_n) \in \mathbb{P}^n$, $\overrightarrow{q} = (q_1, \ldots, q_m) \in \mathbb{P}^m$, and $\varphi(\overrightarrow{x}, \overrightarrow{y})$ is a CML formula called the *transition guard.*

Given a fragment Θ of CML, we denote by CPN[Θ] the class of CPN where all transition guards are formulas in the fragment Θ. Due to the (un)decidability results of section 3.2.2, we focus in the sequel on the classes CPN[Σ_2] and CPN[Σ_1].

The semantics of a CPN is defined formally in [11] in terms of colored markings.

Intuitively, a constrained transition above says that it's firing:

- n different tokens represented by token variables $x_1, \ldots, x_n$ are deleted from the places $p_1, \ldots, p_n$,
- m new different tokens represented by token variables $y_1, \ldots, y_m$ are added to the places $q_1, \ldots, q_m$
- provided that the colors of all these (old and new) tokens satisfy the formula φ,
- and this operation does not modify the rest of the tokens (others than $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$).

CPN can be used to model (unbounded) dynamic networks of parallel processes. We assume w.l.o.g that all processes are identically defined. We consider that a process is defined by a finite control state machine supplied with variables and data structures ranging over potentially infinite domains (such as integer variables, reals, etc). Processes running in parallel can communiate and synchronize using various kinds of mechanisms (rendez-vous, shared variables, locks, etc). Moreover, they can dynamically spawn new (copies of) processes in the network [11].

Given a configuration of the CPN represented by a colored marking $\mathcal{M}$, let post($\mathcal{M}$) be the set of immediate successors of $\mathcal{M}$ by all transitions of the CPN. Similarly, pre($\mathcal{M}$) is the set of immediate predecessors of $\mathcal{M}$.

Since the CML formulas can represent colored marking, the definition of post and pre above can be extended to formulas in CML.

3.4 Verification of CPN's

The Pre-post condition reasoning consists of the following: given a transition τ in S and given two formulas φ and φ', $\langle \varphi, \tau, \varphi' \rangle$ is a hoare triple if whenever the condition φ holds, the condition φ' holds after the execution of τ. In other words, we must have $post_\tau([\![\varphi]\!]) \subseteq [\![\varphi']\!]$, or equivalently that $post_\tau([\![\varphi]\!]) \cap [\![\neg\varphi']\!] = \emptyset$.

This reasonning is used to check the invariant properties. Indeed to check an invariant property I in the system S: if whenever the property I holds, the same property I must hold after the execution of each transition τ of the system S. In other words, we must have $post_\tau([\![I]\!]) \subseteq [\![I]\!]$, or equivalently that $post_\tau([\![I]\!]) \cap [\![\neg I]\!] = \emptyset$.

In the following sections, we will recall the main results provided in [11] about computing the post and pre images, and about checking invariant properties for CPN.

3.4.1 Post and Pre Computation

Let φ be a closed formula, and let τ be a transition $\overrightarrow{x} \in \overrightarrow{p} \hookrightarrow \overrightarrow{y} \in \overrightarrow{q} : \psi$ of the system S. W.l.o.g, we suppose that φ and ψ are in special form. We define hereafter the formulas φ_{post} and φ_{pre} for this single transition. The generalization to the set of all transitions is straightforward.

Intuitively, the idea is to express first the effect of deleting/adding tokens, and then composing these operations to compute the effect of a transition. Two transformations $\ominus$ and $\oplus$ corresponding to deletion and creation of tokens.

The operation $\ominus(\overrightarrow{z}, \texttt{loc}, \texttt{col})$ is parameterized by a vector $\overrightarrow{z}$ of token variables to be deleted, a mapping `loc` associating with token variables in $\overrightarrow{z}$ the places from which they will be deleted, and a mapping `col` associating with each coloring symbol in Γ and each token variable in $\overrightarrow{z}$ a fresh color variable in C. Intuitively, $\ominus$ projects a formula on all variables which are not in $\overrightarrow{z}$.

The operation $\oplus(\overrightarrow{z}, \texttt{loc})$ is parameterized by a vector $\overrightarrow{z}$ of token variables to be added and a mapping `loc` associating with each variable in $z \in \overrightarrow{z}$ a place (in which it will be added). Intuitively, $\oplus$ transforms a formula taking into account that the added tokens by the transition were not present in the previous configuration (and therefore not constrained by the original formula describing the configuration before the transition).

These operations are defined in detail in [11].

φ_{post} and φ_{pre} are defined to be the following formulas:

$$\varphi_{post} = \exists \overrightarrow{y} \in \overrightarrow{q}.\ \exists \overrightarrow{c}.\ ((\varphi \wedge \psi) \ominus (\overrightarrow{x}, \overrightarrow{x} \mapsto \overrightarrow{p}, \Gamma \mapsto (\overrightarrow{x} \mapsto \overrightarrow{c}))) \oplus (\overrightarrow{y}, \overrightarrow{y} \mapsto \overrightarrow{q})$$

$$\varphi_{pre} = \exists \overrightarrow{x} \in \overrightarrow{p}.\ \exists \overrightarrow{c}.\ ((\varphi \oplus (\overrightarrow{x}, \overrightarrow{x} \mapsto \overrightarrow{p})) \wedge \psi) \ominus (\overrightarrow{y}, \overrightarrow{y} \mapsto \overrightarrow{q}, \Gamma \mapsto (\overrightarrow{y} \mapsto \overrightarrow{c}))$$

Bouajjani and all proved the closure properties of the CML fragments under the computation of immediate successors and predecessors for CPN's. The theorem proved is the following:

Theorem 3.1. *Let S be a CPN$[\Sigma_n]$, for $n \in \{1, 2\}$. Then, for every closed formula φ in the fragment Σ_n of* CML*, it's possible to construct two closed for-*

mulas φ_{post} and φ_{pre} in the same fragment Σ_n such that $[\![\varphi_{post}]\!] = post_S([\![\varphi]\!])$ and $[\![\varphi_{pre}]\!] = pre_S([\![\varphi]\!])$

3.4.2 Checking Invariance Properties

An instance of the *invariance cheching problem* is given by a pair of sets of configurations (colored markings) $(Init, Inv)$, and consists in deciding whether starting from any configuration in *Init*, every computation of S can only visit configurations in *Inv*, i.e.,$\bigcup_{k\geqslant 0} post_S^k(Init) \subseteq Inv$. This problem is of course undecidable in general. However, a deductive approach using inductive invariants (provided by the user) can be adopted. Bouajjani and all show that their results allow to automatize the steps of this approach.

A set of configurations $\mathbb{M}$ is an *inductive invariant* if $post_S(\mathbb{M}) \subseteq \mathbb{M}$, or equivalently, if $\mathbb{M} \subseteq pre_S(\mathbb{M})$ [11] .

Theorem 3.2. *If S is a* CPN$[\Sigma_2]$*, then for every formula φ in $B(\Sigma_1)$, the problem of checking whether φ defines an inductive invariant is decidable.*

The deductive approach for establishing an invariance property considers the *inductive invariance checking problem* given by a triple (*Init*,*Inv*,*Aux*) of sets of configurations, and which consists in deciding whether (1) $init \subseteq Aux$, (2) $Aux \subseteq Inv$, and (3) *Aux* is an inductive invariant. Indeed, a (sound and) complete rule for solving an invariance checking problem (*Init*,*Inv*) is in finding a set of configurations *Aux* allowing to solve the inductive invariance checking problem (*Init*,*Inv*,*Aux*).

Theorem 3.3. *If S is a* CPN$[\Sigma_2]$*, then the inductive invariance checking problem is decidable for every instance $(\varphi_{Init}, \varphi, \varphi')$ where $\varphi_{Init} \in \Sigma_2$, and $\varphi,\varphi' \in B(\Sigma_1)$.*

3.5 Example of Modeling: Reader-Writer Lock

Reader-writer is a classical synchronisation scheme used in operating systems or other large scale systems. Several readers and writers work on common data. Readers may read data in parallel but they are exclusive with writers. Writers can only work in exclusive mode with other threads. *Reader-writer lock* is used to implement such kind of synchronization. Readers have to acquire the lock in *read mode*, and writers in *write mode*.

The implementation in Java of atomic operations for acquire and release in read and write mode is classical. This implementation uses an *integer*

threads Writer:	threads Reader:
1: l.acq_write(_pid);	1: l.acq_read(_pid);
2: x = g(x);	2: y = f(x);
3: l.rel_write(_pid);	3: l.rel_read(_pid);
4:	4:

Table 3.1: Example of program using reader-writer lock.

w to identifies the thread holding the lock in write mode or -1 if no such thread exists (threads identifiers are suppsed to be positive integers). Also, an *integer set* r is used to store the identifiers of all threads holding the lock in read mode. Acquire and release operations are accessing variables w and r in mutual exclusion.

Our model of reader-writer lock follows the implementation above. The (global) lock variable is modeled by a place *rw* where each token represents a thread using the lock (i.e., it has acquired but not yet released the lock). Each token in *rw* has two colors: *ty* gives the type of the access to the lock (read or write), and *Id* gives the identifier of the thread represented by the token. The *Id* color is useful to ensure that the releasing of a lock is done by the thread which acquired it. Since acquire and release should be atomic operations, we model them by single transitions (see Table 3.5).

Let consider the program using the reader-writer lock given in Table 3.5. It consists on several Reader and Writer threads, a global reader-writer lock variable l, a global variable x, and a local variable y for Reader threads. Writer threads change the value of the global variable x after acquiring in write mode the lock. Reader threads are setting their local variable y to a value depending on x after acquiring in read mode the lock. (Let us assume that for example the variables range over the domain of positive integers.) Each thread has an unique identifier represented by the _pid local variable.

For this program, the safety property to verify is the absence of race on variable x: value of x should not change while the lock is held in read mode, i.e., a reader thread at line 3 has a value of local variable y equal to f(x).

The CPN model corresponding to this program is given in Table 3.5. We use the logic DL as colored tokens logic. To each control point we associate a place (e.g., place $r3$ for control point corresponding to line 3 of Reader threads) and a transition (e.g., transition r_3 for statement at line 3 of Reader threads). The global variable x is modeled as explained in previous section: we have a place p_x containing an unique token which color α stores the current value of x. With each token in the places corresponding to Reader

$$\begin{array}{rrcl}
w_1: & t \in w1 & \hookrightarrow & t' \in w2, l' \in rw \\
 & & & : \neg(\exists z \in rw.\ true) \ \wedge\ Id(l') = Id(t) \ \wedge\ ty(l') = W \ \wedge\ \Gamma(t', t) \\
w_2: & t \in w2, t_x \in p_x & \hookrightarrow & t' \in w3, t'_x \in p_x \\
 & & & : \alpha(t'_x) = g(\alpha(t_x)) \ \wedge\ \Gamma(t', t) \\
w_3: & t \in w3, l \in rw & \hookrightarrow & t' \in w4 \\
 & & & : Id(l) = Id(t) \ \wedge\ ty(l) = W \ \wedge\ \Gamma(t', t)
\end{array}$$

$$\begin{array}{rrcl}
r_1: & t \in r1 & \hookrightarrow & t' \in r2, l' \in rw \\
 & & & : \neg(\exists z \in rw.\ ty(z) = W) \ \wedge\ Id(l') = Id(t) \ \wedge\ ty(l') = R \ \wedge\ \Gamma(t', t) \\
r_2: & t \in r2, t_x \in p_x & \hookrightarrow & t' \in r3, t'_x \in p_x \\
 & & & : y(t') = f(\alpha(t_x)) \ \wedge\ Id(t') = Id(t) \ \wedge\ \Gamma(t'_x, t_x) \\
r_3: & t \in r3, l \in rw & \hookrightarrow & t' \in r4 \\
 & & & : Id(l) = Id(t) \ \wedge\ ty(l) = R \ \wedge\ \Gamma(t', t)
\end{array}$$

Table 3.2: Model of reader-writer lock.

control points we associate a color y to model the local variable y. We denote by $\Gamma(t', t)$ the (conjunctive) formula expressing that t' and t have the same colors. It can be observed that the obtained model is a $\mathsf{CPN}[\Sigma_2]$.

