Modelling and Analysis of Real Time Systems with Logic Programming and Constraints

Gourinath Banda

AUGUST 2010 ROSKILDE UNIVERSITY COMPUTER SCIENCE RESEARCH REPORT #130

Computer Science
Department of Communication,
Business and Information Technologies
Roskilde University
P. O. Box 260
DK–4000 Roskilde
Denmark

Telephone:	+45 4674 3839
Telefax:	+45 4674 3072
Internet:	http://www.ruc.dk/dat_en/
E-mail:	datalogi@ruc.dk

ISSN 0109–9779

Modelling and Analysis of Real Time Systems with Logic Programming and Constraints

Gourinath Banda

August 11, 2010

Abstract

Embedded systems are increasingly being deployed in a wide variety of applications. Most, if not all, of these applications involve an electronic controller with discrete behaviour controlling a continuously evolving plant. Because of their hybrid behaviour (discrete and continuous) and reactive behaviour, the formal verification of embedded systems pose new challenges. Linear Hybrid Automata (LHA) is a language for specifying systems with linear hybrid behaviour. Abstract interpretation is a formal theory for approximating the semantics of programming languages. Model checking is a technique to verify the reactive behaviour of concurrent systems. Computation Tree Logic (CTL) is a temporal property specification language. Logic programming is a general purpose programming language based on predicate logic.

In this dissertation, the LHA models are verified by encoding them as constraint logic programs. The constraint logic program (CLP) encoding an LHA model is first specialised and then a concrete minimal model (or possibly an abstract minimal model) for the residual program is computed. The abstract minimal model is computed by applying the theory of abstract interpretation. The computed minimal model forms the basis for verifying the LHA model. We consider two techniques to verify the reactive properties specified as CTL formulas: (i) reachability analysis and (ii) model checking.

A systematic translation of LHA models into constraint logic programs is defined. This is mechanised by a compiler. To facilitate forward and backward reasoning, two different ways to model an LHA are defined. A framework consisting of general purpose constraint logic program tools is presented to accomplish the reachability analysis to verify a class of safety and liveness properties. A tool to compute the concrete minimal model is implemented. The model checking of CTL is defined as a concrete CTL-semantic function. Since model checking of infinite state systems, which LHAs are, does not terminate, we apply the theory of abstract interpretation to model checking that ensures termination at the cost of loss in precision. An abstract CTL-semantic function is constructed as an abstract interpretation of the CTL-semantic function. This abstract CTL-semantic function is implemented using a SMT solver resulting in an abstract model checker. We consider two abstract domains: (i) the domain of constraints and (ii) the domain of convex polyhedra, for both abstract model checking and abstract minimal model computation.

We demonstrate the applicability of the proposed theory with examples taken from the literature.

Acknowledgements

The work reported in this dissertation was undertaken at the Computer Science section at Roskilde University, partly supported by an EU-IST project called "Advanced Specialisation and Analysis of Pervasive Systems".

This dissertation is an outcome of the support and help of many people. Therefore it is only appropriate to convey thanks and acknowledge these people.

First and foremost, I must thank my supervisor Professor John P. Gallagher for accepting me as his PhD student. I can still recall my very first meeting with John and particularly his speculative questions concerning my non-Computer Science background and my ambitions to embark on PhD studies in Computer Science. John was right in his concerns, because in the beginning everything seemed no less than snipe hunting for me. Therefore, now, if I have reached the penultimate milestone on this PhD journey, then it is only because of John's patient supervision, guidance and valuable insights. He has taught and made me to learn most of what I know about logic programming, specification languages, modelling, abstract interpretation, model checking etc. This dissertation was enabled and sustained by his vision and ideas. Of course, I remain liable for any errors that remain in this work.

I would like to thank the people I had a chance to visit during my studies. These are Professor Christo Angelov at the Software Engineering Group at the University of Southern Denmark, Sønderborg, and Dr. Henk Muller at the Mobile and Wearable Computing Group at University of Bristol, United Kingdom.

I would like to thank Dr. Kim S. Henriksen for the support he provided in carrying out experiments and Dr. Jan Midtgaard who was catalytical in my understanding of the fundamentals of abstract interpretation.

Anonymous reviewers provided helpful comments on submitted papers. Developers of the tools used in this work kindly provided support. I would therefore thank Professor Michael Leuschel for the support on the Logen partial evaluator, Jesus Correas in getting me started with Ciao Prolog, and Professor Germán Puebla, Daniel Cabeza and Jose Morales for their assistance on Ciao Prolog.

Furthermore, I thank all those researchers who answered my queries via email. This list includes Doctors Amir Pnueli, Jiri Adamek, Joost-Pieter Katoen, Leslie Lamport, Nicolas Halbwachs, Paul Pettersson, Peter Padawitz, Thomas Wahl and Wolfgang Reisig.

I would like to thank the Administrative and IT support staff of Roskilde University, particularly Donald Axel, Søren Døygaard, Søren Hjortkjaer, Heidi Lundquist and Mette Seistrup.

Finally, I will remain grateful to Baalbrahmachari Shri Maharishi Mahesh Yogi for bestowing me the technique of meditation and introducing me to the eternal guru parampara.

Gourinath Banda
April 2010

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Embedded software systems are increasingly being deployed in several *safety critical applications*, which include transportation, health care, power generation, and many others. The consequences of a *software bug*[1] in such systems can be catastrophic, both in terms of life and money. Hence embedded software systems have to be rigorously analysed to make certain that they are *correct*.

Formal verification [120] is a well known approach for analysing the functional and non-functional correctness of hardware and software systems based on precise mathematical foundations employing formal languages and techniques. There exists a rich repository of formal languages and verification techniques as surveyed in [98, 54, 111, 78, 25, 119, 104]. These formal verification techniques could be broadly categorised, following Pnueli [103], into two classes: (i) the class of algorithmic verification techniques and (ii) the class of deductive methods. *Model checking* [39, 106] is a prominent member of the former class; while *Theorem proving* [33, 108] is a prominent member of the latter class. Recent efforts towards synergistic integration of algorithmic and deductive techniques resulted in another class of *hybrid verification techniques* [16]. In all these techniques, the correctness of high-level models of a system is verified with respect to a particular property.

Program analysis is a static compile-time technique for *predicting safe and computable approximations* of the set of values or behaviours possible when a program is executed on a computer. This technique analyses the actual program itself. Program analysis can also be applied to high-level system models by representing them using programs in an appropriate language. Then the formal verification of the represented model can be interpreted from the results of such program analysis.

The thesis of this dissertation is that *general purpose program analyses designed for constraint logic programs can be applied to the formal verification of high-level specifications of embedded systems.*

[1]An *erroneous behaviour* inadvertently implemented in software is called a *software bug.*

Embedded Systems

Most, if not all, embedded systems are *hybrid*, *reactive* and *real-time.* They are hybrid, because some of their variables are evaluated over *discrete domains* and some are evaluated over *continuous domains* [41, 53, 5, 90]. They are *reactive* because they have to continuously interact with their environments and should never terminate [92]. They are real-time because they should produce right computation within predefined time durations. Because of such characteristics, embedded systems software is highly complicated and even more complicated is its verification.

Formal verification in essence involves two phases, namely, the specification (or modelling) phase and the proof phase. In the specification phase, (a) the systems are specified in an appropriate *modelling language* [120]; (b) the properties are specified in an appropriate *property specification language* [120]. In the proof phase, the correctness of a specified model with respect to the specified properties is either *algorithmically checked* or *deductively inferred.*

Because of their hybrid behaviour, modelling of embedded systems requires a formal language that supports both continuous and discrete variables. Also, specifying their reactive properties often requires *temporal logics* [102] that provide temporal operators to formalise such reactive behaviour. In this dissertation, we focus on the high-level specification language of Linear Hybrid Automata [4] as the modelling language and Computation Tree Logic [23] as the property specification language.

Because of the variables that range over continuous domains, the embedded systems are infinite state systems. Consequently their verification by model checking becomes *undecidable* . Undecidability can be dealt with by employing abstraction techniques. In this dissertation, to accomplish the proof phase we apply Static program analysis, Model checking and Abstract interpretation techniques.

A Framework for Verifying Embedded Systems

This dissertation proposes a framework to verify high-level specifications of embedded systems by systematically applying *static program analyses* of *constraint logic programs*[2] and the *model checking technique.* Figure 1.1 outlines this framework.

This framework has three parts: (i) modelling; (ii) reachability analysis and (iii) abstract model checking.

The modelling part (MP) of the framework applies program transformation and specialisation techniques. This part takes in two inputs and generates one output. One input is the model (of the system to be verified) written in a chosen high-level language and the other is a specification of the *computational model* underlying the high-level specification language. The output is a *state transition*

[2]These are programs written in the language of Constraint Logic Programming [69].

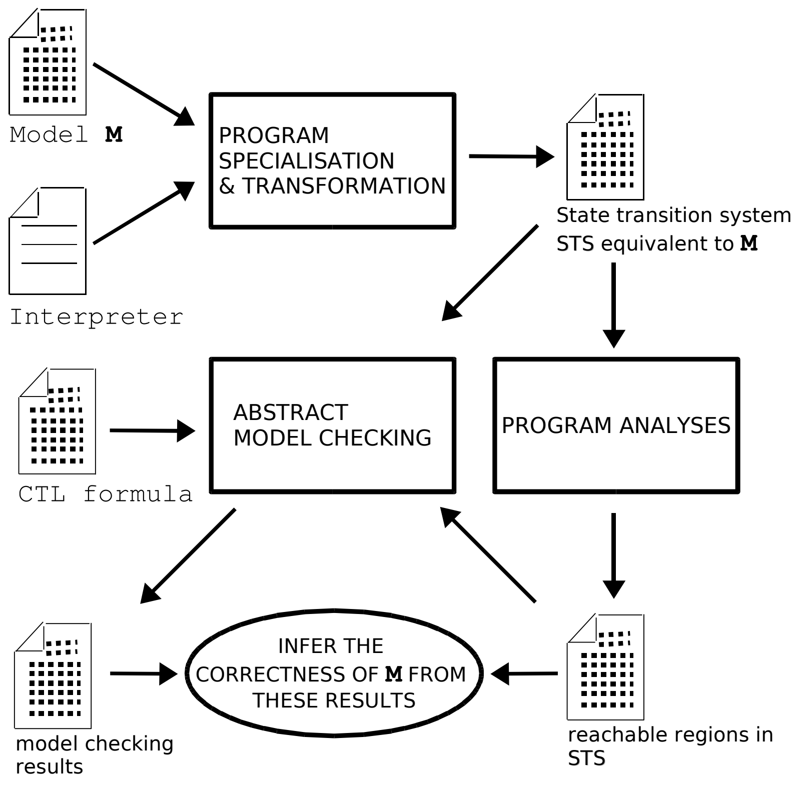
Model M
Interpreter
PROGRAM SPECIALISATION & TRANSFORMATION
State transition system STS equivalent to M
CTL formula
ABSTRACT MODEL CHECKING
PROGRAM ANALYSES
model checking results
INFER THE CORRECTNESS OF M FROM THESE RESULTS
reachable regions in STS

Figure 1.1: A framework for verifying embedded systems

system equivalent of the system model. To be specific, this second input is an interpreter for the high level specification language (in which the system is modelled) written in *constraint logic programming language*. Consequently, the output *state transition system* is a specialised version of the interpreter that was input.

The reachability analysis part (RAP) applies static analysis techniques. It accepts the *state transition system* (output by MP) as its input and outputs the reachable states. This part outputs either the set of concrete reachable states or the set of abstract reachable states. With these sets, the correctness of *a class of safety and liveness properties* could be verified.

The abstract model checking part integrates the theory of abstract interpretation and the model checking technique. This part has three inputs: the state transition system (from MP), the set of reachable states (from RAP), and a CTL formula (specifying the property to be verified), and outputs a set of states where the formula holds. Thus model checked are a class of CTL formulas.

In the following, we briefly explain each of the techniques employed in the framework.

Abstract Interpretation and Model checking

Model checking is an algorithmic technique to verify the properties of finite state systems. We consider *temporal logic model checking* in particular. In this model checking, the system to be verified is modelled as a state transition system and the property is specified in a temporal logic language. We chose a temporal logic language called Computation Tree Logic (CTL). Let STS be the state transition system model of the system with set `InitStates` as its initial states. Let ϕ be the formula written in CTL that specifies the property to be checked. Then the *model checking algorithm* computes $[\![\phi]\!]$, the set of states of STS where ϕ holds. According to the theory of model checking [12], the property ϕ holds in STS if $\texttt{InitStates} \subseteq [\![\phi]\!]$. When a property ϕ holds on STS, we write $\forall s \in \texttt{InitStates} : s, STS \models \phi$ or $STS \models \phi$, in short. When $\texttt{InitStates} \cap [\![\phi]\!] = \emptyset$ it means ϕ does not hold on STS and is written as $\forall s \in \texttt{InitStates} : s, STS \not\models \phi$ (or $STS \not\models \phi$ in short).

Whenever the computation of $[\![\phi]\!]$ becomes either expensive or impossible, we apply the theory of *abstract interpretation* [26, 27] to compute a set $[\![\phi]\!]^a$ which is a safe approximation of $[\![\phi]\!]$ i.e. $[\![\phi]\!]^a$ is a set larger than $[\![\phi]\!]$. Therefore, abstract interpretation-based model checking (AMC) is sound in refuting a property. Since, $STS \not\models \neg\phi \implies STS \models \phi$, abstract interpretation provides a computationally inexpensive alternative to model checking. Particularly, AMC makes model checking of infinite state systems feasible. However, such AMC is sound but not complete. Meaning that not all properties provable with pure model checking can be proved with AMC.

Abstract Interpretation and Static Analysis

Static analysis [95] is a broad term covering program analyses techniques and methods where the *behaviours* of a program are approximately calculated without actually executing the program. Though the calculated behaviours are approximate, the results from static analysis are *sound* i.e. the static analysis results are guaranteed to hold when the software is actually executed. Such soundness is guaranteed because the calculations are always safe approximations of the actual run time behaviours.

The accurate behaviours (seen at run-time) of a program, for example, could be – what are the exact values taken by the program variables at run time, what is the state trace of a program, etc. With static analysis calculating such accurate behaviours might not be possible; rather we could calculate: the range of values possible for program variables, invariants etc. Though these calculations are approximate, we could conclude results such as that program variables never take certain (unsafe) values.

Static analysis involves computing the possible behaviours of a program P as a fixed point of a monotonic function, say F_P (which characterises the semantics of the program P being analysed). Since a fixed point is computed iteratively; for some programs, fixed point computation might be *expensive* or *impossible* i.e. the fixed point cannot be hit within a predefined (finite) amount of time or memory.

Such expensive cases can then be handled by abstract interpretation. In abstract interpretation, the variables are evaluated over abstract domains. Then again, the results from such an abstract analysis are an over approximation of the actual possible behaviours at run time. Let set S and set $S^{'}$ be the results from the actual static analysis and abstract interpretation -based analysis of a program P. Then the theory of abstract interpretation guarantees that $S \subseteq S^{'}$. Therefore, S and $S^{'}$ being the sets of program behaviours, $b \notin S^{'} \implies b \notin S$. Thus, such abstract analyses, though approximate, are indeed helpful in concluding that certain behaviours do not occur in the program.

However, if the abstraction is too imprecise, the set $S^{'}$ might include the forbidden behaviour b, which is never possible in the concrete system. Hence finding the *precise* abstract domains is a challenge in abstraction-based analyses.

Program specialisation and Program transformation

Program transformation [71] is a technique of modifying a program by repeatedly applying a *set of program rewrite rules.* This technique has been extensively used to generate software from high level specifications. The transformed program should preserve the semantics of the original program.

Program specialisation [72] is a source-to-source program transformation technique to specialise a given general purpose program for certain specific application area. The specialised program is computationally more efficient than the original program but has a restricted applicability.

1.1 Thesis Overview

As the proposed verification framework is based on the analysis of logic programs, Chapter 2 introduces logic programming and the bottom-up semantics of the definite logic programs. Chapter 3 introduces reactive systems and a formal specification language called Linear Hybrid Automata to model reactive systems. In this chapter, also explained is a systematic procedure for translating Linear Hybrid Automata (LHA) models into (constraint) logic programs. Chapter 4 introduces the property specification language of *Computation Tree Logic* and explains how different classes of properties are specified in that language. Chapter 5 presents a CLP-based approach to verify reachability properties of LHA models. Chapter 6 introduces the verification technique of model checking and how it can be integrated with the theory of abstract interpretation to verify *temporal properties* specified in CTL. Chapter 7 reports on the experiments. Chapter 8 gives an account of the existing work and Chapter 9 concludes the dissertation.

1.2 Contributions

The work presented in this dissertation is partly funded by the EU-IST-FET project *Advanced Specialization and Analysis for Pervasive Computing*[3] (ASAP) and Roskilde University. One of the goals [105] of the ASAP project was to develop techniques based on high-level languages to verify and analyse pervasive systems. Hence, as said in the thesis statement, the objective of this dissertation is to apply general purpose constraint logic program analysis tools to verify embedded systems. My work towards this objective under the guidance of my supervisor John P. Gallagher resulted in the following contributions:

Modelling of linear hybrid automata in CLP. A systematic translation of Linear Hybrid Automata into constraint logic programming is defined.

CLP-based reachability properties verification of linear hybrid systems. A technique for capturing the possibly infinite set of reachable states of an LHA with a finite set of CLP constraints is defined. This technique applies the static program analyses and the technique of abstract interpretation. A tool to compute the minimal model of constraint logic programs is implemented in CLP and interfaces with the Parma Polyhedra Library (PPL). Different standard recipes for verifying reachability properties are proposed.

Abstract interpretation-based model checking of linear hybrid systems. A standard definition of CTL-semantics function in a suitable form, using only monotonic functions and fixed-point operators is given. Then the standard abstract interpretation framework is applied to get a precise abstraction of the CTL-semantics function. By choosing a constraint-based abstraction of the reachable state space, it is shown how to directly implement an abstract interpretation based model checker (AMC). This model checker is also implemented in CLP and interfaces with the PPL library and a constraint solver, which is based on the satisfiability modulo theories (SMT)-technology.

[3] `http://clip.dia.fi.upm.es/Projects/ASAP/`

The following conference publications reported the above mentioned contributions and contributed to this thesis.

[1] Gourinath Banda and John P. Gallagher. Constraint-Based Abstract Semantics for Temporal Logic: A Direct Approach to Design and Implementation. In Edmund Clarke and Andrei Voronkov, chairs, *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, Dakar, Senegal April 26-30*, 2010, Proceedings. (to appear)

[2] Gourinath Banda and John P. Gallagher. Constraint-Based Abstraction of a Model Checker for Infinite State Systems. In Armin Wolf and Ulrich Geske, chairs, *23rd Workshop on (Constraint) Logic Programming, Potsdam, Germany September 15/16*, proceedings, 2009.

[3] Gourinath Banda and J. P. Gallagher. Analysis of Linear Hybrid Systems in CLP. In M. Hanus, editor, *LOPSTR 2008, volume 5438 of Lecture Notes in Computer Science, pages 5570. Springer*, 2009.

[4] Kim S. Henriksen, Gourinath Banda and John P. Gallagher. Experiments with a Convex Polyhedral Analysis Tool for Logic Programs. In P. Hill and W. Vanhoof, editors, *WLPE-07: Workshop on Logic-Based methods in Programming Environments: ICLP-07 Workshop*, 2007.

[5] John P. Gallagher, Kim S. Henriksen and G. Banda. Techniques for scaling up analyses based on Pre-Interpretations. In M. Gabbrielli and G. Gupta, editors, *Proceedings of the 21st International Conference on Logic Programming, ICLP'2005*, volume 3668 of *Springer-Verlag Lecture Notes in Computer Science*, pages 280-296, 2005.

Chapter 2

Constraint Logic Programming and its Semantics

In this dissertation, we chose the language of *constraint logic programming* (CLP), which is a subset of first order logic, to encode or represent the embedded systems to be verified. This chapter introduces terminology and basic concepts of constraint logic programming.

By formalising a system in *first-order logic*, the standard proof theory of first order logic could be exploited in order to prove properties of the system. The set of possible behaviours of the system can be deduced from the formal model of the system.

Logic programming, a subset of first order logic, provides a sufficiently expressive formalism for representing *non-deterministic state transition systems.* The semantics of formal languages [6, 59, 15] (used for specifying embedded systems) can be conveniently defined with state transition systems. Constraint logic programs and logic programs have themselves been used as models of embedded systems by many [94, 93, 55, 70].

CLP belongs to the class of declarative languages. In declarative programming languages, ideally, the programmer only states *what* is to be computed and not necessarily *how* it is to be computed. The computational engine (underlying the declarative language) computes the solution according to a standard algorithm. In the logic programming paradigm, logic and control are separated [79]. A logic program specifies only the *logic*; while the *control* is implemented in the LP run-time system.

In this dissertation, though we encode state transition systems as logic programs, these logic programs are not necessarily intended to be executed on the LP run-time system. Rather, these programs will be subjected to static analysis. For this reason, we focus only on the model-theoretic semantics of the logic programming language.

We first introduce pure logic programs, which contain only uninterpreted functions and predicates. Since embedded systems involve actions, whose modelling requires arithmetic and constraints, we then introduce the constraint logic programming (CLP) language. CLP is a logic programming language extended with constraint functions and predicates having fixed interpretations.

We choose a subset of logic programming language called *Prolog* (an acronym for PROgramming in LOGic) as the representation language. Prolog is a CLP language where constraint expressions can be embedded in the program as formulas over terms. For instance, if X, Y are two variables (or terms), then arithmetic relations like $X = Y + 1, X > 2 * Y$, *etc.* can be treated as formulas defining the condition that must hold on the variables X, Y. Typically, the variables appearing in the constraints are evaluated over some fixed domain, which could be either the domain of integer numbers, rational numbers or real numbers, or some other user-defined domain.

Chapter Overview

The necessary logic programming terminology and the semantics of logic programs is explained in this chapter.

- Section 2.1 introduces the first order logic.
- Section 2.2 explains how the semantics of logic programs is computed.
- Section 2.3 introduces the language of constraint logic programming.
- Section 2.4 introduces the technique of abstract interpretation and its application to compute the approximate semantics of (constraint) logic programs.

2.1 First-order logic

Syntax

Definition 1 (Alphabet)**.** *The alphabet of a first order language consists of the following sets of symbols:*

- *constants, which will be written as numerals or alphanumeric identifiers beginning with lower-case letters.*
- *functors, which will be written as alphanumeric identifiers beginning with lower-case letters.*

- *variables, which will be written as alphanumeric identifiers beginning with upper-case letters (sometimes subscripted and/or primed).*

- *propositions, which will be written as alphanumeric identifiers beginning with lower-case letters.*

- *predicates, which will be written as alphanumeric identifiers beginning with lower-case letters.*

- *connectives* ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$)

- *quantifiers* ($\forall$, $\exists$)

- *auxiliary symbols like parentheses and comma.*

For any alphabet, only the function and predicate symbol sets may be empty. Every function and predicate symbol has an associated *arity*, which is a natural number greater than zero indicating the number of arguments that the function or predicate takes. If a function (resp. predicate) symbol has an arity n, then we call it an n-ary function (resp. predicate) symbol. Sometimes we treat constants as function symbols with 0 arity; while *propositions* as predicate symbols with 0 arity.

In this chapter, we use:

- Σ to denote the union of the set of function symbols and the set of constant symbols in the alphabet.

- Π to denote the union of the set of predicate symbols and the set of proposition symbols in the alphabet.

- *Var* to denote the set of variables in the alphabet.

For readability, we adopt the following syntactical conventions:

- Variables are denoted by upper case letters selected from the end of the alphabet, for example X, Y, Z. Sometimes with subscripts and/or primes like X_1, Y'.

- Constants are denoted by lower case letters selected from the beginning of the alphabet, for example a, b, c.

- Function symbols are denoted by lower case letters selected from the letter f and the following letters, for example f, g, h.

- Predicates are denoted by lower case letters selected from the group of letters beginning with p and forward, for example p, q, r.

An n-ary predicate of the form $p(X_1, \ldots, X_n)$ can be referred to as p/n.

Definition 2 (Term)**.** *A term is defined inductively as follows:*

- *A variable is a term.*
- *A constant is a term.*
- *If f is an n-ary ($n > 0$) function symbol and $t_1, \ldots, t_n$ are n terms, then $f(t_1, \ldots, t_n)$ is also a term.*

Definition 3 (Formula, Atomic formula)**.** *A (well-formed) formula is defined inductively as following:*

- *if p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are n terms, then $p(t_1, \ldots, t_n)$ is a formula (called an atomic formula or, more simply, an atom).*
- *if F and G are formulas, then so are $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$ and $F \leftrightarrow G$.*
- *let X be a variable and F be a formula, then $\forall X\ F$ and $\exists X\ F$ are also formulas. These are called quantified formulas. A formula is said to be closed if all the variables appearing in it are within the scope of a quantifier for that variable. If any of the variables in a formula is not quantified then the formula is said to be open.*

In logic programming, an implication formula $F \rightarrow G$ is written as $G \leftarrow F$.

Definition 4 (First-order language)**.** *The first order language given by an alphabet consists of the set of all formulas constructible from the symbols of the alphabet.*

In this chapter, we use:

- Term_Σ to denote the set of terms over a given alphabet Σ. When Σ is obvious, we just use Term.
- $\mathsf{Atom}_{\Pi,\Sigma}$ denotes the set of atoms constructed from Π and Term. When Σ and Π is obvious, we just use Atom.

Definition 5 (Ground)**.** *A term or formula is said to be ground if it contains no variables.*

Definition 6 (Literal)**.** *If F is an atomic formula then the formulas F and $\neg F$ are called literals. The literal F is called a* positive *literal, while $\neg F$ is called a* negative *literal.*

Definition 7 (Clause). *A clause is a formula of the form* $\forall X_1 \ldots \forall X_s(H_1 \vee \ldots \vee H_m \leftarrow B_1 \wedge \ldots \wedge B_n)$ *where* $m \geq 0$, $n \geq 0$, $H_1, \ldots, H_m, B_1, \ldots, B_n$ *are all atoms and* $X_1 \ldots X_s$ *are all the variables occurring in* $(H_1 \vee \ldots \vee H_m \leftarrow B_1 \wedge \ldots \wedge B_n)$.

The left hand side of the formula, $H_1 \vee \ldots \vee H_m$ *is called the head of the clause, and the right hand side is called the body of the clause.*

A clause of the form $\forall X_1 \ldots \forall X_s(H_1 \vee \ldots \vee H_m \leftarrow B_1 \wedge \ldots \wedge B_n)$ is often denoted $H_1, \ldots, H_m \leftarrow B_1, \ldots, B_n$. This notation implies that all variables are quantified universally. The commas in the antecedent (or body) $B_1, \ldots, B_n$ denote the conjunction '$\wedge$', while the commas in the consequent (head) denote the disjunction '$\vee$'.

2.1.1 Definite logic programs

In what follows, a subset of first order logic is introduced. This subset and its semantics form the basis for logic programming. Logic programming is adequate to represent state transition systems.

Definition 8 (Horn Clause). *A* Horn clause *is a clause with at most one positive literal in its head.*

Definition 9 (Definite program clause). *A* definite program clause *or* definite clause *is a clause with exactly one positive literal in its head. It is of the form* $H \leftarrow B_1, \ldots, B_n$.

Definition 10 (Unit clause). *A* unit clause *is a definite program clause with an empty body. It is of the form* $H \leftarrow$.

In logic programming, a unit clause is called a *fact*. It conveys the knowledge that is true.

Definition 11 (Definite goal). *A goal is a (Horn) clause with an empty head and a non-empty body. It is of the form* $\leftarrow B_1, \ldots, B_n$ *where each body atom* B_i $(i = 1, \ldots, n)$ *is called a subgoal of the goal.*

Specifically for logic programs the following notation is adopted.

- lists are written using Prolog notation $[H|T]$ and $[\ ]$ where H, T are head (element) and tail (list) of the list, respectively, and $[\ ]$ is the empty list. Here $[H|T]$ stands for a special binary function $cons(H, T)$.

- we adopt the notion of *don't care* from Prolog. A *don't care* variable in Prolog is an anonymous variable and is denoted '_'. Repeated occurrences of a don't care variable in a context stand for distinct anonymous variables.

Definition 12 (Definite program). *A definite program is a finite set of definite program clauses.*

2.1.2 Semantics

The semantics of a first order language (FOL) is defined with respect to a particular universe. Given that particular universe, the meaning of a FOL is given by the set of formulas that evaluate to *true* in that universe. Since formulas are composed from terms and atomic formulas, their evaluation requires assigning meaning to their constituent terms and formulas. The meaning of constant symbols and function symbols is given in terms of the elements and functions in that chosen universe.

The correspondence between the language and a chosen universe is formalised with the following concepts.

Definition 13 (Pre-interpretation)**.** *A* pre-interpretation *of a first order language L consists of the following.*

1. *A non-empty set D, which is the universe of discourse, called the domain of the pre-interpretation.*
2. *Each constant in L is assigned an element in D.*
3. *Each n-ary ($n > 0$) function symbol in L is assigned a function mapping D^n to D.*

More informally, a pre-interpretation maps every constant symbol to an element in the supplied domain D, and maps each function f/n to a function $f^J : D_1 \times \ldots \times D_n \rightarrow D$. In general we denote by f^J the function assigned to function symbol f by pre-interpretation J, and by c^J the domain element assigned to constant c.

In order to assign meaning to a compound term, we need to assign meaning to each of its sub-terms. Sub-terms could be constants, variables and terms. To assign a meaning to a non-ground term, we first need to allocate the variables with values in the domain of pre-interpretation.

Definition 14 (Variable valuation)**.** *A variable valuation function $V : Var \rightarrow D$ assigns to each variable in L an element of D.*

Definition 15 (Term assignment)**.** *Let J be a pre-interpretation of the language L with a domain D, and V be a valuation function over D. A* term assignment *$T_J^V(t)$ is defined for each term t as follows:*

1. *$T_J^V(x) = V(x)$ for each variable x;*
2. *$T_J^V(c) = c^J$ for each constant c;*
3. *$T_J^V(f(t_1, \ldots, t_n)) = f^J(T_J^V(t_1), \ldots, T_J^V(t_n))$, ($n \geq 0$) for each term of the form $f(t_1, \ldots, t_n)$.*

The concept of pre-interpretation and term assignment is illustrated with the following example taken from [97].

Example 1. *Consider a language L with $\Pi = \{even/1\}$ and $\Sigma = C \cup F$ where:*

- *set of constant symbols $C = \{zero\}$;*
- *set of function symbols $F = \{s/1, plus/2\}$;*

Let J be a pre-interpretation of L:

- *with the set of natural numbers as the domain of pre-interpretation i.e. $D = N$;*
- *assigning constant zero to 0 i.e. $zero^J = 0$ and*
- *assigning the function symbols as follows:*
  - *$(s/1)^J : N \to N$ is the function defined as $s^J(x) = 1 + x$ (here $+$ is the standard addition operation over N);*
  - *$(plus/2)_J : N \times N \to N$ is the function defined as $plus^J(x, y) = x + y$ (again $+$ is the standard addition)*

Given a valuation $V = \{X \mapsto 0\}$, the term assignment for term $t = plus(s(zero), X)$ is computed as follows:

$$\begin{aligned} T_J^V(t) &= plus^J(T_J^V(s(zero)), T_J^V(X)) \\ &= s^J(T_J^V(zero))) + V(X) \\ &= s^J(zero^J) + 0 \\ &= s^J(0) + 0 = 1 + 0 = 1 \quad \square \end{aligned}$$

Definition 16 (Substitution). *A binding X_i/t_i consists of a variable X_i and a term t_i with $t_i \neq X_i$. A substitution θ is a finite set of bindings,*

$$\theta = \{X_1/t_1, \ldots, X_n/t_n\}$$

where $X_1, \ldots, X_n$ are distinct.

Just as constant and function symbols are assigned to constants and functions in the domain of pre-interpretation, the predicate symbols of arity n are assigned to relations of the same arity (n) over the domain of pre-interpretation.

Definition 17 (Interpretation)**.** *An interpretation I of a first order language L consists of:*

1. *a pre-interpretation J with domain D of L;*

2. *for each proposition symbol the assignment of* true *or* false*;*

3. *for each n-ary predicate symbol $p \in \Pi$, the assignment of a mapping $D^n \rightarrow \{\mathsf{true}, \mathsf{false}\}$.*

We denote by p^I the relation assigned to p by I.

We now introduce the notion of domain atom in order to construct a convenient representation for interpretations.

Definition 18 (Domain atom)**.** *Let D be the domain of pre-interpretation of a language L and let p be an n-ary predicate symbol from L i.e. $p/n \in \Pi$. Then a domain atom w.r.t D is any atom $p(d_1, \ldots, d_n)$ where $d_i \in D$, $i \in \{1, \ldots, n\}$. The set of domain atoms of p/n over D is $\mathsf{Atom}_D^p = \{p(d_1, \ldots, d_n) \mid p/n \in \Pi, d_i \in D, i \in \{1, \ldots, n\}\}$.*

We denote the set of all domain atoms for a language L as:
$Atom_D = \bigcup\{\mathsf{Atom}_D^p \mid p/n \in \Pi\} \cup \Pi_0$ where Π_0 is the set of propositions.

An interpretation of a language w.r.t. D can be represented as a subset of Atom_D. Let I be a subset of Atom_D. This stands for the interpretation that assigns to predicate p the relation p^I given by:
$p^I = \{(d_1, \ldots, d_n) \in D^n \mid p(d_1, \ldots, d_n) \in I\}$. If $p \in \Pi_0$ then $p^I = \mathsf{true}$ if $p \in I$ otherwise false.

Let $I \subseteq \mathsf{Atom}_D$ be an interpretation, and V be a variable valuation, a truth value is assigned to formulas as follows.

- An atom $p(t_1, \ldots, t_n)$ has the value true if $p(T_I^V(t_1), \ldots, T_I^V(t_n)) \in I$ otherwise false.

- A proposition p has the value true if $p \in I$ otherwise false.

- If the formula has the form $\neg Q$, $Q \wedge R$, $Q \vee R$, $Q \rightarrow R$ or $Q \leftrightarrow R$ (where Q, R, S are atomic formulas that are assigned a truth value by I), then its truth value is given by the following table:

Q	R	$\neg Q$	$Q \wedge R$	$Q \vee R$	$Q \rightarrow R$	$Q \leftrightarrow R$
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

- Formulas of the form $\exists X\ Q$ are true if there exists $t \in \mathsf{Term}$ such that $Q\theta$ where $\theta = \{X/t\}$ is assigned the truth value $\mathtt{true}$ with respect to I and the variable assignment V; otherwise $\exists X\ Q$ is false.

- Formulas of the form $\forall X\ Q$ are true if for all $t \in \mathsf{Term}$ such that $Q\theta$ where $\theta = \{X/t\}$ is assigned the truth value $\mathtt{true}$ with respect to I and the variable assignment V; otherwise $\forall X\ Q$ is false.

Definition 19 (Domain instance)**.** *Let A be an atom of the form $p(t_1, \ldots, t_n)$. Then a domain instance of A based on pre-interpretation J and variable valuation V is the domain atom:*
$p(T_J^V(t_1), \ldots, T_J^V(t_n))$. The set of all domain instances of atom A with respect to J (over D) and some variable assignment V is denoted by $[A]_J$.

It is obvious that $[A]_J \subseteq Atom_D$.

The definition of domain instance extends naturally to formulas. In particular, let C be a clause. Denote by $[C]_J$ the set of all domain instances of the clause with respect to J.

Example 2. *For the language and pre-interpretation in Example 1, the set of domain atoms for L is*
$Atom_D = \{even(d) \mid d \in D\} = \{even(0), even(1), even(2), \ldots\}$; while the domain instance (based on the same pre-interpretation J and variable valuation V) for atom $even(plus(X, zero))$ is $even(0)$. $\square$

Example 3. *Consider the language defined in Example 1 whose interpretation is given as $I = \{even(0), even(2), even(4), even(6), \ldots\}$.*

Then the truth values of the formulas $even(s(s(zero)))$ and $even(s(zero))$ are evaluated based on I as below.

First we evaluate the predicate arguments $s(s(zero))$ and $s(zero)$ by applying the term assignment T_J^V as follows.

Applying J and V, $T_J^V(s(zero)) = 1$ and $T_J^V(s(s(zero))) = 2$.

Since $even(2) \in I$ and $even(1) \notin I$, $even(s(s(zero)))$ evaluates to true*; while $even(s(zero))$ evaluates to* false*.* $\square$

Example 4. *In the interpretation I, from the above example (Example 3), the compound formula $even(s(s(zero))) \wedge even(s(zero))$ evaluates to* false*.* $\square$

Definition 20 (Model)**.** *A* model *of a formula is an interpretation in which the formula has the value* $\mathtt{true}$ *assigned to it.*

If I is a model of the formula ϕ, we write $I \models \phi$.

A model of a set S of formulas is an interpretation in which every formula is assigned the value true. In other words, an interpretation is a model for a program if it is a model for every formula in the program.

Definition 21 (Minimum model). *A model $M \subseteq \mathsf{Atom}_D$ of a program P is called a* minimum model *if there exists no other model M' of P such that $M' \subset M$.*

Two formulas (or programs) are logically equivalent if they have the same set of models.

A formula Q is a *logical consequence* of a set S of formulas, if Q is assigned the value true in all models of S; this is denoted $S \models Q$.