The race-free property that the system must satisfy can be expressed by the following Π_1 formula:

$$RF \ \equiv\ \forall t \in r3.\ \forall t_x \in p_x.\ y(t) = f(\alpha(t_x))$$

Actually, in order to establish the invariance of the property above, it must be strengthened by other auxiliary properties:

- Place p_x contains a single token:

$$A_x \ \equiv\ \forall x, x' \in p_x.\ x = x'$$

- Reader-writer lock is either kept by a set of readers or by a unique writer:

$$\begin{array}{rcl} RW & \equiv & \forall u, u' \in rw.\ \ (ty(u) = ty(u')) \ \wedge\ (ty(u) = W \implies u = u') \\ & & \wedge\ (ty(u) = R \vee ty(u) = W) \end{array}$$

- For all threads in places $w2$ and $w3$ of the Writer, the tokens in the lock place have the same identities and are of writer type:

$$RW_w \quad \equiv \quad \forall w \in \{w2, w3\}.\ \forall l_w \in rw.\ Id(w) = Id(l_w) \wedge ty(l_w) = W$$

- If threads exist in places $r2$ and $r3$ of the Reader, then there is a token in the lock place with reader type:

$$RW_r \quad \equiv \quad (\exists r \in \{r2, r3\}.\ true) \implies (\exists l_r \in rw.\ ty(l_r) = R)$$

It can be seen that all the formulas above are in the fragment $B(\Sigma_1)$.

Part II

Our contribution

Chapter 4

Building a Verification System for CPN

4.1 Introduction

This chapter begins the presentation of the verification system built for verifying invariance properties on CPN. It shows the general architecture of the tool and how it uses the verification system WHY, as well as the tools provided by WHY for the interface with the SMT solvers. The algorithms for efficient invariance checking on CPN and for satisfaction checking for CML are presented in the next chapters.

4.2 Tool Architecture

The architecture of the verification system proposed is presented on Figure 4.1. In this architecture, the greyed parts are not parts of our implementation. Then, we shortly describe here the different parts of our implementation and give the references to the sections and chapters dealing with these parts in detail.

First of all, we use the input language of WHY to define the components of the CML logic, the rules of the CPN model, and the inductive invariant to be checked. This part of our system is described in detail in the remainder of this chapter. From the output of the WHY parser and type-cheker, we obtain an abstract syntax tree which we transform slightly to extract information about the system.

Then, we check that the CPN model and the invariant satisfy the necessary conditions to apply the procedure of invariant checking (see Theorem 6 in [11]). Some optimisations on invariant are then applied in order to accelerate the computation of post images of the invariant. Finally, verification conditions are generated as formulas in CML. This part of the tool is fully described in chapter 5.

The verification formulas generated are given to the SMT-solver for the CML logic which implements in an efficient way the procedure given in [11], as described in detail in chapter 6. This procedure produces first order formula in the theory of the underlying colour logic. These formulas are given to the interface of the WHY with the decision procedures (tool DP) in order to call the Yices tool to test their satisfiability. The results of this call are then interpreted to produce the results of our tool.

In the remainder of this section we present the encoding of the CPN system and invariant property in WHY. The rationale of using WHY is to benefit from the work done in the parsing, type-checking and managing of logic specifications and first-order formulas. We don't use at all the programming language of WHY since this language is not adapted to concurrent programming.

4.3 Encoding CML in WHY

Color Domain: The color domain is encoded in WHY using the existing types ($\mathbb{C} = \mathbb{N}, \mathbb{C} = \mathbb{R}$,...), or by defining a new abstract type. For example, if the color domain is the set of all stacks with integer elements, we can define it as an abstract type in WHY:

```
type stack
```

Functions: Functions of the color logic are expressed in WHY using functions. For the color domain already defined in WHY ($\mathbb{N}$,$\mathbb{R}$, etc.), WHY provides all the necessary functions.

For example, two uninterpreted functions `f` and `g` over integers are encoded in WHY as follows:

```
logic g : int -> int

logic f : int -> int
```

Another example are functions manipulating the stack of integers:

```
logic push : stack -> int -> stack
```

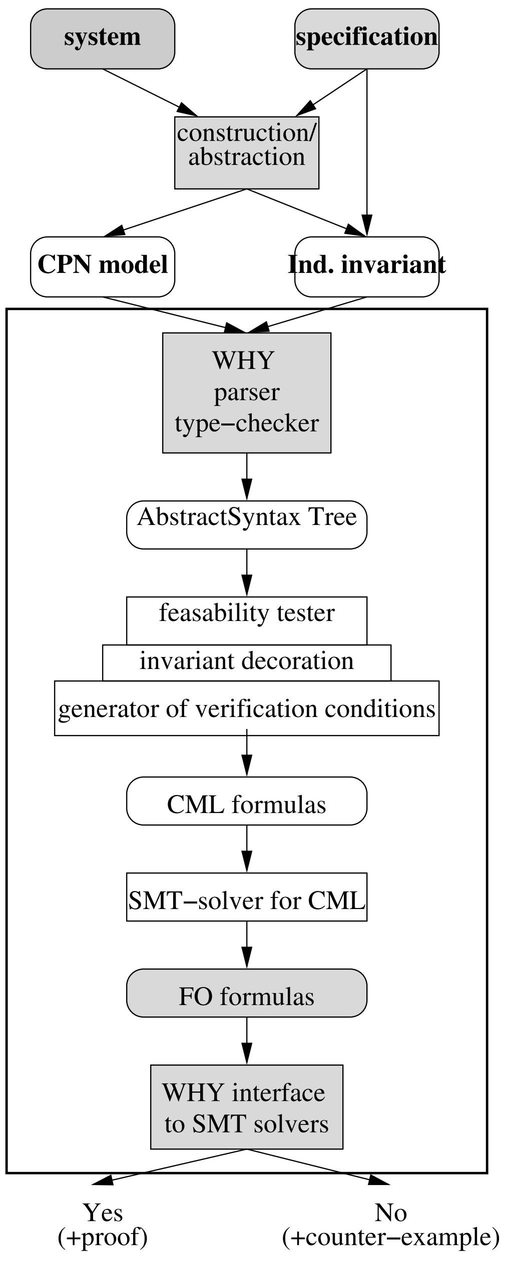

Figure 4.1: The architecture of the verification system for CPN.

```
logic  pop : stack -> stack

logic  top : stack -> int
```

Relations: Relations in the color logic are encoded in using WHY uninterpreted predicates. For example, equality between elements of the color domain can be expressed as follows:

```
logic is_equal : C -> C -> prop
```

For stacks of integers, the empty test may be declared as follows:

```
logic empty : stack -> prop
```

Tokens: An abstract type `token` is introduced to represent the enumerable set of tokens $\mathbb{T}$. The equality between tokens is expressed using an uninterpreted predicate `eq_token`, and we introduce also a the coloring symbol correspoding to the token identity:

```
type  token

logic eq_token : token, token -> prop

logic       id : token -> int
```

The above signature may be transformed into a theory by providing the axioms of the equality predicate, e.g., two equal tokens have the same identity colour:

```
axiom id_axiom :

  forall x:token. forall xp:token. eq_token(x,xp) => id(x)=id(xp)
```

Places: Each place in $\mathbb{P}$ is encoded using an uninterpreted predicate taking a token as a parameter and returning a proposition. A special function `bot` is defined for the special place $\perp$.

For example, let consider the following set of places $\mathbb{P} = \{r_1, w_1, rw\}$. Its encoding is given by the following WHY declarations:

```
logic r1 : token -> prop

logic w1 : token -> prop

logic rw : token -> prop

logic bot : token -> prop
```

The theory on tokens considered in [11] asks that all places in $\mathbb{P}$ are pairwise disjoint and disjoint from $\perp$. This is encoded by the following axiom:

```
axiom token_axiom :
  forall x:token. r1(x) => not (w1(x) or rw(x) or bot(x))
              and  w1(x) => not (r1(x) or rw(x) or bot(x))
              and  rw(x) => not (r1(x) or w1(x) or bot(x))
              and  bot(x) => not (r1(x) or w1(x) or rw(x))
```

This theory may be dealt, under some conditions, by the theorem proving tools, but not by the existing SMT-solver.

Coloring Symbols: The coloring symbols are encoded in WHY using uninterpreted functions mapping tokens to the color domain. For example, a coloring symbol corresponding to the local, integer variable y is encoded by:

```
logic var_y : token -> int
```

Global Colors: Introduced to simplify the specification of global variables in CPN, global colors are syntactic sugar for a place with an unique token which colors provide the values of global variables. The programming language of WHY provides a mean to declare global variables: it is a parameter with a reference type, i.e., which can be modified in place. We use this mean to encode global colors. For example, a global variable x is encoded by:

```
parameter var_x : int ref
```

Formulas: WHY provides a large choice of operators to obtain first order formula. Table 4.1 shows the correspondence between the syntax proposed for CML and the WHY syntax.

Note that the special form proposed for the CML formulas can be represented in WHY only for universally quantified tokens. Indeed, WHY allows to specify *triggers* in universal quantification, i.e., Boolean expressions selecting the variables (here tokens) which are covered by the quantification. For us, the trigger shall be a place uninterpreted function. Then, for the special form of existentially quantified formula, we have to use its translation in ordinary CML formula.

CML	WHY
$x = y$	`eq_token(x,y)`
$p(x)$	`p(x)`
$\xi(t_1, \ldots, t_n)$	`xi(t1,...,tn)`
$\neg\varphi$	not φ
$\varphi \vee \varphi$	φ or φ
$\varphi \wedge \varphi$	φ and φ
$\varphi \Longrightarrow \varphi$	φ -> φ
$\varphi \Leftrightarrow \varphi$	φ <-> φ
$\exists z.\ \varphi$	`exists z:int .` φ
$\forall z.\ \varphi$	`forall z:int .` φ
$\exists x.\ \varphi$	`exists x:token .` φ
$\forall x.\ \varphi$	`forall x:token .` φ
$\exists x \in p.\ \varphi$	`exists x:token . p(x)` and φ
$\forall x \in p.\ \varphi$	`forall x:token [p(x)] .` φ
	`forall x:token . p(x) ->` φ

Table 4.1: Correspondence between the syntax of CML and WHY.

4.4 Encoding CPN in WHY

WHY programming language deal only with sequential programs. Since we are dealing with concurrent programs, we cannot encode directly the CPN in WHY.

The trick we propose is to use WHY parameters to encode each rule. This is somewhat coherent with the semantics given in WHY to parameters.

Indeed, parameters are introduced in WHY to specify external functions (or procedures) by providing their parameters, their pre- and post-condition. For example, an external implementation of a function sorting an array given as parameter is declared as follows:

```
parameter sort :
    t:array ->
       { true }
         array
       { sorted(result) and permutation(t,result) }
```

The function takes as parameter an array (we suppose that such a type is already declares) and output an array; its pre-condition is empty and is post-condition says that the resulting array (keyword `result`) is sorted and it is a permutation of the initial array `t`.

Then, a CPN transition is encoded into a parameter as follows:

- the arguments encode the tokens in the left and right side of the transition,
- the result type is void (written `unit` in WHY),
- the pre-condition is a conjunction of formulas specifying the places of tokens in the left side of the transition and the guard of the transition,
- the post-condition is a conjunction of formulas specifying the places of tokens in the right side of the transition and the effect of the transition.

For example, the following CPN transition *r*:

$$t \in pl \longrightarrow t1 \in pl1, t2 \in pl2 : \neg(\exists z \in pl2.\ true) \wedge id(t) = id(t1) \wedge id(t) = id(t2)$$

is encoded in WHY as follows:

```
parameter r: t:token -> t1:token -> t2:token ->
```

```
{ pl(t) and bot(t1) and bot(t2) and
   not(exists z:token. pl2(z)) }
unit
{ bot(t) and pl1(t1) and pl2(t2) and
   id(t)=id(t1) and id(t)=id(t2) }
```

4.5 Encoding Invariants in WHY

Since Why allows specification of predicates, invariants are naturally written in a form of a predicate with an empty list of parameters. For example, the invariant:

$$RW \equiv \forall u, u' \in rw.\ (ty(u) = ty(u')) \wedge (ty(u) = W \Longrightarrow u = u') \wedge (ty(u) = R \vee ty(u) = W)$$

is encoded in:

```
predicate RW () =
   forall u:token [rw(u)]. forall up:token [rw(up)].
     ty(u)=ty(up) and
      (ty(u)=Write -> eq_token(u,up)) and
      (ty(u)= Read or ty(u)= Write)
```

Note the splitting of the special form universal quantifier in order to properly use the triggers. For this, we use the following equivalence:

$$\begin{aligned}
\forall u, u' \in rw.\ \varphi &\Leftrightarrow \forall u, u'.\ (rw(u) \wedge rw(u')) \Longrightarrow \varphi \\
&\Leftrightarrow \forall u, u'.\ \neg rw(u) \vee \neg rw(u') \vee \varphi \\
&\Leftrightarrow \forall u.\ \neg rw(u) \vee (\forall u'.\ \neg rw(u') \vee \varphi) \\
&\Leftrightarrow \forall u.\ rw(u) \Longrightarrow (\forall u'.\ rw(u') \Longrightarrow \varphi) \\
&\Leftrightarrow \forall u \in rw.\ \forall u' \in rw.\ \varphi
\end{aligned}$$

4.6 Example of Reader-Writer Lock in WHY

We provide in this section the full specification in Why of the reader-writer example given in section 3.5.

```
(* abstract type token *)
type token
logic eq_token : token, token -> prop

(* places *)
logic r1 : token -> prop
logic r2 : token -> prop
logic r3 : token -> prop
logic r4 : token -> prop
logic w1 : token -> prop
logic w2 : token -> prop
logic w3 : token -> prop
logic w4 : token -> prop
logic rw : token -> prop
logic bot : token -> prop

(* abstract type lock *)
type lock
logic Read : lock
logic Write : lock

(* type lock have only 2 values: Read and Write *)
axiom lock_axiom : forall l:lock. l=Read or l=Write


(* coloring symbols *)
(* - global variable x *)
parameter var_x : int ref
```

```
(* - local variables ty, y, id *)
logic ty : token -> lock
logic  y : token -> int
logic id : token -> int

(* uninterpreted functions used on colors *)
logic g : int -> int
logic f : int -> int

(* transition w1:acq_write *)
parameter rule_w1 : t:token -> tp:token -> lp:token ->
        { w1(t) and bot(tp) and bot(lp) and not(exists z:token. rw(z))
 }
        unit
        { bot(t) and w2(tp) and rw(lp) and id(t)=id(tp)
           and ty(lp)=Write and y(tp)=y(t) }



(* transition w2:x=g(x) *)
parameter rule_w2 : t:token -> tp:token  ->
        { w2(t) and bot(tp) }
         unit writes var_x
        { bot(t) and w3(tp) and y(t)=y(tp) and var_x = g(var_x@) }

(* transition w3:rel_write *)
parameter rule_w3 : t:token -> l:token -> tp:token ->
        { w3(t) and rw(l) and bot(tp) and id(l)=id(t) and ty(l)=Write }
        unit
        { w4(tp) and id(t)=id(tp) and y(tp)=y(t) }


(* transition r1:acq_read *)
parameter rule_r1 : t:token -> tp:token -> lp:token ->
        { r1(t) and bot(tp) and bot(lp) and id(lp)=id(t)
            and ty(lp)=Write and not(exists z:token. rw(z) and
 ty(z)=Write) }
        unit
        { r2(tp) and rw(lp) and id(t)=id(tp) and
            ty(lp)=Read and y(tp)=y(t) }


(* transition r2:y=f(x) *)
parameter rule_r2 : t:token -> tp:token ->
        { r2(t) and bot(tp) }
        unit writes var_x
        {r3(tp) and y(tp)=f(var_x@) and
           id(t)=id(tp) and var_x=var_x@ }


(* transition r3:rel_read *)
parameter rule_r3 : t:token -> l:token -> tp:token ->
        { r3(t) and rw(l) and bot(tp) and id(l)=id(t) and ty(l)=Read }
        unit
        { r4(tp) and y(tp)=y(t)}
```

```
(* Invariants *)
predicate RF () =
   forall t:token [r3(t)]. y(t)=f(var_x)

predicate RW () =
  forall u:token [rw(u)]. forall up:token [rw(up)].
    ty(u)=ty(up) and
    (ty(u)=Write -> eq_token(u,up)) and
    (ty(u)= Read or ty(u)= Write)

predicate RW_w () =
  forall w:token [w2(w)| w3(w)]. forall lw:token [rw(lw)].
    id(w)= id(lw) and ty(lw)=Write

predicate RW_r () =
  (forall r:token [r2(r)| r3(r)].false) or
  (exists lr:token. rw(lr) and ty(lr)=Read)
```

4.7 Abstract Syntax Tree of CPN

By calling the WHY parser on a CPN specification built as above, we obtain an abstract syntax tree *A*. On *A*, we call the WHY type-checker in order to check that all declarations type correctly and to obtain type informations on these declarations.

Then, we analyse the abstract syntax tree obtained in order to identify the CPN elements as follows:

- We check the presence of the `token` type and its equality predicate eq_token.

- We build the list of places $\mathbb{P}$ by collecting at all uninterpreted predicate declarations which unique parameter is of type token.
- We build the list of coloring symbols by collecting at all uninterpreted function declarations which unique parameter is of type token.
- We build a list of transitions by collecting parameters declarations with only tokens as parameters and with void result; the places of each token is obtained by analysing the pre- and post-condition of the parameter.
- We build a list of invariants by collecting all predicates with empty list of parameters and using quantifiers over tokens.

All these elements are collected into an abstract syntax tree that we defined in OCAML as follows:

```
type ty_cpn_rule = {
    name : Ident.t;
    leftside : Ident.t Ident.map;  (* mapping token->place *)
    rightside : Ident.t Ident.map; (* mapping token->place *)
    cond : predicate
  }


type ty_inv = {
    subinv : predicate;  (* subinvariant, see next chapter *)
    ...                  (* see next chapters *)
  }


type ty_cpn_model =  {
    places : Ident.set;  (* logic *)
    globals : Ident.set; (* parameters *)
    colors : Ident.set;  (* logic *)
    rules : ty_cpn_rule list;
    inv : ty_inv list    (* see next chapter *)
  }
```

In this tree, we store only identifiers for places, global variables, and coloring symbols, since their definition is already stored in the abstract syntax tree of WHY.

4.8 Feasibility test

Recall from chapter 3 that Theorem 6 in [11] asks that, in order to be able to check invariance properties, the side conditions of transitions shall be Σ_2 formulas and invariants shall be in $B(\Sigma_1)$.

For time optimisation, the feasibility test is done while looking through the abstract syntaxt tree to collect elements of the system: rules and invariants.

The feasibility test checks these conditions for each transition and each invariant. For this, we implemented algorithms to test that a formula is in Σ_2 or $B(\Sigma_1)$. In this section, we present these algorithms.

4.8.1 Checking for Σ_2 fragment

In order to check that a CML formula φ is in the Σ_2 fragment, we traverse the abstract tree of the formula and check that the quantifiers appear in a order given by the Σ_2 definition, i.e., a list of existentially quantified token and (possibly) color variables, followed by a list of universally quantified token and (possibly) color variables, followed by any quantification on color variables. More formally, the word built from the quantifiers appearing in the tree traversal is analysed using the automaton given on Figure 4.2. In this automaton, the initial state is 0 and the accepting state are $\{0, 1, 2, 3\}$. Labels of transitions represent the kind of quantifiers seen through the traversal; we use the following notations: exT (allT) if an `exists x:token` (`forall x:token`) is seen, exC (allC) if an `exists x:color_type` (`forall x:color_type`) is seen. We put a list of labels on a transition when there are several transitions with these labels and the same source and target states.

The word of quantifiers is build on the fly when traversing the tree. Since the tree may contain negations, the traversal keeps a parameter saying if the formula considered is negated or not. This parameter is updates in the recursive call depending on the form of the formula, while applying the de Morgan laws.

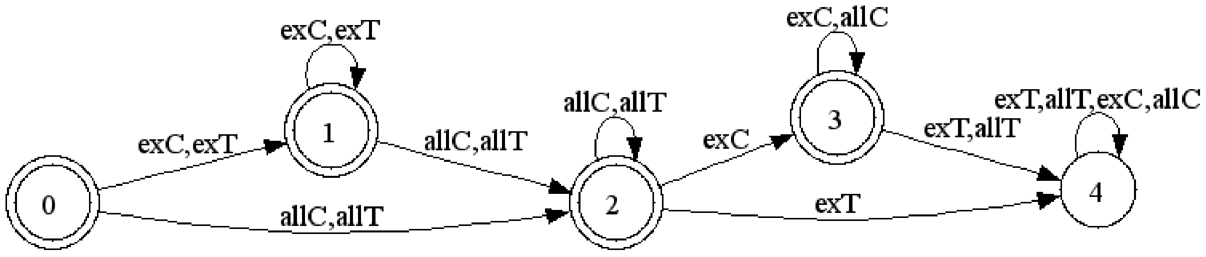

Figure 4.2: Automaton accepting a sequence of quantifiers in Σ_2.

4.8.2 Checking for $B(\Sigma_1)$ fragment

In order to check that a CML formula φ is in the $B(\Sigma_1)$ fragment, we traverse the abstract tree of the formula, ignore all top-most Boolean operators, and, as soon as we traverse a quantifier node, we check if the formula is in Σ_1 fragment (if the quantifier seen is $\exists$) or in Π_1 fragment (if the quantifier seen is $\forall$). The tests for Σ_1 and Π_1 fragments are described formally by automata on Figure 4.3 resp. Figure 4.4. On these figures, we use the same notations as above. The word tested by these automata is built as above, i.e., on the fly with the propagation of negations.