A minimal model of P will be denoted $\mathsf{M}(P)$ in this dissertation. M may be super- or sub-scripted with the pre-interpretation of the model.

2.1.3 Herbrand interpretation

Given a language L, a special kind of domain can be constructed from the language itself and a pre-interpretation of L over this domain. The interpretations arising from such pre-interpretations are called Herbrand interpretations, which are used in model-theoretic analysis of logic programs.

Definition 22 (Herbrand Universe and Base). *The* Herbrand Universe *for the first order language L with constant and function symbols Σ is Term_Σ. This assumes Σ to contain at least one constant. The Herbrand universe is denoted by Term_H.*

The Herbrand Base, $\mathsf{Atom}_{\Pi,\mathsf{Term}_H}$, of the language L, is the set of all domain atoms over the Herbrand universe. We abbreviate $\mathsf{Atom}_{\Pi,\mathsf{Term}_H}$ by Atom_H.

Example 5. *Consider the following definite program P:*
$even(zero)$.
$even(s(s(X))) \leftarrow even(X)$.
We assume here that the language of this program contains only the symbols in the program text, namely:

- *one constant symbol zero,*
- *one unary function symbol s and*
- *one unary predicate symbol even.*

By the above definition, the program's Herbrand universe and Herbrand base, respectively, are as below:
$\mathsf{Term}_H = \{zero, s(zero), s(s(zero)), \ldots\}$
$\mathsf{Atom}_H = \{even(t) \mid t \in \mathsf{Term}_H\} = \{even(zero), even(s(zero)), even(s(s(zero))), \ldots\}$
$\square$

Definition 23 (Herbrand pre-interpretation)**.** *The Herbrand pre-interpretation H for a language L is the pre-interpretation given by the following:*

1. *The domain of the pre-interpretation is the Herbrand universe Term_H.*
2. *Constants in Σ are assigned to themselves i.e. $c^H = c$.*
3. *For all function symbols f/n and all terms $t_1, \ldots, t_n \in \mathsf{Term}_H$, f^H applied to $t_1, \ldots, t_n$ equals $f(t_1, \ldots, t_n)$.*

Definition 24 (Herbrand interpretation and Herbrand model)**.** *Any interpretation based on the Herbrand pre-interpretation H of a language L is a* Herbrand interpretation *of L.*

- *A* Herbrand model *of a definite program P in the language L is any Herbrand interpretation of L that is also a model of P.*
- *A Herbrand model $I \subseteq \mathsf{Atom}_H$ for a program P is a* minimum Herbrand model *if no other $I' \subset I$ is also a Herbrand model of P.*

Example 6. *Consider the program P from the previous example (Example 5). The following are some Herbrand interpretations of P:*
$I_0(P) = \emptyset$
$I_1(P) = \{even(zero)\}$
$I_2(P) = \{even(zero), even(s(s(zero)))\}$
$I_3(P) = \{even(zero), even(s(s(zero))), even(s(s(s(s(zero)))))\}$
$I_4(P) = \{even(s^{2n}(zero)) \mid n \in 0, 1, 2, 3, \ldots\}$
$I_5(P) = \mathsf{Atom}_H$.

Of all these interpretations, only I_4 and I_5 are models of program P. The other interpretations are not models, because:

- *the interpretation I_0 says that the fact $even(zero)$ is false;*
- *the interpretations I_j (where $j \in \{1, 2, 3\}$) include $even(s^{2(j-1)}(zero))$ but not $even(s^{2j}(zero))$ and hence do not satisfy the second clause.*

$\square$

Example 7. *In the above example (Example 6), both I_4 and I_5 (being both a Herbrand interpretation and a model of P) are Herbrand models of the program. But I_5 is not the minimal Herbrand model, because $I_5 \supset I_4$.* $\square$

The minimal Herbrand model captures the meaning of definite program P. It can be shown that it contains exactly the ground atomic logical consequences of the program P. In other words, $\forall A \in \mathsf{Atom}_H : A \in \mathsf{M}^H(P) \equiv P \models A$. Thus the minimal Herbrand model has a special status; if we can find any non-Herbrand model of P in which A is false, then the above equivalence implies $A \notin \mathsf{M}^H(P)$. We will exploit this fact in using non-Herbrand models to establish that certain atoms are not logical consequences of a program [50]. This idea of applying non-Herbrand pre-interpretations to derive certain properties of a definite program was introduced in [18], [17], [47] and [48]. The following example illustrates this concept.

Example 8. *Consider a non-Herbrand pre-interpretation J of the language defined in Example 5:*

- *with the domain of pre-interpretation $D = \{e, o\}$;*
- *assigning constant zero to e i.e. $zero^J = e$ and*
- *assigning the function symbols as follows:*
  - $(s/1)^J : D \to D$ *is a function defined as:* $s^J(o) = e$ *and* $s^J(e) = o$;
  - $plus^J : D \times D \to D$ *is a function defined as:*
    * $plus^J(e, e) = e$,
    * $plus^J(e, o) = o$,
    * $plus^J(o, e) = o$ *and*
    * $plus^J(o, o) = e$.

The set of domain atoms for L over D is $\mathsf{Atom}_D = \{even(e), even(o)\}$. The minimal model (for this non-Herbrand pre-interpretation) for the program P of Example 5 is $\mathsf{M}^J(P) = \{even(e)\}$.

Therefore $even(T_V^J(s(s(s(zero))))) = even(o) \notin \mathsf{M}^J(P)$ which implies that $even(s(s(s(zero)))) \notin \mathsf{M}^H$. $\square$

The primary reason behind computing non-Herbrand models is that for some programs the computation of a Herbrand model becomes very expensive and might not even terminate. Thus such models based on non-Herbrand pre-interpretation provide a safe alternative to check the absence of atoms in the Herbrand model.

2.2 Computing the Semantics of Definite Logic Programs

A logic program can be given interpretations based on different pre-interpretations. This section presents a fixed-point characterisation of the minimal model with respect to a given pre-interpretation. Such a fixed-point definition provides a flexible semantic framework where different pre-interpretations could be plugged in to get corresponding models. If we choose the Herbrand pre-interpretation then it results in the standard semantics which is the minimal Herbrand model [89]. Similarly the choice of pre-interpretation over an abstract domain gives an abstract model. Finally the semantics of constraint logic programs (explained in the last section) can be conveniently expressed in this framework.

2.2.1 Bottom-up semantic frameworks

The minimal model for a pre-interpretation J can be computed as the least fixed point of the function T_P^J which transforms one interpretation into another. This transformation function is called the *immediate consequences operator* [117]. The *immediate consequences operator* with respect to J is defined as below.

Definition 25 (Core bottom-up semantics function T_P^J)**.** *Let P be a definite program, and J a pre-interpretation of the language of P over domain D. Let $Atom_J$ be the set of domain atoms with respect to J.*

$$T_P^J : 2^{Atom_J} \rightarrow 2^{Atom_J}$$

$$T_P^J(I) = \left\{ A' \;\middle|\; \begin{array}{l} A \leftarrow B_1, \ldots, B_n \in P \\ A' \leftarrow B'_1, \ldots, B'_n \in [A \leftarrow B_1, \ldots, B_n]_J \\ \{B'_1, \ldots, B'_n\} \subseteq I \end{array} \right\}$$

$$\mathsf{M}^J[\![P]\!] = \mathsf{lfp}(T_P^J)$$

$\mathsf{M}^J[\![P]\!]$ is the minimal model of P with the pre-interpretation J. Here lfp denotes the least *fixed point.* A *fixed point* of a function is a point that is mapped to itself by the function. It should be noted that not every function has a fixed point; the Knaster-Tarski theorem identifies sufficient conditions on a function to have a least fixed point, which furthermore is the limit of the so called Kleene sequence. These details are discussed in Chapter 6.

The least fixed point for this function is calculated by initially applying the function to the empty interpretation (an empty set meaning that no atomic formula is true), and then iteratively applying it to itself until a fixed point is reached. Algorithm 1 outlines a naive fixed-point computation algorithm.

Algorithm 1 Fix point iterations for transfer function T_P^J

```
initialise:
    $i = 0$;  $I_0 = \emptyset$
repeat
    $I_{i+1} = T_P^J(I_i)$
    $i = i + 1$
until $I_i = I_{i-1}$
```

Example 9. *Consider the pre-interpretation defined in Example 1 and the following program:*

$$P = \left\{ \begin{array}{lcl} \mathit{even}(\mathit{zero}) & \leftarrow & \\ \mathit{even}(\mathit{plus}(X,Y)) & \leftarrow & \mathit{even}(X), \mathit{even}(Y) \end{array} \right\}$$

Let the interpretation be $I = \emptyset$. Then $T_P^J(I) = \{\mathit{even}(0)\}$. The set of domain instances of $[\mathit{even}(\mathit{zero}) \leftarrow]_J$ is $\{\mathit{even}(0) \leftarrow\}$ and of $[\mathit{even}(\mathit{plus}(X,Y)) \leftarrow \mathit{even}(X), \mathit{even}(Y)]$ is $\{(\mathit{even}(e)) \leftarrow \mathit{even}(e_1), \mathit{even}(e_2)) \mid e, e_1, e_2 \in N : e = e_1 + e_2\}$. Applying the immediate consequences operator again: $T_P^J(T_P^J(I)) = \{\mathit{even}(0)\} = T_P^J(I)$. Thus a fixed point is reached, so the minimal model of P with respect to the pre-interpretation J is $\{\mathit{even}(0)\}$. $\square$

The standard semantics will be defined next. It is based on the Herbrand pre-interpretation.

2.2.2 Standard semantics

The standard semantics of a logic program P is given by the core bottom-up semantic function with pre-interpretation J replaced with the Herbrand pre-interpretation H. The minimal Herbrand model can be computed as the least fixed point of the function T_P^H, which is usually called T_P [117] that transforms one Herbrand interpretation into another.

2.2.3 Clark Semantics

The minimal Herbrand model consists of ground atoms. In order to capture information about the occurrence of variables, we extend the domain of pre-interpretation to $\mathsf{Term}_{\Sigma \cup \mathcal{V}}$ where $\mathcal{V} = \{v_0, v_1, v_2, \ldots\}$ is a set of extra elements that is in one-to-one correspondence with the set of variables of the language. The pre-interpretation HV over such an extended domain is identical to the Herbrand pre-interpretation (of Definition 23) with Term_H replaced by $\mathsf{Term}_{\Sigma \cup \mathcal{V}}$.

The elements of $\mathcal{V}$ do not occur in the program or goals, but can appear in atoms in the minimal model $\mathsf{M}^{HV}[\![P]\!]$. Let $\mathcal{C}(P)$ be the set of all atomic logical consequences of the program P, known as the Clark semantics [22]; that is, $\mathcal{C} = \{A \mid P \models \forall A\}$, where A is an atom. Then $\mathsf{M}^{HV}[\![P]\!]$ is isomorphic to $\mathcal{C}(P)$. More precisely, let Ω be some fixed bijective mapping from $\mathcal{V}$ to the variables in L. Let A be an atom; denote by $\Omega(A)$ the result of replacing any constant v_j in A by $\Omega(v_j)$. Then $A \in \mathsf{M}^{HV}[\![P]\!]$ iff $P \models \forall(\Omega(A))$. Using the Clark semantics as a basis, we can construct abstractions [48, 49, 50] that capture instantiation properties such as non-groundness. However these abstractions are not the subject of this work.

Example 10. *Take a program containing a head-only variable, say $P = \{p(X) \leftarrow true,\ p(a) \leftarrow true\}$, and given the set of extra constants $\mathcal{V} = \{v_0, v_1, v_2, \ldots\}$, then the concrete semantics is $\mathsf{M}^{HV}[\![P]\!] = \{p(a), p(v_0), p(v_1), \ldots\}$.* $\square$

2.3 Constraint Logic Programming

Constraint logic programming (CLP) for arithmetic domains is logic programming extended with arithmetic operations and constraint relations. So the alphabet of a CLP language contains the arithmetic function symbols $\Sigma_C = \{+, -, \times\} \cup D_C$, where D_C is the set of arithmetic constants, and constraint relation symbols $\Pi_C = \{\leq, \geq, >, <, =\}$. A program in this language is called a constraint logic program.

The arithmetic function and constraint predicate symbols will be given their usual (pre-)interpretation over a chosen numerical domain. This numerical domain could be either the set of real numbers R or the set of rational numbers Q. The chosen numeric domain is usually stated along with the acronym CLP i.e. for instance CLP($\mathcal{R}$) indicates constraint logic programming where the arithmetic and constraint symbols are interpreted over the domain of real numbers R. In general, $CLP(D_C)$ is a constraint logic programming language parameterized by a fixed interpretation of the additional symbols over domain D_C. We restrict ourselves to linear terms only.

Definition 26 ((Linear) arithmetic term)**.** *A linear arithmetic term (hereafter simply called arithmetic term) t in $CLP(D_C)$ is defined by the following grammar.*

$$t ::= k \mid x \mid k * t \mid t_1 + t_2 \mid t_1 - t_2$$

where $k \in D_C$.

Definition 27 (Atomic linear constraint). *An atomic linear arithmetic constraint (hereafter simply called constraint) over the chosen domain is defined by the following grammar.*

$$c ::= t_1 \leq t_2 \mid t_1 < t_2 \mid t_1 \geq t_2 \mid t_1 > t_2$$

where t_1, t_2 are linear arithmetic terms. The constraint $t_1 = t_2$ is an abbreviation for $t_1 \leq t_2 \wedge t_2 \leq t_1$.

Definition 28 (Constraint logic programming language). *The alphabet of a CLP language $CLP(D_C)$ is as defined in Definition 1, except that the sets of constant, function and predicate symbols are each divided into two disjoint parts: the constraint symbols and the uninterpreted symbols (referred to as user symbols). We call the set of user function symbols Σ_U and the set of user predicate symbols Π_U, which are disjoint from Σ_C and Π_C respectively. A user atomic formula has a predicate from Π_U and arguments from Term_Σ, while a constraint atomic formula has a predicate from Π_C and arguments from Term_{Σ_C}.*

Definition 29 (Constraint logic program). *A constraint logic program is a finite set of clauses of the form:*
$H_0 \leftarrow C_1, \ldots, C_m, H_1, \ldots, H_n \qquad (m, n \geq 0)$
where $C_1, \ldots, C_m$ are constraint atomic formulas and $H_0, \ldots, H_n$ are user atomic formulas.

2.3.1 Semantics of CLP

An interpretation is defined as before, except that the symbols in Σ_C and Π_C should be interpreted consistently with their usual interpretation over D_C. We first define intended interpretation over domains that include D_C, where the constraint symbols are given their usual meanings.

Definition 30 (Arithmetic CLP interpretation). *An arithmetic pre-interpretation JC with domain $D \supseteq D_C$ assigns:*

- *each constant in D_C to itself;*
- *a function $D_C^2 \rightarrow D_C$ to each binary arithmetic function $f \in \Sigma_C$ giving the usual arithmetic meaning to f;*
- *a function $D^n \rightarrow D$ to each n-ary user function $f \in \Sigma_U$.*

Given an arithmetic pre-interpretation with domain $D \supseteq D_C$, an arithmetic interpretation I_C assigns:

- *the standard binary relation over D_C^2 to each constraint predicate in* Π_C*;*
- *an n-ary relation over D^n to each user predicate in* Π_U*.*

As before, an interpretation can be represented by a set of domain atoms. We denote by I_C the set of domain atoms giving the standard interpretations of the constraint predicates in Π_C. For instance, given the domain D_C, for the predicate symbol $\geq \in \Sigma_C$, I_C contains all domain atoms of the form $e_1 \geq e_2$ where $e_1, e_2 \in D_C$ such that e_1 is greater than or equal to e_2. I_C is infinite when D_C is infinite.

We can then introduce arbitrary interpretations so long as they are consistent with the arithmetic interpretation. Let J be an arbitrary pre-interpretation of a $CLP(D_C)$ program P and let I be an interpretation based on J. Then I is a D_C safe model if (i) I is a model of P; (ii) every true atomic constraint A, i.e. $A \in I_C$, is also true in I.

We first present the immediate consequences function for CLP which is identical to the T_P^J defined in Definition 25 except that we include the interpretation I_C.

Definition 31 (Standard semantic function for CLP(D_C) T_P^{JC})**.** *Let P be a constraint logic program, and JC be the pre-interpretation of the language as defined in Definition 30. Let $Atom_{JC}$ be the set of domain atoms with respect to JC and let I_{Π_C} be a D_C safe model of the constraint predicates Π_C.*

$$T_P^{JC} : 2^{Atom_{JC}} \rightarrow 2^{Atom_{JC}}$$

$$T_P^{JC}(I) = \left\{ A' \;\middle|\; \begin{array}{l} A \leftarrow B_1, \ldots, B_n \in P \\ A' \leftarrow B'_1, \ldots, B'_n \in [A \leftarrow B_1, \ldots, B_n]_{JC} \\ \{B'_1, \ldots, B'_n\} \subseteq I \cup I_{\Pi_C} \end{array} \right\}$$

$$\mathsf{M}^{JC}[\![P]\!] = \mathsf{lfp}(T_P^{JC})$$

Here $M^{JC}[\![P]\!] \cup I_{\Pi_C}$ is the minimal D_C safe model of P w.r.t. JC.

2.3.2 Constraint-based definition of the CLP semantics

Definition 32 (Constrained atom)**.** *A constrained atom is a clause of the form $A \leftarrow c(\bar{X})$ where A is an atom containing variables $X_1, \ldots, X_n$ and $c(\bar{X})$ is a linear constraint over $X_1, \ldots, X_k$ ($k \leq n$). It is assumed that A does not include any constraint constants or function symbols as its arguments.*

The set of all constrained atoms in the language $CLP(D_C)$ is denoted by AD_C.

An example of a constrained atom is $state([X'_1, X'_2], [X_1, X_2]) \leftarrow X'_2 \geq X_2 \wedge X'_1 = X_1 - 3 \times (X'_2 - X_2)$.

For the rest of this chapter, we assume a restricted *CLP* language with no user constant and function symbols i.e. $\Sigma_U = \emptyset$. So the previous example reformulated in this restricted language will be:
$state(X_1', X_2', X_1, X_2) \leftarrow X_2' \geq X_2 \wedge X_1' = X_1 - 3 \times (X_2' - X_2)$.

A constraint is *satisfied* by an assignment of numbers to its variables if the constraint evaluates to *true* under this assignment, and is *satisfiable* if there exists some assignment that satisfies it. A constraint can be identified with the set of assignments that satisfy it. Thus a constraint over n variables represents a (possibly infinite) set of n-tuples in D_C^n.

A constrained atom $A \leftarrow c(\bar{X})$ stands for the set of all ground instances $A\theta$ where $c(\bar{X}\theta)$ is true.

We now present the CLP semantics in terms of constrained atoms. This will lead to an implementation taking advantage of constraint libraries. In particular, we use Parma Polyhedra Library (PPL) [11].

Definition 33 (Concrete Semantics)**.** *The immediate consequence operator is defined as:*

$$T_P^{\mathcal{C}} : 2^{AD_C} \rightarrow 2^{AD_C}$$

$$T_P^{\mathcal{C}}(I) = \left\{ A \leftarrow \mathcal{C} \;\middle|\; \begin{array}{l} A \leftarrow B_1, \ldots, B_n \in P \\ \{A_1 \leftarrow \mathcal{C}_1, \ldots, A_n \leftarrow \mathcal{C}_n\} \in I \\ \textit{and } \exists \textit{ substitution } \theta \textit{ such that} \\ mgu((B_1, \ldots, B_n), (A_1, \ldots, A_n)) = \theta \\ \mathcal{C}' = \bigcup\limits_{i=1,\ldots,n} \{\mathcal{C}_i\theta\} \\ \mathsf{SAT}(C') \\ \mathcal{C} = \mathsf{proj}_{Var}(A)(\mathcal{C}') \end{array} \right\}$$

$$M^{\mathcal{C}}[\![P]\!] = \mathsf{lfp}(T_P^{\mathcal{C}})$$

In the above definition, *mgu* is the most general unifier of its arguments [89]. We assume its arguments are standardised apart. $\mathsf{proj}_{\bar{X}}(C)$ returns the projection of constraint C onto the set of variables $\bar{X}$; $\mathsf{SAT}(C)$ returns true if C is satisfiable otherwise false.

This definition will be used in the later chapter as the basis for computing the minimal model of the constraint logic programs representing real time systems.

2.4 Abstract interpretation

We now define basic concepts of abstract interpretation [26], whose aim is to systematically construct approximations of the standard semantic functions. In our case, the model of a program is usually infinite. By abstract interpretation we can construct a finite approximation. This enables us to check program properties that cannot be checked in the standard model. Since we compute the standard model by fixed-point iteration, such a computation might not terminate.

We first present some basics from lattice theory and fixed-point theory.

Definition 34 (Partial order)**.** *A partial order over a set L is a binary relation* $\sqsubseteq: L \times L \rightarrow \{\mathsf{true}, \mathsf{false}\}$ *which is reflexive, transitive and anti-symmetric.*

Definition 35 (Partially ordered set)**.** *A* partially ordered set *is a set with an associated partial ordering. It is also called a* poset *and is written as $\langle L, \sqsubseteq\rangle$. An example of such a poset could be the set of natural numbers ordered by the $\leq$ relation $\langle \mathbb{N}, \leq\rangle$.*

An upper bound of a subset A of poset $\langle L, \sqsubseteq\rangle$ is an element $l \in L$ such that $\forall l_0 \in A : l_0 \sqsubseteq l$. The least upper bound (lub) l of A is an upper bound that for all upper bounds l_0 of A satisfies $l \sqsubseteq l_0$. Similarly a lower bound $l \in L$ of A is an element such that $\forall l_0 \in A : l \sqsubseteq l_0$. The greatest lower bound (glb) l of A is an element that for all lower bounds l_0 of A satisfies $l_0 \sqsubseteq l$.

The glb of a set A is denoted $\bigsqcap A$ and is in some contexts called the *meet* operator. The lub of A is denoted $\bigsqcup A$ and will sometimes be referred to as the *join* operator.

Definition 36 (Complete lattice)**.** *A poset $\langle L, \sqsubseteq\rangle$ is a* complete lattice *if every subset A of L has both a greatest lower bound (*glb*) and a least upper bound (*lub*).*

The *least element* of a complete lattice $\langle L, \sqsubseteq\rangle$ is denoted $\bot = \bigsqcap L$ and the *greatest element* is denoted $\top = \bigsqcup L$. The glb, lub, least element and greatest element can be included in the tuple describing a complete lattice, namely $\langle L, \sqsubseteq, \sqcap, \sqcup, \bot, \top\rangle$. Sometimes we abbreviate this description by $\langle L, \sqsubseteq\rangle$.

Definition 37 (Monotonic function)**.** *Given two posets $\langle L, \sqsubseteq_L\rangle$ and $\langle M, \sqsubseteq_M\rangle$, a function $f : L \rightarrow M$ is called* monotonic *if*

$$\forall l_1, l_2 \in L : l_1 \sqsubseteq_L l_2 \Rightarrow f(l_1) \sqsubseteq_M f(l_2)$$

The composition of monotonic functions is also monotonic i.e. if $f : L_1 \rightarrow L_2$ and $g : L_2 \rightarrow L_3$ are monotonic, then so is $g \circ f : L_1 \rightarrow L_3$.

Definition 38 (Fixed point)**.** *Let $f : L \rightarrow L$ be a function. Then $l \in L$ is a* fixed point *of f, if $f(l) = l$ and is denoted by $\mathit{fix}(f)$.*

Tarski's fixed point theorem [116] states the following:

Theorem 1. *Let L be a complete lattice and $f : L \to L$ be a monotonic function. Then the set of fixed points, $Fix(f) \subseteq L$, is also a complete lattice.*

Since a complete lattice cannot be empty, this guarantees the existence of at least one fixed point of f and even the existence of a least and greatest fixed point. The greatest lower bound of $Fix(f)$ is denoted $\mathsf{lfp}(f)$, *least fixed point* of f. A least upper bound of $Fix(f)$ also exists in L and is denoted by $\mathsf{gfp}(f)$, *greatest fix point* of f.

Definition 39 (Chain)**.** *For a partially ordered set L, a subset $Y \subseteq L$, is a* chain *if*

$$\forall l_1, l_2 \in Y : (l_1 \sqsubseteq l_2) \vee (l_2 \sqsubseteq l_1)$$

A sequence of elements, $\{l_n | n \in \mathbb{N}\}$, in L is an *ascending chain* if $n \leq m \Rightarrow l_n \sqsubseteq l_m$ and *descending chain* if $n \leq m \Rightarrow l_n \sqsupseteq l_m$.

Definition 40 (Ascending Chain Condition)**.** *A partially ordered set, L, satisfies the* Ascending Chain Condition *if every ascending chain $l_1 \sqsubseteq l_2 \sqsubseteq \ldots$ of elements in L is eventually stationary. The chain is stationary if there exists an $n \in \mathbb{N}$ such that $l_m = l_n$ for all $m > n$.*

Definition 41 (Continuous functions between partially ordered sets)**.** *Let L be a complete lattice, then a function $f : L \to L$ is* continuous *if for all $Y \subseteq L$, then $\bigsqcup f(Y) = f(\bigsqcup Y)$ holds.*

Note that if a function is continuous it is also monotonic. Also, when the lattice L happens to be a finite lattice a monotonic function $f : L \to L$ also is a continuous function.

Theorem 2. *For any complete lattice L and any continuous function $f : L \to L$ the $\mathsf{lfp}(f)$ is the least upper bound of the ascending Kleene chain:*

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \ldots$$

A dual characterisation of the above theorem leads to the greatest fixed point, where the chain descends beginning with the greatest element $\top$ and stabilising at the greatest fixed point.

Going back to the LP semantics function $T_P^J : 2^{\mathsf{Atom}_J} \to 2^{\mathsf{Atom}_J}$, the set 2^{Atom_J} is a complete lattice $\langle 2^{\mathsf{Atom}_J}, \subseteq, \cap, \cup, \emptyset, \mathsf{Atom}_J \rangle$ and the function itself is a continuous function. This guarantees the existence of the least fixed point and justifies its computation by constructing the Kleene sequence.

However, when the lattice does not satisfy the ascending chain condition the Kleene sequence might not stabilise in a finite number of iterations. In such cases, we resort to abstract interpretation.

In abstract interpretation we replace the so-called "concrete" semantic function by an abstract semantic function, developed systematically from the concrete semantics with respect to a *Galois connection.* We present the formal framework briefly.

Definition 42 (Galois Connection). $\langle L, \sqsubseteq_L\rangle \underset{\alpha}{\overset{\gamma}{\longleftrightarrow}} \langle M, \sqsubseteq_M\rangle$ *is a* Galois Connection *between the lattices* $\langle L, \sqsubseteq_L\rangle$ *and* $\langle M, \sqsubseteq_M\rangle$ *if and only if* $\alpha : L \to M$ *and* $\gamma : M \to L$ *are monotonic and* $\forall l \in L, m \in M, \alpha(l) \sqsubseteq_M m \leftrightarrow l \sqsubseteq_L \gamma(m)$*.*

In abstract interpretation, $\langle L, \sqsubseteq_L\rangle$ and $\langle M, \sqsubseteq_M\rangle$ are the concrete and abstract semantic domains respectively. Given a Galois connection $\langle L, \sqsubseteq_L\rangle \underset{\alpha}{\overset{\gamma}{\longleftrightarrow}} \langle M, \sqsubseteq_M\rangle$ and a monotonic concrete semantics function $f : L \to L$, we define an abstract semantic function $f^\sharp : M \to M$ such that for all $m \in M$, $(\alpha \circ f \circ \gamma)(m) \sqsubseteq_M f^\sharp(m)$. Furthermore it can be shown that $\mathsf{lfp}(f) \sqsubseteq_L \gamma(\mathsf{lfp}(f^\sharp))$ and that $\mathsf{gfp}(f) \sqsubseteq_L \gamma(\mathsf{gfp}(f^\sharp))$.

Thus the abstract function $f^\sharp$ can be used to compute over-approximations of f, which can be interpreted using the γ function. The case where the abstract semantic function is defined as $f^\sharp = (\alpha \circ f \circ \gamma)$ gives the most precise approximation.

If M is a finite-height lattice, then the non-terminating fixed point computations of $\mathsf{lfp}(f)$ and $\mathsf{gfp}(f)$ over L are approximated with a terminating fixed point computation over the finite lattice M.

2.4.1 Abstract interpretation of logic program semantics

We will now show that any pre-interpretation J induces a Galois connection $\langle \mathsf{Atom}_H, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\longleftrightarrow}} \langle \mathsf{Atom}_J, \subseteq\rangle$. What this means is that by choosing an appropriate pre-interpretation over a finite domain we can derive an abstract semantic function of a program.

We construct the $\alpha : 2^{\mathsf{Atom}_H} \to 2^{\mathsf{Atom}_J}$ and $\gamma : 2^{\mathsf{Atom}_J} \to 2^{\mathsf{Atom}_H}$ functions of the Galois connection using a standard construction as explained in [95] (p235). Given a pre-interpretation J define an *extraction function* $\eta : \mathsf{Atom}_H \to \mathsf{Atom}_J$ defined as: $\eta(A) = A'$ where $[A]_J = \{A'\}$; (since $A \in \mathsf{Atom}_H$, $\mid [A]_J \mid= 1$). Let $I \in 2^{\mathsf{Atom}_H}$; define $\alpha(I) = \{\eta(A) \mid A \in I\}$. Let $S \in 2^{\mathsf{Atom}_J}$; define $\gamma(S) = \{A \in \mathsf{Atom}_H \mid \eta(A) \in S\}$.

It can easily be seen that α and γ define a Galois connection.

$$\begin{aligned} \alpha(I) \subseteq S \ &\equiv \{\eta(A) \mid A \in I\} \subseteq S \\ &\equiv \forall A \in I : \eta(A) \in S \\ &\equiv I \subseteq \gamma(S) \end{aligned}$$

It can be proved that $\forall S : T_P^J(S) \supseteq (\alpha \circ T_P^H \circ \gamma)(S)$ and therefore $\gamma(\mathsf{lfp}(T_P^J)) \supseteq \mathsf{lfp}(T_P^H)$.

For CLP programs we can construct abstract interpretations using pre-interpretations as just explained. However, in this dissertation, we employ another class of Galois connections for CLP programs based on the well known domain of convex polyhedra [30, 57]. For our purposes, we use constraint representations of convex polyhedra; an n-dimensional polyhedron is represented by a conjunction of linear inequalities with n variables ranging over the continuous domain.

Let S be a set of points in k-dimensional space; we assume a function $\mathsf{CH}(S)$ called the convex hull of S, which is the smallest convex polyhedron containing S. This convex polyhedron in k-dimensional space can also be defined with a linear constraint c over k variables $X_1, \ldots, X_k$ corresponding to the k-dimensions respectively. In what follows, we use $\mathsf{CH}(c(\bar{X}))$ where $c(\bar{X})$ is a constraint representation of the set of points in k-dimensional space and $\mathsf{CH}(c(\bar{X}))$ is a constraint representing the convex hull of $c(\bar{X})$. The PPL library provides functions that take constraints as their arguments and return corresponding convex hulls as their results.

As is illustrated in the following example, the constraint $c(\bar{X})$ need not constrain every variable. An unconstrained variable has an unbounded range.

Consequently, an interpretation of a predicate symbol p/n over this domain is a constrained atom of the form: $p(\bar{t}) \leftarrow c(\bar{Y})$, ($\bar{Y} \subseteq \bar{X}$ where $\bar{X} = \mathit{Var}(\bar{t})$).

Example 11. *Consider the points* $(1, -2), (-2, 2), (3, 0)$ *in a* 2*-dimensional space. These points could be envisaged as a constraint* $c(\bar{X})$ *of the form:* $(X_1 = 1 \wedge X_2 = -2) \vee (X_1 = -2 \wedge X_2 = 2) \vee (X_1 = 3 \wedge X_2 = 0)$. *The smallest polyhedron, which is a triangle, enclosing these points is given by the convex hull* $\mathsf{CH}(c(\bar{X}))$ *which is* $X_1 - X_2 \leq 3 \wedge 2 * X_1 + 5 * X_2 \leq 6 \wedge 4 * X_1 + 3 * X_2 \leq -2$. *We use the PPL library to compute such convex hulls.* $\square$

The domain CP (convex polyhedral approximations) formed by the set of the sets of constrained atoms, in which each set contains at most one constrained atom per predicate, also forms a complete lattice $\langle CP, \sqsubseteq_{\mathsf{CH}}, \sqcap_{\mathsf{CH}}, \sqcup_{\mathsf{CH}}, \emptyset, \mathsf{Atom}_{D_C} \rangle$. The partial order, glb and lub operators $\sqsubseteq_{\mathsf{CH}}, \sqcap_{\mathsf{CH}}, \sqcup_{\mathsf{CH}}$ are defined as below. For simplicity, in the following, assume $\bar{t} = \bar{X}$.

Let $S_1, S_2 \in CP$ be two sets of constrained atoms. Then:

$$\begin{array}{lll}
S_1 \sqsubseteq_{\mathsf{CH}} S_2 \equiv & \forall p \in \Pi : & (p(\bar{t}) \leftarrow c_1(\bar{X})) \in S_1 \Rightarrow \\
& & \exists (p(\bar{t}) \leftarrow c_2(\bar{Y})) \in S_2 : c_1(\bar{X}) \Rightarrow c_2(\bar{Y}) \\
S_1 \sqcap_{\mathsf{CH}} S_2 = & \{p(\bar{t}) \leftarrow & \mathsf{CH}(c_1(\bar{X}) \wedge c_2(\bar{Y})) \mid p(\bar{t}) \leftarrow c_1(\bar{X}) \in S_1, \\
& & p(\bar{t}) \leftarrow c_2(\bar{Y}) \in S_2\} \\
S_1 \sqcup_{\mathsf{CH}} S_2 = & \{p(\bar{t}) \leftarrow & \mathsf{CH}(c_1(\bar{X}) \vee c_2(\bar{Y})) \mid p/n \in \Pi : p(\bar{t}) \leftarrow c_1(\bar{X}) \in S_1, \\
& & p(\bar{t}) \leftarrow c_2(\bar{Y}) \in S_2\} \\
& \bigcup & \\
& \{p(\bar{t}) \leftarrow & c_1(\bar{X}) \mid p/n \in \Pi : p(\bar{t}) \leftarrow c_1(\bar{X}) \in S_1, \\
& & p(\bar{t}) \leftarrow c_2(\bar{Y}) \notin S_2\} \\
& \bigcup & \\
& \{p(\bar{t}) \leftarrow & c_2(\bar{Y}) \mid p/n \in \Pi : p(\bar{t}) \leftarrow c_2(\bar{Y}) \in S_2, \\
& & p(\bar{t}) \leftarrow c_1(\bar{X}) \notin S_1\}
\end{array}$$

A pre-interpretation over the domain of convex polyhedra also induces a Galois connection $\langle 2^{\mathsf{Atom}_{D_C}}, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle CP, \sqsubseteq_{\mathsf{CH}} \rangle$ with abstraction function $\alpha : 2^{\mathsf{Atom}_{D_C}} \rightarrow CP$ and concretisation function $\gamma : CP \rightarrow 2^{\mathsf{Atom}_{D_C}}$ defined as:

- $\alpha(I) = \{p(X_1, \ldots, X_n) \rightarrow \mathsf{CH}(Args(I_p)) \mid p/n \in \Pi\}$ where $I_p \subseteq I$ is the set of atoms in I containing predicate p, that is, the interpretation of the n-ary predicate symbol p and $Args(I_p) = \{(t_1, \ldots, t_n) \mid p(t_1, \ldots, t_n) \in I\}$.
- $\gamma(S) = \{p(\bar{r}) \mid \exists p(\bar{X}) \rightarrow c(\bar{X}) \in S \wedge c(\bar{X}/\bar{r})\}$.

Summary

This chapter formalised the bottom-up semantics that form the basis for the static analysis of constraint logic programs. In the next chapter, we explain how to translate a linear hybrid automaton (LHA) into a constraint logic program. The minimal model of the resulting constraint logic program is computed based on the least fixed-point Algorithm 1. Whenever the computation of the concrete minimal model becomes impossible, we define an abstract domain (mapped to the concrete domain via a Galois connection) and the abstract minimal model is computed. The concrete minimal model (resp. abstract minimal model), which is a precise (resp. safe) approximation of the run-time behaviour of an LHA, can be analysed in order to verify the properties of an LHA.

Chapter 3

Formal modelling of Real-time Systems

Introduction

A software bug is an *erroneous behaviour* inadvertently implemented in software. An erroneous behaviour means either the absence of a desired functionality or the presence of a prohibited functionality. There are several factors that contribute to bugs, of which two important ones are: (i) the intricate complexity in the problem targeted by the software; (ii) complexity in implementing the problem's solutions in software. Embedded systems are no less complex than other kinds of software and arguably the presence of hybrid behaviour and real time requirements can increase complexity.

Formal Verification

The most expensive bugs to correct are those that occur earliest in the software development process. Particularly this is true of the bugs in the specification that are not detected until the software testing phase. Such bugs could be caught by formally verifying the system specification against the system requirements.

Formal verification is a technique for mathematically analysing system correctness where first the *system specification* and its *correctness properties* are formally modelled or specified in a formal language and then following a proof method the correctness of the system model with respect to the specified properties is established.