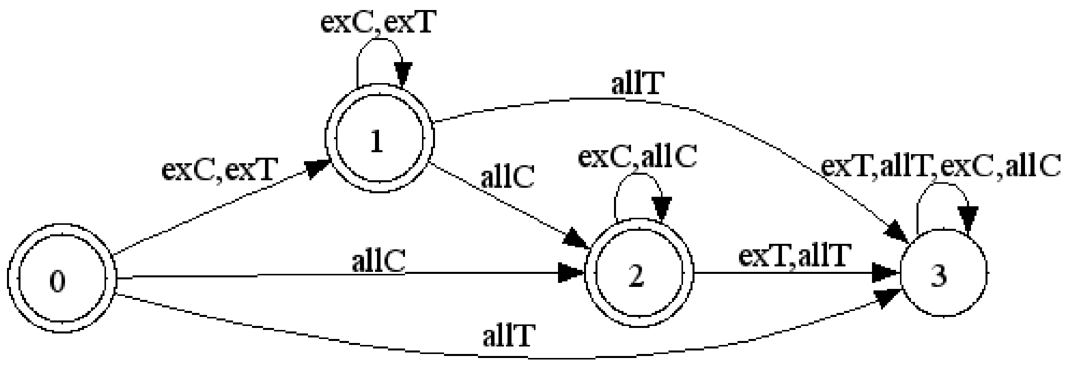

Figure 4.3: Automaton accepting a sequence of quantifiers in Σ_1.

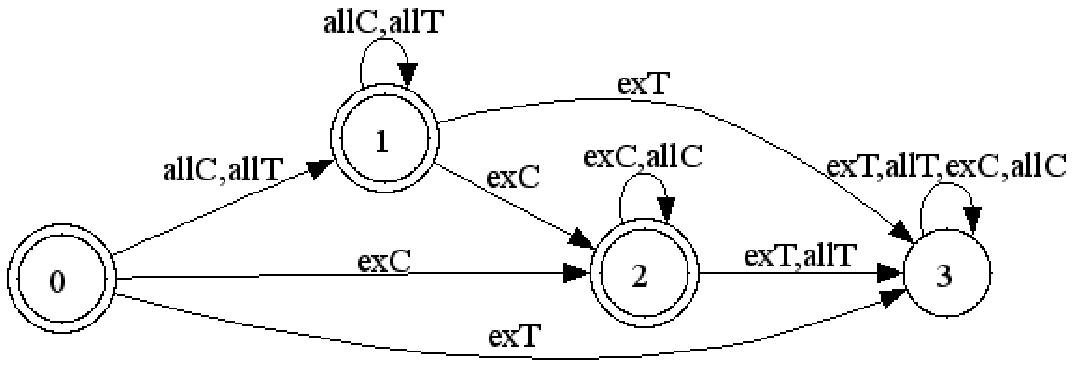

Figure 4.4: Automaton accepting a sequence of quantifiers in Π_1.

4.9 Conclusion

We presented in this chapter the overall architecture of the verification system for CPN, build for invariance checking.

We presented its first steps which consist of encoding the CPN and its logic CML using the WHY syntax. An abstract syntax tree is produced. In this way, we avoid heavy development of parser and type-checker by using the ones provided by the verification system WHY, and we also benefit from the general specification of the color logic allowed by WHY.

Finally, we show how we implement the checks needed to apply the invariance checking procedure described in the next chapter.

Chapter 5

Checking Invariant Properties

5.1 Introduction

This chapter presents the part of our tool which deals with the verification of invariant properties for CPN. The inputs of this part are the CPN and the invariant to be checked. The tool produces as output, a list of CML formulas to be checked for satisfiability by the CML-SAT checker presented in the next chapter. Moreover, the tool stops to produce such formulas as soon as one unsatisfiable formula is found.

Recall that checking an invariant property I is equivalent to check that, e.g., $\mathsf{post}(I) \wedge \neg I$ is unsatisfiable. To do efficiently this checking, we propose three techniques, each of them being discussed in the following sections. Let us give here an informal description of them.

First, we observe that a simpler form of invariants may optimize the computation of both $\mathsf{post}(I)$ part and $\neg I$ part. This simple form is the CNF form where atoms are Σ_1 and Π_1 formulas. Such form may be obtained from any kind of $B(\Sigma_1)$ formulas by using an algorithm similar to one use in SAT-solvers. As shown in section 5.4, using this simple form of invariants, we can avoid the full computation of $\mathsf{post}(I)$. Indeed, we have to compute the post effect only on conjuncts of I which refer to places concerned by the transition of post.

This last remark asks to compute the places concerned by each conjunct of the invariants I. We describe this computation as a second optimization technique in section 5.3.

Third, we prove how these optimizations can be combined to generate a small number of verification conditions.

5.2 Simple Form for Invariants

Theoretical Results

In order to obtain efficient invariant checking, we consider that the invariant is in *conjunctive normal form* (CNF), each elementary subformula being a Σ_1 or a Π_1 formula. Formally, the allowed syntax for invariants is the following:

$$\begin{array}{rcl} I & = & \bigwedge_{i=1..n} I_i \\ & \text{where} & \\ I_i & = & \bigvee_{j=1..m} [\neg]\, I_{ij} \\ & \text{where} & \\ I_{ij} & = & \begin{cases} (\exists\overrightarrow{t} \mid \exists\overrightarrow{c})^*(\exists\overrightarrow{c} \mid \forall\overrightarrow{c})^*.\ \psi_{ij} & \in \Sigma_1 \\ (\forall\overrightarrow{t} \mid \forall\overrightarrow{c})^*(\exists\overrightarrow{c} \mid \forall\overrightarrow{c})^*.\ \psi_{ij} & \in \Pi_1 \end{cases} \end{array}$$

with all $t \in T$ (set of token variables), all $c \in C$ (set of color variables), and for all i and j, ψ_{ij} are quantifier free formulas in Σ_0.

Note that I_{ij} may appear negated, but their negation is still a Σ_1 or Π_1 formula. Indeed, for example if I_{ij} has the form $(\exists\overrightarrow{t} \mid \exists\overrightarrow{c})^*(\exists\overrightarrow{c} \mid \forall\overrightarrow{c})^*.\ \psi_{ij}$, then:

$$\begin{array}{rcl} \neg I_{ij} & = & \neg(\exists\overrightarrow{t} \mid \exists\overrightarrow{c})^*(\exists\overrightarrow{c} \mid \forall\overrightarrow{c})^*.\ \psi_{ij} \\ & = & (\forall\overrightarrow{t} \mid \forall\overrightarrow{c})^*(\forall\overrightarrow{c} \mid \exists\overrightarrow{c})^*.\ \neg\psi_{ij} \end{array}$$

In the reader-writer lock example presented in section 3.5, the invariant $I \equiv (RF \wedge RW \wedge RW_w \wedge RW_r)$ is written according to the simple form defined above. Let us recall each one of these formulas:

- $RF \equiv \forall t \in r3.\ \forall t_x \in p_x.\ y(t) = f(\alpha(t_x))$
- $RW \equiv \forall u, u' \in rw.\ \ (ty(u) = ty(u'))\ \wedge\ (ty(u) = W \implies u = u')$
  $\wedge\ (ty(u) = R \vee ty(u) = W)$
- $RW_w \equiv \forall w \in \{w2, w3\}.\ \forall l_w \in rw.\ Id(w) = Id(l_w) \wedge ty(l_w) = W$
- $RW_r \equiv (\exists r \in \{r2, r3\}.\ true) \implies (\exists l_r \in rw.\ ty(l_r) = R) \equiv RW_{r1} \vee RW_{r2}$

Then, the invariant I corresponds indeed to a conjunction of subformulas such that, RF, RW, RW_w are in Π_1 and $RW_r \in B(\Sigma_1)$ since RW_{r1} is a Π_1 formula and RW_{r2} is a Σ_1 formula.

Implementation Issues

As already presented in section 4.7, the invariant is not manipulated in its abstract syntax tree but as a list of predicates data structure. Each predicate corresponds to a subinvariant I_i. For example, in the reader-writer, the invariant I is represented by a list $[RF, RW, RW_w, RW_r]$.

The invariant introduced by the user, using the Why syntax, is first checked to make sure that the simple form is respected. This is done according to the following algorithme:

We look through the predicate of each subinvariant I_i,

1. if $\forall$ is found, goto 4
2. if $\exists$ is found, goto 5
3. if $I_i = p_1 \vee p_2$, goto 1 for p_1, goto 1 for p_2
4. check that it's a Π_1 formula (see Figure 4.4)
5. check that it's a Σ_1 formula (see Figure 4.3)

Example 1. Consider the formula I below saying that all tokens in the place *rw* have a unique type and if this type is writer, then the place *rw* contains a unique token:

$$\begin{aligned} I \equiv \exists u.\ \forall u' \in rw.\ \ & (ty(u) = ty(u')) \ \wedge\ (ty(u) = W \implies u = u') \\ & \wedge (ty(u) = R \vee ty(u) = W) \end{aligned}$$

This formula is written in Why using the following syntax:

```
predicate I :bool ->
      {}
      bool
      { exists u:token. forall up:token [rw(up)]. ty(u)=ty(up)
          and (ty(u)=Write -> eq_token(u,up))
          and (ty(u)=Read or ty(u)=Write) }
```

When checked, this invariant doesn't match with the simple form defined above. It's not accepted because it is a Σ_2 formula. The procedure is then stopped and an exception of ”Bad_invariant” is raised.

When the invariant introduced respects the simple form defined in this section, a set of places is computed and associated to each conjunct of this invariant as described in the following section.

5.3 Localizing Invariants

This section shows how we compute an over-approximation of places concerned by each conjunct of an invariant in simple form. This computation is useful for the optimization of post computation.

Theoretical Results

Recall from section 3.2 that the set of CML formulas written in special form is given by the grammar:

$$\varphi ::= x = y \mid \xi(t_1, \ldots, t_n) \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists z.\ \varphi \mid \exists x \in p.\ \varphi$$

where $x, y \in T$, $z \in C$, $p \in \mathbb{P} \cup \{\bot\}$, $\xi \in \Xi$, and $t_1, \ldots, t_n$ are token color terms.

This special form locate quantified token in places, which are shorthand notations as follows:

$$\exists t \in p.\varphi \equiv \exists t.p(t) \wedge \varphi$$

and

$$\forall t \in p.\varphi \equiv \forall t.p(t) \Rightarrow \varphi$$

When several tokens are quantified by these formulas, the shorthand notations are extended naturally:

$$\exists \overrightarrow{t} \in \overrightarrow{p}.\ \varphi \equiv \exists \overrightarrow{t}.\ \bigwedge_{i=1}^{n} p_i(t_i) \wedge \varphi$$

and

$$\forall \overrightarrow{t} \in \overrightarrow{p}.\ \varphi \equiv \forall \overrightarrow{t}.\ (\bigwedge_{i=1}^{n} p_i(t_i)) \Rightarrow \varphi \equiv \forall \overrightarrow{t}.\ \neg(\bigwedge_{i=1}^{n} p_i(t_i)) \vee \varphi \equiv \forall \overrightarrow{t}.\ (\bigvee_{i=1}^{n} \neg p_i(t_i)) \vee \varphi$$

Then, for each conjunct I_i of an invariant I in simple form, we define $Places(I_i)$ to be the set of places concerned by the subinvariant I_i as The union of all the sets of places concerned by the subinvariants I_{ij}:

$$Places(I_i) = \cup_j Places(I_{ij})$$

where $Places(I_{ij})$ is over-approximated by the following definition:

$$Places(I_{ij}) = \begin{cases} \overrightarrow{p} & \text{if } I_{ij} = (\exists \overrightarrow{t} \mid \exists \overrightarrow{c})^*(\exists \overrightarrow{c} \mid \forall \overrightarrow{c})^*.\ \bigwedge_{k=1}^{n} p_k(t_k) \wedge \psi_{ij} \\ \mathbb{P} & \text{if } I_{ij} = (\exists \overrightarrow{t} \mid \exists \overrightarrow{c})^*(\exists \overrightarrow{c} \mid \forall \overrightarrow{c})^*.\ \bigwedge_{k=0}^{m<n} p_k(t_k) \wedge \psi_{ij} \\ \overrightarrow{p} & \text{if } I_{ij} = (\forall \overrightarrow{t} \mid \forall \overrightarrow{c})^*(\exists \overrightarrow{c} \mid \forall \overrightarrow{c})^*.\ (\bigwedge_{k=1}^{n} p_k(t_k)) \Longrightarrow \psi_{ij} \\ \mathbb{P} & \text{if } I_{ij} = (\forall \overrightarrow{t} \mid \forall \overrightarrow{c})^*(\exists \overrightarrow{c} \mid \forall \overrightarrow{c})^*.\ (\bigwedge_{k=0}^{m<n} p_k(t_k)) \Longrightarrow \psi_{ij} \end{cases}$$

where $\overrightarrow{t} = (t_1, ..., t_n) \in T^n$, and $\overrightarrow{p} = (p_1, ..., p_m) \in \mathbb{P}^m$. Indeed, if a quantified token is not located in the formula, then the set of places concerned by this formula is the set of places $\mathbb{P}$, as the quantified token can be located in any place.

Hereafter, we present an example to illustrate the computation of the set of places for a given formula in CML. Let's consider the subinvariant RW_r of the reader-writer example:

$$RW_r \equiv (\exists r \in \{r_2, r_3\}.\ true) \Longrightarrow (\exists l_r \in rw.\ ty(l_r) = R)$$

This invariant is not in the simple form defined above, therefore, we transform it as follows:

$$RW_r \equiv (\exists r.\ r_2(r) \vee r_3(r)) \Longrightarrow (\exists l_r.\ rw(l_r) \wedge ty(l_r) = R)$$

Then, we rewrite the implication ($\Rightarrow$) in terms of the negation ($\neg$) and the disjunction ($\vee$) :

$$RW_r \equiv \neg(\exists r.\ r_2(r) \vee r_3(r)) \vee (\exists l_r.\ rw(l_r) \wedge ty(l_r) = R)$$

We can then push negations over quantifiers and obtain:

$$RW_r \equiv \forall r.\ (\neg r_2(r) \wedge \neg r_3(r)) \vee (\exists l_r.\ rw(l_r) \wedge ty(l_r) = R)$$

Following our definition, the set of $Places(RW_r)$ is therefore be equal to $\mathbb{P}$.

However, by applying the definition of the special form, we can also write:

$$RW_r \equiv (\forall r \in \{r_1, r_3\}.\ false) \vee (\exists l_r..\ rw(l_r) \wedge ty(l_r) = R)$$

and $Places(RW_r)$ is therefore equal to $\{r2, r3, rw\}$. Note then that the second formula gives a more accurate set of $Places(RW_r)$

Implementation Issues

As it was mentionned previously, we consider the simple form of an invariant to be a conjunction of disjunction of formulas in Σ_1 and Π_1.

- Formulas in Π_1 have the form: $(\overrightarrow{\forall t} \mid \overrightarrow{\forall c})^*(\overrightarrow{\exists c} \mid \overrightarrow{\forall c})^*.\ \psi_{ij}$. For these formulas, the locations of tokens are collected in the triggers and in the left side of the implications when looking through the quantified free formula ψ_{ij}. For example, in the formula : $I_i = \forall t.\ p(t) \Rightarrow \phi$, the place p $\in Places(I_i)$

- Formulas in Σ_1 have the form: $(\overrightarrow{\exists t} \mid \overrightarrow{\exists c})^*(\overrightarrow{\exists c} \mid \overrightarrow{\forall c})^*.\ \psi_{ij}$. For these formulas, the locations of tokens are collected in the sides of conjunctions when looking through the quantified free formula ψ_{ij}. For example, in the formula : $I_i = \exists t.\ p(t) \wedge \phi$, the place p $\in Places(I_i)$

Let's consider the following formula: `forall t [p(t)]. q(t) =>` ψ. This formula respects the simple form defined and is therefore accepted by the system. However, this formula is false considering the semantic of the CPN system because the same token can't be located in two different places at the same time.

Note that the negation conserves the location of tokens as follows:

$$\neg(\forall x \in p.\psi) \equiv \neg(\forall x.p(x) \Rightarrow \psi) \equiv \exists x.p(x) \wedge \psi$$

$$\neg(\exists x \in p.\psi) \equiv \neg(\exists x.p(x) \wedge \psi) \equiv \forall x.\neg p(x) \vee \neg\psi \equiv \forall x.p(x) \Rightarrow \neg\psi$$

Considering the reader writer example, the results of the computation of the set of places concerned by each subinvariant are the following:

- $Places(RF) = \{r3\}$
- $Places(RW) = \{rw\}$
- $Places(RW_w) = \{w2, w3, rw\}$
- $Places(RW_r) = \{r2, r3, rw\}$

5.4 Computing Post/Pre Images of Rules

Theoretical Results

Let φ be a Σ_2 closed formula, and let τ be a transition $\overrightarrow{x} \in \overrightarrow{p} \hookrightarrow \overrightarrow{y} \in \overrightarrow{q} : \psi$ of the CPN. In [11], it has been shown that the pre/post-images of φ formula by the transition τ is given by:

$$\begin{aligned} Post_\tau(\varphi) &= \exists\overrightarrow{y} \in \overrightarrow{q}.\ \exists\overrightarrow{c}.\ ((\varphi \wedge \psi) \ominus (\overrightarrow{x}, \overrightarrow{x} \mapsto \overrightarrow{p}, \Gamma \mapsto (\overrightarrow{x} \mapsto \overrightarrow{c}))) \oplus (\overrightarrow{y}, \overrightarrow{y} \mapsto \overrightarrow{q}) \\ Pre_\tau(\varphi) &= \exists\overrightarrow{x} \in \overrightarrow{p}.\ \exists\overrightarrow{c}.\ ((\varphi \oplus (\overrightarrow{x}, \overrightarrow{x} \mapsto \overrightarrow{p})) \wedge \psi) \ominus (\overrightarrow{y}, \overrightarrow{y} \mapsto \overrightarrow{q}, \Gamma \mapsto (\overrightarrow{y} \mapsto \overrightarrow{c})) \end{aligned}$$

where operations $\oplus$ and $\ominus$ are defined in Tables 5.1 and 5.2.

Intuitively, the operation $\ominus$ is parameterized by a vector $\overrightarrow{z}$ of token variables to be deleted, a mapping `loc` associating with token variables in $\overrightarrow{z}$ the places from which they will be deleted, and a mapping `col` associating with each coloring symbol in Γ and each token variable in $\overrightarrow{z}$ a fresh color variable in C. Intuitively, $\ominus$ projects a formula on all variables which are not in $\overrightarrow{z}$. Rule $\ominus_1$ substitutes in a color formula $\xi(\overrightarrow{t})$ all occurences of colored tokens in $\overrightarrow{z}$ by fresh color variables given by the mapping `col`. A formula $x = y$ is unchanged by the application of $\ominus$ if the token variables x and y are not in $\overrightarrow{z}$; otherwise, rule $\ominus_2$ replaces $x = y$ by true if it is trivially true (i.e., we have the same variable in both sides of the equality) or by false if x or y is in $\overrightarrow{z}$. Indeed, each token variable in $\overrightarrow{z}$ represents (by the semantics of CPN) a different token, and since this token is deleted by the transition rule, it cannot appear in the reached configuration. Rules $\ominus_3$ and $\ominus_4$ are straightforward. Rule $\ominus_5$(resp. $\ominus_6$) does a case splitting according to the fact whether a deleted token is precisely the one referenced by the existential (resp. universal) token quantification or not. Rule $\ominus_7$ (resp. $\ominus_8$) does the same thing as rule $\ominus_5$ (resp. $\ominus_6$) with the difference that the substitution made over tokens is not based on their location as the token quantified existentially (resp. universally) is not located in a specific place. As the formula $x \in p$ in the special form is written $p(x)$ in a predicate (as described in section 4.2.), the rule $\ominus_9$ was added to replace by true $p(x)$ if one of the deleted tokens is located in p, false otherwise; the formula remains unchanged if x is not to be deleted.

The operation $\oplus$ is parameterized by a vector $\overrightarrow{z}$ of token variables to be added and a mapping `loc` associating with each variable in $z \in \overrightarrow{z}$ a place (in which it will be added). Intuitively, $\oplus$ transforms a formula taking into account that the added tokens by the transition were not present in the previous configuration (and therefore not constrained by the original formula describing the configuration before the transition). Then, the application of $\oplus$ has no effect on color formulas $\xi(\overrightarrow{t})$ (rule $\oplus_1$). When equality of tokens is tested, rule $\oplus_2$ takes into account that all added tokens are distinct and different from the existing tokens. For token quantification, rule $\oplus_5$ says that quantified tokens of the previous configuration cannot be equal to the added tokens. Rule $\oplus_7$(resp. $\oplus_8$) deals with token quantification without location. Rule$\oplus_9$ was added to deal with formulas not in special form.