The formal languages, called specification languages, could be broadly categorised [120] into two classes: (a) *modelling languages* and (b) *property specification languages*. Examples of modelling languages used for specifying embedded systems include CSP [67], Petri nets [101], Timed Automata [6] and Hybrid Automata [59] among several others [78]. Examples of property specification languages used for specifying reactive properties include temporal logics [92], TLA [84] etc.

There are several *models of computation* [86, 1] to choose from while modelling a system. For instance, state transition systems [77, 81], Petri nets [101, 32], data flow networks [74], etc. are some models of computation. A state transition system [81] has *state variables* whose values identify its state and the *change in the variable values* identify the state transition. The state and state transition capture the state and action of the modelled reactive system; while the sequence of states generated by the state machine capture the reactive behaviour. The state transition system model can be conveniently extended [1] for modelling reactive systems.

Most of the modelling languages provide an expressive syntax relevant for specifying reactive systems and have semantics based on the state transition system model. In this dissertation, we focus on a modelling language called Linear Hybrid Automata.

Formal Modelling of Embedded Systems

There exist a wide range of formal modelling languages. In a formal model, we use mathematical structures such as sets, functions, relations, and arithmetic. Modelling a system using mathematical structures can be difficult. To make formal modelling easier, several high-level formal modelling languages for embedded systems have been developed. These modelling languages, like high-level programming languages that both abstract the target hardware-related intricacies and provide several language constructs oriented towards embedded systems, hide some of the mathematics and provide high-level expressiveness.

To choose an appropriate modelling language, we need to establish the main features and functionality of the target embedded systems that are to be modelled; this is discussed in the following section.

Chapter Overview

- Section 3.1 introduces the features of embedded systems.
- Section 3.2 introduces the language of Linear Hybrid Automata.
- Section 3.3 presents a standard scheme to translate LHA models into constraint logic programs.
- Section 3.4 explains how the constraint logic program forms the common basis to analyse the encoded LHA model.

3.1 Embedded Systems

Embedded systems are microcontroller-based systems that are embedded inside another system in order to control its operation. For instance, a car heating system contains an embedded system that decides when to turn on the heater. Embedded systems are predominantly employed as controllers in *digital control systems.*

3.1.1 Digital control systems

A *digital control system* comprises: (i) a digital computer (which is an embedded system); (ii) a sensor; (iii) a (continuous) plant and (iv) an actuator. Figure 3.1 gives a generic block diagram of a digital control system.

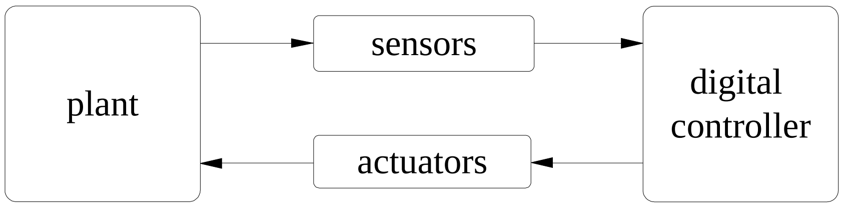

Figure 3.1: Block diagram of a digital control system [40].

The digital computer controls the state of the plant, which is a *physical process*, by executing a *control program*. A control program, from time to time, depending on the plant state, computes the necessary *control actions* to maintain the plant state within certain desired limits. The sensors measure the *observable* physical properties (like temperature, pressure, etc.) of the plant, while the actuators stimulate the plant by generating the physical signals (inputing some energy, etc.) corresponding to the computed control actions. Once stimulated, the plant state reacts according to certain continuous laws. Figure 3.2 shows a generic control program.

```
repeat forever
    read the state of the plant via sensors;
    compute the control action according to a pre-defined control law;
    imposes the computed control on the plant via actuators;
```

Figure 3.2: A control program

Thus, analysing an embedded system (digital computer) formally requires modelling not only the control program but also the plant, the sensors and the actuators. In the following, we formalise the behaviour of plants, sensors, controllers and actuators.

Example 12. *Consider the digital control system shown in Figure 3.3. The control system is expected to maintain the water level in a tank, which gets emptied at variable rate. This system has two actuators, namely the pump and the valve, and a water-level sensor. When turned on the pump delivers water into the tank at a constant rate, while the valve empties the tank at a constant rate. The sensor measures the level of water in the tank.* □

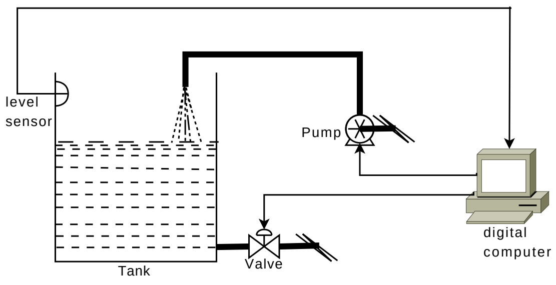

Figure 3.3: A water level control system.

3.1.2 Formal behaviour

Plant

If the plant has n variables then its behaviour is formally modelled by the *continuous function*[1] $p : T \to R^n$ ($T \in [0, +\infty]$), which is defined as:

$$p(t) = p(t_0) + \int_{t_0}^{t} \dot{p}(t)dt \tag{3.1}$$

where:

- $t \in T$ is the variable representing time;
- t_0 is the time of initialisation and $p(t_0)$ gives the plant state at instance t_0;
- $\dot{p}(t)$ is the *rate of change* in the plant state and is defined as below: $\dot{p}(t) : R^n \times T \to R^n$ is a function of plant state and time that defines the rate at which the plant state changes;

[1]Here continuous means the function (resp. behaviour) is continuous with respect to time.

It should be noted that the variable t, which represents time, takes values that range over the set T, which is a domain of *real numbers*. Since the domain of real numbers is *dense*[2], embedded systems fall under the category of *infinite state systems*.

The plant state might change either linearly or non-linearly with respect to time. Also such dynamics could be either deterministic or non-deterministic.

Deterministic Vs. Non-deterministic plant dynamics A plant can evolve either *deterministically* or *non-deterministically*. This can be explained by defining a total function $ps : T \rightarrow 2^{R^n}$ ($T \in [0, +\infty]$) that for a given time instance returns the set of states possible for a plant. A plant is deterministic if and only if $\forall t \in T : \mid ps(t) \mid = 1$; while a plant is non-deterministic if $\exists t \in T : \mid ps(t) \mid > 1$.

Linear vs. Non-linear plant dynamics Finally, a plant can evolve either *linearly* or *non-linearly* with respect to time and other variables. Since non-linear systems are not amenable to automatic verification, they are often approximated either with the sum of squares or with the linear equations [60], and are then verified. In this dissertation, we focus on linear hybrid systems only.

Sensor

A sensor measures some continuous property of a plant. Formally, such a measurement is a function accepting a time value t and property value $sp \in R$ as its input and returns a discrete value $vd_t \in Disc$, where $Disc$ is a set of discrete symbols (like integers or limited precision floating-point numbers).

Let n sensors measure n state variables. Then such a measurement is a function $sensor : T \times R^n \rightarrow Disc^n$ mapping n-dimensional dense real space into an n-dimensional finite discrete state space.

Digital computer

The digital computer executes a control program of the form shown in Figure 3.2. In more detail each step is as follows:

1. *Read the state of the plant.* The sensors are either read *continuously* or at *regular time points.* The continuous sensing is modelled by assuming that the measurements are available in the control step at all times; while in

[2]Between any two distinct real (resp. rational) numbers there exists another real (resp. rational) number.

$$
control(y_1, \ldots, y_n, t) = \begin{cases} (v_1^1, \ldots, v_m^1) & \text{if } (y_1, \ldots, y_n) \in Region_1 \\ (v_1^2, \ldots, v_m^2) & \text{if } (y_1, \ldots, y_n) \in Region_2 \\ \vdots & \\ (v_1^{cr}, \ldots, v_m^{cr}) & \text{if } (y_1, \ldots, y_n) \in Region_{cr} \end{cases}
$$

Figure 3.4: A generic control function

the regular mode the measurements are available at certain pre-determined periodic points.

2. *Compute the control action.* A control law defines the values of control variables as a function of plant state and time. Figure 3.4 gives a generic control function $control : R^n \times T \rightarrow R^m$ of a system with n state variables $(y_1, \ldots, y_n)$ and m control variables $(u_1, \ldots, u_m)$. In the Figure, $Region_i \subseteq R^n$ (where $i \in \{1, \ldots, cr\}$ and $cr \in N$) are regions in n-dimensional real space and the tuple assignments $(u_1, \ldots, u_m) := (v_1, \ldots, v_m)$ means $u_i := v_i$ (for $i \in \{1, \ldots, m\}$).

   Such control laws can be modelled with discrete automata as illustrated in Example 13.

   Deterministic vs. Non-deterministic automata. Again the control law might be either deterministic or non-deterministic. Accordingly the control program could be modelled either with a deterministic automaton or non-deterministic automaton. In a deterministic automaton, at any given time, at most one transition is enabled. In a non-deterministic automaton, there exists some time point where multiple transitions are enabled.

   Discrete vs. Continuous control action. The function $control : R^n \times T \rightarrow R^m$ (seen in Figure 3.4) is a continuous valued function. But it could well be a discrete valued function like $control : R^n \times T \rightarrow Disc^m$. Accordingly, the control action could be modelled either as a *discrete control function* or *continuous control function.*

3. *Impose the computed control.* Once the control values are computed, the actuators are driven with those values.

Actuator

An actuator accepts the computed control values, which are discrete symbols, and translates them into an equivalent *continuous* stimulus on the plant (if the plant is continuous). Such a stimulus in turn induces a change in the rate of change i.e. $\dot{p}(t)$.

Thus, if there are m control variables to monitor n state variables, then the actuators can be modelled as a function $actuator : Disc^m \rightarrow R^n$ mapping the discrete controls to the rate of change they induce; that is, the ith component of the result is the rate of change of the ith state variable for the given values of the actuators. This concept is illustrated with Example 14.

Example 13. *The embedded system (of Example 12) maintains the water level w between 5 units and 10 units by executing the control program shown in Figure 3.5. p and v are control variables corresponding to the two actuators pump and valve, respectively. If the control variable is set, then the actuator gets turned on when the control is imposed.*

```
repeat forever
    sampling: Read $w$ from the sensor;
    control law:
            if $(w \leq 6)$ then $p = 1, v = 0$;
            else if $(6 < w < 9)$ then $p = 0, v = 0$;
            else if $(w \geq 9)$ then $p = 0, v = 1$;
    Actuation: Drive the pump and valve accordingly.
```

Figure 3.5: The control sequence for the water level controller.

Thus, at any given instance, the controller will be in one of the three modes:
(a) $mode_{00}$, the mode where both pump and valve are signalled off;
(b) $mode_{10}$: the mode where pump and valve are signalled on and off, respectively; or
(c) $mode_{01}$: the mode where pump and valve are signalled off and on, respectively. Since the control program repeats the control law forever, the mode changes instantaneously depending on the plant state.

This controller is modelled with the discrete automaton shown in Figure 3.6. The value of the variable w is made available at the controller by the sensor. □

Example 14. *The system has two control variables, namely pump and valve and one state variable namely w. Depending on whether the pump and valve are turned on or off, the water level in the plant varies at different rates. While the pump and valve are turned off the water level w drops 0.1 unit per unit time i.e. $\dot{w} = -0.1$. When the pump is on and valve is off $\dot{w} = 0.5$; while pump is off and valve is on $\dot{w} = -1$.*

This actuation behaviour is given by the partial function $actuation : \{1,0\} \times \{1,0\} \rightarrow R$ defined as:
$actuation(0,0) = -0.1$
$actuation(1,0) = 0.5$

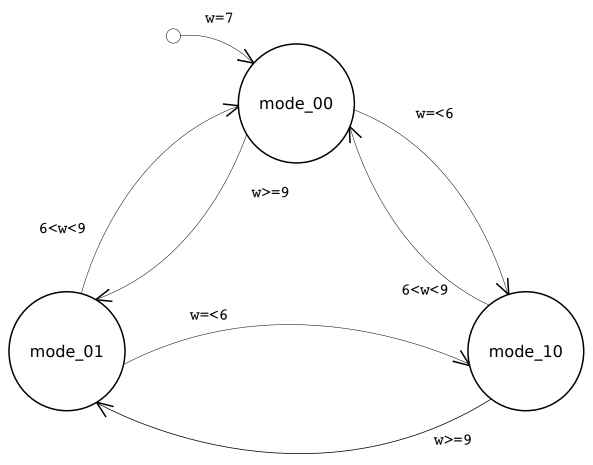

Figure 3.6: The discrete automaton modelling the control program in Example 13.

actuation$(0, 1) = -1$.

Let the system be initialised at $t_0 = 0$ *with water level* $w = 7$. *Then the water level after a lapse of 11 seconds is given by:*

$$w(11) = w(0) + \int_0^{11} \dot{w}(t)dt \qquad [\textit{By Equation } 3.1] \tag{3.2}$$

Since the pump and valve are turned on and off depending on w, *the rate of change* $\dot{w}(t)$ *changes. We can see that in the first 10 seconds the automaton stays in* $mode_{00}$ *where* $\dot{w}(t) = -0.1$ *in the last second it spends in* $mode_{10}$. *The above equation translates to:*

$$\begin{aligned} w(11) &= w(0) + \int_0^{10} (-0.1)dt + \int_{10}^{11} (0.5)dt \\ &= 7 + (-0.1) * \int_0^{10} dt + (0.5) * \int_{10}^{11} dt \\ &= 7 + (-0.1) * [t]_0^{10} + (0.5) * [t]_{10}^{11} \\ &= 7 + (-0.1) * [10 - 0] + (0.5) * [11 - 10] \\ &= 7 - 1 + 0.5 \\ &= 6.5 \end{aligned}$$

□

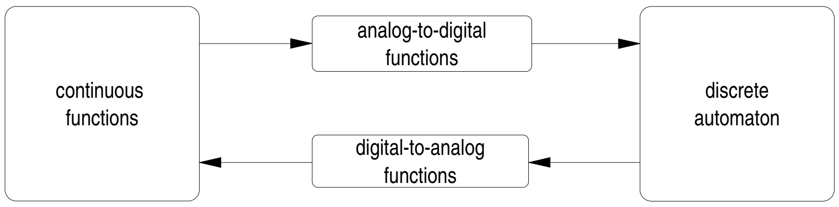

Figure 3.7: Block diagram of a hybrid model.

Thus modelling of embedded systems, on one hand, requires formal languages that allow for modelling discrete behaviour and on the other hand requires languages that allow for specifying continuous dynamics of the plant. For instance, finite state automata provide an ideal formalism for specifying discrete behaviour, while *differential equations* provide an ideal formalism for specifying continuous behaviour.

Several *hybrid modelling languages* [80] have been developed that combine both differential equations and automata formalisms. Of such modelling languages, Hybrid automata and Timed automata are the most popular ones [68].

Figure 3.7 gives a generic block diagram of a hybrid model corresponding to a discrete control system.

In this dissertation, we focus on the language of Linear Hybrid Automata as the modelling language. This language allows for modelling control systems with:

- plants that evolve *continuously*, and *linearly*;
- control programs that *continuously* read the sensors and *non-deterministically* compute the (discrete) control actions.

Initially the language was intended to model deterministic plants and control programs, but were later extended to model non-deterministic ones also. We do not model the sensors and actuators explicitly. Their functionality is incorporated in the control automaton.

3.2 Linear Hybrid Automata

3.2.1 The language of Linear Hybrid Automata

Following [59], we formally define a linear hybrid automaton as a 6-tuple $\langle \texttt{Loc}, \texttt{Trans}, \texttt{Var}, \texttt{Init}, \texttt{Inv}, \mathcal{D} \rangle$, with:

- A finite set `Loc` of locations also called control nodes, corresponding to control modes of a controller/plant.
- A finite set $\texttt{Var} = \{x_1, x_2, \ldots, x_n\}$ of real valued variables, where n is the number of variables in the system. The state of the automaton is a tuple (l, X), where X is the valuation vector in $\mathbb{R}^n$, giving the value of each variable. Associated with variables are two sets:
  - $\dot{\texttt{Var}} = \{\dot{x}_1, \ldots, \dot{x}_n\}$, where $\dot{x}_i$ represents the first derivative of variable x_i w.r.t time;
  - $\texttt{Var}' = \{x'_1, \ldots, x'_n\}$, where x'_i represents x_i at the end of a transition.
- Three functions `Init`, `Inv` and $\mathcal{D}$ that assign to each location $l \in \texttt{Loc}$ three predicates respectively: $\texttt{Init}(l)$, $\texttt{Inv}(l)$ and $\mathcal{D}(l)$. The free variables of $\texttt{Init}(l)$ and $\texttt{Inv}(l)$ range over `Var`, while those of $\mathcal{D}(l)$ range over $\texttt{Var} \cup \dot{\texttt{Var}}$. An automaton can start in a particular location l only if $\texttt{Init}(l)$ holds. So long as it stays in the location l, the system variables evolve as constrained by the predicate $\mathcal{D}(l)$ not violating the invariant $\texttt{Inv}(l)$. The predicate $\mathcal{D}(l)$ constrains the rate of change of system variables.
- A set of discrete transitions $\texttt{Trans} = \{\tau_1, \ldots, \tau_t\}$; $\tau_k = \langle k, l, \gamma_k, \alpha_k, l' \rangle$ is a transition
  - uniquely identified by integer k, $0 \leq k \leq t$;
  - corresponding to a discrete jump from location l to location l'; and
  - guarded by the predicate γ_k and with actions constrained by the predicate α_k.

  In the above, the symbols γ_k and α_k denote the 3-ary *guard predicate* $\gamma(k, l, l')$ and the 4-ary action predicate $\alpha(k, l, (l', X), (l', X'))$, respectively, where $X \subseteq \texttt{Var}$ and $X' \subseteq \texttt{Var}'$.

In the following example (Example 15), we illustrate the LHA model of a water-level control system. This system is an adapted version of the water-level control system defined in [57].

Example 15. *Consider a water-level monitoring system similar to the one shown in Figure 3.3 with pump as the actuator and level-sensor as the sensor. There is a constant discharge of water from the tank and the system is expected to maintain the water-level between* 1 *unit-height and* 12 *unit-height by turning the pump ON or OFF from time to time. The reaction time of the pump is 2 time-units i.e. to switch from ON to OFF (or OFF to ON) mode it takes 2 time-units. The LHA model of this system is shown in Figure 3.8.*

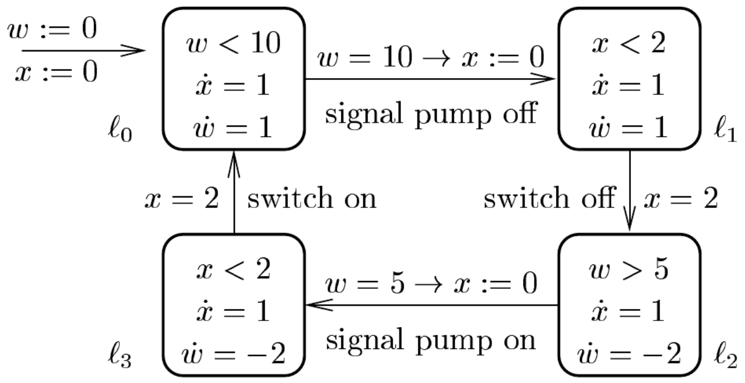

Figure 3.8: An LHA model of the water-level monitor.

This graphical model when expressed in the LHA language (defined previously) looks like:

1. $\texttt{Loc} = \{l_0, l_1, l_2, l_3\}$; *That is there are four locations in the LHA identified by* l_0, l_1, l_2 *and* l_3.

2. $\texttt{Init}(l_0) : w = 0, x = 0$, $\texttt{Init}(l_1) : \mathsf{false}$, $\texttt{Init}(l_2) : \mathsf{false}$ *and* $\texttt{Init}(l_3) : \mathsf{false}$; *That is* l_0 *is the initial location.*

3. $\texttt{Var} = \{w, x\}$ *and* $\dot{\texttt{Var}} = \{\dot{w}, \dot{x}\}$;
   The variable w *corresponds to the water level in the tank, while variable* x *records the time lapsed in a location.*

4. $\texttt{Inv}(l_0) : w < 10$, $\texttt{Inv}(l_1) : x < 2$, $\texttt{Inv}(l_2) : w > 5$ *and* $\texttt{Inv}(l_3) : x < 2$;

5. $\mathcal{D}(l_0) : \dot{w} = 1\ \dot{x} = 1$, $\mathcal{D}(l_1) : \dot{w} = 1\ \dot{x} = 1$, $\mathcal{D}(l_2) : \dot{w} = -2, \dot{x} = 1$ *and* $\mathcal{D}(l_3) : \dot{w} = -2, \dot{x} = 1$;

6. $\texttt{Trans} = \{\tau_0, \tau_1, \tau_2, \tau_3\}$ *where*

   - $\tau_1 = \langle 1, l_0, \gamma_1, \alpha_1, l_1 \rangle$ *where* $\gamma_1 : w = 10$ *and* $\alpha_1 : x := 0$;
   - $\tau_2 = \langle 2, l_1, \gamma_2, \alpha_2, l_2 \rangle$ *where* $\gamma_2 : x = 2$ *and* $\alpha_2 :$;
   - $\tau_3 = \langle 3, l_2, \gamma_3, \alpha_3, l_3 \rangle$ *where* $\gamma_3 : w = 5$ *and* $\alpha_3 : x := 0$
   - $\tau_4 = \langle 4, l_3, \gamma_4, \alpha_4, l_0 \rangle$ *where* $\gamma_4 : x = 2$ *and* $\alpha_4 :$
     That is there are four discrete transitions τ_1, τ_2, τ_3 *and* τ_4 *from* l_0 *to* l_1, l_1 *to* l_2, l_2 *to* l_3 *and* l_3 *to* l_0 *respectively. In the above, since* τ_2 *and* τ_4 *have no actions hence* α_2 *and* α_4 *are empty.*

□

3.2.2 LHA Semantics as a Transition System

At any time instance, an LHA has a state (l, X) defined by the control location ($l \in \texttt{Loc}$) and the state of its variables ($X \in \mathbb{R}^n$) at that time point. With the passage of time either the location or variables might change thus resulting in a overall state transition. Such state transition semantics of LHAs can be formally described as consisting of *runs* of a *labelled transition system.*

Definition 43. *A labelled transition system is a 4-tuple $\langle S, S_0, A, \xrightarrow{a} \rangle$, $a \in A$ with:*

- *A possibly infinite set $S \subseteq \texttt{Loc} \times \mathbb{R}^n$, called the* state space *defined as:* $S = \{(l, X) \mid l \in \texttt{Loc}, X \in \mathbb{R}^n : X \models \texttt{Inv}(l)\}$*, which is the set of all* $(l, v_1, \ldots, v_n) \in \texttt{Loc} \times \mathbb{R}^n$ *where predicate* $\texttt{Inv}(l)$ *holds at valuation* X *i.e.* $[\overline{x} := \overline{v}]$*,* $\overline{x}$ *means* $(x_1, \ldots, x_n)$ *while* $\overline{v} = (v_1, \ldots, v_n)$*.*

- *A non-empty set $S_0 \subseteq S$, called initial states of LHA, defined as:* $S_0 = \{(l, X) \mid (l, X) \models \texttt{Init}(l)\}$*. It is a set of states satisfying the initialisation predicate.*

- *A possibly infinite set $A = \mathbb{R}_{\geq 0}$ of positive reals, over which the time durations range.*

- *A binary* transition relation $\xrightarrow{a}$ *on the state space S.*

The transitions are of two kinds: *delay transitions* and *discrete transitions.* A delay transition corresponds to the state change that occurs because of the passage of time staying in the same location; while discrete transition corresponds to the state change that occurs because of the change of the location.

Definition 44 (delay transition)**.** *A* delay transition *is defined as:* $(l, X) \xrightarrow{\delta} (l^\delta, X^\delta)$ *iff* $l = l^\delta$*, where*

- $\delta \in \mathbb{R}_{\geq 0}$ *is the duration of time passed* staying *in the location l, during which the predicate* $\texttt{Inv}(l)$ *continuously holds;*

- *X and X^δ are the variable valuations in l such that $\mathcal{D}(l)$ and* $\texttt{Inv}(l)$*, the predicates on location l, hold. The predicate $\mathcal{D}(l)$ constrains the variable derivatives* $\dot{\texttt{Var}}$ *such that* $X^\delta = X + \delta * \dot{X}$*.*

Definition 45 (discrete transition)**.** *A* discrete transition *is defined by* $(l, X) \rightarrow (l', X')$*, where*

- *there exists a transition* $\tau = \langle k, l, \gamma, \alpha, l' \rangle \in \texttt{Trans}$ *identified by k;*

- *the guard predicate $\gamma(k, l, l')$ holds at the valuation X in location l and k identifies the guarded transition;*

- *the associated action predicate $\alpha(k, l, (l', X), (l', X'))$ holds at valuation X' with which the transition ends entering the new location l' (k identifies the transition that triggers the action).*

If there are events in the system, then such events are raised by the discrete transition. In that case, the discrete transition is labelled with an associated event, which is raised along that transition.

Run

A *run* $\sigma = s_0 s_1 s_2 \cdots$ is an infinite sequence of *states* $(l, X) \in \texttt{Loc} \times \mathbb{R}^n$, where l is the location and X is the valuation. In a run σ, the transition from state s_i to state s_{i+1} are related by either a delay transition or a discrete transition. As the domain of time is dense, the number of states possible via delay transitions becomes infinite following the infinitely fine granularity of time. Hence the delay transitions and their derived states are abstracted by the duration of time (δ) spent in a location. Thus a run σ of an *LHA* is defined as:

$$\sigma = (l_0, X_0, t_0) \rightarrow^{\delta_0}_{(\gamma_0, \alpha_0)} (l_1, X_1, t_1) \rightarrow^{\delta_1}_{(\gamma_1, \alpha_1)} (l_2, X_2, t_2) \cdots$$

where:

1. t_0 is the initialisation instance and t_i (for $i \geq 1$) are the instances at which the γ_{i-1} become true and the location l_i is entered by applying the action α_{i-1} ;
2. $\delta_j = t_{j+1} - t_j$ ($j \geq 0$) is the duration of time spent in l_j;

In the above run, the new state (l_{j+1}, X_{j+1}) is entered with valuation X_{j+1} as constrained by α_j. Further $\tau_j = \langle j, l_j, \gamma_j, \alpha_j, l_{j+1} \rangle \in \texttt{Trans}$. Again during this time duration δ_j, the defined invariant $\texttt{Inv}(l_j)$ on l_j continues to hold, and the invariant $\texttt{Inv}(l_{j+1})$ holds at valuation X_{j+1}. Most importantly, between (l_i, X_i, t_i) and $(l_{i+1}, X_{i+1}, t_{i+1})$, if $t_{i+1} \neq t_i$ then there is an infinite path connecting the states that evolve continuously with time according to $\mathcal{D}(l_i)$. Finally, whenever there is a non-deterministic discrete transition, the above run branches out accordingly.

The following example (Example 16) illustrates the semantics of an LHA model.

Example 16. *Consider the LHA model from the previous example (Example 15). The interpretation of this LHA model is as follows:*

In locations l_0 and l_1, water-level increases[3] *at the rate of 1 unit/unit-time i.e. $\dot{w} = 1$. In the locations l_2 and l_3, the water-level decreases*[4] *at the rate of*

[3]The locations l_0, l_1 correspond to that mode of the controller where the actuator "pump" is turned ON and the actuator "valve" is turned OFF.

[4]The locations l_2 and l_3 correspond to that mode of the controller where the valve is turned ON and the pump is turned OFF.

1 unit/unit time i.e. $\dot{w} = -1$. In all the locations, the clock increases at the rate of 1*unit/ unit-time.*

The automaton is initialised in location l_0 with $w := 0$ and $x := 0$. The control stays in a location as long as the invariant holds. Once in a location l_k, the water level varies following $\mathcal{D}(l_k)$. When the invariant (of a location l_k) is violated and the guard associated on the transition outgoing from that location (l_k) holds, the control shifts to the new location. This continues for ever.

Every location has an invariant. They are $w < 10$, $x < 2$, $w > 5$ and $x < 2$ on locations l_0, l_1, l_2 and l_3 respectively. □

3.3 Translation of LHA into CLP

Table 3.1 shows a scheme for translating an LHA specification into CLP clauses. In what follows, each row of this table is explained.

The first row in the table shows how the state is modelled in CLP. We add an explicit "time stamp" to a state, extending $\texttt{Var}, \texttt{Var}'$ with time variables t, t' respectively giving $\texttt{Var}_t, \texttt{Var}'_t$. We model a state (along with its time stamp) as a list, whose last element gives the time stamp of the state and with location as its first element.

The predicates (`timeOf`/2 and `locOf`/2) defined in the second row are used to extract particular information about a given state. Namely, given a state S, `timeOf`(S, T) gives its time stamp T, which is the last element of the list modelling the state S (see the first row of the table); while `locOf`(S, L) returns its location L, which is the first element in the list modelling the state S. The other predicate `before`$(S, S1)$, which defines the temporal order, relates the two states $S, S1$ if S has a time stamp less than $S1$.

Example 17. *Consider an LHA with two locations i.e. $Loc = \{l1, l2\}$ and two state variables $\texttt{Var} = \{x_1, x_2\}$ whose values range over the real number domain. Then its state is modelled with a list $[L, X1, X2, T]$, where L corresponds to the LHA location; $X1, X2$ model the variables and T models the state's time stamp. Let $l1$ be the location of initialisation where the variables are initialised to $(x_1, x_2) = (0, 0)$. Then its initial state $S = [l1, 0, 0, 0]$ (the last variable corresponds to time). Now the following is the illustration of `timeOf/2`, `locOf/2` and `before/2`:*

$$\begin{array}{l}\texttt{timeOf}([l1, 0, 0, 0], 0).\\ \texttt{locOf}([l1, 0, 0, 0], l1).\\ \texttt{before}([l1, 0, 0, 0], [\_, \_, \_, T1]) \leftarrow T1 > 0.\end{array}$$

□

LHA	*CLP*

location l
state variables $x_1, \ldots, x_n$
state with time t and location l
	`L`
	`X1,...,Xn`
	`S = [L,X1,...,Xn,T]`

state time
state location
temporal order on states
	`timeOf(S,T)` $\leftarrow$ `lastElementOf(S,T).`
	`locOf(S,L)` $\leftarrow$ `S = [L|_].`
	`before(S,S1)` $\leftarrow$ `timeOf(S,T),`
	`timeOf(S1,T1),`
	`T<T1.`

`Init`(l)	`init(S)` $\leftarrow$ `locOf(S,L),`
	`to_clp(Init(`l`)).`

`Inv`(l)	`inv(L,S)` $\leftarrow$ `locOf(S,L),`
	`to_clp(Inv(`l`)).`

$D(l)$ (using the
derivative relation $D_t(l)$
explained in the text)
	`d(S,S1)`$\leftarrow$ `locOf(S,L),`
	`timeOf(S,T),`
	`locOf(S1,L),`
	`timeOf(S1,T1),`
	`to_clp(`$D_t(l)$`).`

LHA transition $\langle k, l, \gamma_k, \alpha_k, l' \rangle$
γ_k
 	
α_k
	
	`gamma(K,L,S)` $\leftarrow$ `locOf(S,L1),`
	`to_clp(`γ_k`).`
	`alpha(K,L,S1,S2)` $\leftarrow$ `locOf(S1,L1),`
	`locOf(S2,L1),`
	`to_clp(`α_k`).`

delay transition
 	
discrete transition
	`delaytransition(S0,S1)` $\leftarrow$
	`locOf(S0,L0),`
	`before(S0,S1),`
	`d(S0,S1),`
	`inv(L0,S1).`
	`discretetransition(S0,S2)`$\leftarrow$
	`locOf(S0,L0),`
	`before(S0,S1),`
	`d(S0,S1),`
	`gamma(K,L0,S1)`
	`alpha(K,L0,S1,S2).`

transition	`transition(S0,S2)` $\leftarrow$
	`discretetransition(S0,S1),`
	`delaytransition(S1,S2).`

Table 3.1: Translation of LHAs to CLP

The translation of LHAs is direct apart from the handling of the constraints on the derivatives on location l, namely $\texttt{D}(l)$ which is a conjunction of linear constraints on $\dot{\texttt{Var}}$.

The predicates $\texttt{Init}(l)$ and $\texttt{Inv}(l)$ being linear constraints (over state variables) are represented, following [70], as a CLP conjunction via `to_clp(.)`. `to_clp(.)` means the translation of the linear constraints in the corresponding LHA predicates into a CLP conjunction. For instance, in the previous example, $l1$ being the initial location, in the LHA, $\texttt{Init}(l1)$ is a linear constraint encoding the initial states of the state variables; it is of the form: $x_1 = 0 \wedge x_2 = 0$.
So $\mathit{to\_clp}(\texttt{Init}(l1))$ means a constraint of the form $X1 = 0 \wedge X2 = 0$.

The predicate $D(l)$ in LHA specifies the rate at which the state variables change with respect to time. This requires keeping track of the time lapsed using the time stamps of the states. The constraint $D_t(l)$ is a conjunction of linear constraints on $\texttt{Var}_t \cup \texttt{Var}'_t$, obtained by replacing each occurrence of $\dot{x_j}$.in $D(l)$ by $(x'_j - x_j)/(t' - t)$ in $D_t(l)$, where t', t represent the time stamps associated with x'_j, x_j respectively. In LHAs, the state variables change at a constant rate with respect to time. So, $\dot{x_i} = c$ (where $c \in R$) means a constraint $(x^{'}_i - x_i)/(t^{'} - t) = c$ where $x^{'}_i, x_i$ are the values of the variable at times $t^{'}, t$ respectively.

Discrete transitions, i.e. $\tau_k = \langle k, l, \gamma_k, \alpha_k, l' \rangle$ (where k is the id of a transition), translate to three CLP predicates:

1. $\texttt{gamma}(K, L, S)$ modelling the guard γ associated with a transition τ and is defined as:

$$\begin{array}{l}\texttt{gamma}(K, L, S) \leftarrow \\ \qquad \texttt{locOf}(S, L1), \\ \qquad \texttt{to\_clp}(\gamma_k).\end{array}$$

   where:

   - the argument K models the identifier τ_k;
   - the arguments $L, L1$ model locations of the transition predecessor state and successor state, respectively;
   - the argument S is the list $[L, X1, \ldots, Xn, T]$ modelling the state;
   - the linear constraint corresponding to the guard γ_k is represented by the CLP conjunction via $\texttt{to\_clp}(\gamma_k)$.

   This predicate evaluates to true, whenever τ_k can fire in state S.

2. $\texttt{alpha}(K, L, S1, S2)$ modelling the action (α_k) associated with a discrete

transition and is defined as:

$$
\begin{aligned}
\texttt{alpha}(K, L, S1, S2) \leftarrow &\\
&\texttt{locOf}(S1, L1),\\
&\texttt{locOf}(S2, L1),\\
&\texttt{to\_clp}(\alpha_k).
\end{aligned}
$$

where:

- K models to the transition id τ_k;
- $S1, S2$ are lists modelling the states before and after the action;
- $L, L1$ models the locations of transition predecessor and successor;
- the assignments in an action translate to a CLP conjunction via $\texttt{to\_clp}(\alpha_k)$.

3. $\texttt{discretetransition}(S0, S2)$ modelling the discrete transition is defined as:

$$
\begin{aligned}
\texttt{discretetransition}(S0, S2) \leftarrow\ & \texttt{locOf}(S0, L0),\\
& \texttt{before}(S0, S1),\\
& \texttt{d}(S0, S1),\\
& \texttt{gamma}(K, L0, S1),\\
& \texttt{alpha}(K, L0, S1, S2).
\end{aligned}
$$

   where the arguments $S0, S2$ are the respective states before and after the transition, $\texttt{d}/2$ is the CLP translation of $D(l)$ as explained earlier.

What this transition predicate says is “there is a discrete transition from $S0$ to $S2$ such that: (i) there is a state $S1$ after $S0$ where the guard gets enabled, (ii) then the state $S2$ is entered by applying the action associated with the transition”.

Similarly a delay transition translates to the CLP predicate $\texttt{delaytransition}/2$ defined as:

$$
\begin{aligned}
\texttt{delaytransition}(S0, S1) \leftarrow\ & locOf(S0, L0),\\
& \texttt{before}(S0, S1),\\
& \texttt{d}(S0, S1),\\
& \texttt{inv}(L0, S1).
\end{aligned}
$$

where $S0, S1$ are the predecessor and successor states respectively. What this transition predicate says is “the LHA at a state $S0$ evolves to state $S1$ over a passage of time during which both the variables evolve as specified by $\texttt{d}(S0, S1)$ and the invariant specified by $\texttt{inv}(L0, S1)$ holds”.

Instead of differentiating between a delay and a discrete transition, we define a new transition by integrating both kinds of transitions. This transition is any discrete transition that is followed by a delay transition. This transition is defined by the predicate `transition/2`, which is then used in the CLP translation of the labelled state transition system (Definition 43), which defines the semantics of an LHA (Figure 3.9).

3.3.1 Formal translation of an LHA into a CLP program

In the following, we formalise the translation scheme presented earlier.

An $LHA = \langle \texttt{Loc}, \texttt{Trans}, \texttt{Var}, \texttt{Init}, \texttt{Inv}, \mathcal{D} \rangle$ is translated into the CLP program $CLP = C_0 \cup C_1 \cup C_{\texttt{Init}} \cup C_{\texttt{Inv}} \cup C_{\mathcal{D}} \cup C_{\texttt{Tr}}$ where C_0, C_1, $C_{\texttt{Init}}$, $C_{\texttt{Inv}}$, $C_{\mathcal{D}}$ and $C_{\texttt{Tr}}$ are CLP clauses defined as below.