Implementation Issues

In order to obtain the $Post_\tau(\varphi)$ formula, we first compute the predicate:

$$(\varphi \wedge \psi).\ \ominus(\overrightarrow{x}, \overrightarrow{x} \mapsto \overrightarrow{p}, \Gamma \mapsto (\overrightarrow{x} \mapsto \overrightarrow{c})).\ \oplus(\overrightarrow{y}, \overrightarrow{y} \mapsto \overrightarrow{q})$$

and we prefix it afterthat with: $\exists \overrightarrow{y} \in \overrightarrow{q}.\ \exists \overrightarrow{c}$

$$
\begin{array}{rrcl}
\ominus_1: & \xi(\vec{t}) \ominus (\vec{z}, \texttt{loc}, \texttt{col}) & = & \xi(\vec{t})[\texttt{col}(\gamma)(z)/\gamma(z)]_{\gamma\in\Gamma, z\in\vec{z}} \\
\ominus_2: & (x = y) \ominus (\vec{z}, \texttt{loc}, \texttt{col}) & = & \begin{cases} x = y & \text{if } x, y \notin \vec{z} \\ \text{true} & \text{if } x \equiv y \\ \text{false} & \text{otherwise} \end{cases} \\
\ominus_3: & (\neg\varphi) \ominus (\vec{z}, \texttt{loc}, \texttt{col}) & = & \neg(\varphi \ominus (\vec{z}, \texttt{loc}, \texttt{col})) \\
\ominus_4: & (\varphi_1 \vee \varphi_2) \ominus (\vec{z}, \texttt{loc}, \texttt{col}) & = & (\varphi_1 \ominus (\vec{z}, \texttt{loc}, \texttt{col})) \vee (\varphi_2 \ominus (\vec{z}, \texttt{loc}, \texttt{col})) \\
\ominus_5: & (\exists x \in p.\ \varphi) \ominus (\vec{z}, \texttt{loc}, \texttt{col}) & = & \exists x \in p.\ (\varphi \ominus (\vec{z}, \texttt{loc}, \texttt{col})) \vee \bigvee_{z\in\vec{z}:\texttt{loc}(z)=p} (\varphi[z/x]) \ominus (\vec{z}, \texttt{loc}, \texttt{col}) \\
\ominus_6: & (\forall x \in p.\ \varphi) \ominus (\vec{z}, \texttt{loc}, \texttt{col}) & = & \forall x \in p.\ (\varphi \ominus (\vec{z}, \texttt{loc}, \texttt{col})) \wedge \bigwedge_{z\in\vec{z}:\texttt{loc}(z)=p} (\varphi[z/x]) \ominus (\vec{z}, \texttt{loc}, \texttt{col}) \\
\ominus_7: & (\exists x.\ \varphi) \ominus (\vec{z}, \texttt{loc}, \texttt{col}) & = & \exists x.\ (\varphi \ominus (\vec{z}, \texttt{loc}, \texttt{col})) \vee \bigvee_{z\in\vec{z}} (\varphi[z/x]) \ominus (\vec{z}, \texttt{loc}, \texttt{col}) \\
\ominus_8: & (\forall x.\ \varphi) \ominus (\vec{z}, \texttt{loc}, \texttt{col}) & = & \forall x.\ (\varphi \ominus (\vec{z}, \texttt{loc}, \texttt{col})) \wedge \bigwedge_{z\in\vec{z}} (\varphi[z/x]) \ominus (\vec{z}, \texttt{loc}, \texttt{col}) \\
\ominus_9: & p(x) \ominus (\vec{z}, \texttt{loc}, \texttt{col}) & = & \begin{cases} \text{true} & \text{if } x = z \in \vec{z}, loc(z) = p \\ \text{false} & \text{if } x = z \in \vec{z}, loc(z) \neq p \\ p(x) & \text{if } x \notin \vec{z} \end{cases}
\end{array}
$$

Table 5.1: Definition of the $\ominus$ operator.

$$
\begin{array}{lrcl}
\oplus_1 : & \xi(\vec{t}) \oplus (\vec{z}, \texttt{loc}) & = & \xi(\vec{t}) \\
\oplus_2 : & (x = y) \oplus (\vec{z}, \texttt{loc}) & = & \begin{cases} x = y & \text{if } x, y \notin \vec{z} \\ \text{true} & \text{if } x \equiv y \\ \text{false} & \text{otherwise} \end{cases} \\
\oplus_3 : & (\neg\varphi) \oplus (\vec{z}, \texttt{loc}) & = & \neg(\varphi \oplus (\vec{z}, \texttt{loc})) \\
\oplus_4 : & (\varphi_1 \vee \varphi_2) \oplus (\vec{z}, \texttt{loc}) & = & (\varphi_1 \oplus (\vec{z}, \texttt{loc})) \vee (\varphi_2 \oplus (\vec{z}, \texttt{loc})) \\
\oplus_5 : & (\exists x \in p.\, \varphi) \oplus (\vec{z}, \texttt{loc}) & = & \exists x \in p.\, (\varphi \oplus (\vec{z}, \texttt{loc})) \wedge \bigwedge_{z \in \vec{z} : \texttt{loc}(z) = p} \neg(x = z) \\
\oplus_6 : & (\forall x \in p.\, \varphi) \oplus (\vec{z}, \texttt{loc}) & = & \forall x \in p.\, (\varphi \oplus (\vec{z}, \texttt{loc})) \vee \bigvee_{z \in \vec{z} : \texttt{loc}(z) = p} (x = z) \\
\oplus_7 : & (\exists x.\, \varphi) \oplus (\vec{z}, \texttt{loc}) & = & \exists x.\, (\varphi \oplus (\vec{z}, \texttt{loc})) \wedge \bigwedge_{z \in \vec{z}} \neg(x = z) \\
\oplus_8 : & (\forall x.\, \varphi) \oplus (\vec{z}, \texttt{loc}) & = & \forall x.\, (\varphi \oplus (\vec{z}, \texttt{loc})) \vee \bigvee_{z \in \vec{z}} (x = z) \\
\oplus_9 : & p(x) \oplus (\vec{z}, \texttt{loc}) & = & \begin{cases} \text{true} & \text{if } x = z \in \vec{z}, loc(z) = p \\ \text{false} & \text{if } x = z \in \vec{z}, loc(z) \neq p \\ p(x) & \text{if } x \notin \vec{z} \end{cases}
\end{array}
$$

Table 5.2: Definition of the $\oplus$ operator.

As mentionned before, the invariant φ has the form of a list of predicates. We compute the post only for subinvariants concerned by the rule τ as it's explained in section 5.5, the post is then computed for each predicate of the list I_τ as well as the formula ψ of the transition τ.

We start by applying $\ominus$ on each formula of the list I_τ and ψ and we cumulate the results.

The vector $\overrightarrow{z}$ of the tokens to be deleted and the mapping `loc`, associating with token variables in $\overrightarrow{z}$ the places from which they will be deleted, are collected in the leftside of the transition τ.

the first optimization made in the computation of the $\ominus$ operation concerns the mapping `col` associating with each coloring symbol in Γ and each token variable in $\overrightarrow{z}$ a fresh color variable in C. Indeed, not all the coloring symbols in Γ are considered, only those mensionned in the formula to compute. This reduces significantly the number of color variables $\overrightarrow{c}$ quantified existentially that pefix the Post formula. The coloring symbols to be substitute are collected while looking through the predicate. The new color variables are cumulated to prefix the Post formula.

The second optimization made is that the $\ominus$ rules and the color substitution are made at the same time. This reduces the execution time.

5.5 Invariant Splitting

Theoretical Results

Let consider an invariant I and the following transition τ:

$$\vec{x} \in \vec{p} \hookrightarrow \vec{y} \in \vec{q} : \varphi$$

the successor of I over τ is computed as follows:

$$Post_\tau(I) = \exists \vec{y} \in \vec{q}, \exists \vec{c}_x.\ [\varphi \wedge I] \ominus (\vec{x}, \vec{x} \mapsto \vec{p}, \Gamma \mapsto (\vec{x} \mapsto \vec{c}_x)) \oplus (\vec{y}, \vec{y} \mapsto \vec{q})$$

where $\oplus$ and $\ominus$ are the operators defined in tables 5.1 and 5.2. We remark that these operators don't have any effect on formulas that are not concerned with places in which the tokens are added or removed. therefore, it's interessting to split the invariant I into a conjunction of two new formulas:

$$I = I_\tau \wedge I_{\neg\tau}$$

where $I_{\neg\tau}$ is $\{Q_i t_i.\ \varphi_i \mid Places(t_i) \cap \{\vec{p}, \vec{q}\} = \emptyset\}$

Let consider the transition $w2$ and the property RW of the reader-writer example :

$$w_2 : t \in w2, t_x \in p_x \hookrightarrow t' \in w3, t'_x \in p_x : \alpha(t'_x) = g(\alpha(t_x)) \wedge \Gamma(t', t)$$

In this transition $\{\vec{p}, \vec{q}\} = \{w2, p_x, w3\}$.

$$\begin{aligned} RW \equiv \forall u, u' \in rw.\ & (ty(u) = ty(u')) \ \wedge\ (ty(u) = W \implies u = u') \\ & \wedge\ (ty(u) = R \vee ty(u) = W) \end{aligned}$$

Since $Places(RW) = \{rw\} \cap \{w2, p_x, w3\} = \emptyset$, therefore $RW \in I_{\neg w2}$

We note that if $I = \forall x.\ \varphi$, then $Places(x) = \mathbb{P} \cap \{\vec{p}, \vec{q}\} \neq \emptyset$, therefore, this formula is an I_τ

The post computation of the invariant I becomes:

$$\begin{aligned} & Post_\tau(I_\tau \wedge I_{\neg\tau}) \\ & = \exists \vec{y} \in \vec{q}.\ \exists \vec{c}_x.\ [\varphi \wedge I_\tau \wedge I_{\neg\tau}] \ominus (\vec{x}, \vec{x} \mapsto \vec{p}, \Gamma \mapsto (\vec{x} \mapsto \vec{c}_x)) \oplus (\vec{y}, \vec{y} \mapsto \vec{q}) \\ & = \exists \vec{y} \in \vec{q}.\ \exists \vec{c}_x. \\ & \qquad [((\varphi \wedge I_\tau) \ominus (\vec{x}, \vec{x} \mapsto \vec{p}, \Gamma \mapsto (\vec{x} \mapsto \vec{c}_x)) \oplus (\vec{y}, \vec{y} \mapsto \vec{q})) \\ & \qquad \wedge (I_{\neg\tau} \ominus (\vec{x}, \vec{x} \mapsto \vec{p}, \Gamma \mapsto (\vec{x} \mapsto \vec{c}_x)) \oplus (\vec{y}, \vec{y} \mapsto \vec{q}))] \\ & = \exists \vec{y} \in \vec{q}.\ \exists \vec{c}_x. \\ & \qquad [((\varphi \wedge I_\tau) \ominus (\vec{x}, \vec{x} \mapsto \vec{p}, \Gamma \mapsto (\vec{x} \mapsto \vec{c}_x)) \oplus (\vec{y}, \vec{y} \mapsto \vec{q})) \\ & \qquad (\bigwedge_i Q_i t_i : token.\ \varphi_i \ominus (\vec{x}, \vec{x} \mapsto \vec{p}, \Gamma \mapsto (\vec{x} \mapsto \vec{c}_x)) \oplus (\vec{y}, \vec{y} \mapsto \vec{q}))] \end{aligned}$$

By applying the rules $\ominus_3$, $\ominus_4$,$\ominus_5$, there is no effect on $I_{\neg\tau}$:

$$\begin{aligned} Post_\tau(I_\tau \wedge I_{\neg\tau}) = \ & \exists \vec{y} \in \vec{q}.\ \exists \vec{c}_x. \\ & [((\varphi \wedge I_\tau) \ominus (\vec{x}, \vec{x} \mapsto \vec{p}, \Gamma \mapsto (\vec{x} \mapsto \vec{c}_x)) \oplus (\vec{y}, \vec{y} \mapsto \vec{q})) \\ & \wedge (\bigwedge_i Q_i t_i : token.\ \varphi_i \oplus (\vec{y}, \vec{y} \mapsto \vec{q}))] \end{aligned}$$

By applying the rules $\oplus_3$, $\oplus_4$,$\oplus_5$, there is no effect on $I_{\neg\tau}$:

$$\begin{aligned} Post_\tau(I_\tau \wedge I_{\neg\tau}) = \ & \exists \vec{y} \in \vec{q}.\ \exists \vec{c}_x. \\ & [((\varphi \wedge I_\tau) \ominus (\vec{x}, \vec{x} \mapsto \vec{p}, \Gamma \mapsto (\vec{x} \mapsto \vec{c}_x)) \oplus (\vec{y}, \vec{y} \mapsto \vec{q})) \\ & \wedge (\bigwedge_i Q_i t_i : token.\ \varphi_i)] \end{aligned}$$