- $C_0 = \{$
  $(\texttt{timeOf}(S,T) \leftarrow \texttt{lastElelementOf}(S,T)),$
  $(\texttt{locOf}(S,L) \leftarrow [L|\_]),$
  $(\texttt{before}(S,S1) \leftarrow (\texttt{timeOf}(S,T), \texttt{timeOf}(S1,T1), T \leq T1))$
  $\};$

- $C_1 = \{$
  $(\texttt{discretetransition}(S0,S2) \leftarrow \texttt{locOf}(S0,L0),$
  $\texttt{before}(S0,S1),$
  $\texttt{d}(S0,S1),$
  $\texttt{gamma}(K,L0,S1),$
  $\texttt{alpha}(K,L0,S1,S2)),$
  $(\texttt{delaytransition}(S0,S1) \leftarrow \texttt{locOf}(S0,L0),$
  $\texttt{before}(S0,S1),$
  $\texttt{d}(S0,S1),$
  $\texttt{inv}(L0,S1))$
  $\};$

- $C_{\texttt{Init}} = \{$
  $(\texttt{init}(S) \leftarrow$
  $\texttt{state}(S),$
  $\texttt{locOf}(S,L),$
  $\texttt{toclp}(\texttt{Init}(L), S, \texttt{Var}))|L \in \texttt{Loc}\}$

- $C_{\texttt{Inv}} = \{$
  $(\texttt{inv}(L,S) \leftarrow$
  $\texttt{state}(S),$
  $\texttt{locOf}(S,L),$
  $\texttt{toclp}(\texttt{Inv}(L), S, \texttt{Var}))|L \in \texttt{Loc}\}$

- $C_{\mathcal{D}} = \{$ $(\texttt{d}(S,S1) \leftarrow \texttt{state}(S),$
  $\texttt{state}(S1),$
  $\texttt{locOf}(S,L),$
  $\texttt{locOf}(S1,L), \texttt{timeOf}(S,T),$
  $\texttt{timeOf}(S1,T),$
  $\texttt{toclp}(D_t(L), S, S1, \texttt{Var}, \texttt{Var}'))|L \in Loc\}$

- $C_{\texttt{Tr}} = \{(\texttt{alpha}(K, L, S1, S2) \leftarrow \texttt{state}(S1),$
  $\texttt{state}(S2),$
  $\texttt{locOf}(S1, L1),$
  $\texttt{locOf}(S2, L2),$
  $\texttt{toclp}(\alpha_K, S1, S2, \texttt{Var}))|\tau_K \in \mathit{Trans}\}$
  $\cup$
  $\{(\texttt{gamma}(K, L, S) \leftarrow \texttt{state}(S),$
  $\texttt{locOf}(S, L),$
  $\texttt{toclp}(\gamma_K, S, \texttt{Var}))|\tau_K \in \mathit{Trans}\}$

In the above clauses, the symbols `toclp`/3, `toclp`/4 and `toclp`/5 are functions that accept constraint expressions whose free variables are from `Var` and `Var`′ and return their equivalent expressions where each occurrence of state variable $v \in \texttt{Var}$ that is in lower case is replaced with a logic variable that is in upper case. The `toclp` function is defined as below. In the following: (i) the arguments $E1$ and $E2$ appearing in `toclp` functions are linear arithmetic expressions over the variables $\texttt{Var} \cup \texttt{Var}'$; (ii) the arguments S is a list of $n + 2$ logic variables (where n is the number of elements in `Var`) as explained in the previous section.

-
  - $\texttt{toclp}(E1 < E2, S, \texttt{Var}) = (\texttt{toclp}(E1, S, \texttt{Var}) < \texttt{toclp}(E2, S, \texttt{Var}))$.
  - $\texttt{toclp}(E1 \leq E2, S, \texttt{Var}) = (\texttt{toclp}(E1, S, \texttt{Var}) \leq \texttt{toclp}(E2, S, \texttt{Var}))$.
  - $\texttt{toclp}(E1 > E2, S, \texttt{Var}) = (\texttt{toclp}(E1, S, \texttt{Var}) > \texttt{toclp}(E2, S, \texttt{Var}))$.
  - $\texttt{toclp}(E1 \leq E2, S, \texttt{Var}) = (\texttt{toclp}(E1, S, \texttt{Var}) \leq \texttt{toclp}(E2, S, \texttt{Var}))$.
  - $\texttt{toclp}(E1 + E2, S, \texttt{Var}) = (\texttt{toclp}(E1, S, \texttt{Var}) + \texttt{toclp}(E2, S, \texttt{Var}))$.
  - $\texttt{toclp}(E1 - E2, S, \texttt{Var}) = (\texttt{toclp}(E1, S, \texttt{Var}) - \texttt{toclp}(E2, S, \texttt{Var}))$.
  - $\texttt{toclp}(E1 = E2, S, \texttt{Var}) = (\texttt{toclp}(E1, S, \texttt{Var}) = \texttt{toclp}(E2, S, \texttt{Var}))$.
  - $\texttt{toclp}(n, S, \texttt{Var}) = n$. (where n numeric constant)
  - $\texttt{toclp}(x, S, \texttt{Var}) = X$, where `lookup(x,Var,S) = X` The `lookup`/3 returns the logic variable X modelling the variable $x \in \texttt{Var}$. The variables $x_1, \ldots, x_n$ are mapped to the logic variables $X_1, \ldots, X_n$. Let $S = [L, X_1, \ldots, X_n, T]$ and $\texttt{Var} = [x_1, \ldots, x_n]$, then $\texttt{lookup}(x_i, \texttt{Var}, S)$ returns the $(i + 1)^{th}$ element of S i.e. X_i.

-
  - $\texttt{toclp}(E1 < E2, S, S1, \texttt{Var}, \texttt{Var}') =$
    $(\texttt{toclp}(E1, S, S1, \texttt{Var}, \texttt{Var}') < \texttt{toclp}(E2, S, S1, \texttt{Var}, \texttt{Var}'))$
  - $\texttt{toclp}(E1 \leq E2, S, S1, \texttt{Var}, \texttt{Var}') =$
    $(\texttt{toclp}(E1, S, S1, \texttt{Var}, \texttt{Var}') \leq \texttt{toclp}(E2, S, S1, \texttt{Var}, \texttt{Var}'))$
  - $\texttt{toclp}(E1 > E2, S, S1, \texttt{Var}, \texttt{Var}') =$
    $(\texttt{toclp}(E1, S, S1, \texttt{Var}, \texttt{Var}') > \texttt{toclp}(E2, S, S1, \texttt{Var}, \texttt{Var}'))$

- $\texttt{toclp}(E1 \geq E2, S, S1, \texttt{Var}, \texttt{Var}') =$
  $(\texttt{toclp}(E1, S, S1, \texttt{Var}, \texttt{Var}') \geq \texttt{toclp}(E2, S, S1, \texttt{Var}, \texttt{Var}'))$
- $\texttt{toclp}(E1 = E2, S, S1, \texttt{Var}, \texttt{Var}') =$
  $(\texttt{toclp}(E1, S, S1, \texttt{Var}, \texttt{Var}') = \texttt{toclp}(E2, S, S1, \texttt{Var}, \texttt{Var}'))$
- $\texttt{toclp}(x, S, S1, \texttt{Var}, \texttt{Var}') = X$, where $\texttt{lookup}(x, \texttt{Var}, S) = X$
- $\texttt{toclp}(x', S, S1, \texttt{Var}, \texttt{Var}') = X$, where $\texttt{lookup}(x', \texttt{Var}', S1) = X$

- $\texttt{toclp}(E1 := E2, S1, S2, \texttt{Var}) = (\texttt{toclp}(E1, S2, \texttt{Var}) = \texttt{toclp}(E2, S1, \texttt{Var}))$

3.3.2 What does the `transition`/2 predicate model?

With the above defined `transition`/2 predicate it is not possible to exactly model the continuous dynamics of the modelled LHA. The resulting CLP model represents a system having the same dynamics as the original LHA but every (linear) run of the LHA modelled gets *abstracted* with an infinite set of *pseudo-continuous* runs.

Recall that the run of an LHA is of the form:
$\sigma = (l_0, X_0, t_0) \rightarrow^{\delta_0}_{(\gamma_0, \alpha_0)} (l_1, X_1, t_1) \rightarrow^{\delta_1}_{(\gamma_1, \alpha_1)} (l_2, X_2, t_2) \cdots$.

For any delay $\delta_k > 0$ after a state (l_k, X_k, t_k) the system state evolves *continuously* resulting in the set of states:
$\{(l_k, X_{k,0}, t_{k,0}), \ldots, (l_k, X_{k,i}, t_{k,i}), \ldots, (l_k, X_{k,\infty}, t_{k,\infty})\}$
where $(l_0, X_{k,0}, t_{k,0}) = (l_k, X_k, t_k)$ and the last element is the state when the guard γ_k holds firing the discrete transition into $(l_{k+1}, X_{k+1}, t_{k+1})$ and $t_{k,\infty} = t_{k+1}$. This set is a *dense set*. The run corresponding to the delay following (l_k, X_k, t_k) is defined by a trajectory formed by the temporally ordered states from that dense set.

On this run for any state $(l_k, X_{k,i}, t_{k,i})$ if $t_k < t_{k,i} \leq t_{k+1}$ there is a dense set of predecessors and successors within the same location. But according to our transition predicate, there is no transition predecessor or successor to any state within the same location. Because of this abstraction, the run of an LHA is modelled by an infinite set of runs of the form:
$(l_0, X_0, t_0) \overset{\delta_{0,1}}{\rightarrow} (l_1, X_{1,2}, t_{1,2}) \overset{\delta_{1,2}}{\rightarrow} (l_2, X_{2,3}, t_{2,3}) \cdots$ where $\delta_{i,i+1} \in [\delta_i, \delta_i + \delta_{i+1}]$.

What this means is, *there is at least one run in our model* that does not contain a state (corresponding to a location l_i) that is always visited at some time instance $t_i + \delta$ where $\delta \in [\delta_i, \delta_i + \delta_{i+1}]$. Consequently our CLP model does not preserve *all kinds* of universal liveness and existential safety properties. Nevertheless, as will be explained later, universal liveness properties of a certain kind are preserved by the CLP model. Besides these all kinds of existential liveness and the universal safety properties are preserved by our model. Various kinds of safety and liveness properties are presented in Chapter 4. In the following two paragraphs, we make use of the concepts that are introduced in the latter chapters.

In the temporal language of CTL, state universal liveness and state existential safety are specified with the formulas *AFp* and *EGp*, respectively, where p is an atomic proposition. In brief, the CTL formula *AFp* states that "along all runs of a system, in future, a state where proposition p holds is visited". Similarly, the formula *EGp* states that "there exists a run, where on all states, proposition p holds". Usually these properties are proved by checking that there does not exist any counter-examples to the stated properties.

The counter-example for the property *AFp* is any run where $EG\neg p$ holds. If the proposition p holds exactly at one time point during a delay in some location, then as explained earlier there is at least one run where such time point is not visited in our model run. This translates to presence of a run where $\neg p$ holds forever. So if the proposition p holds only at one time point during the delay in some location, then the properties of the form *AFp* are not preserved. However, if the proposition p happens to be an invariant for the entire delay period (in some location), then we do preserve *AFp* properties. When we project this condition on to the modelled LHA, what it turns out to be is that "if the proposition p is *entailed* by the invariants on the locations of the LHA, then *AFp* is preserved". If p entails the invariant, then we can refine the locations of the LHA model by partitioning its locations in a standard way such that we get a refined location whose invariant entails p. Such a refinement of the LHA model ensures that the consequent CLP model preserves the universal liveness. The refined LHA in all respects has the same dynamics as the original LHA.

Since the property *EGp* is a negation of $AF\neg p$, the same refinement meant for preserving *AFp* also preserves this property. The Algorithm 2 outlines a procedure to check whether a given CTL formula ϕ is preserved or not. If the algorithm returns `true` then the property is preserved else it is not. This algorithm should be input with: (a) the invariants on every location in the LHA; (b) the propositions that are within the *immediate scope* of *AF* and *EG* operators. For a given CTL formula, a proposition (or a boolean combination of two or more propositions) is said to be in the immediate scope of a *AF* or *EG* if and only if it is the innermost CTL operator quantifying the proposition. In the algorithm: n_l is number of locations in the LHA; n_p is the number of propositions in the formula that are in the immediate scope of *AF* and/or *EG*; $\mathtt{inv}_i$ is the invariant of the states visited while in location l_i (for $i = 0$ to $n_l - 1$); and p_j (for $j = 0$ to $n_p - 1$) is a proposition.

Algorithm 2 Algorithm to check the property preservation.

```
for ($j = 0; j \leq n_p - 1; j = j + 1$) do
    for ($i = 0; i \leq n_l - 1; i = i + 1$) do
        if $p_j \implies \mathtt{inv}_i$ then break;
    od
    if $i \leq n_l - 1$ then break;
od
if ($j \leq n_p - 1$) or ($i \leq n_l - 1$) then return(false) ;
else return(true)
```

3.3.3 Different CLP Representations of a State Transition System

In the above, we presented a scheme to translate the transition relation of an LHA into CLP. In what follows, we show how the full state transition system is represented in CLP.

State transition systems can be conveniently represented as logic programs. Since such a representation is intended for verification purposes, the chosen representation should meet the following criteria:

1. the values of variables can be continuous or discrete;
2. the transitions can be deterministic or non-deterministic;
3. the (possibly) infinite set of reachable states can be finitely represented and computed;
4. the paths or traces can be represented and
5. we can reason both *forward* from an initial state and *backward* from a chosen target state.

Since we use CLP($\mathcal{Q}$)as the representation/programming language, where the variables are interpreted over the set of rational numbers, our representations in CLP($\mathcal{Q}$)meet the first criteria by default. Furthermore, since an LHA involves linear constraints over variables and constants ranging over real numbers (that include both rational and irrational numbers), the modelling of LHAs in CLP($\mathcal{Q}$)is justified under the implicit assumption that all constants are such reals which are also rationals.

Logic programming is by default a non-deterministic language which directly allows for modelling non-determinism. This is illustrated by the example below.

Example 18. *Consider a constraint logic program (CLP(Q)) defined by the following set of clauses:*

$$\begin{aligned}
&\boldsymbol{p}([1]) \leftarrow \\
&\boldsymbol{t}([X_2],[X_1]) \leftarrow X_2 \geq X_1 + 10 \\
&\boldsymbol{t}([X_2],[X_1]) \leftarrow X_2 \leq X_1 - 10 \\
&\boldsymbol{r}([X]) \leftarrow \boldsymbol{p}([X]) \\
&\boldsymbol{r}([X_2]) \leftarrow \boldsymbol{r}([X_1]), \\
&\qquad\qquad \boldsymbol{t}([X_1],[X_2])
\end{aligned}$$

The above program specifies a state transition system with a single state variable x, whose values range over a set of rational numbers, where predicates: $p/1$ defines the initial state; $t/2$ defines the transition relation and $r/1$ defines the reachable states of a state transition system. Meaning, the next state after the initial state could be either $x \geq 11$ or $x \leq -9$. □

With constraints, infinite sets of states can sometimes be finitely represented. For instance, the infinite set of positive rational numbers could be finitely represented by a clause of the form: $element(X) \leftarrow X >= 0$. The two remaining criteria are met depending on how the sequences of states arising from the transition system are modelled. For instance, a transition could be applied either forwards or backwards.

3.3.4 State transition driver

We refer to that part of the representation which encodes the actual state transition behaviour (of a state transition system) as the *state transition driver* or simply as the *driver*.

Employing a particular driver makes it possible to compute the set of reachable states; while employing another (richer) driver makes it possible to represent and compute the paths. Moreover some properties could be proved by travelling backwards from a chosen target state, while others could be proved by travelling forwards from an initial state. Therefore we use two kinds of drivers, one each for forward reasoning and backward reasoning.

Forwards reasoning driver capturing reachable states

We model the forward transition behaviour with the predicate `rstate`/1. Figure 3.9 gives the definition of `rstate`/1 predicate.

The predicates `init`(S) and `transition`($S1, S2$) appearing in the definition of the driver are specific to a system and are extracted from the high level LHA specifications. The arguments of predicates `init`/1, `rstate`/1 and `transition`/2 are states including the location and time stamps as mentioned earlier.

```
rstate(S2) ←
     transition(S1,S2),
     rstate(S1).
rstate(S0) ←
     init(S0).
```

Figure 3.9: Reachable states driver

```
qstate(S1) ←
     transition(S1,S2),
     qstate(S2).
qstate(Sk) ←
     target(Sk).
```

Figure 3.10: Reaching states driver

Therefore, a complete representation of a state transition system STS is given by the CLP program P, which is the union of the clauses that define predicates `init`/1, `transition`/2 and the `rstate`/1 driver.

Example 19. *The CLP program encoding the LHA model from the previous example (Example 15) is shown in Figure 7.3. This is a forward reasoning driver based CLP encoding.*

Backwards reasoning driver capturing reaching states

The `rstate`/1 predicate was all about travelling forwards from an initial state. By contrast, we could also travel backwards from a particular state. The state transition driver shown in Figure 3.10 allows for reasoning backwards from a specific state, which is called *target state*.

The states that reach a target state are referred to as *reaching states*. The backward driver is useful for computing the set of reaching states for a given target state. This driver is useful to check a sub-class of safety and liveness properties.

The predicates `target`(S_k) and `transition`(S_1, S_2) appearing in the definition of the backward driver are specific to a system and are extracted from the high level LHA specifications. The arguments of predicates `target`/1, `qstate`/1 and `transition`/2 are states.

Therefore, a complete representation of a state transition system STS is given by the CLP program P, which is a union of the clauses that define predicates `init`/1 (or `target`/1), `transition`/2 and the driver (either `rstate`/1 or `qstate`/1). This is illustrated later by the Example 37 in Chapter 7.

3.3.5 Parallel Composition of Linear Hybrid Automata

A system might be composed from two or more LHAs that interact by notifying each other with events. To incorporate such systems, the language of LHA is extended with a set of events $Events = \{evt_1, \ldots, evt_{ne}\}$ (where ne is the number of events in the component LHA). An LHA can raise an event along its discrete transitions. The restriction is that at most one event can be mapped on to a discrete transition. If there are two or more interacting LHAs in a system, all the discrete transitions (across different LHAs) marked with the same event must synchronize. If the discrete transition τ should synchronize on the event evt_e, then that transition is labelled with that event. In the following, we briefly present the rules of synchronization [5].

The *discrete transitions* of two automata LHA_1 and LHA_2 with events $Events_1$ and $Events_2$ respectively, synchronise on an event evt as following:

- if $evt \in Events_1 \cap Events_2$, then the discrete transitions $\tau_i \in \texttt{Trans}_1$ and $\tau_j \in \texttt{Trans}_2$ labelled with the event evt must synchronize;
- if $evt \notin Events_2$ but $evt \in Events_1$, then the discrete transition τ_i of LHA_1 can occur simultaneously with a zero duration delay transition of LHA_2, and similarly if $evt \in \Sigma_2$ and $evt \notin Events_1$ then the discrete transition τ_j of LHA_2 can occur simultaneously with a zero duration delay transition of LHA_1.

Finally, a *delay transition* of LHA_1 with a duration δ must synchronize with a delay transition of LHA_2 of the same duration.

The above rules of synchronisation are enforced by constructing a parallel product of the involved LHAs.

3.4 What does the resulting CLP model capture?

We now relate LHA semantics with the minimal model of the constraint logic program (based on `rstate` and `qstate` drivers).

Recall that the semantics of an LHA is given by runs, which are infinite sequences of the form $(l_0, X_0, t_0) \rightarrow^{\delta_0}_{(\gamma_0,\alpha_0)} (l_1, X_1, t_1) \rightarrow^{\delta_1}_{(\gamma_1,\alpha_1)} (l_2, X_2, t_2) \cdots$. Let us denote by a *state* a triple (l, X, t) where l is a location, X a valuation of the state variables and t a time stamp. Any prefix of a run is called run-prefix. A *reachable state* of the LHA is a triple $s = (l_i, \tilde{X}_i, \tilde{\delta}_i)$ where $\tilde{X}_i = X_i + \tilde{\delta}_i * \dot{X}_i$, and $0 \leq \tilde{\delta}_i \leq \delta_i$. In other words, a state is reachable if it is the state immediately resulting from a discrete transition, or it is the state reached by delaying in a location for some time period. (Note that in the definition of reachable state we use a local time stamp which is set to zero on entering a location).

Let P_r be the constraint logic program encoding the LHA model based on the reachable states (`rstate`) driver and let P_q be the constraint logic program encoding based on the reaching states (`qstate`) driver.

We now show in essence that the set of ground instances of the `rstate`/1 predicate in the minimal model gives the set of reachable states of an LHA. In the following, we give a more formal presentation of this assertion.

Lemma 1. *Let $A = \langle \mathit{Loc}, \mathit{Trans}, \mathit{Var}, \mathit{Init}, \mathit{Inv}, \mathcal{D}\rangle$ be an LHA and P_r be the CLP encoding of A. Suppose $\tau_k = \langle k, l, \alpha_k, \gamma_k, l'\rangle$ is a discrete transition of A. Then for every pair of states (s, s')such that:*
$\sigma = (l_0, X_0, t_0) \rightarrow^{\delta_0}_{(\gamma_0,\alpha_0)} (l_1, X_1, t_1) \rightarrow^{\delta_1}_{(\gamma_1,\alpha_1)} (l_2, X_2, t_2) \cdots$ is a valid run of A and
$s = (l_i, \tilde{X}_i, \tilde{t}_i)$ with $\tilde{t}_i = \tilde{\delta}_i,\ 0 \le \tilde{\delta}_i \le \delta_i$ and $\tilde{X}_i = X_i + \tilde{\delta}_i * \dot{X}_i$
$s' = (l_{i+1}, \tilde{X}_{i+1}, \tilde{t}_{i+1})$ with $\tilde{t}_{i+1} = \tilde{\delta}_{i+1},\ 0 \le \tilde{\delta}_{i+1} \le \delta_{i+1}$, $\tilde{X}_{i+1} = X_{i+1} + \tilde{\delta}_{i+1} * \dot{X}_{i+1}$
there exists `transition`$(s, s') \in \mathsf{M}[\![P_r]\!]$.

Proof. To reach s' from s involves a (possibly zero-length) delay transition followed by a discrete transition followed by a (possibly zero-length) delay transition. The proof directly follows from the definitions of the predicates `transition`/2, `delaytransition`/2 and `discretetransition`/2. □

Theorem 3. *Let $A = \langle \mathit{Loc}, \mathit{Trans}, \mathit{Var}, \mathit{Init}, \mathit{Inv}, \mathcal{D}\rangle$ be an LHA and P_r be the reachable state based CLP encoding of A. Then for every $s = \langle l_i, \tilde{X}_i, \tilde{\delta}_i\rangle$ with $0 \le \tilde{\delta}_i \le \delta_i$ on the run $\sigma = (l_0, X_0, t_0) \rightarrow^{\delta_0}_{(\gamma_0,\alpha_0)} (l_1, X_1, t_1) \rightarrow^{\delta_1}_{(\gamma_1,\alpha_1)} (l_2, X_2, t_2) \cdots$ of A there exists* `rstate`$(s) \in \mathsf{M}[\![P_r]\!]$.

Proof. We prove this by induction on the length of a run-prefix (the index i in a run). For a given run $(l_0, X_0, t_0) \rightarrow^{\delta_0}_{(\gamma_0,\alpha_0)} (l_1, X_1, t_1) \rightarrow^{\delta_1}_{(\gamma_1,\alpha_1)} (l_2, X_2, t_2) \cdots$, define the set of states R_i to be the set $\{(l_i, \tilde{X}_i, \tilde{\delta}_i) \mid 0 \le \tilde{\delta}_i \le \delta_i, \tilde{X}_i = X_i + \tilde{\delta}_i * \dot{X}_i\}$. Every reachable state in a run is in R_i for some i.

1. Base case ($i = 0$):

   P_r contains a clause `rstate(S') :- init(S), delaytransition(S,S')` as shown in Figure 3.9. Following the definitions of (i) the `init`/2 predicate and (ii) the `discretetransition`/2 predicate, for every $s \in R_0$ there exists `rstate`$(s) \in \mathsf{M}[\![P_r]\!]$.

2. Inductive case ($i > 0$): Assume that the proposition holds for all states in $R_0, \ldots, R_{i-1}$. P_r contains a clause
   ```
     rstate(S') :- rstate(S),transition(S,S').
   ```

For the instance `S` $=s$, `S'` $=s'$, we have `transition`$(s, s') \in \mathsf{M}[\![P_r]\!]$ by Lemma 1, and `rstate`$(s) \in \mathsf{M}[\![P_r]\!]$ by the inductive hypothesis. Hence `rstate`(s') is derivable and so `rstate`$(s') \in \mathsf{M}[\![P_r]\!]$.

□

Similarly, it can be shown that the ground instances of the `qstate`/1 predicate in the minimal model of P_q give the states reaching a target state s_t i.e. `qstate`$(s) \in \mathsf{M}[\![P_q]\!] \Rightarrow s \in S_{\to s_t}$ where $S_{\to s_t}$ is the set of states reaching state s_t.

A minimal model is computed iteratively as defined in the previous chapter. Since it is not always possible to compute a concrete minimal model, as an alternative an abstract minimal model is computed. In the following chapters, it will be explained how minimal models provide a basis for proving certain properties.

Summary

In this chapter, we presented a systematic translation of LHA into a constraint logic program. This constraint logic program preserves the reachable states. Besides the reachable properties, the model also preserves a kind of liveness properties. In the latter chapters Chapter 5 and Chapter 6, presented are techniques that make use of the minimal model of the constraint logic programs to verify different CTL properties. The next chapter (Chapter 4) introduces the language of Computational Tree Logic and illustrates how to specify different classes of reactive properties.

Chapter 4

Computation Tree Logic: A temporal property specification language

Typically, formal verification comprises two phases: (i) the specification phase where the system behaviour and the properties are specified and (ii) the proof phase, where the system correctness is proved or disproved with respect to the properties. *System specification* was the topic of the previous chapter, while *property specification* is the topic of this chapter. This chapter presents a *formal property specification language* called *Computation Tree Logic* and how to express different classes of system properties in that language. In this dissertation, the properties to be verified are specified in the Computation Tree Logic.

The correctness of a *system behaviour* is established with respect to the *required behaviours*. A required behaviour is usually identified with the property to be possessed by the system.

In the case of transformational systems, which terminate after transforming inputs into outputs, a property is specified with a pair of predicates: a *pre-condition*, and a *post-condition*. The predicate pre-condition is a condition that should hold on the input, while the post-condition is a condition that should hold on the output. The input-to-output transformational behaviour is formally specified with a *transformation function*. Finally, a property is verified to hold if this function transforms the inputs, where the pre-condition holds, into the outputs, where the post-condition holds. Such verification approaches due to Floyd [44] and Hoare [66] are not adequate for verifying reactive systems, because their correctness depends not only on their transformational behaviour but also on their behaviour over time.

In the case of reactive systems, which do not terminate, the behaviour is given by the *infinite sequence* of states:

$\sigma = s_0\ s_1\ s_2\ \ldots$
where each state following the initial state s_0 results by applying a transition in the preceding state. This sequence preserves the *temporal order* in which the states are visited in real-time. Consequently, specifying a property mandates defining such infinite sequences that possess that property. Let *Prop* be the set of sequences that possess the property *prop*. Then the property *prop* can be verified [2, 3] by checking whether the system behaviour σ is included in the property set *Prop*. The property holds iff $\sigma \in Prop$. Hence specifying a reactive system's property also requires defining infinite state sequences possessing the property. With *Computation Tree Logic* such state sequences can be specified.

System properties can be broadly grouped into *three classes*[1] [91]: (i) *Safety properties*, which specify the invariants characterising the system safety; (ii) *Liveness properties*, which specify the system progress; and (iii) *Precedence properties*, which specify certain precedence ordering among the system properties.

Chapter Overview

- Section 4.1 presents the syntax and semantics of Computation Tree Logic.
- Section 4.2 introduces different classes of temporal properties and their specification in Computation Tree Logic.
- Section 4.3 discusses the use of CTL formulas to specify properties of hybrid systems.

Temporal Logic

Temporal logic is a formalism used for describing infinite sequences. There exists a rich collection of temporal logic languages [38, 73] that are used for various purposes. Depending on how the time is modelled, there are two kinds of temporal logics: (a) Linear-time temporal logics and (b) Branching-time temporal logics.

In the linear-time logics, time is modelled *linearly* i.e. each time-point has got only one successor, whereas in branching-time logics a time-point could have more than one successor. The *Computation Tree Logic* is a branching-time logic.

4.1 Computation Tree Logic

Computation Tree Logic, or CTL in short, is a branching-time temporal logic proposed by Clarke and Emerson in [39] for specifying properties of reactive systems.

[1]In [92] several other classes are mentioned.

The syntax and semantics of CTL are defined as below.

4.1.1 CTL Syntax

The set of CTL formulas ϕ is inductively defined by the following grammar:

$$\begin{aligned}\phi ::= \;& \mathsf{true} \mid p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \phi_1 \leftrightarrow \phi_2 \mid AX\phi \mid EX\phi \mid AF\phi \\ & \mid EF\phi \mid AG\phi \mid EG\phi \mid AU[\phi_1, \phi_2] \mid EU[\phi_1, \phi_2] \mid AR[\phi_1, \phi_2] \mid ER[\phi_1, \phi_2]\end{aligned}$$

where p ranges over a set of atomic formulas `Prop`.

The paired symbols AF, EF, AG, EG, AX, EX, AR, ER, AU and EU are called *CTL-operators* or *CTL-quantifiers*. The operators AR, ER, AU and EU are binary operators; while the AF, EF, AG, EG, AX and EX are unary operators. Each operator is pronounced as following:

- $AG\phi$ "on all paths globally (at every state) ϕ holds";
- $EG\phi$ "there exists a path along which globally (at every state) ϕ holds";
- $AF\phi$ "on all paths in some future state ϕ holds";
- $EF\phi$ "there exists a path along which in some future state ϕ holds";
- $AR[\phi_1, \phi_2]$ "on all paths ϕ_2 is released when ϕ_1 holds";
- $ER[\phi_1, \phi_2]$ "there exists a path where ϕ_2 is released when ϕ_1 holds";
- $AU[\phi_1, \phi_2]$ as "on all paths ϕ_1 holds until ϕ_2 holds";
- $EU[\phi_1, \phi_2]$ as "there exists a path where ϕ_1 holds until ϕ_2 holds";;
- $AX\phi$ "on all paths in the next state ϕ holds";
- $EX\phi$ "there exists a path where in the next state ϕ holds";

Example 20. *Let* ***Prop*** $= \{x = 2, x \leqslant 1, x \geqslant 5\}$ *be the set of atomic propositions. Following are some well formed CTL formulas:*
(i) $EF(x = 2)$*; (ii)* $AG(x \leqslant 1) \vee (x \geqslant 5)$*; (iii)* $AU[(x \leqslant 1), (x \geqslant 5)]$. $\square$

A formula with either a CTL-operator or a boolean connective is constructed from one or more well-formed CTL formulas. Such contained formulas are called *sub-formulas* of the immediate *top level formula*. A proposition is said to be in the *immediate scope* of a CTL-operator if and only if that operator is the nearest operator quantifying that proposition.

Definition 46 (Negation Normal Form). *A CTL-formula is said to be in negation normal form (NNF) if and only if every negation (if any) is adjacent to the atomic propositions in the formula.*

Example 21. *Consider the three CTL-formulas:* $\neg AG\, p$*,* $EG(\neg(p \rightarrow EF\, q))$ *and* $AF\ AG(\neg p)$ *where* p, q *are atomic propositions. The first two formulas are not in NNF, because there are negations that are adjacent to the CTL operators. The third formula is in NNF.* $\square$

Since every CTL-operator has a corresponding dual CTL-operator, any non-NNF CTL-formula can be reduced to a semantically equivalent NNF CTL-formula by applying the De Morgan's laws and elimination of double negation together with the Table 4.1. Table 4.1 summarises the equivalence between the dual CTL-operators.

4.1.2 CTL Semantics

CTL semantics are defined with respect to a state transition system called *Kripke structure.* This *Kripke structure* defines a temporal frame in which a CTL-formula evaluates to either true or false depending on a particular *time-point* (*state*) in that frame. Thus given a Kripke structure K, a CTL-formula ϕ is true in some states and false in the other. The meaning of a CTL formula ϕ with respect to K is the set of all states of K in which ϕ evaluates to `true`. In the following, the Kripke structure and how a CTL-formula is evaluated w.r.t a Kripke structure are explained.

Definition 47 (Kripke Structure). *A Kripke structure* K *is a transition system* $\langle \mathtt{States}, \mathtt{InitStates}, \mathtt{Trans}, \mathtt{Prop}, \mathtt{Label} \rangle$ *where:*

- $\mathtt{States}$ *is a (non-empty) set of states;*
- $\mathtt{InitStates} \subseteq \mathtt{States}$ *is a set of initial states;*
- $\mathtt{Trans} \subseteq \mathtt{States} \times \mathtt{States}$ *is a total relation on* $\mathtt{States}$*, which relates to each state* $s \in States$ *one or more successor states;*
- $\mathtt{Prop}$ *is a set of atomic propositions that is closed under negation, meaning if* $p \in \mathtt{Prop}$ *then* $\neg p \in \mathtt{Prop}$ *or vice versa;*
- $\mathtt{Label} : \mathtt{States} \rightarrow 2^{\mathtt{Prop}}$ *is a labelling function which labels each state* $s \in \mathtt{States}$ *with one or more atomic propositions that are true in that state;*

A Kripke structure is also called *Kripke model* or simply *model.*

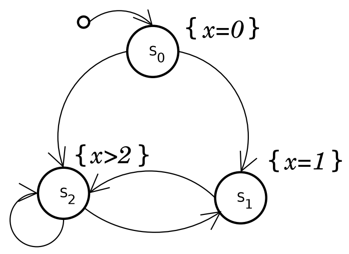

Figure 4.1: A Kripke structure

Example 22. *Consider the Kripke structure* $K_1 = \langle \texttt{States}_1, \texttt{InitStates}_1, \texttt{Trans}_1, \texttt{Prop}_1, \texttt{Label}_1 \rangle$ *where:*

- $\texttt{States}_1 = \{s_0, s_1, s_2\}$;
- $\texttt{InitStates}_1 = \{s_0\}$;
- $\texttt{Trans}_1 = \{(s_0, s_1), (s_0, s_2), (s_2, s_1), (s_1, s_2), (s_2, s_2)\}$;
- *the set of atomic propositions* $\texttt{Prop}_1 = \{x = 1, x = 0, x \geq 2, x \neq 1, x \neq 0, x < 2\}$;
- *the state labels are as below:*
  $\texttt{Label}_1(s_0) = \{x = 0\}, \texttt{Label}_1(s_1) = \{x = 1\}$ *and* $\texttt{Label}_1(s_2) = \{x \geq 2\}$

Figure.4.1 shows K_1, where the states are depicted as circles and the relation $\texttt{Trans}$ *is denoted by arrows. There is an arrow from state s to state $s^{'}$ iff $(s, s^{'}) \in$* $\texttt{Trans}$. *The labelling function is denoted by the sets* $\texttt{Label}_1(s_i)$ *($i \in \{0, 1, 2\}$), which are positioned next to the corresponding states. The computation tree corresponding to this system is constructed by unfolding the transition relation beginning with the initial state. The computation tree for K_1 is shown in Figure. 4.2. As the computation tree is infinite, the figure gives only a (prefix of) the possible computation tree.* □

Example 23. *Consider the Kripke structure from the previous example (22) and the following CTL-formulas:*

1. *EF$(x = 1)$ This formula states that "there exists a path where in future (x=1)". From the computation tree it is clear that this property holds in all the states.*
2. *AF$(x = 1)$ This formula, which states that "along all paths in future (x=1)", does not hold in any of the states.*

3. *$EG(x \geq 2)$ This formula, which states that "there exists a path along which globally (x=2)", holds only in state s_2.*

4. *$AG((x = 0) \vee (x = 1) \vee (x \geq 2))$ This formula, which states that "on all paths globally x remains either equal to 0 or 1 or greater than 2", holds in each and every state.*

$\square$

Definition 47 is the standard definition of a Kripke structure. But in this dissertation, we sometimes replace the labelling function `Label` with a function $\mathsf{states} : \mathtt{Prop} \rightarrow 2^{\mathsf{States}}$ which returns the set of states where an atomic proposition holds. This function, called the *allocation function*, is defined as:
$\mathsf{states}(p) = \{s \in \mathtt{States} \mid p \in \mathtt{Label}(s)\}$.

Definition 48 (Path). *A path of a model K beginning at state s is an infinite sequence of states $s_0\ s_1\ s_2\ \ldots$ such that $s_0 = s$ and $(s_i, s_{i+1}) \in$ **`Trans`** for all $i \geqslant 0$.*

Let S^ω be the set of infinite sequences and S^* be the set of finite sequences formed from the states of the model. For some path $\sigma(\in S^\omega) = a_0\ a_1\ a_2 \ldots$, we use $\sigma[i]$ to denote the $(i+1)$th element of σ i.e. $\sigma[i] = a_i$. Also we use σ_i to denote a finite path-prefix $a_0\ a_1\ \ldots\ a_i$ of length $i+1$. An infinite path with a finite prefix σ_k is represented with $\sigma_k\beta$ where $\beta \in S^\omega : \beta[0] = \sigma[k]$.