Since $\vec{y}$ and $\vec{c}_x$ are free variables in $I_{\neg\tau}$, so we obtain:

$$\begin{aligned} Post_\tau(I_\tau \wedge I_{\neg\tau}) = \ & \exists \vec{y} \in \vec{q}.\ \exists \vec{c}_x. \\ & [((\varphi \wedge I_\tau) \ominus (\vec{x}, \vec{x} \mapsto \vec{p}, \Gamma \mapsto (\vec{x} \mapsto \vec{c}_x)) \oplus (\vec{y}, \vec{y} \mapsto \vec{q})) \\ & \wedge I_{\neg\tau} \\ = \ & I_{post} \wedge I_{\neg\tau} \end{aligned}$$

where

$$I_{post} = \exists \vec{y} \in \vec{q}.\ \exists \vec{c}_x.\ [\varphi \wedge I_\tau] \ominus (\vec{x}, \vec{x} \mapsto \vec{p}, \Gamma \mapsto (\vec{x} \mapsto \vec{c}_x)) \oplus (\vec{y}, \vec{y} \mapsto \vec{q})$$

Implementation Issues

Let's recall that we consider in our implementation the invariant to be a list of predicates representing the subinvariants. Before checking the invariant given in the model on each rule, the set of places concerning each subinvariant is first computed independently from each rule as described in section 5.3. The intersection with the places in the left side and the right side of a rule τ is next made to determine I_τ and $I_{\neg\tau}$. The lists of predicates I_τ and $I_{\neg\tau}$ are stored in two different lists. The post computation will only be performed on predicates in the I_τ list.

Hereafter, we will give examples and concrete results about how this splitting reduces the computation of the post and how it makes the procedure efficient.

Reasonning about the reader-writer example, we give the lists I_τ and $I_{\neg\tau}$ for each transition of the system:

- $I_{w_1} = [RW, RW_w, RW_r]$ and $I_{\neg w_1} = [RF]$
- $I_{w_2} = [RW_w]$ and $I_{\neg w_2} = [RF, RW, RW_r]$
- $I_{w_3} = [RW, RW_w, RW_r]$ and $I_{\neg w_3} = [RF]$
- $I_{r_1} = [RW, RW_w, RW_r]$ and $I_{\neg r_1} = [RF]$
- $I_{r_2} = [RF, RW_r]$ and $I_{\neg r_2} = [RW, RW_w]$
- $I_{r_3} = [RF, RW, RW_w, RW_r]$ and $I_{\neg r_3} = []$

The splitting of the invariant reduces the post computation of formulas, and therefore reduces the complexity and the size of these formulas. Indeed, the post computation increases the number of quantified variables and therefore increases the complexity of formulas.

5.6 Invariant Checking

Theoretical Results

To check that I is invariant through a given transition τ, we have to prove the following:

$$Post_\tau(I) \in I \qquad valid$$
$$Post_\tau(I) \in I_\tau \wedge I_{\neg\tau} \qquad valid$$
$$\neg Post_\tau(I) \vee (I_\tau \wedge I_{\neg\tau}) \qquad valid$$
$$Post_\tau(I) \wedge (\neg I_\tau \vee \neg I_{\neg\tau}) \qquad unsat$$
$$(Post_\tau(I) \wedge \neg I_\tau) \vee (Post_\tau(I) \wedge \neg I_{\neg\tau}) \qquad unsat$$
$$(I_{post} \wedge I_{\neg\tau} \wedge \neg I_\tau) \vee (I_{post} \wedge I_{\neg\tau} \wedge \neg I_{\neg\tau}) \qquad unsat$$
$$I_{post} \wedge I_{\neg\tau} \wedge \neg I_\tau \qquad unsat$$
$$I_{post} \wedge (\bigwedge_i I_{i\neg\tau}) \wedge (\bigvee_j \neg I_{j\tau}) \qquad unsat$$
$$\bigvee_j I_{post} \wedge (\bigwedge_i I_{i\neg\tau}) \wedge \neg I_{j\tau} \qquad unsat$$

We end up with a disjunction of formulas. Each formula can be given separately to the CML sat-solver. This disjunction could be built considering the simple form of the invariants described in section 5.2.

Implementation Issues

For each transition τ and for each jth formula of the list I_τ, we build the formula $I_{post} \wedge (\bigwedge_i I_{i\neg\tau}) \wedge \neg I_{j\tau}$. A SAT problem is then generated for this formula. If the problem is not unsat, then the invariant is not true. The procedure stops at the first rule so that the invariant is not checked. an exception "Invariant_not_true" is then raised.

5.7 Conclusion

We presented in this section the techniques adopted to make the invariants checking more efficient.

One technique is to consider a CNF form of invariants which help have a disjunction of SAT formulas. Each one can be solved separately by a solver. Another technique is to associate a set of places to each conjunct of invariant. For each rule, we consider two lists with the intersection of the places concerned by each rule. The post computation is applied on subinvariants concerned by the rule and the others stay unchanged.

When combined, these techniques help us gain in execution time and formulas complexity.

After the steps described in this chapter, we end up with a list of verification conditions which are CML SAT problems. These formulas have to be reduced and then be given to SAT solvers as described in the following chapter.

Chapter 6

Checking Satisfiability for CML

6.1 Introduction

The WHY tool provides an interface with a wide set of provers (Coq, Isabelle, etc.) and SAT-solvers (Yices, CVC, etc.). Most of these SAT-solvers provides algorithms for satisfaction modulo theory (SMT) of first order formulas over classical theories like integers with common operations, reals, etc. Due to a very known competition between SMT-solvers, all these tools supports as input formulas in the SMTlib format (see `combination.cs.uiowa.edu/smtlib/`).

The CML logic does not correspond to a theory supported by SMT solvers. However, formulas in the Σ_0 fragment of CML (which are also formulas of the first-order logic of color L) can be solved using the SMT-solvers for some classical theories of colors.

Then, the formulas in the Σ_2 fragment, which are produced by the invariant checking, shall be reduced into formulas in Σ_0 fragment in order to check their satisfiability. Theorem 2 in [11] provides a reduction procedure from the satisfiability problem of Σ_2 formulas to the satisfiability problem of Σ_0 formulas. We provide a quick view of this procedure in 6.2.

Then, we consider each step of this general algorithms and try to optimize its implementation.

First, we show how to use the localization of tokens to optimize the first step of the reduction, from Σ_2 formulas to Σ_1 formulas.

Second, we observe that the second step of the reduction (from Σ_1 formulas to Σ_0 formulas) can be simplified for Σ_1 formulas which don't have $p(x)$ sub-formulas (localization of the token x in place p). Indeed, such formulas can be checked for satisfiability by some SMT-solvers (e.g., Yices) by considering that tokens belongs to the theory of an abstract data type with equality, and coloring symbols are uninterpreted functions.

Finally, we conclude with experimental results.

6.2 General Procedure for Satisfiability

We recall the main result proved in [11] about the decidability of the satisfiability problem of CML.

Theorem 6.1. *(Theorem 2) Let L be a colored tokens logic. If the satisfiability problem of L is decidable, then the fragment Σ_2 of* $\mathsf{CML}(L)$ *is decidable.*

We shortly resume here the proof of this theorem, which proceeds by reduction of the satisfiability problem of the Σ_2 fragment to the one of the Σ_0 fragment.

Let φ be a closed formula in Σ_2 in prenex normal form:

$$\varphi = \exists \vec{x}.\ \exists \vec{z}.\ \forall \vec{y}.\ \phi$$

where $\vec{x}$, $\vec{y}$ are token variables and $\vec{z}$ color variables, and let assume that all variables are different.

First, the satisfiability problem can be reduced from Σ_2 to Σ_1 due to the "small model" property of the Σ_2 fragment. This property says that if there exists a model for φ, then there exists also a "small model" of size at most equal to the number of existentially quantified variables $\vec{x}$. In this small model, universally quantified variables (in $\vec{y}$) will be taken in the set of the existential variables. The, we consider all mappings $\sigma \in [\vec{y} \to \vec{x}]$ from elements of $\vec{y}$ to elements of $\vec{x}$ and φ becomes:

$$\varphi = \exists \vec{x}.\ \exists \vec{z}.\ \bigwedge_{\sigma \in [\vec{y} \to \vec{x}]} \phi[\vec{y} \to \sigma(\vec{y})]$$

The satisfiability problem can then be reduced from Σ_1 to Σ_0 with the following transformations:

1. We eliminate sub-formulas corresponding to token equality by enumerating all the possible equivalence classes for equality between the finite number of variable in $\vec{x}$.

2. We eliminate sub-formulas of the form $p(x)$ by enumerating all the possible mappings from a token variable x to the set of places.

3. We replace terms of the form $\gamma(x)$ by fresh color variables.

Finally, the formula obtained after these transformations is in the fragment Σ_0 and has the following form:

$$\varphi = \exists \vec{z'}.\ \phi''$$

6.3 Discussion

The problem with the general procedure of satisfiability described above is the explosion of the size of formulas. In fact, to reduce a Σ_2 formula to a Σ_1 formula, we need to map each token variable quantified universally to each token variable quantified existentially. Therefore, if the number of token variable quantified existentially is $|x|$ and the number of token variable quantified universally is $|y|$, then the number of formulas generated after the mapping is $|x|^{|y|}$.

We propose to optimize this process by using the localization of tokens to chose mappings which are not trivially inconsistent with the localization. Indeed, a consistent mapping shall map a variable y on a variable x only if the place in which y is localized by the formula is a place where x is localized. Of course, if a variable y is not localized by the formula, it can be mapped on any variable of $\vec{x}$. In practice, this optimization reduces significantly the number of formulas added. Indeed, existential variables are usually localized in the right side of transitions and universal variables are localized in general by invariant.

Generating all possible equivalence classes is very expensive. Instead, recent researches on SMT solvers propose efficient algorithms to deal with theory with equality and uninterpreted functions. Then, formulas in Σ_1 obtained are given to such SMT solver (e.g., Yices) to decide of their satisfiability if they don't contain $p(x)$ sub-formulas. Therefore, we don't need to implement the full procedure to reduce formulas from Σ_1 to Σ_0, but only the step 2 which eliminates $p(x)$ formulas based on localization of tokens.

6.4 Implementation Issues

After the steps performed for the invariant checking and described in chapter 5, for each rule τ, we obtain formulas with the following form:

$$\bigvee_j I_{post} \wedge (\bigwedge_i I_{i\neg\tau}) \wedge \neg I_{j\tau}$$

Each jth CML formula in Σ_2 has to be reduced to Σ_0 using the following steps:

1. Rename redundant quantified variables in formulas.
2. Compute the prenex normal form of these formulas.
3. Reduce formulas from Σ_2 to Σ_1 without $p(x)$ sub-formulas.
4. Call Yices for the Σ_1 formulas obtained.

The remaining of this section details each step above.

6.4.1 Rename redundant variables

As shown in section 6.2, the reduction procedure supposes that the Σ_2 formula φ to be reduced is prenex form and all quantified variables are different. We have to ensure this before performing the reduction.

Due to the invariant splitting, we note a redundancy of quantified variables in formulas I_{post} and $\neg I_{j\tau}$. Indeed, they are both based on the formula $I_{j\tau}$.