Definition 49 (Set of Paths starting in a state). *The set of paths starting in state s of a model K is defined as $P_K(s) = \{\sigma \in S^\omega \mid \sigma[0] = s\}$.*

For any Kripke structure $K = \langle \mathtt{States}, \mathtt{InitStates}, \mathtt{Trans}, \mathtt{Prop}, \mathtt{Label} \rangle$, we can construct an infinite computation tree as follows: (i) the root node is labelled $s_0 \in \mathtt{InitStates}$ and (ii) a node labelled s has a successor node s' iff $(s,s') \in \mathtt{Trans}$. This was illustrated in the Example 22.

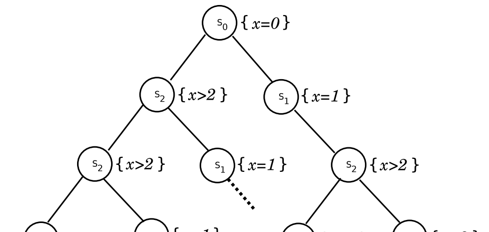

Figure 4.2: A (portion of) the infinite tree

Definition 50 (CTL Semantics)**.** *The CTL semantics with respect to a Kripke structure $K = \langle \texttt{States}, \texttt{InitStates}, \texttt{Trans}, \texttt{Prop}, \texttt{Label} \rangle$ is given by the satisfaction relation "$\models$" between K, one of its states s and a formula ϕ. If the computation tree of K rooted in a state s agrees with the CTL formula ϕ, then the satisfaction relation holds at that state i.e. $K, s \models \phi$. When the specifics of the Kripke structure K is clear from the context, we drop the K in the satisfaction relation i.e. $K, s \models \phi$ is simply written as $s \models \phi$. The satisfaction relation for each CTL-operator is defined as below.*

1. *$K, s \models true$ and $K, s \not\models false$ The proposition true is valid in all states, while proposition $false$ does not hold in any of the states.*

2. *$K, s \models p$ iff $s \in \mathsf{states}(p)$ The formula p holds in a state s if the proposition holds in the state i.e. proposition p labels the state s.*

3. *$K, s \models \neg\phi$ iff $K, s \not\models \phi$ The formula $\neg\phi$ holds in a state s if the formula ϕ does not hold in the state.*

4. *$K, s \models \phi_1 \vee \phi_2$ iff $K, s \models \phi_1$ or $K, s \models \phi_2$ The formula $\phi_1 \vee \phi_2$ holds in a state s if at state s either formula ϕ_1 or formula ϕ_2 holds.*

5. *$K, s \models \phi_1 \wedge \phi_2$ iff $K, s \models \phi_1$ and $K, s \models \phi_2$ The formula $\phi_1 \wedge \phi_2$ holds in a state s if and only if both the formula ϕ_1 and formula ϕ_2 holds in state s.*

6. *$K, s \models AX\phi$ iff $\forall \sigma \in P_K(s) : \sigma[1] \models \phi$ The formula $AX\phi$ holds in state s if and only if along all paths σ starting in s such that in the very next state along all these paths the property ϕ holds.*

7. *$K, s \models EX\phi$ iff $\exists \sigma \in P_K(s) : \sigma[1] \models \phi$ The formula $EX\phi$ holds in state s if and only if there exists some path σ starting in s such that in the very next state along this path the property ϕ holds.*

8. *$K, s \models AF\phi$ iff $\forall \sigma \in P_K(s) : (\exists j \geqslant 0 : \sigma[j] \models \phi)$ The formula $AF\phi$ holds in state s, if and only if along all paths starting in s in some future state (including s) the property ϕ holds.*

9. *$K, s \models EF\phi$ iff $\exists \sigma \in P_K(s) : (\exists j \geqslant 0 : \sigma[j] \models \phi)$. The formula $EF\phi$ holds in state s, if and only if there exists a path starting in s along which in some future state (including s) the property ϕ holds.*

10. *$K, s \models AG\phi$ iff $\forall \sigma \in P_K(s) : (\forall j \geqslant 0 : \sigma[j] \models \phi)$ The formula $AG\phi$ holds in state s, if and only on all paths starting in s along which on all states i.e. globally the property ϕ holds.*

11. $K, s \models EG\phi$ *iff* $\exists \sigma \in P_K(s) : (\forall j \geqslant 0 : \sigma[j] \models \phi)$ *The formula $EG\phi$ holds in state s, if and only if there exists a path starting in s where along all states i.e. globally the property ϕ holds.*

12. $K, s \models AU[\phi_1, \phi_2]$ *iff* $\forall \sigma \in P_K(s) : (\exists j \geqslant 0 : (\sigma[j] \models \phi_2 \wedge (\forall (0 \leqslant k < j) : \sigma[k] \models \phi_1)))$ *The formula $AU[\phi_1, \phi_2]$ holds in state s if and only if along all paths σ starting in s such that in some future state (including s) the property ϕ_2 holds and at all the states preceding this state, property ϕ_1 holds.*

13. $K, s \models EU[\phi_1, \phi_2]$ *iff* $\exists \sigma \in P_K(s) : (\exists j \geqslant 0 : (\sigma[j] = \phi_2 \wedge (\forall (0 \leqslant k < j) : \sigma[k] \models \phi_1)))$ *The formula $EU[\phi_1, \phi_2]$ holds in state s if and only if there exists some path σ starting in s such that in some future state (including s) the property ϕ_2 holds and at all the states preceding this state, property ϕ_1 holds.*

14. $K, s \models AR[\phi_1, \phi_2]$ *iff* $\forall \sigma \in P_K(s) : (\exists j \geqslant 0 : (\sigma[j] = \phi_1 \wedge (\forall (0 \leqslant k \leq j) : \sigma[k] \models \phi_2)))$ *The formula $AR[\phi_1, \phi_2]$ holds in state s if and only if along all paths σ starting in state s either in some future state (including s) the property ϕ_1 holds for the first time and at all the states preceding and including that state, property ϕ_2 holds.*

15. $K, s \models ER[\phi_1, \phi_2]$ *iff* $\forall \sigma \in P_K(s) : (\exists j \geqslant 0 : (\sigma[j] = \phi_1 \wedge (\forall (0 \leqslant k \leq j) : \sigma[k] \models \phi_2)))$ *The formula $ER[\phi_1, \phi_2]$ holds in state s if and only if there exists a path σ starting in state s along which in some future state (including s) the property ϕ_1 holds for the first time and at all the states preceding and including that state, property ϕ_2 holds.*

The semantics of a CTL-formula ϕ is represented as $[\![\phi]\!]$, which is the set of all states where the formula is satisfied i.e. $[\![\phi]\!] = \{s \in \texttt{States} \mid s \models \phi\}$. In a later chapter (Chapter 6), we define a CTL-semantics function $[\![\cdot]\!] : CTL \to 2^{\texttt{States}}$ that returns the $[\![\phi]\!]$ for a given CTL-formula ϕ.

4.1.3 Important equivalences between CTL formulas

Definition 51 (Equivalent CTL-formulas)**.** *Two CTL-formulas ϕ and ψ are said to be semantically equivalent to each other if any state in a model which satisfies one of them also satisfies the other. Such an equivalence is denoted by $\phi \equiv \psi$.*

Duality between CTL-operators The CTL-operators AF, AG, AU, AR, AX are duals of EG, EF, ER, EU, EX respectively. Because of this duality, by De Morgan's laws, we have equivalences as summarised in Table 4.1.

$\phi \equiv \neg(\neg\phi)$
$AF\phi \equiv \neg(EG(\neg\phi))$
$AG\phi \equiv \neg(EF(\neg\phi))$
$AR[\phi_1, \phi_2] \equiv \neg EU[\neg\phi_1, \neg\phi_2]$
$AU[\phi_1, \phi_2] \equiv \neg ER[\neg\phi_1, \neg\phi_2]$
$AX\phi \equiv \neg(EX(\neg\phi))$

Table 4.1: Equivalent CTL-formulas involving dual operators

4.1.4 CTL Axioms

Also, from the CTL-semantics Definition 50, we get the following axioms (equivalences), which define each of the six CTL operators in terms of itself and a next state operator i.e. either *EX* or *AX*.

1. $AF\phi \equiv \phi \vee AX(AF\phi)$
2. $AG\phi \equiv \phi \wedge AX(AG\phi)$
3. $EF\phi \equiv \phi \vee EX(EF\phi)$
4. $EG\phi \equiv \phi \wedge EX(EG\phi)$
5. $AU[\phi_1, \phi_2] \equiv \phi_2 \vee (\phi_1 \wedge AX(AU[\phi_1, \phi_2]))$
6. $EU[\phi_1, \phi_2] \equiv \phi_2 \vee (\phi_1 \wedge EX(EU[\phi_1, \phi_2]))$
7. $AR[\phi_1, \phi_2] \equiv \phi_2 \wedge (\phi_1 \vee AX(AR[\phi_1, \phi_2]))$
8. $ER[\phi_1, \phi_2] \equiv \phi_2 \wedge (\phi_1 \vee EX(ER[\phi_1, \phi_2]))$

These axioms form the basis for the fixed-point characterisation of CTL semantics. Such a fixed-point characterisation is the mathematical foundation for a verification technique called *Model Checking*. Model checking is the topic of the next chapter.

4.1.5 Types of CTL formulas

Depending on their syntactic structure, we differentiate between three types of CTL formulas: (i) simple formulas; (ii) next state formulas and (ii) nested formulas.

Definition 52 (Simple CTL-formula)**.** *A simple CTL-formula is a CTL-formula with exactly one of the eight CTL operators AF, EF, AG, EG, AR, ER, AU and EU.*

In Chapter 5, we present proof techniques to verify a class of simple safety and liveness formulas.

Definition 53 (Next State CTL-formula)**.** *A next state formula is a CTL-formula with either AX or EX as its top-level operator.*

Definition 54 (Nested CTL-formula)**.** *A nested formula is a non-next state CTL-formula with two or more CTL-operators.*

In Chapter 6, we present the technique of abstract interpretation based model checking with which arbitrary CTL formulas can be verified.

Evaluating a formula with no CTL operator can be done given a state alone; while the simple and nested formulas can be evaluated when the paths are computed. The following examples illustrate these evaluation concepts.

Example 24. *Consider a state transition system* $STS = \langle S, S_0, \rightarrow \rangle$ *(of the form defined in Chapter 3 in Definition 43) with only two state variables, identified with* x *and* y *that are evaluated over the set of integers i.e.* $S = Z \times Z$. *The (binary) state transition relation* $\rightarrow$ *defined as:*
$(x, y) \rightarrow (x^{'}, y^{'}) : x \leq 2 \wedge y \leq 4 \wedge x^{'} = y \wedge y^{'} = y + 3$
$(x, y) \rightarrow (x^{'}, y^{'}) : x \leq 2 \wedge y > 4 \wedge x^{'} = y + 1 \wedge y^{'} = x - 2$
$(x, y) \rightarrow (x^{'}, y^{'}) : x > 2 \wedge x^{'} = y - 5 \wedge y^{'} = y$

A state s_i *is a tuple* $(x_i, y_i) \in S$. *The path is given by*
$\sigma : (0,0)\ ((0,3)\ (3,6)\ (1,6)\ (7,-1)\ (-6,-1)\ (-1,2)\ (2,5)\ (6,0)\ (-5,0))^{\omega}$.
The state-sequence that is marked with the super-script ω *repeats infinitely often. Collecting the states reached along a run gives the set of reachable states*
$\mathbb{S}_{\mathbb{R}} = \{(0,0), (0,3), (3,6), (1,6), (7,-1), (-6,-1), (-1,2), (2,5), (6,0), (-5,0)\}$.

Consider two formulas: (i) $p_1 = (x > 0) \wedge (y \leq 7)$ *and (ii)* $p_2 = (x > -10)$ *that state "the value of state variable* x *is greater than* 0 *and value of* y *is less than or equal to* 7*" and "the value of state variable* x *is greater than* -10*" respectively. The formulas* p_1, p_2*, being void of a CTL-operator are just propositions, and can be evaluated at any state* $s \in$ `States` *given that state alone. For instance at state* $s = (0,3)$ *formula* p_1 *evaluates to false since* $s(x) = 0 \not> 0$*, while* p_2 *holds i.e. evaluates to true.* $\square$

Example 25. *Given the transition system STS in Example 24, consider two more formulas: (i)* $\phi_3 = (x = 1 \wedge y = 6) \rightarrow AX(x = 7 \wedge y = -1))$ *and (ii)* $\phi_4 = (x = 3) \rightarrow AF(x = 6 \wedge y = 0)$. *The next state formula* ϕ_3 *states that "the state (1,6) is immediately succeeded by the state (7,-1)"; while* ϕ_4 *states that "a state where variable x equals 3 leads to a state (6,0)".*

To evaluate ϕ_3 *it is necessary that two adjacent states (temporally) along a path be given. Such ordered adjacent states in* σ *are:* $(0,0), (0,3),\ (0,3), (3,6),$

$(3,6),(1,6),\ (1,6),(7,-1),\ \ldots,\ (-5,0),(0,3)$. *These pairs are also defined in the transition relation.*

Similarly ϕ_4 *can be evaluated if the states temporally appearing after the state* $(1,6)$ *along all runs are given. There are finite but several such ordered pairs of states, but we give here only those pairs where the later state is* $(6,0)$*. Such state pairs are* $(0,0),(6,0),\ (0,3),(6,0),\ (3,6),(6,0),\ (1,6),(6,0),\ (7,-1),(6,0),\ (-6,-1),(6,0),\ (-1,2),(6,0),\ (2,5),(6,0),\ (5,0),(6,0)$. *Each pair is an abstraction of the section of a run from the first state to the second state in the pair.* □

Any two distinct states s_j, s_k appearing on a run σ are said to be temporally ordered if the state s_j appears before s_k on σ or vice versa.

Example 26. *The system STS of Example 24 being deterministic has only one run:*
$\sigma = (0,0)\ ((0,3)\ (3,6)\ (1,6)\ (7,-1)\ (-6,-1)\ (-1,2)\ (2,5)\ (6,0)\ (-5,0))^{\omega}$.
Any two instances of states along σ *can be arranged as a temporally ordered pair.* □

Example 27. *Consider two path formulas* q_1 *and* q_2 *from Example 25. The formula* $\psi_1 = AG\ q_1$ *which says "along all runs always the state* $(1,6)$ *is immediately followed by the state* $(7,-1)$*". The formula* $\psi_2 = AG\ q_2$ *means "there exists a run with a state where* $x = 3$ *precedes a state* $(6,0)$*". Both these formulas are safety formulas whose sub-formulas are propositions.* □

4.2 Specifying Properties in CTL

Three classes of properties: Safety, Liveness and Precedence Properties

Naively, system correctness can be stated as "a system is expected to be either *free of forbidden behaviour* or *possess a desired behaviour*". The features that demarcate forbidden behaviours and those that demarcate correct behaviours need to be formalised for specifying properties.

Formally, the system properties assert that during a system execution either "something bad doesn't happen" or "something good will happen". Accordingly, Lamport in [82] grouped properties into two classes, namely: (i) *safety properties* and (ii) *liveness properties.* In [91], besides safety and liveness, a third class of properties named *precedence properties* is mentioned.

In [2], it is proved that that any property that is neither a safety property nor a liveness property is an intersection of safety and liveness properties. This mean every property can be expressed as a combination of safety and liveness properties.

4.2.1 Safety Properties

Informally, a safety property asserts that "during a system execution *something bad never happens*". In a temperature control system that maintains the temperature between T_{min} and T_{max}, a safety property could be "temperature never exceeds T_{max}", in which the *proscribed state* is any state with temperature exceeding T_{max}.

Other examples for safety properties in a multi-process system are: *mutual-exclusion*, *deadlock-freedom* etc. In a mutual-exclusion property, the proscribed *bad state* is "two or more processes executing their critical sections *simultaneously*". In a deadlock-freedom, the proscribed state is a *dead state* (the state with no successor state).

Specifying safety in CTL

In [92], the syntactic definition of a safety formula is given. That definition translates to the following in CTL.

Definition 55 (Safety property/ Safety formula). *A safety formula is a CTL-formula of the form $AG\phi$ or $EG\phi$.*

- *The safety formulas of the form $AG\phi$ are called universal safety formulas.*
- *The safety formulas of the form $EG\phi$ are called existential safety formulas.*

If $\neg\phi$ specifies an *unsafe condition*, then the universal (resp. existential) safety property is specified with a formula of the form $AG\phi$ (resp. $EG\phi$). Formula $AG\phi$ states that "along all paths at every state ϕ holds"; while $EG\phi$ states that "exists a path on which at every state ϕ holds".

The safety formula $AG\phi$ is called a *universal simple safety formula* if the subformula ϕ is either a an atomic proposition or a combination of two or more atomic propositions, otherwise it is called a *universal nested safety formula*. Similarly defined are an *existential nested safety* and an *existential simple safety* formula.

Strictly speaking, *in the sense of safety defined by Lamport*, only universal simple safety formulas qualify to be called safety properties. The nested safety formula if violated will not necessarily lead to an unsafe state.

Following the CTL-semantics definition, a universal simple safety formula AGp holds in a state if there is no reachable state where $\neg p$ holds.

Example 28. *Consider two safety formulas of a floor heating system: (i) $\psi_a = AG(t \leq 25)$ and (ii) $\psi_b = AG((t \geq 19 \wedge t \leq 20) \Rightarrow AF(t > 20))$. Formula ψ_a means "always the temperature remains less than or equal to 20° C", while the formula ψ_b means "always globally, a state with temperature between 19 and 20°C always is followed in future by a state where temperature is greater than*

20° C". Since its sub-formula $\psi_{a,1} = (t \leq 25)$ is a proposition, ψ_a is a simple safety formula. On the contrary ψ_b is a nested safety formula since the sub-formula $\psi_{b,1} = (t \geq 19 \wedge t \leq 20) \Rightarrow AF(t > 20)$ is a simple CTL-formula.

4.2.2 Liveness Properties

Informally, a liveness property asserts that during an execution *something good happens.* In a temperature control system, one liveness property could be "eventually the temperature reaches T°_{max} Celsius".

Other examples for liveness properties in multi-process systems include: (i) starvation freedom, (ii) termination. A starvation freedom property guarantees the progress of every scheduled process. Here the *good thing is making progress.* A termination property asserts that the process will terminate. Here the good thing is termination. In the case of temperature stabilisation, the good thing is temperature hitting a desired temperature of 25°.

An important point about a liveness property is that every partial execution (lacking a good thing until any point in the course of its execution) is always *remediable* in future. For this reason liveness properties are called *eventuality properties.*

Specifying liveness in CTL

[92] gives the canonical form of a liveness formula. That definition translates to the following in CTL.

Definition 56 (Liveness). *A liveness formula is of the form $EF\phi$ or $AF\phi$.*

- *The formulas of the form $AF\phi$ are called universal liveness formulas.*
- *The formulas of the form $EF\phi$ are called existential liveness formulas.*

If ϕ specifies a required good behaviour, then $AF\phi$ (or $EF\phi$) specifies the liveness property.

The existential liveness formula $EF\phi$ (resp. $AF\phi$) specifies that there exists a path (resp. along all paths) along which in some future state ϕ holds.

A liveness formula with simple formulas as its sub-formula is called a *nested liveness property*; while that with propositions as its sub-formula is called a *simple liveness property.* The formula $AF\phi$ is called a *universal liveness property*; while $EF\phi$ is called an *existential liveness property.*

Following the CTL-semantics definition, an existential simple liveness EFp holds if and only if there exists one or more states where p holds.

Example 29. *In the temperature control system, formula* $AF(t = 25)$ *specifies a universal simple liveness property that always in future the temperature* 25 *is reached; while* $AF(t = 25 \rightarrow AG(t = 25))$ *is a universal nested safety property. Replacing the top-level quantifier AF with EF results in the existential liveness versions.*

4.2.3 Precedence Properties

A precedence property asserts that during an execution "eventually something happens and until then at all preceding states something else holds". In a temperature control system, a precedence property looks like "the temperature increases until 25°C is reached". In a telephone system, "the telephone rings until the receiver is lifted off the hooks". In a traffic signalling system, "a green signal is followed by a yellow signal that inturn is followed by a red signal".

A precedence property states that a property ϕ_2 eventually holds and at all preceding states property ϕ_1 holds. It is modelled by an infinite path $\sigma = s_0, s_1, s_2 \ldots$ if in some future state s_j (j≥0) ϕ_2 holds and at all preceding states s_j $(0 \leq i < j)$ ϕ_1 holds.

Specifying precedence in CTL

A precedence property is syntactically defined as below.

Definition 57 (Precedence formula)**.** *A precedence formula is a formula of the form* $AU(\phi_1, \phi_2)$ *or* $EU(\phi_1, \phi_2)$*.*

- *A precedence formula of the form* $AU(\phi_1, \phi_2)$ *is called a universal precedence formula.*
- *A precedence formula of the form* $EU(\phi_1, \phi_2)$ *is called a existential precedence formula.*

The formula $AU[\phi_1, \phi_2]$ (resp. $EU[\phi_1, \phi_2]$) says that along all paths (resp. exists some path) ϕ_2 holds in some future state and until then ϕ_1 holds.

Example 30. *In the temperature control system with a heating element and a power on switch, whose states are modelled with* x *and* y *respectively, the formula* $AU[(x = 1 \wedge y = 1 \wedge t < 25), (x = 1 \wedge y = 0 \wedge t >= 25)]$ *defines a precedence property.*

4.3 CTL and Hybrid systems

As explained in the previous section, the semantics of a CTL formula is given w.r.t. a Kripke structure. Since the constraint logic program modelling an LHA happens to be a Kripke structure, defined with constraints, the CTL formulas have their usual meaning on the CLP model. But the LHAs, whose property we wish to specify, are not pure discrete systems. In this section, the meaning of a CTL formula with respect to the LHA model is presented informally.

We avoid formal presentation, for it involves considering either the topological semantics [9] of hybrid systems or the very powerful modal logics such as Spatial logic [109], both of which are beyond the scope of this dissertation. A formal investigation into modal logics for continuous systems is undertaken in [34]. However, we can bypass these concepts of topologies and modal logics altogether, because Alur *et. al* in [8] formally established that hybrid systems can be *safely abstracted* with discrete systems. Here safety means the resulting discrete abstraction preserve all kinds of temporal properties present in the original hybrid system. Thus, given a CTL formula ϕ, a discrete system preserving ϕ could be constructed. Therefore, for this reason, in various other hybrid system verification frameworks: (a) CTL continues to be used as a property specification language and also (b) to verify CTL properties the continuous dynamics are abstracted with discrete transition systems.

4.3.1 CTL formulas and the runs of an LHA

Recall from Chapter 3 Section 3.2.2, that the run of an LHA is:
$\sigma = (l_0, X_0, t_0) \rightarrow^{\delta_0}_{(\gamma_0,\alpha_0)} (l_1, X_1, t_1) \rightarrow^{\delta_1}_{(\gamma_1,\alpha_1)} (l_2, X_2, t_2) \cdots$
For an LHA involving one or more non-deterministic discrete transitions, there are two or more different runs. Given an LHA, the meanings for each of the CTL operators excepting *EX* and *AX* are as below:

1. $AG\phi$: The formula ϕ holds on all runs at every $t \geq t_r$;
2. $EG\phi$: The formula ϕ holds at every $t \geq t_r$ along one or more runs;
3. $AF\phi$: The formula ϕ holds at some point $t \geq t_r$ along all runs;
4. $EF\phi$: The formula ϕ holds at some $t \geq t_r$ along one or more runs;
5. $AU[\phi_1, \phi_2]$: On all runs, ϕ_1 holds till some time point $t \geq t_r$ at which ϕ_2 holds;
6. $EU[\phi_1, \phi_2]$: On one or more runs, ϕ_1 holds till some time point $t \geq t_r$ at which ϕ_2 holds;

7. $AR[\phi_1, \phi_2]$: On all runs, ϕ_2 holds until some time point $t \geq t_r$ at which ϕ_1 holds;

8. $ER[\phi_1, \phi_2]$: On one or more runs, ϕ_2 holds until some time point $t \geq t_r$ at which ϕ_1 holds.

where t_r is called *reference time* or *root time*. This root time is inherited from the immediate top-level operator. If the top most operator happens to be a CTL operator, then $t_r = t_0$ for that operator; while for the inner operators it takes values based on its immediate top-level operator. If the top most operator is a conjunction (resp. disjunction), then for each conjunct (resp. disjunct) $t_r = t_0$. If the top most operator is an implication (resp. equality), then the antecedent has $t_r = t_0$ while the consequent's root time is that instance $t \geq t_0$ when the antecedent evaluates to true. Given a CTL-formula $Op\phi$ where $Op \in \{AG, EG, AF, EF, AU, EU, AR, ER\}$, if t_r is root time for the top level CTL-operator Op, then the root times for the sub-formulas are defined as following:

1. $AG\phi$: $\forall t_{r\phi} \in [t_r, \infty]$;

2. $EG\phi$: $\forall t_{r\phi} \in [t_r, \infty]$;

3. $AF\phi$: $\exists t_{r\phi} \in [t_r, \infty]$;

4. $EF\phi$: $\exists t_{r\phi} \in [t_r, \infty]$;

5. $AU[\phi_1, \phi_2] : \forall t_{r\phi_1} \in [t_r, t_{r\phi_2})$ and $t_{r\phi_2} \geq t_r$ is the first time when ϕ_2 holds;

6. $EU[\phi_1, \phi_2] : \forall t_{r\phi_1} \in [t_r, t_{r\phi_2})$ and $t_{r\phi_2} \geq t_r$ is the first time when ϕ_2 holds;

7. $AR[\phi_1, \phi_2] : \forall t_{r\phi_2} \in [t_r, t_{r\phi_1}]$ and $t_{r\phi_1} \geq t_r$ is the first time when ϕ_1 holds;

8. $ER[\phi_1, \phi_2] : \forall t_{r\phi_2} \in [t_r, t_{r\phi_1}]$ and $t_{r\phi_1} \geq t_r$ is the first time when ϕ_1 holds.

Though the conditions like $t \geq t_r$ appear, it should not raise philosophical concerns about the continuum of time etc., because as discussed in [94], for a given temporal formula, we only need to consider the discrete instances when the propositions appearing in the formula evaluate to true. This is explained later. Also, in the above, each inequality involving times are evaluated along the same time branch.

Example 31. *Of an LHA consider three CTL-formulas: AFp, $AG(p \rightarrow AF(q \vee r))$ and $AF(EGp)$ where p, q, r are propositions. The formula AFp specifies that on every run, there is some future where p holds. The nested formula $AG(p \rightarrow AF(q \vee r))$ means that along every run globally whenever p holds on every run passing through that state in some future either q or r holds. The third formula states that along all paths from some future onwards there exists a run where p holds for ever.*

To verify these formulas against an LHA, it is not necessary to consider the entire continuum during the delay periods on a run, rather it is sufficient to consider only one instance in each delay duration during which the individual propositions (appearing in the CTL formula to be verified) remain invariant. This means, a formula ϕ, can be verified with our CLP model, iff every run of the CLP model preserves at least one state from every delay duration where a proposition (in ϕ) remains invariant. Recall that our CLP model's run includes only one state (corresponding to some time instant) from each delay segment in the run. Now recall that a delay happens while staying in a location, on which an invariant is defined. What this translates to is "if each invariant (corresponding to each location of an LHA) entails each proposition appearing in a CTL-formula ϕ then the CLP model's run preserves the LHA's property ϕ".

The procedure outlined in Algorithm 2 in Chapter 3 performs exactly this entailment check. If this check fails i.e. the algorithm returns false, then the original LHA needs to be refined. Each location l_i where a proposition (in the formula ϕ) entails invariant $\texttt{inv}_i$ is partitioned into two or more locations locations so that the (refined) invariants corresponding to the refined locations satisfy the entailment check. This refinement is repeated for each proposition. Since the number of propositions in a CTL-formula is finite, the refinement will always terminate. Recall, as explained in the previous chapter, that our definition of the transition relation results in an infinite number of runs corresponding to every linear run in an LHA. On every CLP run there is at least one state from every location. So having refined the LHA model by introducing refined locations with invariants corresponding to the propositions in a given CTL-formula, every run of our model preserves one state on which a proposition remains invariant. Thus model checking results for the CTL formulas on our CLP model do hold on the LHA model.

Summary

In this chapter, we introduced the property specification language of CTL and how different classes of properties are specified in this language. The next chapter defines verification techniques to verify a class of safety and liveness CTL-properties.

Chapter 5

Practical Proof Techniques I

This and the following chapter present practical methods to verify temporal properties. In this chapter we focus on a restricted class of CTL formulas, while the next chapter focuses on arbitrary CTL formulas.

The proof phase of a formal verification involves generating evidence that formally establishes the correctness of a system with respect to a property. In so-called *automated formal verification*, the proof phase is completely automated i.e. given the system specification and the property, the verifier says either YES if the system possesses the property or NO otherwise. This chapter presents a *static analysis*-based approach for *automatic formal verification.*

Essentially, in this chapter, the proof phase comprises two steps. First is the *computation step* where the set of behaviours of a system is computed. Second is the *query step* where it is checked whether the computed set of behaviours possesses the desired property. But computation of all possible behaviours, which are infinite structures is in general impossible. However, certain properties can be verified by computing an abstraction of the total set of behaviours. In this chapter, we consider the set of reachable states or an over approximation of this set in the computation step.

Starting with the constraint logic programs representing LHA models, we compute the set of reachable/reaching states as the minimal model of the respective drivers introduced in Section 3.3.4. We noted that the minimal model of the respective programs captures the set of reachable states from an initial state or the set of reaching states to a given target state, respectively.

Let P be the constraint logic program representing a given LHA *Sys*; the minimal model $\mathsf{M}[\![P]\!]$ capturing the set of reachable or reaching states is represented by constraint facts. In the second proof step, which is the *querying step* of verification, $M[\![P]\!]$ is queried with a suitable goal. Depending on the property to be proved, we might be seeking a presence or lack of a solution to this goal.

In this chapter, we show that universal simple safety and existential simple liveness properties can be verified with this two step proof method based on computing and querying the set of reachable states and/or the set of reaching states. However, for some systems and/or CLP representations, the computation step might not terminate within a predetermined time and/or finite memory. Such cases are then handled by *abstract interpretation*, where instead of a *concrete minimal model* an *abstract minimal model* is computed by interpreting the logic program over abstract domain(s). This *abstract minimal model* (which being an over approximation contains reachable/reaching states that are not visited by the concrete system) is queried. As will be explained latter, whether or not abstraction is useful depends on the property.

Chapter Overview

- Section 5.1 presents two general schemes for verifying certain kinds of simple CTL properties.

- Section 5.2 instantiates the two general schemes presented in the previous section for proving universal simple safety properties.

- Section 5.3 instantiates the two general schemes for proving existential liveness properties.

- Section 5.4 explains the concept of refining a property.

5.1 Verification of simple CTL properties

In Chapter 3, it is shown how an LHA is translated into a CLP program. This CLP program could then be analysed to verify the LHA represented by it. In Chapter 2, it is explained how to compute the minimal model of constraint logic programs (Definitions 31, 33). Furthermore, in cases where a finitely representable model cannot be computed, following Section 2.4 an over approximation of the model can be computed. The minimal model of constraint logic programs is computed using the iterative Algorithm 1. In this section, we show how certain CTL properties can be verified directly using such minimal models or approximations.

In Section 3.3.4, we showed two drivers for encoding an LHA, namely: (i) the reachable states driver `rstate` for forwards reasoning, (ii) the reaching states driver `qstate` for backwards reasoning.

As shown in Section 2.3.2, the models can be represented in extensional form as constrained facts. For instance, the clause `rstate`$(X) \leftarrow X \leq 2, X \geq -2$ is a constrained fact. In this form, the models can be queried using a standard CLP($\mathcal{Q}$)system. For instance, to check whether `rstate`$(3/2)$ has a solution in the model, the model (of constraint logic program based on `rstate` driver) is queried with the goal $\leftarrow \texttt{rstate}(X), 2 * X = 3$.

In the following, we outline two proof methods capable of handling a class of CTL formulas. The first method is for the reachable states driver based representation and the second one is for the reaching states driver based representation.

5.1.1 Method based on reachable states.

Given an LHA model of a system *Sys* and a CTL formula ϕ, the method consists of the following steps:

1. Translation of LHA *Sys* into a CLP program P^r_{Sys}, which is the CLP representation of `Sys` based on the `rstate` driver as explained in Section 3.3.4;

2. Translation of formula ϕ into one or more CLP goals of the form $\leftarrow q$ where q is an atomic proposition;

3. Computation of the minimal model $\mathsf{M}[\![P^r_{Sys}]\!]$ or the abstract model $\mathsf{M}^a[\![P^r_{Sys}]\!]$;

4. Verification of the formula ϕ by querying the (approximate) model with the obtained queries.

5.1.2 Method based on reaching states.

Given an LHA model of a system *Sys* and a CTL formula ϕ, this method consists of the following steps:

1. Translation of sub-formulas of ϕ into one or more CLP clauses of the form $\texttt{qstate}(\bar{X}) \leftarrow c_q(\bar{X})$ where $c_q(\bar{X})$ is a linear constraint representing the target state;

2. Translation of LHA *Sys* into a CLP program P^q_{Sys}, which is the CLP representation of `Sys` based on the `qstate` driver as explained in Section 3.3.4;

3. Computation of the minimal model $\mathsf{M}[\![P^q_{Sys}]\!]$ or the abstract model $\mathsf{M}^a[\![P^q_{Sys}]\!]$;

4. Verification of the formula ϕ by querying the (approximate) model for the initial states.

In the next sections, we show how to verify universal simple safety and existential simple liveness properties using these two methods.

5.2 Universal simple safety

5.2.1 Method based on reachable states.

We now instantiate the `rstate` method, for $\phi = AGp$ where p is an atomic proposition.

Translation of LHA. The LHA *Sys* is translated into the `rstate` based constraint logic program P^r_{Sys}. This translation follows the schema presented in Table 3.1.

Translation of formula. The atomic proposition p is first negated and then the negated proposition $\neg p$ is translated into a CLP query of the form:
$\leftarrow \texttt{rstate}(\bar{X}), c_{\neg p}(\bar{X})$
where $c_{\neg p}(\bar{X})$ is a linear constraint encoding $\neg p$. Recall that any atomic proposition in `Prop` can be characterised as a linear constraint on the state variables of the system *Sys*.

Computation of minimal model. The minimal model $\mathsf{M}[\![P^r_{Sys}]\!]$ is computed using the fixed point algorithm. If the algorithm does not terminate within a predefined (finite) amount of time, then the abstract model $\mathsf{M}^a[\![P^r_{Sys}]\!]$ is computed.

Verification. The minimal model is then queried with the goal:
$\leftarrow \texttt{rstate}(\bar{X}), c_{\neg p}(\bar{X})$.
Depending on which model (concrete or abstract) is queried we have following cases.

1. Concrete minimal model: If $\mathsf{M}[\![P^r_{Sys}]\!]$ has no solution for the query, then the property AGp holds else the property fails;

2. Abstract minimal model:

   (a) If $\mathsf{M}^a[\![P^r_{Sys}]\!]$ has no solution for the query, then the property AGp holds;

   (b) If $\mathsf{M}^a[\![P^r_{Sys}]\!]$ has a solution for the query, then nothing can be concluded about the correctness of the property AGp;

Correctness: The correctness of the above method is based on the CTL-semantics (Definition 50) of the safety property AGp. Given a Kripke structure K, AGp holds at a state s_0, i.e. $K, s_0 \models AGp$ iff $\forall \sigma \in P_k(s_0) : \forall i \geq 0 : \sigma[i] \models p$ where $P_k(s_0)$ is the set of paths originating in state s_0. This means that "AGp holds if and only if

```
rstate([X2,Y2]) ←
     transition([X1,Y1],[X2,Y2]),
     rstate([X1,Y1]).
rstate(S0) ←
     init([X0,Y0]).

init([0,0]).

transition([X1,Y1],[X2,Y2]) ←
     X1 > 2,
     X2 = Y1-5,
     Y2 = Y1.
transition([X1,Y1],[X2,Y2]) ←
     X1 =< 2,
     Y1 =< 4,
     X2 = Y1,
     Y2 = Y1+3.
transition([X1,Y1],[X2,Y2]) ←
      X1 =< 2,
      Y1 > 4,
      X2 = Y1+1,
      Y2 = X1-2.
```

Figure 5.1: The constraint logic program encoding Sys_{24}.

the proposition p holds on each and every state appearing on every path originating in the initial state s_0". This translates to checking that p holds at every state reachable from the initial state. Thus in the above method, in the verification step we query the minimal model, which encodes the set of reachable states, for a state where $\neg p$ holds. Thus universal simple safety can be verified by checking that $\neg p$ holds at none of the reachable states i.e. $\leftarrow$ `rstate`$(\bar{X}), c_{\neg p}(\bar{X})$ fails. When the concrete minimal model cannot be computed, we compute the abstract minimal model. Since the abstract minimal model is an over approximation of the concrete minimal model, the absence of an unsafe state in the set of abstract reachable states also implies their absence in the concrete reachable state space. However if there is a solution, we cannot conclude whether the safety property holds or not.

This method is illustrated in the following example (Example 32).