We chose to rename variables of the formula $\neg I_{j\tau}$. We give new time-stamps to variables and thus each variable is defined by a unique couple: name and time-stamp.

6.4.2 Computing PNF

Then, we have to put the formula in a prenex normal form. We note that each one of the formulas I_{post}, $(\bigwedge_i I_{i\neg\tau})$, and $\neg I_{j\tau}$ is (or can be put easily) in the prenex normal form. Naturally, their conjunction is not in PNF.

The problem now is how to build a Σ_2 prenex normal form of a conjunction of $B(\Sigma_1)$ formulas. Let's consider two formulas φ_1 and φ_2 in $B(\Sigma_1)$, each one in a prenex normal form. We want to compute the prenex normal form of the formula $\varphi = \varphi_1 \wedge \varphi_2$, so that $\varphi \in \Sigma_2$.

The idea is to look through the list of quantified variables of φ_1 simultaneously with the list of quantified variables of φ_2 from right to left (i.e., from Σ_0 to Σ_2). The merging of two lists is done such that variables quantified universally are brought out before variables quantified existentially. The list obtained by merging is reversed in order to obtain the final list.

Collecting Places for Quantified Token Variables: During the computation of the prenex normal form, the set of possible places for each token variable is collected. This will help in the next step of the reduction from Σ_2 to Σ_1.

For variables quantified universally, their possible places are collected from triggers. For variables quantified existentially, the possible places are collected by looking at $p(x)$ conjuncts; if the formula is not a conjunction, the full set of places is considered. A new data structure `quant_ty` is defined to store this useful information.

Let's note that for variables quantified universally, we don't look through the whole formula for implications to collect places, we only look in the triggers. The, the set of places we obtain it's an over-approximation of the exact result. However, it allows to make a significant optimization.

6.4.3 Reduction from Σ_2 to Σ_1

At this point, we have a closed formula in Σ_2 in prenex normal form ready to be reduced:

$$\varphi = \exists\vec{x}.\ \exists\vec{z}.\ \forall\vec{y}.\ \phi$$

The, we have to map all token variables quantified universally $\vec{y}$ into token variables quantified existentially $\vec{x}$. The optimization we propose is to map only variables which are located at the same place. The collection of the set of places for each variable is done during the prenex normal form.

Formally, the mapping is done using a set of substitution σ which is consistent with the localization of tokens. Moreover, in order to reduce the number of traversals of the formula built, its sub-formulas $p(x)$ are simplified to true or false, depending on the localization considered for $\vec{x}$:

$$\varphi = \exists\vec{x}.\ \exists\vec{z}..\ \bigwedge_{\sigma\in[\vec{y}\to\vec{x}]_p} \phi[\sigma][\vec{x}\to\vec{p}]$$

Let's consider a formula of the Reader-Writer lock example to reduce, with the following quantified token variables (we mention for each variable its location):

- Existential quantification: `u:rw; up:rw; lp:rw; tp:w2; l:rw`
- Universal quantification: `t:r3; r:r2,r3; w:w2,w3; lw:rw; u: rw; up: rw`

where `u, up, lp, tp, l, t, rw, lw` are token variables, and `rw, w2, r2, r3, w2, w3` are places.

Taking this localization information into consideration, we remark that:

- It is not consistent to map variable `lw` on `tp`, since $places(lp) \cap places(tp) = \emptyset$.
- Since $places(t) \cap places(u, up, lp, lr) = \emptyset$, only one mapping is consistent for `t`, i.e., the one to `tp`.

Then, the algorithm implemented for the reduction is the following:

1. Build two mappings:

   (a) exmap: p → { set of token variables quantified existentially in p} which associates to each place p the set of token variables quantified existentially and localized in p. For example, rw → {u, up, lp, lr}

   (b) allmap: p → { set of token variables quantified universally in p} which associates to each place p the set of token variables quantified universally and in localized in p. Example, rw → {lw, u, up}

2. Build a partial substitution (list of pairs) $\sigma_1 \in [\vec{y}' \to \vec{x}]$ such that at most one choice is possible for each variable in $\vec{y}' \subseteq \vec{y}$, i.e., quantified universally. Then, substitution σ_1 does the mapping for universal variables which have only one consistent choice w.r.t. localization. It is easily computed by considering places p such that $allmap(p)$ is not empty and $exmap(p)$ has at most an element.

3. Apply σ_1 on each predicate of the list of predicates and simplify following the localization chosen. A new list of predicates is produced.

4. For the variables quantified universally which don't have a unique mapping, build a list of substitutions L_{σ_2} to variables quantified existentially localized in the same place.

5. Apply each substitution $\sigma_2 \in L_{\sigma_2}$ on each formula of the list of formulas and simplify sub-formulas $p(x)$ according to the localization.

6.4.4 Calling Yices for Σ_1

The last step to do is to print the formulas obtained into a file in the SMTlib format and to call Yices. Recall (from section 5.6) that, in order to prove that the invariant is inductive, all the formulas obtained have to be checked for unsatisfiability since they all belongs to a disjunction.

Then, for each formula a query is generated to the solver and the result is monitored:

- If the result is "`unsat`", it means that the sub-formula checked is unsat, but we have to continue to check that there is not another satisfiable formula.

- If the result is "`sat`", it means that the disjunction checked is satisfiable, then we interrupt all the process of invariant checking. This is done by raising an exception.

- If the result is "`unknown`", it means that the SMT-solver is not powerful enough to solve this problem and the reduction from Σ_1 to Σ_0 shall be done. In our experiments with Yices, we don't find such formulas.

6.5 Experimental Results

The results of the implementation of the verification system for CPN were tested using the Reader-Writer lock example as described in [11]. The results obtained are the following: 25 queries are generated for Yices; each query has a size varying from 65 to 566 lines with a total of 5218 lines for the 25.

Yices was able to solve all of them, but as there are still some bugs, Yices answers SAT when it should answer UNSAT.

The time of execution for the example of the Reader-Writer lock on the 4GHZ, bi-processor, Pentium V is of 0,697sec.

The results are very promising. We keep on fixing the bugs and on trying the verifying system on other examples.

6.6 Conclusion

We presented in this chapter the last step for the invariant checking which is the satisfiability checking of the formulas obtained in the previous chapter. For this, these Σ_2 formulas need to be reduced into Σ_0 formulas which can be solved by the existing solvers. However, due to the improvement of techniques in the SMT-solvers field, for example with Yices, we can stop the reduction to a subset of Σ_1 formulas.

Besides the reduction, we presented some necessary work done: (1) the renaming of quantified variables in formulas and (2) the building of the prenex normal form of formulas in the fragment Σ_2.

Chapter 7

Conclusion and Perspectives

In [11] is proposed a generic framework for reasoning about parametrized and dynamic networks of concurrent processes which can manipulate (local and global) variables over infinite data domains. The framework consists of the expressive model CPN, and of the first order logic CML.

The purpose of this work is to build an efficient verifying system for CPN. This has been done using the verification system WHY to avoid heavy development of parser and type-checker, by using the ones provided by the verification system WHY. We also benefit from the general specification of the color logic allowed by WHY and the SMT solver Yices that he supports.

Several techniques are adopted to make efficient computations, starting with a CNF simple form of invariants, and by taking into account the places associated to each subinvariant. As a result, small SAT formulas in Σ_2 are generated to check the invariant introduced.

Hereafter, the formulas are reduced into Σ_1 using also some optimization techniques. The resulted formulas can be given directly to Yices to decide of their satifiability.

The system built as a result for this work was applied to the reader_writer example and gave satisfying results.

This work has several perspectives. One of them is improve the existing system and to apply it to other examples.

[11] are working on another system of verification, which is an extension of CPN. This new system considers different data domains in order to deal with other classes of systems such as multithreaded programs where each process (thread) has an unbounded stack (due to procedure calls). Building a new verifying system, or improving the existing one, to take into account the new verification system defined by [11] can be considered as another important perspective of our work.

Bibliography

[1] The free encyclopedia. http://www.wikipedia.org/.

[2] The objective caml language. http://caml.inria.fr/.

[3] Rfc 3208. http://www.javvin.com/protocol/rfc3208.pdf.

[4] The why verification tool. http://why.lri.fr/.

[5] Yices: An SMT solver. http://yices.csl.sri.com/.

[6] N. Amla, R. Kurshan, K. McMillan, and R. Medel. Experimental Analysis of Different Techniques for Bounded Model Checking. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 34–48. Springer, April 2003.

[7] André Arnold and Irène Guessarian. *Mathematics for Computer Science*. Prentice-Hall, Masson. Prentice-Hall, Masson, 1996.

[8] Roberto J. Bayardo and Robert C. Schrag. Using CSP lookback techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence(AAAI)*, pages 203–208, July 1997.

[9] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer, July 1999.

[10] P. Bjesse, T. Leonard, and A. Mokkedem. Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers. In G´erard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV)*, volume

2102 of *Lecture Notes in Computer Science*, pages 454–464. Springer, July 2001.

[11] A. Bouajjani, Y. Jurski, and M. Sighireanu. A generic framework for reasoning about dynamic networks of infinite-state processes. In O. Grumberg and M. Huth, editors, *Proceedings of the 13rd Intern. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *LNCS*, pages 690–705. Springer-Verlag, March 2007.

[12] H. B¨uning and T. Lettmann. *Propositional logic: Deduction and Algorithms*, volume 48 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1999.

[13] F. Copti, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of Bounded Model Checking in an Industrial Setting. In G´erard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science*, pages 436–453. Springer, July 2001.

[14] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[15] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[16] L. de Moura. *System Description : Yices 0.1*.

[17] J. Filliatre. Why: a multi-language multi-prover verification condition generator.

[18] J. Filliatre. *Why: an Intermediate Language for Program Verification*, 2007.

[19] J. Filliatre. *The WHY Verification Tool: Tutorial and Reference Manual*, version 2.03 edition, apr 2007.

[20] J. Filliatre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification (CAV07). jul 2007.

[21] C. Flanagan, S.N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *ESOP*, pages 262–277, 2002.

[22] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[23] E. Goldberg and Y. Novikov. Berkmin: a Fast and Robust Sat-Solver. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 142–149, March 2002.

[24] C. Marché, C. Paulin, and X.Urbain. *The Krakatoa Tool for JML/Java Program Certification.* Submitted to JLAB. http://www.lri.fr/ marche/Krakatoa/.

[25] Jo?ao P. Marques-Silva and Karem A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, may 1999.

[26] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC)*, pages 530–535, June 2001.

[27] William F. Ogden, Joseph E. Hollingsworth, Joan Krone, Murali Sitaraman, and Bruce W. Weide. The resolve software verification vision.

[28] M.R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. 2005.

[29] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise Strategies for Improving Local Search. In *Proceedings of the 12th National Conference on Artificial Intelligence(AAAI)*, pages 337–343, July 1994.

[30] Bart Selman, Hector J. Levesque, and David Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the 10th National Conference on Artificial Intelligence(AAAI)*, pages 440–446, July 1992.

[31] H. Zhang. SATO: An Efficient Propositional Prover. In *Proceedings of the 14th International Conference on Automated Deduction (CADE)*, volume 1249, pages 272–275. Springer, July 1997.

[32] L. Zhang and S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. In Ed Brinksma and Kim G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 17–36. Springer, July 2002.