Example 32. *Consider the transition system from the Example 24 and a property $AG(-6 \leq x \leq 7)$. Its CLP encoding P^r_{Sys} based on the reachable states driver is as shown in Figure 5.1.*

```
rstate([X,Y]) ← 1*X=0,1*Y=0.
rstate([X,Y]) ← 1*X=0,1*Y=3.
rstate([X,Y]) ← 1*X=3,1*Y=6.
rstate([X,Y]) ← 1*X=1,1*Y=6.
rstate([X,Y]) ← 1*X=7,1*Y= -1.
rstate([X,Y]) ← 1*X= -6,1*Y= -1.
rstate([X,Y]) ← 1*X= -1,1*Y=2.
rstate([X,Y]) ← 1*X=2,1*Y=5.
rstate([X,Y]) ← 1*X=6,1*Y=0.
rstate([X,Y]) ← 1*X= -5,1*Y=0.
```

Figure 5.2: The constrained atoms corresponding to `rstate` predicate in $\mathsf{M}[\![P^r_{Sys}]\!]$

The minimal model of this program is shown in Figure 5.2. In the verification step, since the negation of the proposition $-6 \leq x \leq 7$ *is the disjunction* $-6 > x \vee x > 7$, $\mathsf{M}[\![P^r_{Sys}]\!]$ *is queried with the goals* $\leftarrow \texttt{rstate}(X, \_), X < -6$ *and* $\leftarrow \texttt{rstate}(X, \_), X > 7$.

The minimal model of Figure 5.2 defines:
$\mathsf{M}[\![P^r_{Sys}]\!] = \{\texttt{rstate}([0,0]), \texttt{rstate}([0,3]), \texttt{rstate}([3,6]), \texttt{rstate}([1,6]), \texttt{rstate}([7,-1]), \texttt{rstate}([-6,-1]), \texttt{rstate}([-1,2]), \texttt{rstate}([2,5]), \texttt{rstate}([6,0]), \texttt{rstate}([-5,0]), \}$. *It is clear that both goals fail, thus the system is verified to hold the safety property* $AG(-6 \leq x \leq 7)$. $\square$

In the above example, since P^r_{Sys} has three predicates `rstate`/1, `transition`/2 and `init`/1, $\mathsf{M}[\![P^r_{Sys}]\!]$ usually contains constrained atoms corresponding to all the three predicates. But here, before computing the minimal model, we specialised P^r_{Sys} by unfolding the calls to `transition`/2 and `init`/1. Hence the minimal model $\mathsf{M}[\![P^r_{Sys}]\!]$ only contains constrained atoms corresponding to `rstate`/1.

5.2.2 Method based on reaching states.

We now instantiate the `qstate` method, for $\phi = AGp$ where p is an atomic proposition.

Translation of formula: The atomic proposition p is first negated and then the negated proposition $\neg p$ is translated into a CLP clause of the form $\texttt{target}(\bar{X}) \leftarrow c_{\neg p}(\bar{X})$ where $c_{\neg p}(\bar{X})$ is a linear constraint encoding $\neg p$. Thus we defined the target state as any state violating the proposition p. These target states are unsafe states.

Translation of LHA: The LHA *Sys* is translated into the `qstate` based constraint logic program P^q_{Sys}. This translation follows the schema presented in Table 3.1. Recall that a `qstate` driver is with respect to a particular target state, which is specified with the predicate `target`/1. We already specified this in the previous step.

Computation of minimal model: The minimal model $\mathsf{M}[\![P^q_{Sys}]\!]$ is computed using the fixed point algorithm. If the algorithm does not terminate within a (predefined) finite time or memory, then the abstract model $\mathsf{M}^a[\![P^r_{Sys}]\!]$ is computed.

Verification: The minimal model is then queried with the goal:
$\leftarrow \texttt{rstate}(\bar{X}), c_{init}(\bar{X})$
where $c_{init}(\bar{X})$ is the constraint characterising the set of initial states in the LHA *Sys*. Depending on which model (concrete or abstract) is queried we have the following cases.

1. Concrete minimal model: If $\mathsf{M}[\![P^q_{Sys}]\!]$ has no solution for the query, then the property AGp holds else the property fails;
2. Abstract minimal model:
   (a) If $\mathsf{M}^a[\![P^q_{Sys}]\!]$ has no solution for the query, then the property AGp holds;
   (b) If $\mathsf{M}^a[\![P^q_{Sys}]\!]$ has a solution for the query, then nothing can be concluded about the validity of AGp.

Correctness: In backward reasoning, checking universal simple safety translates to proving that none of the initial states *reach* the states where *safety is violated.* Thus we compute the reaching states of a target state, which is marked by the negated proposition $\neg p$, and the resulting set of reaching states is queried for the initial states. In the cases where abstraction of reaching states is computed, since the abstraction is always an over approximation of the set of the concrete reaching states, the presence of initial states in the abstraction does not guarantee their presence in the set of concrete reaching states. Thus, in the reaching states method, we cannot establish the correctness of universal simple safety properties with abstraction.

This method is illustrated with the Example 33.

Example 33. *Now we verify the same system and property from the previous example but with the reaching states method. Figure 5.3 shows the program* P^q_{Sys}. *The negation of the propositional part of* $AG(-6 \leq x \leq 7)$ *is* $\neg p = x < -6 \vee x >$

```
qstate([X1,Y1]) ←
     transition([X1,Y1],[X2,Y2]),
     qstate([X2,Y2]).
qstate([X,Y]) ←
     target([X,Y]).

target([X,_]) ← X > 7.
target([X,_]) ← X < -6.

transition([X1,Y1],[X2,Y2]) ←
     X1 > 2,
     X2 = Y1-5,
     Y2 = Y1.
transition([X1,Y1],[X2,Y2]) ←
     X1 =< 2,
     Y1 =< 4,
     X2 = Y1,
     Y2 = Y1+3.
transition([X1,Y1],[X2,Y2]) ←
     X1 =< 2,
     Y1 > 4,
     X2 = Y1+1,
     Y2 = X1-2.
```

Figure 5.3: The CLP representation of Sys_{24} with the `qstate` driver.

7. *Thus we define the target state as any state satisfying this negated proposition. Then we compute the minimal model* $\mathsf{M}[\![P^q_{Sys}]\!]$*. Figure 5.4 shows the constrained atoms contained in model* $\mathsf{M}[\![P^q_{Sys}]\!]$*. Since the initial state is* $(0,0)$ *i.e. a state where* $X = 0 \wedge Y = 0$ *holds, we query the minimal model with the goal* $\leftarrow qstate([X,Y]), X = 0, Y = 0$*. Since there is no solution for this query the safety property* AGp *holds.* $\square$

5.3 Existential simple liveness

5.3.1 Method based on reachable states

Recall (from Section 4.2.2) that the existential simple liveness property is specified with the CTL formula EFp where p is an atomic proposition. We now instantiate the `rstate` method, for $\phi = EFp$.

```
qstate([X,Y]) ← -1*X>6.
qstate([X,Y]) ← 1*X>7.
qstate([X,Y]) ← -1*Y>1,1*X>2.
qstate([X,Y]) ← 1*Y>12,1*X>2.
qstate([X,Y]) ← -1*X>= -2,-1*Y>6.
qstate([X,Y]) ← -1*X>= -2,1*Y>6.
qstate([X,Y]) ← 1*X>2,-1*Y>= -7,1*Y>6.
qstate([X,Y]) ← -1*X> -1,1*Y>4.
qstate([X,Y]) ← 1*X>2,-1*Y> -6,1*Y>4.
qstate([X,Y]) ← -1*X>= -2,-1*Y>= -4,1*Y>3.
qstate([X,Y]) ← 1*X>2,-1*Y>= -4,1*Y>3.
qstate([X,Y]) ← -1*X>= -2,-1*Y> -3,1*Y>2.
qstate([X,Y]) ← -1*X>= -2,-1*Y>= -1,1*Y>0.
qstate([X,Y]) ← 1*X>2,-1*Y> -3,1*Y>2.
qstate([X,Y]) ← 1*X>2,-1*Y>= -1,1*Y>0.
qstate([X,Y]) ← -1*X>= -2,-1*Y>0,1*Y> -1.
qstate([X,Y]) ← -1*X>= -2,-1*Y>=2,1*Y> -3.
qstate([X,Y]) ← 1*X>2,-1*Y>0,1*Y> -1.
qstate([X,Y]) ← -1*X>= -2,-1*Y>3,1*Y> -4.
qstate([X,Y]) ← -1*X>= -2,-1*Y>=5,1*Y> -6.
qstate([X,Y]) ← 1*Y>4,-1*X> -2,1*X>1.
qstate([X,Y]) ← -1*X>= -2,-1*Y> -2,1*Y>1.
qstate([X,Y]) ← 1*X>2,-1*Y> -2,1*Y>1.
qstate([X,Y]) ← -1*X>= -2,-1*Y>1,1*Y> -2.
qstate([X,Y]) ← -1*X>= -2,-1*Y>4,1*Y> -5.
```

Figure 5.4: The constrained atoms corresponding to `qstate` predicate in $\mathsf{M}[\![P^q_{Sys}]\!]$

Translation of LHA: The LHA *Sys* is translated into the `rstate` based constraint logic program P^r_{Sys}. This translation follows the schema of Table 3.1.

Translation of formula: The atomic proposition p is translated into a CLP query of the form $\leftarrow$ `rstate`$(\bar{X}), c_p(\bar{X})$ where $c_p(\bar{X})$ is a linear constraint encoding the proposition p.

Computation of minimal model: The minimal model $\mathsf{M}[\![P^r_{Sys}]\!]$ is computed using the fixed point algorithm. If the algorithm does not terminate within a predefined (finite) amount of time, then the abstract model $\mathsf{M}^a[\![P^r_{Sys}]\!]$ is computed.

Verification: The minimal model is then queried with the goal:

$\leftarrow \texttt{rstate}(\bar{X}), c_p(\bar{X})$.

Depending on which model (concrete or abstract) is queried we have the following cases.

1. Concrete minimal model: If $\mathsf{M}[\![P^r_{Sys}]\!]$ has a solution for the query, then the property EFp holds else the property fails;
2. Abstract minimal model:
   (a) If $\mathsf{M}^a[\![P^r_{Sys}]\!]$ has no solution for the query, then the property EFp fails;
   (b) If $\mathsf{M}^a[\![P^r_{Sys}]\!]$ has a solution for the query, then nothing can be said about the correctness of the property EFp;

Correctness: The correctness of the above method is based on the definition for the liveness property EFp (Definition 56). The existential simple liveness holds i.e. $K, s_0 \models EFp$ iff $\exists \sigma \in P_K(s_0) : \exists i \geq 0 : \sigma[i] \models p$, which means there exists a reachable state where p holds. Thus in the above method, the reachable states is computed as the minimal model $\mathsf{M}[\![P^r_{Sys}]\!]$, which is queried for a state where p holds with the goal $\leftarrow \texttt{rstate}(\bar{X}), c_p(\bar{X})$. If this query succeeds unconditionally, then p holds in the (concrete) reachable state space, which implies that EFp holds. If the query fails the property does not hold. We later discuss the case where the goal succeeds with residual constraints.

In the approximated reachable state space, since we over approximate the reachable states, the absence of a state where p fails means EFp fails.

This method is illustrated in the following example (Example 34).

Example 34. *Of the same system in the previous example (Example 33), consider the existential liveness formula $\phi_l = EF(X = 1) \wedge (Y = 6)$, which means "there exists a path along which the state (1,6) is reached". This property can be verified by computing the concrete minimal model.*

In the reachable driver method, the minimal model $\mathsf{M}[\![P^r_{Sys}]\!]$ of Figure 5.2 is queried with the goal $\texttt{rstate}([X, Y]), X = 1, Y = 6$. *Since this query has a solution i.e.* $\texttt{rstate}(1, 6) \in \mathsf{M}[\![P^r_{Sys}]\!]$, *the property ϕ_l holds.* □

5.3.2 Method based on reaching states

We now instantiate the `qstate` method, for $\phi = EFp$.

Translation of formula: The atomic proposition p is translated into a CLP clause of the form $\texttt{target}(\bar{X}) \leftarrow c_p(\bar{X})$ where $c_p(\bar{X})$ is a linear constraint encoding p. That is we defined the states where p holds as the target states.

Translation of LHA: The LHA *Sys* is translated into the `qstate` based constraint logic program P^q_{Sys}. This translation follows the schema presented in Table 3.1. Recall that a `qstate` driver is defined with respect to a particular target state, which is specified with the predicate `target`/1. We already specified this in the previous step.

Computation of minimal model: The minimal model $\mathsf{M}[\![P^q_{Sys}]\!]$ is computed using the fixed point algorithm. If the algorithm does not terminate within a predefined (finite) amount of time, then an abstract model $\mathsf{M}^a[\![P^r_{Sys}]\!]$ is computed.

Verification: The minimal model is then queried with the goal:

$\leftarrow \texttt{rstate}(\bar{X}), c_{init}(\bar{X})$

where $c_{init}(\bar{X})$ is the constraint characterising the set of initial states in the LHA *Sys*. Depending on which model (concrete or abstract) is queried we have the following cases.

1. Concrete minimal model: If $\mathsf{M}[\![P^q_{Sys}]\!]$ has a solution for the query, then the property EFp holds (with that state as initial state) else the property fails;

2. Abstract minimal model:

   (a) If $\mathsf{M}^a[\![P^q_{Sys}]\!]$ has no solution for the query, then the property EFp fails;

   (b) If $\mathsf{M}^a[\![P^q_{Sys}]\!]$ has a solution for the query, then nothing can be concluded about the correctness of the property EFp;

Correctness: While reasoning backwards, checking existential liveness EFp translates to checking that from the initial states a state where p holds is reachable. If the initial states are included in $\mathsf{M}[\![P^q_{Sys}]\!]$, which is the set of concrete states reaching a target state where p holds, then the property EFp holds. This inclusion is checked by the unconditional success of the goal $\leftarrow \texttt{rstate}(\bar{X}), c_{init}(\bar{X})$. We consider in a later section the case where the goal succeeds with a residual constraint. In the case of $\mathsf{M}^a[\![P^q_{Sys}]\!]$ where an abstract set of states is computed and if it includes the initial states then nothing can be concluded about the correctness of the existential simple liveness.

This method is illustrated in the following example (Example 35).

Example 35. *To verify the same property in the reaching driver method, we define the state* $(1, 6)$ *as target state. The resulting program is same as the program of Figure 5.3 except for the definition of predicate* `target`/1. *It is replaced with the following definition:*
`target`$([X, Y]) \leftarrow X = 1, Y = 6$. *For this program, the concrete model computing*

algorithm does not terminate. Therefore we compute the abstract minimal model $\mathsf{M}^a[\![P^q_{Sys}]\!] = \{\texttt{qstate}(X,Y) \leftarrow \texttt{true}\}$*, which means there are no constraints on X and Y. Since* $\mathsf{M}^a[\![P^q_{Sys}]\!]$ *is always an over approximation of* $\mathsf{M}[\![P^q_{Sys}]\!]$*, though the initial state* $\texttt{qstate}(X,Y) \leftarrow X = 0, Y = 0$ *has a solution in* $\mathsf{M}^a[\![P^q_{Sys}]\!]$ *nothing can be concluded about the correctness of the property.* $\square$

As seen in the above examples (Examples 35, 34), for some systems, the reachable states method might fare better than the reaching states method and vice versa.

5.4 Refinement of system or a property

In the above instantiations of the two methods, there were verification cases where we could not conclude whether a property holds or not. For example, in the verification of property AGp based on the reachable states driver, step 2(b) is one such case. The results from these inconclusive cases can be exploited to either refine the property or strengthen the initial conditions of the system.

5.4.1 Universal simple liveness

Method based on reachable states

Recall that to verify AGp using an abstract minimal model, in the method based on reachable states, the abstract minimal model is queried for the states where $\neg p$ holds. That is, if P^r_{Sys} is the constraint logic program encoding the LHA *Sys* with the reachable states driver, then $\mathsf{M}^a[\![P^r_{Sys}]\!]$ is queried with the goal $\leftarrow \texttt{rstate}(\bar{X}), c_{\neg p}(\bar{X})$. If there are solutions to this goal then nothing can be concluded about the correctness of AGp. Let $c_{p'}(\bar{X})$ be the constraint that captures these solutions. Say this linear constraint is characterised by a proposition p', then we define a refined property $AG(p \vee p')$. Such a refined property, which is weaker than the original safety property, holds on the system.

Method based on reaching states:

Recall that to verify AGp using an abstract minimal model, in the method based on the reaching states driver, a target state is defined as any state where $\neg p$ holds; then the minimal model of the constraint logic program representing the system with the reaching states driver is queried for the initial states. That is, if P^q_{Sys} is the constraint logic program encoding the LHA *Sys* with the reaching states driver, then $\mathsf{M}^a[\![P^q_{Sys}]\!]$ is queried with the goal $\leftarrow \texttt{rstate}(\bar{X}), c_{init}(\bar{X})$. If there are solutions to this goal then nothing can be concluded about the correctness of AGp. Let

$c_{init'}(\bar{X})$ be the constraint that captures these solutions. Say this linear constraint is characterised by a proposition *init'*, then the initialisation *init* is strengthened as $init \wedge \neg init'$. The safety property AGp holds on such a strengthened system.

5.4.2 Existential simple liveness

In a similar way, we could exploit the results from the cases where an existential liveness property fails. However this is restricted to the cases involving the concrete minimal model.

Method based on reachable states:

Recall that to verify EFp with the concrete minimal model, in the method based on reachable states, the model is queried for the states where $\neg p$ holds. That is, if P^r_{Sys} is the constraint logic program encoding the LHA *Sys* with the reachable states driver, then $\mathsf{M}^a[\![P^r_{Sys}]\!]$ is queried with the goal $\leftarrow \texttt{rstate}(\bar{X}), c_{\neg p}(\bar{X})$. If there are solutions to this goal but with a residual constraint, then nothing can be concluded about the correctness of EFp. Let $c_{p'}(\bar{X})$ be the residual constraint. Say this linear constraint is characterised by a proposition p', then we define a refined property $EF(p \wedge p')$. Such a refined property, which is stronger than the original liveness property, holds on the system.

Method based on reaching states

Recall that to verify EFp with a concrete minimal model, in the method with reaching states driver, the minimal model is queried for the initial states. That is, if P^q_{Sys} is the constraint logic program encoding the LHA *Sys* with the reachable states driver, then $\mathsf{M}[\![P^q_{Sys}]\!]$ is queried with the goal $\leftarrow \texttt{rstate}(\bar{X}), c_{init}(\bar{X})$. If the goal has solutions with residual constraints, then nothing can be concluded about the correctness of EFp. Let $c_{init'}(\bar{X})$ be the residual constraint. Say this linear constraint is characterised by a proposition *init'*, then we strengthen the system's initialisation as $init \wedge init'$. Now the system with such a strengthened initial condition possesses the property EFp.

Summary

In this chapter, we saw methods to prove formulas of the form AGp and EFp that make use of minimal models of CLP programs capturing reachable and reaching states of a system. These methods make direct use of CLP querying. In cases where the desired properties cannot be proved, we can use the results of the CLP query either to strengthen the initial conditions or to modify the property so that it becomes provable. We also saw how over approximations (abstract minimal models) can be used to prove universal safety properties.

In the next chapter, we present a proof method for arbitrary CTL formulas. This method will make use of the techniques of abstract interpretation and model checking.

Chapter 6

Practical Proof Techniques II

The proof methods described in the previous chapter were restricted to simple properties alone. This chapter describes a method to prove nested properties. This proof method has its basis in the technique of *model checking* and exploits the framework of *abstract interpretation.* In contrast to most previous work on abstract model checking, the application of the theory of abstract interpretation is completely standard, and gives results for all CTL properties.

Model checking [106, 23, 118] is a verification technique to automatically verify temporal properties of state transition systems. Only model checking of finite state systems is decidable. But the state transition systems underlying hybrid systems are seldom finite. Therefore, to model-check such infinite state systems, it becomes necessary to finitely represent their state space. Abstraction is a technique for abstracting infinite state systems where a finite or infinite number of *original states* are collectively represented with a single *abstract state.* Besides reduction in state space, abstraction also induces information loss[1]. Therefore, due to such information loss, abstraction based analyses in general are not accurate. Although inaccurate, such abstraction-based analyses, when coined appropriately, provide sufficient precision to prove both the safety and liveness properties.

Model checking is all about checking whether a transition system is a model of a given temporal formula. This involves computing the set of states $[\![\phi]\!]$ where the temporal formula ϕ holds and checking whether the initial state(s) `InitStates` of the system is part of the computed set i.e. $\texttt{InitStates} \subseteq [\![\phi]\!] \Rightarrow M \models \phi$. In this dissertation, we consider the temporal language of Computation Tree Logic (CTL) as the property specification language. This chapter focusses on CTL model checking.

Cousot and Cousot [27, 26] proposed the *framework of Abstract Interpreta-*

[1]Here the information that is lost is the ability to distinguish between the original states that are now jointly represented.

tion, which formalises the abstraction techniques. Though the abstraction based analyses are approximate, this framework guarantees sound but incomplete model checking of CTL. In abstraction interpretation based analyses, first an *abstract domain* is defined and then so called *Galois connections* are defined. Given a concrete domain C and an abstract domain A, which are lattices, a *Galois connection* consists of two functions: one called the abstraction function $\alpha : C \rightarrow A$ and another called the concretisation function $\gamma : A \rightarrow C$. Then the actual analysis is carried out on the abstract domain. Owing to the Galois connections, the result from such abstract analysis will always be a *safe approximation* of the actual result, had the actual analysis been possible.

In our case, model checking is the analysis and the infinite state space is the concrete domain. In essence, the language of CTL will be interpreted over an abstract domain using an interpretation function constructed from a Galois connection. Consequently, abstract model checking returns a set of abstract states $[\![\phi]\!]^a$, which, because of the framework, for any CTL-formula ϕ in negated normal form, is always an over-approximation of the set of concrete states $[\![\phi]\!]$ where ϕ holds i.e. $\gamma([\![\phi]\!]^a) \supseteq [\![\phi]\!]$[2]. Thus the absence of initial states `InitStates` in an over-approximation ensures its absence in the actual set i.e. $\texttt{InitStates} \not\subseteq \gamma([\![\phi]\!]^a) \Rightarrow \texttt{InitStates} \not\subseteq [\![\phi]\!] \Rightarrow M \not\models \phi$. Since $M \not\models \phi \Rightarrow M \models \neg\phi$, abstract interpretation based model checking analysis is accurate in refuting any property $\neg\phi$. And since a negation of any CTL formula can be reduced into NNF, the abstract interpretation based model checking is accurate in proving ϕ.

6.1 CTL Model checking

CTL Model checking is the process of computing an answer to the question of "whether a Kripke structure $K = \langle \texttt{States}, \texttt{Trans}, \texttt{InitStates}, \texttt{Label}, \texttt{Prop} \rangle$ models a CTL formula ϕ". This model checking question translates to checking whether $\forall s \in \texttt{InitStates} : M, s \models \phi$ where ϕ is a well formed CTL formula. Thus CTL model checking involves: (i) computing the set of states $[\![\phi]\!] \subseteq \texttt{States}$ where the CTL formula ϕ holds and (ii) checking whether this set contains all the initial states of K i.e. $\texttt{InitStates} \subseteq [\![\phi]\!]$. If $\texttt{InitStates} \subseteq [\![\phi]\!]$ then K models ϕ i.e. $\forall s \in \texttt{InitStates} : M, s \models \phi$ meaning that K possesses the property ϕ. Otherwise K does not possess the property ϕ.

We define a function $[\![\cdot]\!] : CTL \rightarrow 2^{\texttt{States}}$ that returns the set of states where the CTL-formula, which is in negation normal form (NNF), holds. This function, called the *CTL-semantics function*, is specific to a given Kripke structure. "Model checking algorithm" is the algorithmic incarnation of this function only.

[2]Here γ is an interpretation for $[\![\phi]\!]^a$ in the same domain as $[\![\phi]\!]$.

Before defining the CTL-semantics function, we define three functions: (i) $\mathsf{pred}_\exists : 2^{\mathsf{States}} \rightarrow 2^{\mathsf{States}}$ called the *Predecessor-exists* function; (ii) $\mathsf{pred}_\forall : 2^{\mathsf{States}} \rightarrow 2^{\mathsf{States}}$ called the *Predecessor-for-all* function and (iii) $\mathsf{states} : \mathtt{Prop} \rightarrow 2^{\mathsf{States}}$ called the *Allocation function.* These three functions are specific for a given Kripke structure $K = \langle \mathtt{States}, \mathtt{Trans}, \mathtt{InitStates}, \mathtt{Label}, \mathtt{Prop} \rangle$.

Definition 58 ($\mathsf{pred}_\exists : 2^{\mathsf{States}} \rightarrow 2^{\mathsf{States}}$)**.** *Given a Kripke structure K, the predecessor-exists function* $\mathsf{pred}_\exists : 2^{States} \rightarrow 2^{States}$ *is defined as:*
$\mathsf{pred}_\exists(S') = \{s \mid \exists s^{'} \in S' : (s, s^{'}) \in \mathtt{Trans}\}$.

Definition 59 ($\mathsf{pred}_\forall : 2^{\mathsf{States}} \rightarrow 2^{\mathsf{States}}$)**.** *Given a Kripke structure K, the predecessor-forall function* $\mathsf{pred}_\forall : 2^{States} \rightarrow 2^{States}$ *is defined as:*
$\mathsf{pred}_\forall(S^{'}) = \mathsf{pred}_\exists(S^{'}) \backslash \mathsf{pred}_\exists(\mathsf{compl}(S^{'}))$. *Here the function* $\mathsf{compl}(X) = \mathtt{States} \backslash X$ *is the usual set complement function with set* $\mathtt{States}$ *as the universe.*

Informally, given a set of states $S' \subseteq \mathtt{States}$ of a Kripke structure K, $\mathsf{pred}_\exists(S')$ returns the set of states having at least one of their successors in S', while $\mathsf{pred}_\forall(S^{'})$ returns the set of states having all of their successors in $S^{'}$.

Definition 60 ($\mathsf{states} : \mathtt{Prop} \rightarrow 2^{\mathsf{States}}$)**.** *Given a Kripke structure K, the allocation function* $\mathsf{states} : \mathtt{Prop} \rightarrow 2^{States}$ *is defined as:*
$\mathsf{states}(p) = \{s \in S \mid p \in \mathtt{Label}(s)\}$.

Informally, given an atomic proposition $p \in \mathtt{Prop}$ and Kripke structure K, $\mathsf{states}(p)$ returns the set of states where p holds.

Definition 61 (CTL-semantics function)**.** *Given a Kripke structure $K = \langle \mathtt{States}, \mathtt{Trans}, \mathtt{InitStates}, \mathtt{Label}, \mathtt{Prop} \rangle$, the semantic function $[\![.]\!] : CTL \rightarrow 2^{States}$ is* *recursively defined as below.*

$$\begin{array}{ll}
[\![true]\!] & = \mathtt{States} \\
[\![false]\!] & = \emptyset \\
[\![p]\!] & = \mathsf{states}(p) \\
[\![\neg p]\!] & = \mathsf{states}(\neg p) \\
[\![\phi_1 \vee \phi_2]\!] & = [\![\phi_1]\!] \cup [\![\phi_2]\!] \\
[\![\phi_1 \wedge \phi_2]\!] & = [\![\phi_1]\!] \cap [\![\phi_2]\!] \\
[\![AX\phi]\!] & = \mathsf{pred}_\forall([\![\phi]\!]) \\
[\![EX\phi]\!] & = \mathsf{pred}_\exists([\![\phi]\!]) \\
[\![AF\phi]\!] & = \mu Z.f_{(AF,\phi)}(Z) \textit{ where } f_{(AF,\phi)}(Z) = [\![\phi]\!] \cup \mathsf{pred}_\forall(Z) \\
[\![EF\phi]\!] & = \mu Z.f_{(EF,\phi)}(Z) \textit{ where } f_{(EF,\phi)}(Z) = [\![\phi]\!] \cup \mathsf{pred}_\exists(Z) \\
[\![AG\phi]\!] & = \nu Z.f_{(AG,\phi)}(Z) \textit{ where } f_{(AG,\phi)}(Z) = [\![\phi]\!] \cap \mathsf{pred}_\forall(Z) \\
[\![EG\phi]\!] & = \nu Z.f_{(EG,\phi)}(Z) \textit{ where } f_{(EG,\phi)}(Z) = [\![\phi]\!] \cap \mathsf{pred}_\exists(Z) \\
[\![AR[\phi_1,\phi_2]]\!] & = \nu Z.f_{(AR,[\phi_1,\phi_2])}(Z) \\
 & \qquad \textit{where } f_{(AR,[\phi_1,\phi_2])}(Z) = [\![\phi_2]\!] \cap ([\![\phi_1]\!] \cup \mathsf{pred}_\forall(Z)) \\
[\![ER[\phi_1,\phi_2]]\!] & = \nu Z.f_{(ER,[\phi_1,\phi_2])}(Z) \\
 & \qquad \textit{where } f_{(ER,[\phi_1,\phi_2])}(Z) = [\![\phi_2]\!] \cap ([\![\phi_1]\!] \cup \mathsf{pred}_\exists(Z)) \\
[\![AU[\phi_1,\phi_2]]\!] & = \mu Z.f_{(AU,[\phi_1,\phi_2])}(Z) \\
 & \qquad \textit{where } f_{(AU,[\phi_1,\phi_2])}(Z) = [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathsf{pred}_\forall(Z)) \\
[\![EU[\phi_1,\phi_2]]\!] & = \mu Z.f_{(EU,[\phi_1,\phi_2])}(Z) \\
 & \qquad \textit{where } f_{(EU,[\phi_1,\phi_2])}(Z) = [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathsf{pred}_\exists(Z))
\end{array}$$

Here:

- ϕ, ϕ_1, ϕ_2 *are CTL-formulas in NNF;*
- $\mathsf{pred}_\exists, \mathsf{pred}_\forall$ *and* states *are functions as defined in Definitions 58, 59 and 60, respectively;*
- *the functions* $f_{(AF,\phi)}, f_{(EF,\phi)}, f_{(AG,\phi)}, f_{(EG,\phi)}, f_{(AR,[\phi_1,\phi_2])}, f_{(ER,[\phi_1,\phi_2])}, f_{(AU,[\phi_1,\phi_2])}$ *and* $f_{(EU,[\phi_1,\phi_2])}$ *are called* ***CTL characteristic functions*** *and are of type* $2^{States} \rightarrow 2^{States}$*;*
- $\mu Z.(F(Z))$ *stands for the least fixed point of the function* $\lambda Z.F(Z)$*;*
- $\nu Z.(F(Z))$ *stands for the greatest fixed point of the function* $\lambda Z.F(Z)$*;*

In the above definition, the fixed point functions of the formulas $AF\phi$, $EF\phi$, $AG\phi$, $EG\phi$, $AR[\phi_1, \phi_2]$, $ER[\phi_1, \phi_2]$, $AU[\phi_1, \phi_2]$ and $EU[\phi_1, \phi_2]$ are derived directly from the CTL axioms listed in 4.1.4.

For illustration, in the following, we derive the fixed point function for the formula $[\![AU[\phi_1, \phi_2]]\!]$.

Beginning with the CTL axiom $AU[\phi_1, \phi_2] = \phi_2 \vee (\phi_1 \wedge AX(AU[\phi_1, \phi_2]))$:
$[\![AU[\phi_1, \phi_2]]\!]$
$=$ [*Applying the CTL-semantic function to the above axiom*]
$[\![\phi_2 \vee (\phi_1 \wedge AX(AU[\phi_1, \phi_2]))]\!]$
$=$ [*By CTL semantic function definition* $[\![\psi_1 \vee \psi_2]\!] = [\![\psi_1]\!] \cup [\![\psi_2]\!]$]
$[\![\phi_2]\!] \cup [\![(\phi_1 \wedge AX(AU[\phi_1, \phi_2]))]\!]$
$=$ [*By CTL semantic function definition* $[\![\psi_1 \wedge \psi_2]\!] = [\![\psi_1]\!] \cap [\![\psi_2]\!]$]
$[\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap [\![AX(AU[\phi_1, \phi_2])]\!])$
$=$ [*By CTL semantic function definition* $[\![AX\psi]\!] = \mathsf{pred}_\forall([\![\psi]\!])$]
$[\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathsf{pred}_\forall([\![AU[\phi_1, \phi_2]]\!]))$
$\Leftrightarrow$ [*Substituting* $Z = [\![AU[\phi_1, \phi_2]]\!]$]
$Z = [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathsf{pred}_\forall(Z))$

This is a fixed point equation of the form $Z = f_{(AU,[\phi_1,\phi_2])}(Z)$, where we substituted $f_{(AU,[\phi_1,\phi_2])}(Z) = [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathsf{pred}_\forall(Z)$ is the CTL characteristic function $f_{(AU,[\phi_1,\phi_2])} : 2^{\mathsf{States}} \rightarrow 2^{\mathsf{States}}$ for the operator AU. Actually the semantics of $AU[\phi_1, \phi_2]$ is given by the least fixed point i.e. $Z = \mu Z.f_{(AU,[\phi_1,\phi_2])}(Z)$. The answer to the question "whether to compute the least fixed point or the greatest fixed point for a given formula" is explained in [12].

Similarly the semantics of the formulas $EU[\phi_1, \phi_2], AF\phi$ and $EF\phi$ are given by the least fixed points $\mu Z.f_{(EU,[\phi_1,\phi_2])}(Z)$, $\mu Z.f_{(AF,\phi)}(Z)$ and $\mu Z.f_{(EF,\phi)}(Z)$, respectively. The semantics of the formulas $AG\phi, EG\phi, AR[\phi_1, \phi_2]$ and $ER[\phi_1, \phi_2]$ are given by the greatest fixed points $\nu Z.f_{(AG,\phi)}(Z), \nu Z.f_{(EG,\phi)}(Z), \nu Z.f_{(AR,[\phi_1,\phi_2])}(Z)$ and $\nu Z.f_{(ER,[\phi_1,\phi_2])}(Z)$, respectively.

6.1.1 Criteria for Fixed-point computation

Tarski's fixed point theorem [116] specifies two sufficient conditions that guarantee the existence of fixed-point/s.

These two conditions are:

1. the domain 2^{States} of the characteristic functions (i.e. $f_{(AF,\phi)}, \ldots, f_{(EU,[\phi_1,\phi_2]}$), whose fixed points are to be computed, should form a complete lattice;

2. the characteristic functions should be monotonic.

Domain 2^{States} is a complete lattice: The power-set of any set forms a complete lattice. So 2^{States} forms the complete lattice $\langle 2^{\mathsf{States}}, \subseteq, \cup, \cap, \emptyset, \mathtt{States}\rangle$.

Characteristic functions are monotonic: By Definition 37, a function $F : 2^{\mathsf{States}} \rightarrow 2^{\mathsf{States}}$ is monotonic if and only if $Z_1 \subseteq Z_2 \implies F(Z_1) \subseteq F(Z_2)$. This can be proved for each of the eight characteristic functions. For illustration, we only prove the monotonicity of $f_{(AF,\phi)}(Z)$.

We first check the monotonicity of various operators and subsidiary functions that are used in the characteristic functions. These operators include:

1. set-union $\cup$ and set-intersection $\cap$, which are *monotonic*;
2. set-complement operator compl, since $Z_1 \subseteq Z_2 \Leftrightarrow \mathsf{compl}(Z_2) \subseteq \mathsf{compl}(Z_1)$, is anti-monotonic or anti-tonic;
3. functions $\mathsf{pred}_\exists$ and $\mathsf{pred}_\forall$.

Lemma 2. *The function* $\mathsf{pred}_\exists : 2^{States} \rightarrow 2^{States}$ *is monotonic.*

Proof. Let $Z_1, Z_2 \in 2^{\mathsf{States}}$ such that $Z_1 \subseteq Z_2$. Then

$$\begin{array}{l}
\mathsf{pred}_\exists(Z_1) \\
= [\textit{By definition of } \mathsf{pred}_\exists] \\
\{s \mid \exists s^{'} \in Z_1 : (s, s^{'}) \in \mathtt{Trans}\} \\
\subseteq [\textit{Because } Z_1 \subseteq Z_2] \\
\{s \mid \exists s^{'} \in Z_2 : (s, s^{'}) \in \mathtt{Trans}\} \\
= [\textit{By definition of } \mathsf{pred}_\exists] \\
\mathsf{pred}_\exists(Z_2)
\end{array}$$

□

Hence the function $\mathsf{pred}_\exists : 2^{\mathsf{States}} \rightarrow 2^{\mathsf{States}}$ is monotonic.

Lemma 3 (The function $\mathsf{pred}_\forall : 2^{\mathsf{States}} \rightarrow 2^{\mathsf{States}}$ is monotonic)**.**

Proof. Let $Z_1, Z_2 \in 2^{\mathsf{States}}$ such that $Z_1 \subseteq Z_2$. Then:

$$\begin{array}{l}
\mathsf{pred}_\forall(Z_1) \\
= [\textit{By definition of } \mathsf{pred}_\forall] \\
\mathsf{pred}_\exists(Z_1) \setminus \mathsf{pred}_\exists(\mathsf{compl}(Z_1)) \\
\subseteq [\textit{Because } Z_1 \subseteq Z_2 \Rightarrow \mathsf{pred}_\exists(Z_1) \subseteq (Z_2)] \\
\mathsf{pred}_\exists(Z_2) \setminus \mathsf{pred}_\exists(\mathsf{compl}(Z_1)) \\
\subseteq [\textit{Because } Z_1 \subseteq Z_2 \implies \mathsf{pred}_\exists(\mathsf{compl}(Z_1)) \supseteq \mathsf{pred}_\exists(\mathsf{compl}(Z_2)] \\
\mathsf{pred}_\exists(Z_2) \setminus \mathsf{pred}_\exists(\mathsf{compl}(Z_2)) \\
= [\textit{By definition of } \mathsf{pred}_\forall] \\
\mathsf{pred}_\forall(Z_2)
\end{array}$$

□

Therefore, the universal predecessor function $\mathsf{pred}_\forall$ is monotonic.

Every CTL characteristic function is monotonic: Since a composition of monotonic functions is also a monotonic function, each characteristic function is monotonic. In the following lemma, for illustration, we prove the monotonicity of $f_{(AF,\phi)} : 2^{\mathsf{States}} \rightarrow 2^{\mathsf{States}}$:

Lemma 4. *The function $f_{(AF,\phi)} : 2^{States} \rightarrow 2^{States}$ is monotonic.*

Proof. Let $Z_1, Z_2 \in 2^{\mathsf{States}}$, such that $Z_1 \subseteq Z_2$.

$$\begin{array}{l}
f_{(AF,\phi)}(Z_1) \\
= [\textit{By definition of } f_{(AF,\phi)}] \\
\llbracket\phi\rrbracket \cup \mathsf{pred}_{\forall}(Z_1)\} \\
\subseteq [\textit{Because } Z_1 \subseteq Z_2, \textit{ and } \mathsf{pred}_{\forall} \textit{ is monotonic }] \\
\llbracket\phi\rrbracket \cup \mathsf{pred}_{\forall}(Z_2) \\
= [\textit{By definition of } f_{(AF,\phi)}] \\
f_{(AF,\phi)}(Z_2)
\end{array}$$

□

Therefore irrespective of the arbitrary CTL formula ϕ, $Z_1 \subseteq Z_2 \implies f_{(AF,\phi)}(Z_1) \subseteq f_{(AF,\phi)}(Z_2)$.

6.1.2 Fixed-point algorithm

The algorithms for computing fixed-point (both least and greatest) are by-products of the *Kleene's fixed point theorem* (Theorem 2).

Least fixed point algorithm

The Kleene theorem states "for any continuous function $f : L \rightarrow L$, when L being a complete lattice, the least fixed point $\mu Z.f(Z)$ is the least upper bound of the *ascending Kleene chain* (Definition 40):
$\bot \subseteq f(\bot) \subseteq f(f(\bot)) \subseteq \ldots$".

The algorithm to compute a least fixed point for the CTL-characteristic functions is outlined in 3. In this algorithm, the function F could be any of the CTL characteristic functions $f_{(AF,\phi)}, f_{(EF,\phi)}, f_{(AU,[\phi_1,\phi_2])}$ or $f_{(EU,[\phi_1,\phi_2])}$.

Greatest fixed point algorithm

The Kleene theorem states "for any continuous function $f : L \rightarrow L$, when L being a complete lattice, the greatest fixed point $\nu(Z).f(Z)$ is the greatest lower bound of the *descending Kleene chain* (explained in Section 2.4):
$\top \supseteq f(\top) \supseteq f(f(\top)) \supseteq \ldots$".

Algorithm 3 Least fixed point algorithm for CTL characteristic functions

```
initialise:
    $i = 0;\quad Z_0 = \emptyset$
repeat
    $Z_{i+1} = F(Z_i)$
    $i = i + 1$
until $Z_i = Z_{i-1}$
```

The algorithm to compute a greatest fixed point for the CTL-characteristic functions is outlined in 4. We begin the algorithm with the initialisation $Z =$ `States`. In this algorithm, the function F could be any of the CTL characteristic functions $f_{(AG,\phi)}, f_{(EG,\phi)}, f_{(AR,[\phi_1,\phi_2])}$ or $f_{(ER,[\phi_1,\phi_2])}$.

Algorithm 4 Greatest fixed point algorithm for CTL characteristic functions

```
initialise:
    $i = 0;\quad Z_0 = \texttt{States}$
repeat
    $Z_{i+1} = F_(Z_i)$
    $i = i + 1$
until $Z_i = Z_{i-1}$
```

Thus the computation step of CTL model checking requires implementing the CTL semantic function which in essence is a *fixed point computation.* Though fixed points exist their computation is not always possible i.e. the fixed point algorithms outlined in the above might not terminate. Particularly this is the case when the lattices are infinite. Thus the model checking of infinite state systems, whose state space being infinite forms an infinite lattice, becomes undecidable. For this reason [12], model checking is restricted to the verification of finite state systems. This restriction can be overcome by applying the theory of *abstract interpretation* which makes possible to approximate the $[\![.]\!]$.

In the next section, we explain how to apply the theory of abstract interpretation, which was introduced in Chapter 2, to model check infinite state systems.

6.2 Abstract Interpretation of CTL

For model-checking to be decidable, a system normally should be a finite state system. But hybrid systems are infinite state systems. Hence to model check such systems their state space – the infinite domain over which the system state is evaluated – should be finitely represented. This typically involves a loss of information[3]. Then a new CTL semantic function is defined corresponding to this finite representation. All these steps – of relating an infinite domain to a finite domain and defining the new semantic function – can be formalised in the framework of *abstract interpretation.*

Recall the concepts of abstract interpretation presented in Section 2.4.

6.2.1 How is the theory of abstract interpretation applied?

First, the infinite concrete lattice $\langle 2^{\texttt{States}}, \subseteq, \cup, \cap, \emptyset, \texttt{States}\rangle$ is approximated with a finite abstract lattice $\langle 2^{\texttt{AStates}}, \subseteq, \cup, \cap, \emptyset, \texttt{AStates}\rangle$ and then the Galois connection (Definition 42) $\langle 2^{\texttt{States}}, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\longleftrightarrow}} \langle 2^{\texttt{AStates}}, \subseteq_A\rangle$ is established between these lattices. Recall that the functions $\alpha : 2^{\texttt{States}} \rightarrow 2^{\texttt{AStates}}$ and $\gamma : 2^{\texttt{AStates}} \rightarrow 2^{\texttt{States}}$ are the abstraction function and the concretisation function, respectively,

Now, each CTL characteristic functions $f : 2^{\texttt{States}} \rightarrow 2^{\texttt{States}}$ (whose fixed point needs to be computed) is abstracted by a function $f^\sharp : 2^{\texttt{AStates}} \rightarrow 2^{\texttt{AStates}}$. We refer to function $f^\sharp$ as an abstract CTL characteristic function. The theory of abstract interpretation mandates that $f^\sharp$ be defined such that:

$$\alpha \circ f \circ \gamma \subseteq f^\sharp \tag{6.1}$$

where α and γ form a Galois connection.

So the most precise definition for $f^\sharp : M \rightarrow M$ is:

$$f^\sharp = \alpha \circ f \circ \gamma \tag{6.2}$$

Because of 6.1, we always have the following relation between the fixed points of f and $f^\sharp$:

$$\mu C.f(C) \subseteq \gamma(\mu A.f^\sharp(A)) \tag{6.3}$$

$$\nu C.f(C) \subseteq \gamma(\nu A.f^\sharp(A)) \tag{6.4}$$

where $C \in 2^{\texttt{States}}$ and $A \in 2^{\texttt{AStates}}$ and A is an abstraction of C i.e. $\alpha(C) = A$.

The above subset relations are direct consequences of the equation 6.1 and the Galois connection.

Thus the abstract characteristic function $f^\sharp$ can be used to compute over-approximations of the fixed points of the original characteristic function f. The

[3]A finite representation should not be understood as necessarily a loss of information. The state space remains infinite, but the representation becomes finite. For instance, consider a variable x whose value ranges over the dense interval $[1, 2]$, which is an infinite set of reals. Such an infinite set can be represented finitely with the constraint $1 \leq x \wedge x \leq 2$

case where the abstract semantic function is defined as $f^\sharp = (\alpha \circ f \circ \gamma)$ gives the most precise approximation.

We next apply this general framework to abstraction of the CTL semantic function, and illustrate with a specific abstraction in Section 6.3.

6.2.2 Abstract Interpretation of the CTL Semantic function

In this section we consider abstractions based on Galois connections of the form $\langle 2^{\mathtt{States}}, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^{\mathtt{AStates}}, \subseteq\rangle$, where the abstract domain $2^{\mathtt{AStates}}$ consists of sets of abstract states. In fact the abstract domain could be any lattice but for the purposes of this chapter we consider state-based abstractions, which will be further discussed in Section 6.3.

Definition 62. *Let* $\mathsf{pred}_\exists$, $\mathsf{pred}_\forall$, *and* states *be the functions (defined in Definitions 58, 59 and 60) used in the CTL semantic function. Given a Galois connection* $\langle 2^{States}, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^{AStates}, \subseteq\rangle$, *we define* $\mathsf{apred}_\exists : 2^{AStates} \rightarrow 2^{AStates}$, $\mathsf{apred}_\forall : 2^{AStates} \rightarrow 2^{AStates}$ *and* $\mathsf{astates} : \boldsymbol{Prop} \rightarrow 2^{AStates}$ *as below:*

$$\mathsf{apred}_\exists = \alpha \circ \mathsf{pred}_\exists \circ \gamma \qquad \mathsf{apred}_\forall = \alpha \circ \mathsf{pred}_\forall \circ \gamma \qquad \mathsf{astates} = \alpha \circ \mathsf{states}$$

It follows directly from the properties of Galois connections that for all $S' \subseteq \mathtt{States}$, $\alpha(\mathsf{pred}_\exists(S')) \subseteq \mathsf{apred}_\exists(\alpha(S'))$ and $\alpha(\mathsf{pred}_\forall(S')) \subseteq \mathsf{apred}_\forall(\alpha(S'))$.

Definition 63 (Abstract CTL semantics function). *Given a Galois connection* $\langle 2^{States}, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^{AStates}, \subseteq\rangle$, *the abstract CTL semantic function* $[\![.]\!]^a : CTL \rightarrow 2^{AStates}$ *is defined as follows.*

$$\begin{array}{ll}
[\![true]\!]^a & = \texttt{AStates} \\
[\![false]\!]^a & = \emptyset \\
[\![p]\!]^a & = \mathsf{astates}(p) \\
[\![\phi_1 \vee \phi_2]\!]^a & = [\![\phi_1]\!]^a \cup [\![\phi_2]\!]^a \\
[\![\phi_1 \wedge \phi_2]\!]^a & = [\![\phi_1]\!]^a \cap [\![\phi_2]\!]^a \\
[\![AX\phi]\!]^a & = \mathsf{apred}_\forall([\![\phi]\!]^a) \\
[\![EX\phi]\!]^a & = \mathsf{apred}_\exists([\![\phi]\!]^a) \\
[\![AF\phi]\!]^a & = \mu Z.f^a_{(AF,\phi)}(Z) \ \textit{where} \ f^a_{(AF,\phi)}(Z) = [\![\phi]\!]^a \cup \mathsf{apred}_\forall(Z) \\
[\![EF\phi]\!]^a & = \mu Z.f^a_{(EF,\phi)}(Z) \ \textit{where} \ f^a_{(EF,\phi)}(Z) = [\![\phi]\!]^a \cup \mathsf{apred}_\exists(Z) \\
[\![AG\phi]\!]^a & = \nu Z.f^a_{(AG,\phi)}(Z) \ \textit{where} \ f^a_{(AG,\phi)}(Z) = [\![\phi]\!]^a \cap \mathsf{apred}_\forall(Z) \\
[\![EG\phi]\!]^a & = \nu Z.f^a_{(EG,\phi)}(Z) \ \textit{where} \ f^a_{(EG,\phi)}(Z) = [\![\phi]\!]^a \cap \mathsf{apred}_\exists(Z) \\
[\![AR[\phi_1,\phi_2]]\!]^a & = \nu Z.f^a_{(AR,[\phi_1,\phi_2])}(Z) \\
 & \qquad \textit{where} \ f^a_{(AR,[\phi_1,\phi_2])}(Z) = [\![\phi_2]\!]^a \cap ([\![\phi_1]\!]^a \cup \mathsf{apred}_\forall(Z)) \\
[\![ER[\phi_1,\phi_2]]\!]^a & = \nu Z.f^a_{(ER,[\phi_1,\phi_2])}(Z) \\
 & \qquad \textit{where} \ f^a_{(ER,[\phi_1,\phi_2])}(Z) = [\![\phi_2]\!]^a \cap ([\![\phi_1]\!]^a \cup \mathsf{apred}_\exists(Z)) \\
[\![AU[\phi_1,\phi_2]]\!]^a & = \mu Z.f^a_{(AU,[\phi_1,\phi_2])}(Z) \\
 & \qquad \textit{where} \ f^a_{(AU,[\phi_1,\phi_2])}(Z) = [\![\phi_2]\!]^a \cup ([\![\phi_1]\!]^a \cap \mathsf{apred}_\forall(Z)) \\
[\![EU[\phi_1,\phi_2]]\!]^a & = \mu Z.f^a_{(EU,[\phi_1,\phi_2])}(Z) \\
 & \qquad \textit{where} \ f^a_{(EU,[\phi_1,\phi_2])}(Z) = [\![\phi_2]\!]^a \cup ([\![\phi_1]\!]^a \cap \mathsf{apred}_\exists(Z))
\end{array}$$

Here:

- $\cup$ *and* $\cap$ *are the usual set-union and set-intersection operators, respectively;*
- $\mathsf{apred}_\exists, \mathsf{apred}_\forall$ *and* $\mathsf{astates}$ *are the functions as defined in Definition 62, respectively;*
- *the functions* $f^a_{(AF,\phi)}, f^a_{(EF,\phi)}, f^a_{(AG,\phi)}, f^a_{(EG,\phi)}, f^a_{(AR,[\phi_1,\phi_2])}, f^a_{(ER,[\phi_1,\phi_2])}, f^a_{(AU,[\phi_1,\phi_2])}$ *and* $f^a_{(EU,[\phi_1,\phi_2])}$ *are called* ***abstract CTL characteristic functions*** *and are of type* $2^{\texttt{AStates}} \longrightarrow 2^{\texttt{AStates}}$*;*
- $\mu Z.(F^a(Z))$ *stands for the least fixed point of the function* $\lambda Z.F^a(Z)$*;*
- $\nu Z.(F^a(Z))$ *stands for the greatest fixed point of the function* $\lambda Z.F^a(Z)$*;*

Having defined the abstract CTL semantic function over a given abstract domain and Galois connection, we next consider: (i) how to obtain an abstract domain and (ii) the corresponding Galois connection.

Given a transition system $K = \langle \texttt{States}, \texttt{Trans}, \texttt{InitStates}, \texttt{Label}, \texttt{Prop} \rangle$, from now on, the (original) state space `States` will be called *concrete state space* and the lattice $\langle 2^{\texttt{States}}, \subseteq, \cup, \cap, \emptyset, \texttt{States}\rangle$ is called the *concrete lattice* or the *concrete domain.*

If K is a system with n state variables ranging over domains $D_1, \ldots, D_n$, then the concrete state space $\texttt{States} \subseteq D_1 \times \ldots \times D_n$. Thus $\texttt{States}$ can be viewed as an n-dimensional space.

We are going to consider abstract state spaces that are (not necessarily disjoint) regions of the concrete state space. Usually we will consider such abstractions that are partitions of the concrete state space.

6.2.3 Abstract Domain Construction

Abstract Domain

Typically, an n-dimensional space can be envisaged as a union of one or more n-dimensional regions. Such regions could be either closed or open and might overlap each other. We abstract the infinite concrete state space $\texttt{States} \subseteq \mathbf{R}^n$ by a finite number of n-dimensional regions. Consequently, an *infinite set* of states gets *abstracted with a finite set* of regions. Each element of this finite set collectively represents the (possibly infinite) states in that region.

Each region could be conveniently defined with a set of constraint expressions over the state variables. In this dissertation, we consider the domain of linear constraints as the abstract state space.

Abstract Domain Construction

Such an *abstract state space* is constructed automatically during the static analysis. Recall from the previous chapter (Chapter 5) that we compute the set of reachable states as the minimal model of the constraint logic program representing an LHA with the reachable states driver. A minimal model, whether concrete $\mathsf{M}[\![P_r]\!]$ or abstract $\mathsf{M}^a[\![P_r]\!]$, is a finite set of constrained facts i.e.
$\mathsf{M}[\![P_r]\!] = \{(\texttt{rstate}(\bar{X}) \leftarrow c_1(\bar{X})), \ldots, (\texttt{rstate}(\bar{X}) \leftarrow c_{n_r}(\bar{X}))\}$. Each of these n_r constrained facts collectively represents (possibly infinite) number of states satisfying the linear constraint $c_i(\bar{X})$ (for $i = 1$ to n_r). We compose the *abstract state space* with these constraints as its elements i.e. $\texttt{AStates} = \{v_1, \ldots, v_{n_r}\}$ where each abstract element v_i (for $i = 1$ to n_r) is a region formed by the set of states satisfying the respective constraint c_i (for $i = 1$ to n_r). We then define the abstract domain as the complete lattice formed by the power-set $2^{\texttt{AStates}}$.

Restriction on `AStates`: In this dissertation, we only consider abstractions that partition the concrete state space. That is $\forall v_j, v_k \in \texttt{AStates} : v_j \cap v_k = \emptyset$. This means the conjunction of two constraints in the minimal model are unsatisfiable i.e. $c_j(\bar{X}) \wedge c_k(\bar{X}) \equiv \mathit{false}$. The abstract sets that are not partitions can be made into partitions by either partitioning or merging the overlapping regions.

6.2.4 Constructing the Galois connection

To construct the Galois connection $\langle 2^{\texttt{States}}, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^{\texttt{AStates}}, \subseteq\rangle$, we need to define two functions, namely, the abstraction function $\alpha : 2^{\texttt{States}} \rightarrow 2^{\texttt{AStates}}$, and the concretisation function $\gamma : 2^{\texttt{AStates}} \rightarrow 2^{\texttt{States}}$.

Abstraction function

Before defining the abstraction function $\alpha : 2^{\texttt{States}} \rightarrow 2^{\texttt{AStates}}$, we define how each of the concrete states $s \in \texttt{States}$ is represented with one or more abstract states in the set $AS \in 2^{\texttt{AStates}}$. Since every abstract element is a constraint, elemental abstraction is straightforward. This elemental abstraction is formalised with a function called a *representation function.*

Definition 64 (Representation function $\beta : \texttt{States} \rightarrow 2^{\texttt{AStates}}$)**.** *A representation function $\beta : \texttt{States} \rightarrow 2^{\texttt{AStates}}$ is defined as $\beta(s) = \{v_j \mid s \in v_j\}$.*

Definition 65 (Abstraction function $\alpha : 2^{\texttt{States}} \rightarrow 2^{\texttt{AStates}}$)**.** *The abstraction function $\alpha : 2^{\texttt{States}} \rightarrow 2^{\texttt{AStates}}$ is defined as $\alpha(CS) = \bigcup\{\beta(s) \mid s \in CS\}$. Recall that $\beta : \texttt{States} \rightarrow 2^{\texttt{AStates}}$.*

Concretisation function

The concretisation of an abstract set $AS \in 2^{\texttt{AStates}}$ results in a set of such concrete states *all* of whose abstract representations are contained in AS.

Definition 66 (Concretisation function $\gamma : 2^{\texttt{AStates}} \rightarrow 2^{\texttt{States}}$)**.** *The concretisation function $\gamma : 2^{\texttt{AStates}} \rightarrow 2^{\texttt{States}}$ is defined as: $\gamma(AS) = \{s \in \texttt{States} \mid \beta(s) \subseteq AS\}$.*

Before proceeding any further, it is necessary to know whether the abstraction and concretisation functions form a Galois connection between the concrete and abstract lattices. The following theorem proves that $\langle 2^{\texttt{States}}, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^{\texttt{AStates}}, \subseteq\rangle$.

Theorem 4. *$\langle 2^{\texttt{States}}, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^{\texttt{AStates}}, \subseteq\rangle$.*

Proof. By Definition 42, we need to check whether:

1. the abstraction function $\alpha : 2^{\texttt{States}} \rightarrow 2^{\texttt{AStates}}$ is monotonic;

2. the concretisation function $\gamma : 2^{\texttt{AStates}} \rightarrow 2^{\texttt{States}}$ is monotonic;

3. $\forall CS \in \texttt{States} : CS \subseteq \gamma \circ \alpha(CS)$ and

4. $\forall AS \in \texttt{AStates} : \alpha \circ \gamma(AS) \sqsubseteq AS$.

1. **α is monotonic**:
   Let $CS_1, CS_2 \in 2^{\texttt{States}}$ such that $CS_1 \subseteq CS_2$. Then:

$$\begin{array}{l} \alpha(CS_1) = [\textit{By the definition of } \alpha] \\ \bigcup\{\beta(s) \mid s \in CS_1\} \\ \subseteq [\textit{Because } CS_1 \subseteq CS_2] \\ \bigcup\{\beta(s) \mid s \in CS_2\} \\ = [\textit{By the definition of } \alpha] \\ \alpha(CS_2). \end{array}$$

   Thus α is monotonic.

2. **γ is monotonic:**
   Let $AS_1, AS_2 \in 2^{\texttt{AStates}}$ such that $AS_1 \sqsubseteq AS_2$. Then:

$$\begin{array}{l} \gamma(AS_1) \\ = [\textit{By definition of } \gamma] \\ \{s \in \texttt{States} \mid \beta(s) \subseteq AS_1\} \\ \subseteq [\textit{Because } AS_1 \sqsubseteq AS_2] \\ \{s \in \texttt{States} \mid \beta(s) \subseteq AS_2\} \\ = [\textit{By definition of } \gamma] \\ \gamma(AS_2) \end{array}$$

   Thus γ is monotonic.

3. The parts 3 and 4 of the theorem hold iff:
   $\forall CS \in 2^{\texttt{States}}, \forall AS \in 2^{\texttt{AStates}} : \alpha(CS) \subseteq AS \Leftrightarrow CS \subseteq \gamma(AS)$.

   The proof is taken from [95]:

$$\begin{array}{ll} \alpha(CS) \sqsubseteq AS & \Leftrightarrow \bigcup\{\beta(s) \mid s \in CS\} \sqsubseteq AS \\ & \Leftrightarrow \forall s \in CS : \beta(s) \sqsubseteq AS \\ & \Leftrightarrow CS \subseteq \gamma(AS) \end{array}$$

Therefore α and γ form the $\langle 2^{\texttt{States}}, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\longleftrightarrows}} \langle 2^{\texttt{AStates}}, \subseteq\rangle$. □

Since all the operators appearing in the abstract CTL-semantic are monotonic, the fixed-point expressions and hence the abstract semantic function is well defined. The following soundness theorem is the basis of our abstract model checking approach.

Theorem 5 (Safety of Abstract CTL Semantics)**.** *Let K be a Kripke structure, $\langle 2^{States}, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\longleftrightarrow}} \langle 2^{AStates}, \subseteq\rangle$ be a Galois connection and ϕ any CTL-formula in negation normal form. Then $\alpha(\llbracket\phi\rrbracket) \subseteq \llbracket\phi\rrbracket^a$ and $\gamma(\llbracket\phi\rrbracket^a) \supseteq \llbracket\phi\rrbracket$.*

The proof follows from the fact that α is a join-morphism: that is, that $\alpha(S_1 \cup S_2) = \alpha(S_1) \cup \alpha(S_2)$ and the fact that $\alpha(S_1 \cap S_2) \subseteq \alpha(S_1) \cap \alpha(S_2)$.

Theorem 6.

$$\forall \phi \in CTL : \alpha(\llbracket\phi\rrbracket) \subseteq \llbracket\phi\rrbracket^a$$

Proof. Proof by induction on the depth of ϕ :

1. Base case $k = 1$ i.e. $\phi = p$

$$\begin{array}{l}\alpha(\llbracket p\rrbracket) \\ = [\textit{By definition of } \llbracket p\rrbracket] \\ \alpha \circ \mathsf{states}(p) \\ = [\textit{By definition of } \mathsf{astates}] \\ \mathsf{astates}(p) \\ = [\textit{By definition of } \llbracket p\rrbracket^a] \\ \llbracket p\rrbracket^a\end{array}$$

   Thus $\alpha(\llbracket p\rrbracket) \subseteq \llbracket p\rrbracket^a$.

2. Inductive case: Depth $k > 1$

   Inductive hypothesis: For all ϕ of depth $j < k$, assume that $\alpha(\llbracket\phi\rrbracket) \subseteq \llbracket\phi\rrbracket^a$.

   Induction: There are several cases depending on the form of ϕ as following:

   (a) $\phi_1 \cup \phi_2$ and $\phi_1 \cap \phi_2$:

$$\begin{array}{l}\alpha(\phi_1 \cup \phi_2) \\ = [\textit{Since } \alpha \textit{ is additive}] \\ \alpha(\phi_1) \cup \alpha(\phi_2) \\ \subseteq [\textit{By induction hypothesis}] \\ \llbracket\phi_1\rrbracket^a \cup \llbracket\phi_2\rrbracket^a \\ = [\textit{By definition of } \llbracket\rrbracket^a] \\ \llbracket\phi_1 \cup \phi_2\rrbracket^a\end{array}$$

   Therefore, $\alpha(\phi_1 \cup \phi_2) \subseteq \llbracket\phi_1 \cup \phi_2\rrbracket^a$. Similarly we can prove $\alpha(\phi_1 \cap \phi_2) \subseteq \llbracket\phi_1 \cap \phi_2\rrbracket^a$.

(b) $EX\phi$ and $AX\phi$:

$$\begin{aligned}
&\alpha([\![EX\phi]\!])\\
&= [\textit{By definition of } [\![]\!]]\\
&\alpha(\mathsf{pred}_\exists([\![\phi]\!]))\\
&\subseteq [\textit{Since}[\![\phi]\!] \subseteq \gamma(\alpha([\![\phi]\!]))\textit{and } \mathsf{pred}_\exists \textit{ is monotonic}]\\
&\alpha(\mathsf{pred}_\exists(\gamma \circ \alpha([\![\phi]\!])))\\
&=\\
&\alpha(\mathsf{pred}_\exists(\gamma(\alpha([\![\phi]\!]))))\\
&= [\textit{By definition of } \mathsf{apred}_\exists]\\
&\mathsf{apred}_\exists(\alpha([\![\phi]\!]))\\
&\subseteq [\textit{By induction hypothesis and } \mathsf{apred}_\exists \textit{ being monotonic}]\\
&\mathsf{apred}_\exists([\![\phi]\!]^a)\\
&= [\textit{By definition of } [\![]\!]^a]\\
&[\![EX\phi]\!]^a
\end{aligned}$$

Thus $\alpha([\![EX\phi]\!]) \subseteq [\![EX\phi]\!]^a$. Similarly it can be proved that $\alpha([\![AX\phi]\!]) \subseteq [\![AX\phi]\!]^a$.

(c) We can prove the same for the semantic functions involving fixed point computations i.e. $AF\phi, \ldots, EU[\phi_1, \phi_2]$.

$$\begin{aligned}
&\alpha([\![Op\ \phi]\!])[\textit{where the place holder } Op \in \{AF, \ldots, EG\}]\\
&= [\textit{By CTL semantic function}]\\
&\alpha(\mathit{fix}(f_{(Op,\phi)}))\\
&\subseteq [\textit{By the Lemma 5}]\\
&\mathit{fix}(f^a_{(Op,\phi)})\\
&=\\
&[\![Op\ \phi]\!]^a
\end{aligned}$$

In the above, depending on the operator *Op*, *fix*($f_{(Op,\phi)}$) stands for either the least fixed point or the greatest fixed point of the function $f_{(Op,\phi)}$.

Thus $\alpha([\![Op\ \phi]\!]) \subseteq [\![Op\ \phi]\!]^a$ where $Op \in \{AF, \ldots, EG\}$. We can prove the same for the operators AU, EU, AR and ER. □

Lemma 5. *Let* $f : 2^{States} \to 2^{States}$, $f^a : 2^{AStates} \to 2^{AStates}$ *be two monotonic function over the complete lattices* 2^{States} *and* $2^{AStates}$ *such that* $\alpha \circ f \subseteq f^a \circ \alpha$ *where* $\langle 2^{States}, \alpha, \gamma, 2^{AStates}\rangle$ *is the Galois connection. By Theorems 7.1.0.2 and 7.1.0.4 from [27], we have* $\alpha(\mathsf{lfp}(f)) \subseteq \mathsf{lfp}(f^a)$ *and* $\alpha(\mathsf{gfp}(f)) \subseteq \mathsf{gfp}(f^a)$.

Following this lemma, since $f_{(Op,\phi)}$ is a monotonic function, given the Galois connection $\langle 2^{\mathsf{States}}, \alpha, \gamma, 2^{\mathsf{AStates}}\rangle$, if $\alpha(\mathsf{lfp}(f_{(Op,\phi)})) \subseteq \mathsf{lfp}(f^a_{(Op,\phi)})$, then $\alpha(\mathsf{lfp}(f_{(Op,\phi)}))$

$\subseteq \mathsf{lfp}(f^a_{(Op,\phi)})$ and $\alpha(\mathsf{gfp}(f_{(Op,\phi)})) \subseteq \mathsf{gfp}(f^a_{(Op,\phi)})$. In Theorem 7, it is proved that $\alpha(f_{(Op,\phi)}) \subseteq f^a_{(Op,\phi)}$.

Theorem 7. $\alpha(f_{(Op,\phi)}) \subseteq f^a_{(Op,\phi)}$ *where* ϕ *is a CTL formula and* $Op \in \{AF, \ldots, EG\}$.

Proof. Assume that $\alpha(\llbracket\phi\rrbracket) \subseteq \llbracket\phi\rrbracket^a$.

We prove when $Op = AF$

$\alpha(f_{(AF,\phi)}(Z))$
$= [\textit{By definition } \llbracket\rrbracket]$
$\alpha(\llbracket\phi\rrbracket \cup \mathsf{pred}_\forall(Z))$
$= [\textit{Because } \alpha \textit{ is additive}]$
$\alpha(\llbracket\phi\rrbracket) \cup \alpha(\mathsf{pred}_\forall(Z))$
$\subseteq [\textit{By the Theorem 6}]$
$\llbracket\phi\rrbracket^a \cup \alpha(\mathsf{pred}_\forall(Z))$
$\subseteq [\textit{Because of the Galois connection and the monotonicity of } \mathsf{pred}_\forall]$
$\llbracket\phi\rrbracket^a \cup \alpha(\mathsf{pred}_\forall(\gamma(\alpha(Z))))$
$=$
$\llbracket\phi\rrbracket^a \cup \alpha \circ \mathsf{pred}_\forall \circ \gamma(\alpha(Z))$
$= [\textit{By the definition of } \mathsf{pred}_\forall]$
$\llbracket\phi\rrbracket^a \cup \mathsf{apred}_\forall(\alpha(Z))$
$= [\textit{By the abstract CTL function definition}]$
$\llbracket f^a_{(AF,\phi)}(\alpha(Z))\rrbracket$

Thus $\alpha \circ f_{(AF\phi)} \subseteq \llbracket f^a_{(AF\phi)} \circ \alpha\rrbracket$. We can prove a similar result for the rest of the CTL characteristic functions. □

Verification based on abstract semantics: The above theorem provides us with a sound abstract model checking procedure for any CTL formula ϕ. As noted previously, $K \models \phi$ iff $\llbracket\neg\phi\rrbracket \cap \texttt{InitStates} = \emptyset$ (where $\neg\phi$ is converted to negation normal form). It follows from Theorem 5 that this follows if $\gamma(\llbracket\neg\phi\rrbracket^a) \cap \texttt{InitStates} = \emptyset$. Of course, if $\gamma(\llbracket\neg\phi\rrbracket^a) \cap \texttt{InitStates} \supseteq \emptyset$ nothing can be concluded.

Abstract-interpretation based model checking recipe For a transition system TS, the Abstract interpretation based CTL Model Checking recipe is as below:

1. Negate the CTL-formula ϕ to be verified. Say $\neg\phi = \psi$ where ψ is in negated normal form.
2. Compute the abstract states $\llbracket\psi\rrbracket^a$.
3. Compute the concrete sets $\gamma(\llbracket\psi\rrbracket^a)$.
4. Check whether $\gamma(\llbracket\psi\rrbracket^a) \cap \texttt{InitStates} = \emptyset$

6.3 Abstract Model Checking in Constraint-based Domains

The abstract semantics given in Section 6.2.2 is not always implementable in practice for a given Galois connection $\langle 2^C, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\longleftrightarrow}} \langle 2^A, \subseteq\rangle$. Here A, C are `AStates` and `States`, respectively. In particular, the function γ yields a value in the concrete domain, which is typically an infinite object. Thus evaluating the functions $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)$ and $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)$ might not be feasible.

In this section we show that the construction is implementable for transition systems and abstract domains expressed using linear constraints.

6.3.1 Constraint Representation of Transition Systems

Recall from Section 7.1, that we represent the transition systems with linear constraints. We review the basic terminology and operations on linear constraints. Consider the set of linear arithmetic constraints (hereafter simply called constraints) over the real numbers.

$$c ::= t_1 \leq t_2 \mid t_1 < t_2 \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \mid \neg c$$

where t_1, t_2 are linear arithmetic terms built from real constants, variables and the operators $+$, $*$ and $-$. The constraint $t_1 = t_2$ is an abbreviation for $t_1 \leq t_2 \wedge t_2 \leq t_1$. Note that $\neg(t_1 \leq t_2) \equiv t_2 < t_2$ and $\neg(t_1 < t_2) \equiv t_2 \leq t_2$, and so the negation symbol $\neg$ can be eliminated from constraints if desired by moving negations inwards by Boolean transformations and then applying this equivalence.

To identify the variables $\bar{X}$ occuring in a constraint c, we sometimes write $c(\bar{X})$. A constraint is *satisfied* by an assignment of real numbers to its variables if the constraint evaluates to *true* under this assignment, and is *satisfiable* if there exists some assignment that satisfies it. A constraint can be identified with the set of assignments that satisfy it. Thus a constraint over n real variable represents a set of points in R^n.

A constraint can be projected onto a subset of its variables. Denote by $\mathsf{proj}_V(c)$ the projection of c onto the set of variables V.

Let us consider a transition system defined over the state-space R^n. Let $\bar{x}, \bar{x}_1, \bar{x}_2$ etc. represent n-tuples of distinct variables, and $\bar{r}, \bar{r}_1, \bar{r}_2$ etc. represent tuples of real numbers. Let $\bar{x}/\bar{r}$ represent the assignment of values $\bar{r}$ to the respective variables $\bar{x}$. We consider transition systems in which the transitions can be represented as a finite set of *transition rules* of the form $\bar{x}_1 \xrightarrow{c(\bar{x}_1,\bar{x}_2)} \bar{x}_2$. This represents the set of all transitions from state $\bar{r}_1$ to state $\bar{r}_2$ in which the constraint $c(\bar{x}_1, \bar{x}_2)$ is satisfied by the assignment $\bar{x}_1/\bar{r}_1, \bar{x}_2/\bar{r}_2$. Such transition systems can be used to model real-time control systems [57, 13].

6.3.2 Computation of the CTL semantic function using constraints

A constraint representation of a transition system allows a constraint solver to be used to compute the functions $\mathsf{pred}_\exists$, $\mathsf{pred}_\forall$ and states in the CTL semantics. Let T be a finite set of transition rules. Let $c'(\bar{y})$ be a constraint over variables $\bar{y}$. It is assumed that the set of propositions in the Kripke structure used in the semantics is the set of linear constraints.

$$\begin{array}{l}
\mathsf{pred}_\exists(c'(\bar{y})) = \bigvee\{\mathsf{proj}_{\bar{x}}(c'(\bar{y}) \wedge c(\bar{x},\bar{y})) \mid \bar{x} \xrightarrow{c(\bar{x},\bar{y})} \bar{y} \in T\} \\
\mathsf{pred}_\forall(c'(\bar{y})) = \mathsf{pred}_\exists(c'(\bar{y})) \wedge \neg(\mathsf{pred}_\exists(\neg c'(\bar{y}))) \\
\mathsf{states}(p) = p
\end{array}$$

In the definition of states, we use p both as the proposition (the argument of states) and as a set of points (the result) where p holds.

6.3.3 Abstract Domains Based on a Disjoint State-Space Partition

Suppose we have a transition system with n state variables; we take as the *concrete domain* the complete lattice $\langle 2^C, \subseteq\rangle$ where $C \subseteq \mathbf{R}^n$ is some non-empty, possibly infinite set of n-tuples including all the reachable states of the system.

We build an abstraction of the state space based on a disjoint partition of C say $A = \{d_1, \ldots, d_k\}$ such that $\bigcup A = C$. Such a partition can itself be constructed by an abstract interpretation of the transition relation [13]. Define a representation function $\beta : C \to 2^A$, such that $\beta(\bar{x}) = \{d \in A \mid \bar{x} \in d\}$. We extend the representation function to sets of points, obtaining the abstraction function $\alpha; 2^C \to 2^A$ given by $\alpha(S) = \bigcup\{\beta(\bar{x}) \mid \bar{x} \in S\}$. Define the concretisation function $\gamma : 2^A \to 2^C$, as $\gamma(V) = \{\bar{x} \in C \mid \beta(\bar{x}) \subseteq V\}$. As shown in [95, 27], $(2^C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (2^A, \subseteq)$ is a Galois connection. Because the partition A is disjoint the value of $\beta(\bar{x})$ is a singleton for all x, and the γ function can be written as $\gamma(V) = \bigcup\{\gamma(\{d\}) \mid d \in V\}$.

6.3.4 Representation of Abstraction Using Constraints

A constraint can be identified with the set of points that satisfies it. Suppose that each element d of the partition A is representable as a linear constraint c_d over the variables $x_1, \ldots x_n$. The β function can be rewritten as $\beta(\bar{x}) = \{d \mid (\bar{x})$ satisfies $c_d\}$. Assuming that we apply α to sets of points represented by a linear constraint over $x_1, \ldots x_n$, we can rewrite the α and γ functions as follows.

$$\alpha(c) = \{d \mid \mathsf{SAT}(c_d \wedge c)\} \qquad \gamma(V) = \bigvee\{c_d \mid d \in V\}$$

6.3.5 Computation of α and γ functions using constraint solvers

The constraint formulations of the α and γ functions allows them to be effectively computed. The expression $\mathsf{SAT}(c_d \wedge c)$ occurring in the α function means "$(c_d \wedge c)$ is satisfiable" and can be checked by an SMT solver. In our experiments we use the SMT solver Yices [37]. The γ function simply collects a disjunction of the constraints associated with the given set of partitions; no solver is required.

6.3.6 Implementation of constraint-based abstract semantics

Combining the constraint-based evaluation of the functions $\mathsf{pred}_\exists$ and $\mathsf{pred}_\forall$ with the constraint-based evaluation of the α and γ functions gives us (in principle) a method of computing the abstract semantic counterparts of $\mathsf{pred}_\exists$ and $\mathsf{pred}_\forall$, namely $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)$ and $(\alpha \circ \mathsf{pred}_\forall \circ \gamma)$. This gives us a sound abstract semantics for CTL as discussed previously. The question we now address is the feasibility of this approach. Taken naively, the evaluation of these constraint-based functions (in particular $\mathsf{pred}_\forall$) does not scale up. We now show how we can transform these definitions to a form which can be computed much more efficiently, with the help of an SMT solver.

Consider the evaluation of $(\alpha \circ \mathsf{pred}_\forall \circ \gamma)(V)$ where $V \in 2^A$ is a set of disjoint partitions represented by constraints.

$$\begin{aligned}
(\alpha \circ \mathsf{pred}_\forall \circ \gamma)(V) &= (\alpha \circ \mathsf{pred}_\forall)(\bigvee\{c_d \mid d \in V\}) \\
&= \alpha(\mathsf{pred}_\exists(\bigvee\{c_d \mid d \in V\}) \wedge \neg(\mathsf{pred}_\exists(\neg(\bigvee\{c_d \mid d \in V\}))) \\
&= \alpha(\mathsf{pred}_\exists(\bigvee\{c_d \mid d \in V\}) \wedge \neg(\mathsf{pred}_\exists(\bigvee\{c_d \in A \setminus V\}))
\end{aligned}$$

In the last step, we use the equivalence $\neg(\bigvee\{c_d \mid d \in V\}) \leftrightarrow \bigvee\{c_d \in A \setminus V\})$, which is justified since the abstract domain A is a disjoint partition of the concrete domain; thus $A \setminus V$ represents the negation of V restricted to the state space of the system. The computation of $\mathsf{pred}_\exists(\bigvee\{c_d \in A \setminus V\})$ is much easier to compute (with available tools) than $\mathsf{pred}_\exists(\neg(\bigvee\{c_d \mid d \in V\}))$. The latter requires the projection operations proj to be applied to complex expressions of the form $\mathsf{proj}_{\bar{x}}(\neg(c_1(\bar{y}) \vee \cdots \vee c_k(\bar{y})) \wedge c(\bar{x}, \bar{y}))$, which involves expanding the expression (to d.n.f. for example); by contrast the former requires evaluation of simpler expressions of the form $\mathsf{proj}_{\bar{x}}(c_d(\bar{y}) \wedge c(\bar{x}, \bar{y}))$.

6.3.7 Further Optimisation by Pre-Computing Predecessor Constraints

We now show that we can improve the computation of the abstract function $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)$. Let $\{c_i\}$ be a set of constraints, each of which represents a set of points. It can easily seen that $\mathsf{pred}_\exists(\bigvee\{c_i\}) = \bigvee\{\mathsf{pred}_\exists(c_i)\}$. Consider the evaluation of $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)(V)$ where $V \in 2^A$ is a set of disjoint partitions represented by constraints.

$$\begin{aligned}(\alpha \circ \mathsf{pred}_\exists \circ \gamma)(V) &= (\alpha \circ \mathsf{pred}_\exists)(\bigvee\{c_d \mid d \in V\}) \\ &= \alpha(\bigvee\{\mathsf{pred}_\exists(c_d) \mid d \in V\})\end{aligned}$$

Give a finite partition A, we pre-compute the constraint $\mathsf{pred}_\exists(c_d)$ for all $d \in A$. Let $Pre(d)$ be the stored predecessor constraint for partition element d. The results can be stored as a table, and whenever it is required to compute $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)(V)$ where $V \in 2^A$, we simply evaluate $\alpha(\bigvee\{Pre(d) \mid d \in V\})$. The abstraction function α is evaluated efficiently using the SMT solver, as already discussed.

Note that expressions of the form $\alpha(\mathsf{pred}_\exists(\bigvee\{\cdots\}))$ occur in the transformed expression for $(\alpha \circ \mathsf{pred}_\forall \circ \gamma)(V)$ above. The same optimisation can be applied here too. Our experiments show that this usually yields a considerable speed-up (2-3 times faster) compared to dynamically computing the $\mathsf{pred}_\exists$ function during model checking.

6.4 Implications of Abstraction

Typically, the precision of (abstraction-based) verification techniques does suffer from the loss of information introduced by over-approximation. However the proposed techniques, because of the theory of abstract interpretation, are sound but not complete. That means, if a property ϕ is proved to hold in the abstraction, then, because of the soundness, the property is deemed to hold in the original system. However, because of the lack of completeness, we might not be able to conclude the correctness of each and every property with these techniques.

Abstraction and Synchronisation In the proposed framework, a system comprising more than one LHA is verified by first constructing the product of its constituent LHAs and then, depending on the property, one of the proposed techniques is applied. Thus, since the abstraction is done after the product LHA is constructed, synchronisation gets inbuilt into the constraints corresponding to (guards and actions of) the transitions of product LHA. However, loss of information due to abstraction can affect synchronisation in the same way that it can affect any other property of the system.

Summary

In this chapter, we presented the abstract model checking framework to verify arbitrary CTL formulas. Given the minimal model $\mathsf{M}[\![P^r_{Sys}]\!]$ or $\mathsf{M}^a[\![P^r_{Sys}]\!]$ corresponding to an LHA *Sys* and a CTL formula ϕ, abstract model checking involves the following steps:

1. Constructing the abstract state space `AStates` from $\mathsf{M}[\![P^r_{Sys}]\!]$ or $\mathsf{M}^a[\![P^r_{Sys}]\!]$ (as explained in Section 6.2.3). This abstract state space is a finite set of regions partitioning the reachable states.
2. Defining the Galois connection between the concrete domain $2^{\texttt{States}}$ and the abstract domain $2^{\texttt{AStates}}$ (as explained in Section 6.2.4);
3. Computing the abstract states $[\![\neg\phi]\!]^a$ by using the abstract CTL semantic function defined in Definition 63;
4. Giving a definition of α and γ in terms of constraint operations;
5. Computing the set $\gamma([\![\neg\phi]\!]^a) \cap \texttt{InitStates}$ where `InitStates` is the set of initial states of *Sys*.
   (a) If $\gamma([\![\neg\phi]\!]^a) \cap \texttt{InitStates} = \emptyset$[4] the property ϕ holds;
   (b) If $\gamma([\![\neg\phi]\!]^a) \cap \texttt{InitStates} \supseteq \emptyset$ and $\gamma([\![\phi]\!]^a) \cap \texttt{InitStates} \supseteq \emptyset$ the correctness of property ϕ is not known. Proving such properties often mandates refining the abstraction.

In the next chapter, we present case studies that make use of AMC.

[4]Alternatively, because of the Galois connection, a property ϕ holds if $\alpha(\texttt{InitStates}) \cap [\![\neg\phi]\!]^a = \emptyset$.

Chapter 7

Experiments

This chapter describes experiments with the modelling and verification techniques proposed in this dissertation. We chose some example systems from the literature for experimentation.

The method described in this chapter is a walk through of the framework outlined in Figure 1.1 of Chapter 1. It comprises translating LHA into CLP, reachability analysis and model checking. In the translation step, we employ program transformation techniques, namely, compilation and partial evaluation. The reachability analysis step is based on the static analysis as explained in Chapter 5. The model checking step applies the abstract model checking that was presented in Chapter 6.

Program compilation is a technique to transform a *source program* in one language into an equivalent *target program* in another language. The languages of source and target programs are called source and target languages respectively. Compilation typically involves parsing and code generation. The LHA models are graphical and can be encoded in a textual language. We compile these LHA text programs into constraint logic program. The code generation step automates the translation scheme explained in Chapter 3.

Partial evaluation is a technique of specialising a program with respect to one or more of its inputs. The generic LHA driver is specialised w.r.t. to a given LHA model. The resulting specialised program is the constraint logic program capturing the state transition semantics of the LHA model. We use the LOGEN partial evaluator [87] to do this specialisation.

Static analysis is a compile-time technique to analyse the run-time behaviour of a program. For a constraint logic program, its minimal model includes all the possible run-time behaviours. A minimal model is computed following the iterative algorithms presented in Chapter 2.

Model checking is a technique to verify temporal properties of reactive systems. Based on the results from static analysis, certain temporal properties could be verified following the abstract model checking technique that was presented in the previous chapter.

Chapter Overview

- Section 7.1 presents the tools to realise the translation of an LHA into CLP;
- Section 7.2 presents the tools to realise the proof techniques I and II;
- Section 7.3 presents experiments of verifying various systems.

7.1 Tools in the Framework

In what follows are explained how the translation of LHA models into CLP programs and the proof techniques (presented in the previous chapter) are accomplished with a chain of tools implemented in CLP. Some of these tools are existing general purpose CLP program analysis tools; while some are developed for the purpose of analysing LHA models and model checking transition systems.

Translating LHA models into CLP programs

7.1.1 Text LHA language

Actually the language of LHA is *graphical.* To encode such graphical LHA models as text programs, we defined a simple text language named *Text LHA*. The syntax for this language is shown in Figure 7.1.

The following example (Example 36) illustrates the textual LHA LHA_{text} encoding a graphical LHA model.

Example 36. *Consider the water-level monitoring system that was introduced in Example 15 (Figure 3.8). Recall that this monitor is expected to maintain the water-level (W) between 1 and 12 (inclusive). The* LHA_{text} *textual encoding of this LHA is shown in Figure 7.2.* □

7.1.2 LHA2CLP **Compiler**

The LHA to CLP translation scheme presented in Chapter 3 is mechanised with a tool written in Ciao Prolog. This tool, named LHA2CLP compiler, compiles a Text LHA program into a CLP program. The parser part, which parses a Text LHA program, of this compiler is generated using the JavaCC tool.

Recall that across different LHA models, only the definitions for predicates `alpha`/4, `gamma`/3, `inv`/1, etc. change; while the definitions of the `rstate`/1,

```
LHA ::= <Var_Decl>+ <Event_Decl>* <Loc_Decl>+ <Trans_Decl>+

Var_Decl ::= 'variable(' VariableType ',' VariableID ').'
Event_Decl ::= 'event(' EventID ').'
Loc_Decl ::= 'location(' LocID ',' <RateDecl> ',' <InvariantDecl> ').'
Trans_Decl ::= 'transition(' <LocPair> ',' <GuardDecl> ',' <ActionDecl>
               ').'

VarType ::= 'numeric'
RateDecl ::= '(' <RateRel> ')' | '(' <RateRelComma>+ <RateRel>')'
RateDeclComma ::= <RateRel> ','
RateRel ::= 'rate(' VariableID ')' <LinRelSymb> <LinExpr>
InvariantDecl ::= '(' <LinRel> ')' | '(' <LinRelAndOr>+ <LinRel>')'
LocPair ::= '(' LocID ',' LocID ')'
GuardDecl ::= '(' <LinRel> ')' | '(' <LinRelComma>+ <LinRel>')'
ActionDecl ::= '()' | '(' <LinRel> ')' | '(' <LinRelComma>+ <LinRel>')'
               | '(' <LinRelComma>* <EventDecl>')'
EventDecl ::= EventID ':= 1'

LinRelComma ::= <LinRel> ','
LinRelAndOr ::= <LinRel> '&' | <LinRel> '|'
LinRel ::= 'true' | <LinExpr> <LinRelSymb> <NUMBER>
LinExpr ::= NUMBER | DECIMALNUMBER | VariableID
            | <Expression> <ArithOp> <Expression>
LinRelSymnb ::= '>' | '>=' | '<' | '=<' | '=' | '=='
ArithOp ::= '+' | '-' | '*'

NUMBER ::= ['1'-'9'] ['0'-'9']* | '0'
DECIMALNUMBER ::= <NUMBER> '.'  <NUMBER>
EventID ::= 'event_'<LETTER> (<LETTER>|<DIGIT>)*
LocID ::= 'loc_' <NUMBER>
VariableID ::= <LETTER> (<LETTER>|<DIGIT>)*
LETTER ::= (['A'-'Z'] | ['a'-'z'] | "_")
DIGIT ::= ['0'-'9']
```

Figure 7.1: The Syntax of the Text-LHA language

```
variable(numeric,x).
variable(numeric,w).

location(loc_0,(rate(x)=+1,rate(w)=+1),(w<10)).
location(loc_1,(rate(x)=+1,rate(w)=+1),(x<2)).
location(loc_2,(rate(x)=+1,rate(w)=-2),(w>5)).
location(loc_3,(rate(x)=+1,rate(w)=-2),(x<2)).

init(loc_0,(x=0,w=0)).

transition((loc_0,loc_1),(w==10),(x=0)).
transition((loc_1,loc_2),(x==2),()).
transition((loc_2,loc_3),(w==5),(x=0)).
transition((loc_3,loc_0),(x==2),()).
```

Figure 7.2: The textual encoding of the water-level LHA model

`qstate`/1 and `transition`/2 predicates remain the same. So the LHA2CLP compiler generates first the CLP program fragment defining all the predicates appearing in the transition predicate `transition`/2 and then appends either the reachable states driver (of Figure 3.9) or the reaching states driver (of Figure 3.10) to this CLP fragment resulting in the total CLP program encoding of a given LHA model. We have two versions of this LHA2CLP compiler, namely, LHA2CLP$_R$ and LHA2CLP$_Q$.

The LHA2CLP$_R$ compiler generates the CLP program encoding an LHA with the reachable states driver; while LHA2CLP$_Q$ generates the CLP program encoding an LHA with the reaching states driver. We need to specify the initial state (resp. target state) in the case of the LHA2CLP$_R$ (resp. LHA2CLP$_Q$) compiler. In this chapter, as we only focus on the forward reasoning driver we do not demonstrate the LHA2CLP$_Q$ compiler.

Example 37. *The* LHA2CLP$_R$ *compiler, when input with the textual LHA program from the previous example, generates the CLP program as shown in Figure 7.3. In this CLP program, the list argument corresponds to the state tuple. The first element of this list is the location variable, the next two elements are the state variables x, w (declared in the Text LHA program) and the last element is the time variable t.*

```
locationOf([Loc,_,_,_],Loc).

before([_,_,_,C],[_,_,_,F]) :-     C=<F.

stateSpace([Loc,_,_,_]) :-     location(Loc).

location(loc_0).
location(loc_1).
location(loc_2).
location(loc_3).
init([loc_0,+0,+0,0]).

invariant(loc_0,[loc_0,_,B,_]) :-        B< +10.
invariant(loc_1,[loc_1,A,_,_]) :-        A< +2.
invariant(loc_2,[loc_2,_,B,_]) :-        B> +5.
invariant(loc_3,[loc_3,A,_,_]) :-        A< +2.

d([loc_0,A,B,C],[_,D,E,F]) :-
       1*D=1*A+1*(F-C),
       1*E=1*B+1*(F-C).
d([loc_1,A,B,C],[_,D,E,F]) :-
       1*D=1*A+1*(F-C),
       1*E=1*B+1*(F-C).
d([loc_2,A,B,C],[_,D,E,F]) :-
       1*D=1*A+1*(F-C),
       1*E=1*B+ -2*(F-C).
d([loc_3,A,B,C],[_,D,E,F]) :-
       1*D=1*A+1*(F-C),
       1*E=1*B+ -2*(F-C).

gamma(0,loc_0,[loc_1,_,B,_]) :-        B= +10.
gamma(1,loc_1,[loc_2,A,_,_]) :-        A= +2.
gamma(2,loc_2,[loc_3,_,B,_]) :-        B= +5.
gamma(3,loc_3,[loc_0,A,_,_]) :-        A= +2.

alpha(0,loc_0,[loc_1,A,B,_],[loc_1,G,H,0]) :-    G= +0,    H=B.
alpha(1,loc_1,[loc_2,A,B,_],[loc_2,G,H,0]) :-    G=A,      H=B.
alpha(2,loc_2,[loc_3,A,B,_],[loc_3,G,H,0]) :-    G= +0,    H=B.
alpha(3,loc_3,[loc_0,A,B,_],[loc_0,G,H,0]) :-    G=A,      H=B.
```

Figure 7.3: The constraint logic program encoding the LHA in Figure 3.8

Specialisation

In this step, we specialise the CLP program (encoding a given LHA) to reduce the computational expenses to be incurred in the verification phase.

Recall (from both the verification recipes of Chapter 5 and the model checking recipe of Chapter 6) that proving a property mandates computing the minimal model of the CLP program encoding the LHA to be verified. The minimal model is computed, because the set of reachable (resp. reaching states) is given by the ground instances of the `rstate`/1 (resp. `qstate`/1) predicate. But (recall from Chapter 3) the complete CLP model CLP_{LHA} of an LHA model LHA is the union of clauses defining the predicates `transition`/2, `d`/1, `alpha`/2, `gamma`/2, `invariant`/2, `before`/2, and either the `rstate`/1 and `init`/1 or `qstate`/1 and `target`/1. Thus computing the minimal model $\mathsf{M}[\![CLP_{LHA}]\!]$ means computing the ground instances for all the mentioned predicates. But, since we are only interested in instances of either the `rstate`/1 (or `qstate`/1) predicate, computing the ground instances for uninteresting predicates could be avoided to reduce the computation costs. This is achieved by first specialising the CLP_{LHA} into CLP_{Spec}, which contains only the specialised versions of `rstate`/1 (or `qstate`/1) predicate, and then computing the minimal model $\mathsf{M}[\![CLP_{Spec}]\!]$. We use an existing CLP tool called LOGEN to specialise CLP_{LHA} into CLP_{spec}.

To get an idea of the computational efficiency we gain by specialisation, consider the computation of ground instances for the `transition`/2 predicate in the CLP_{LHA} program. This involves computing *all* the states, both reachable and unreachable, that are connected with a transition. Besides this computational efficiency, we also gain precision while computing the abstract minimal model $\mathsf{M}^a[\![LHA_{spec}]\!]$.

7.1.3 LOGEN

LOGEN is a partial evaluator that specialises a CLP program according to an annotation for this CLP program. It accepts two programs, (a) the CLP program to be specialised and (b) the annotation of that program, and outputs the specialised CLP program.

We could view the specialisation of the CLP program CLP_{LHA} encoding an LHA model as partially evaluating the reachable states driver (resp. reaching states driver) w.r.t the *transition relation* (i.e. predicate `transition`/2) corresponding to a given LHA model.

The following example (Example 38) illustrates the use of LOGEN partial evaluator.

Example 38. *The partial evaluator* LOGEN *when input with two files: (a) the CLP_{LHA} program shown in Figure 7.3 and (b) the annotation for the CLP_{LHA} as* *shown in Figure 7.4 outputs the specialised program CLP_{spec} seen in Figure 7.5. The annotation file is also mechanically generated by the* LHA2CLP *compiler. The* LOGEN *partial evaluator even specialises the* `rstate`/1 *predicate with respect to the locations. Hence in the specialised program there are four predicates* `rstate__`1*,* `rstate__`2*,* `rstate__`4 *and* `rstate__`4*, which are specialised versions of* `rstate`/1 *with respect to the locations loc_0, loc_1, loc_2 and loc_3, respectively. Besides this,* LOGEN *also flattens the list (of state variables) and changes the order of variables. The location variable which was the first element in the list is placed as the last argument. So in this specialised program, arguments A, B, C, D corresponds to x, w, t and location, respectively. The last argument D in the predicate* $\texttt{rstate}_i/4$ *(for i= 1 to 4) takes the value $D = i + 3$.*

7.2 Proof techniques

7.2.1 Proof techniques I

The verification recipes presented in Chapter 5 essentially involve two steps: (i) computing the minimal model of the CLP program encoding the LHA to be verified; (ii) querying this minimal model with an appropriate goal. We use two *general purpose* CLP program analyses tools, namely TPCLP *analyser* and CHA *analyser* to accomplish the first step; while we use a CLP run time system to accomplish the second step.

Minimal model computation

TPCLP **analyser**

This tool computes the concrete minimal model (CMM) of a CLP program. It implements the Algorithm 1, where the transfer function is as defined in Definition 31, to compute this minimal model. This TPCLP analyser when input with a CLP program outputs its minimal model. We implemented this tool in Ciao Prolog [20] and makes use of the Parma Polyhedra Library [11].

Furthermore, TPCLP tool can analyse only the CLP programs with flat terms i.e. the arguments in the program clauses should only be variables and/or constants but not lists. Whenever there are list arguments in the program, the LOGEN tool, besides specialising the program, also flattens such lists. Therefore, the specialised programs from LOGEN can be analysed with this TPCLP tool.

Example 39 illustrates the application of TPCLP *analyser*.

Example 39. *The* TPCLP *tool when input with the specialised program of Figure 7.5 outputs the minimal model shown in Figure 7.6. The constrained atoms with*

```
logen(locationOf/2,locationOf([Loc,_,_,_],Loc)).
logen(before/2,before([_,_,_,C],[_,_,_,F])) :- logen(rescall,C=<F).
logen(stateSpace/1,stateSpace([Loc,_,_,_])) :-
                              logen(unfold,location(Loc)).
logen(location/1,location(loc_0)).
logen(location/1,location(loc_1)).
logen(location/1,location(loc_2)).
logen(location/1,location(loc_3)).
logen(init/1,init([loc_0,+0,+0,0])).

logen(invariant/2,invariant(loc_0,[loc_0,_,B,_])) :-
       logen(rescall,B< +10).
logen(invariant/2,invariant(loc_1,[loc_1,A,_,_])) :-
       logen(rescall,A< +2).
logen(invariant/2,invariant(loc_2,[loc_2,_,B,_])) :-
       logen(rescall,B> +5).
logen(invariant/2,invariant(loc_3,[loc_3,A,_,_])) :-
       logen(rescall,A< +2).

logen(d/2,d([loc_0,A,B,C],[_,D,E,F])) :-
       logen(rescall,1*D=1*A+1*(F-C)), logen(rescall,1*E=1*B+1*(F-C)).
logen(d/2,d([loc_1,A,B,C],[_,D,E,F])) :-
       logen(rescall,1*D=1*A+1*(F-C)), logen(rescall,1*E=1*B+1*(F-C)).
logen(d/2,d([loc_2,A,B,C],[_,D,E,F])) :-
       logen(rescall,1*D=1*A+1*(F-C)), logen(rescall,1*E=1*B+ -2*(F-C)).
logen(d/2,d([loc_3,A,B,C],[_,D,E,F])) :-
       logen(rescall,1*D=1*A+1*(F-C)), logen(rescall,1*E=1*B+ -2*(F-C)).

logen(gamma/3,gamma(0,loc_0,[loc_1,_,B,_])) :- logen(rescall,B= +10).
logen(gamma/3,gamma(1,loc_1,[loc_2,A,_,_])) :- logen(rescall,A= +2).
logen(gamma/3,gamma(2,loc_2,[loc_3,_,B,_])) :- logen(rescall,B= +5).
logen(gamma/3,gamma(3,loc_3,[loc_0,A,_,_])) :- logen(rescall,A= +2).

logen(alpha/4,alpha(0,loc_0,[loc_1,A,B,_],[loc_1,G,H,0])) :-
       logen(rescall,G= +0), logen(rescall,H=B).
logen(alpha/4,alpha(1,loc_1,[loc_2,A,B,_],[loc_2,G,H,0])) :-
       logen(rescall,G=A), logen(rescall,H=B).
logen(alpha/4,alpha(2,loc_2,[loc_3,A,B,_],[loc_3,G,H,0])) :-
       logen(rescall,G= +0), logen(rescall,H=B).
logen(alpha/4,alpha(3,loc_3,[loc_0,A,B,_],[loc_0,G,H,0])) :-
       logen(rescall,G=A), logen(rescall,H=B).
```

Figure 7.4: The annotation of the constraint logic program of Figure 7.3

```
rstate__1(A,B,C,D):-
       D=4, 0=<C, 1*A=1*0+1*(C-0),
       1*B=1*0+1*(C-0), B<10.

rstate__1(A,B,C,D):-
       rstate__4(E,F,G,H),
       D=4, H=7, G=<I, 1*J=1*E+1*(I-G),
       1*K=1*F+ -2*(I-G) J=2, L=J, M=K,
       0=<C, 1*A=1*L+1*(C-0),
       1*B=1*M+1*(C-0), B<10.

rstate__2(A,B,C,D):-
       rstate__1(E,F,G,H),
       D=5, H=4, G=<I,
       1*J=1*E+1*(I-G),
       1*K=1*F+1*(I-G),
       K=10, L=0, M=K,
       0=<C,
       1*A=1*L+1*(C-0),
       1*B=1*M+1*(C-0),
       A<2.

rstate__3(A,B,C,D):-
       rstate__2(E,F,G,H),
       D=6, H=5, G=<I,
       1*J=1*E+1*(I-G),
       1*K=1*F+1*(I-G),
       J=2,L=J,M=K,0=<C,
       1*A=1*L+1*(C-0),
       1*B=1*M+ -2*(C-0),
       B>5.

rstate__4(A,B,C,D):-
       rstate__3(E,F,G,H),
       D=7, H=6, G=<I,
       1*J=1*E+1*(I-G),
       1*K=1*F+ -2*(I-G),
       K=5, L=0, M=K, 0=<C,
       1*A=1*L+1*(C-0),
       1*B=1*M+ -2*(C-0),
       A<2.
```

Figure 7.5: The specialised forward reasoning driver w.r.t the LHA of Figure 7.3.

```
rstate__1(A,B,C,D) :-
       D=1, -1*A> -10, 1*A>=0, 1*A+ -1*B=0, 1*A+ -1*C=0.
rstate__2(A,B,C,D) :-
       D=2, -1*C> -2, 1*C>=0, 1*A+ -1*C=0, 1*B+ -1*C=10.
rstate__3(A,B,C,D) :-
       D=3, -2*C> -7, 1*C>=0, 1*A+ -1*C=2, 1*B+2*C=12.
rstate__4(A,B,C,D) :-
       D=4, -1*C> -2, 1*C>=0, 1*A+ -1*C=0, 1*B+2*C=5.
rstate__1(A,B,C,D) :-
       D=1, -1*C> -9, 1*C>=0, 1*A+ -1*C=2, 1*B+ -1*C=1.
```

Figure 7.6: The minimal model of the program modelling the water-level monitor.

heads `rstate__`*i (for i= 1 to 4) identifies the set of reachable states corresponding to the location loc_i* $-$ 1.

CHA **Analyser**

We use an *already existing* tool called CHA *analyser* developed by Kim Henriksen [58] to compute the abstract minimal model (AMM) of a CLP program. It implements the iterative fixed point algorithm where the transfer function is the abstract semantics function as defined in [58]. This CHA analyser when input with a CLP program outputs its abstract minimal model. The application of this tool is presented in a later section.

Querying the minimal model

So computed minimal model (whether concrete or abstract) is queried with a goal corresponding to the existential-liveness or universal-safety property. We use the SICStus Prolog with the CLPQ package to query the minimal model.

Safety property verification: Recall (from Chapter 5) that to verify the safety property AGp, the minimal model is is queried with a goal of the form

$\leftarrow c_{\neg p}(\bar{X}), \texttt{rstate}(\bar{X})$

where $c_{\neg p}(\bar{X})$ is the constraint corresponding to the proposition $\neg p$. The failure of this goal means there is no reachable state where the safety is violated and hence the safety property holds. To verify the existential property *EFp*, the goal is of the form

$\leftarrow c_p(\bar{X}), \texttt{rstate}(\bar{X})$

where $c_p(\bar{X})$ is the constraint encoding the proposition p. This is illustrated in the following example (Example 40). As mentioned earlier, the LOGEN tool even specialises the `rstate`/1 predicate. Consequently, the minimal model has constrained atoms corresponding to each of the specialised version of the `rstate`/1 predicate. So if there are n locations then the minimal model is comprised by the constrained facts with the heads `rstate__1`/1, ..., `rstate__`n/1. Then the atom `rstate`/1 in the goal is replaced with the disjunction $\bigvee_{i=1}^{i=n}(\texttt{rstate\_\_}i(\bar{X}))$.

Example 40. *Consider the following two properties of the water-level monitor: (i) $AG(w \leq 12)$ and (ii) $EF(w = 10)$. The safety property is verified by querying the minimal with the goal:*

$$\leftarrow (\texttt{rstate\_\_1}(\_, W, \_, \_) \vee \texttt{rstate\_\_2}(\_, \_, W, \_) \vee \texttt{rstate\_\_3}(\_, \_, W, \_) \vee \texttt{rstate\_\_4}(\_, W, \_, \_)) \wedge (W > 12)$$

where W is the variable corresponding to water level w. To verify the liveness property, the constraint in the previous goal is replace with the constraint $W = 10$. Since the minimal is queried using the SICStus Prolog system, first the minimal model is rewritten in a form accepted by SICStus and is queried with the mentioned goals. The goal corresponding to $AG\ W > 12$ (resp. $EF\ W = 10$) fails (resp. holds), hence the property is proved to hold.

The tool chain to accomplish the verification recipes of Chapter 5 is shown in Figure 7.7. Whenever TPCLP does not terminate, the TPCLP tool in the tool chain is replaced with the CHA tool to compute the abstract minimal model. To handle such cases, the TPCLP tool in the tool chain is replaced with the CHA tool. The computation times of the TPCLP and CHA tools are summarised in Tables 7.5 and 7.6, respectively.

7.2.2 Proof techniques II

The verification recipes explained in Chapter 6 make use of the model checking algorithms. The essential steps in these recipes are: (i) construct the abstract state space and the Galois connection; (ii) perform the abstract model checking following the verification recipes presented in Chapter 6.

7.2.3 Abstract domain construction step

In this step, the abstract state space is constructed as explained in Chapter 6. This construction phase is also integrated into the TPCLP tool. Recall that the minimal model of the specialised program in minimal model computation phase is a set of constraint atoms of the form $\texttt{rstate}(\bar{X}) \leftarrow c_1(\bar{X}), \ldots, \texttt{rstate}(\bar{X}) \leftarrow c_n(\bar{X})$. The TPCLP tool defines the abstract domain whose elements are the regions defined by each constraint $c_i(\bar{X})$ $(1 \leq i \leq n)$. Example 41 illustrate the construction of abstract domain. Defining the Galois connections becomes trivial

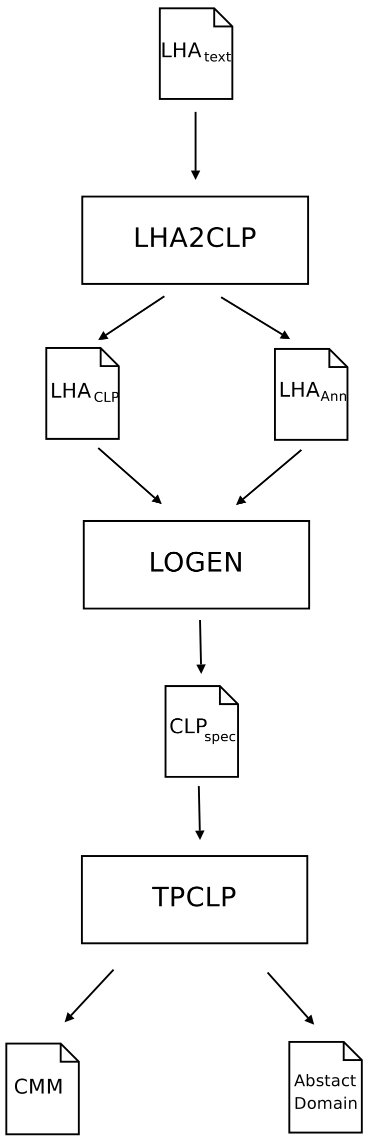

Figure 7.7: Minimal model tool chain

```
 version(1,rstate__1(A,B,C,D),
         [D=4,-1*A> -10,1*A>=0,1*A+ -1*B=0,1*A+ -1*C=0]).
version(2,rstate__2(A,B,C,D),
         [D=5,-1*C> -2,1*C>=0,1*A+ -1*C=0,1*B+ -1*C=10]).
version(3,rstate__3(A,B,C,D),
         [D=6,-2*C> -7,1*C>=0,1*A+ -1*C=2,1*B+2*C=12]).
version(4,rstate__4(A,B,C,D),
         [D=7,-1*C> -2,1*C>=0,1*A+ -1*C=0,1*B+2*C=5]).
version(5,rstate__1(A,B,C,D),
         [D=4,-1*C> -9,1*C>=0,1*A+ -1*C=2,1*B+ -1*C=1]).
```

Figure 7.8: The set of abstract states corresponding to minimal model of Figure 7.6

if the abstract state space is a partition of the concrete state space. For this reason we can construct abstract state space that is a partition.

Example 41. *The set of abstract reachable states generated by* TPCLP *is shown Figure 7.8. So there are five abstract states in this case, one each corresponding to the five constrained facts in the minimal model. Each of the version/3 fact defines an abstract state. The first argument is the identifier for the abstract state; the second argument identifies the location of the abstract state; while the third argument, which is a list, defines the set of concrete states abstracted by this abstract state.*

In the above example, the abstract states were disjoint[1]. But we do not always get a *disjoint* abstract state space. Several refinement and reduction strategies exist to construct a disjoint partition from a non-disjoint partition. In this dissertation, we consider a simple strategy. When the states are not disjoint, we define a new abstract state as the union of the overlapping abstract states and thus define a new disjoint abstraction. Such new abstract state abstracts the union of the sets of concrete states abstracted by the non-disjoint abstract states.

Abstract model checking step

In this phase, the system is model checked for a given CTL formula.

AMC Tool An abstract model checking tool named *AMC* is written by implementing the abstract CTL semantics function of Definition 63. This tool accepts

[1]Two abstract states are said to be disjoint if their correspondent sets of concrete sets are disjoint.

three inputs: (i) the specialised transition system CLP_{Spec} from LOGEN ; (ii) the minimal model $\mathsf{M}[\![CLP_{Spec}]\!]$ output from the TPCLP or CHA tool and (iii) the CTL formula ϕ to be checked, and returns the set of abstract states, $[\![\neg\phi]\!]^a$, where the negated CTL formula $\neg\phi$ holds. This tool is also written in Ciao Prolog and makes use of the Yices SMT solver and the Parma Polyhedra Library (as explained in 6.3.5). The following example illustrates the application of the AMC.

Example 42. *We verify the water-level monitor for the following properties:*

1. $AF(w \geq 10)$*: This formula expresses the property of "on all paths the water level (W) eventually reaches at least a level of 10".*
2. $EF(w = 10)$*: This formula expresses the property of "on some path w equals 10".*
3. $AG(0 \leq w \wedge w \leq 12)$*: This formula expresses the safety property of "always globally w remains between 0 and 12"*
4. $AF(AG(1 \leq w \wedge w \leq 12))$*: This formula states that "on all paths in future the water level ranges between 1 and 12".*
5. $AG(w = 10 \rightarrow AF(w < 10 \vee w > 10))$*: This formula states that "on every path the water level cannot get stuck at 10".*
6. $EU(w < 12, AU(w < 12, w \geq 12))$*: This formula states that "there is a path such that W remains less than 12 until it hits 12".*
7. $AG(AG(AG(AG(AG(0 \leq w \wedge w \leq 12)))))$*: This formula is equivalent to the safety formula* $AG(0 \leq w \wedge w \leq 12)$*.*

To model check the water level monitor, the abstract model checker is input with the three inputs, namely:

1. *the specialised program of Figure 7.5;*
2. *the set of abstract reachable state space that is derived (as explained in 6) from the minimal model of Figure 7.6;*
3. *the CTL formula to be verified.*

The $[\![\neg\phi]\!]^a$ *output for each of the properties is shown in Table 7.1.*

The abstract state corresponding to the initial state is 1 *i.e.* $\mathsf{astates}(w = 0 \wedge x = 0) = \{1\}$*. Following the abstract model checking recipe, for the first four formulas in the table (Table 7.1), since* $[\![\neg\phi]\!]^a \cap \{1\} = \emptyset$*, the formulas are proved to hold; while for the fifth and seventh formulas since* $[\![\neg\phi]\!]^a \cap \{1\} \neq \emptyset$ *we cannot conclude anything about their correctness. For the sixth formula, we have not proved whether the state where* $w = 10$ *is actually visited, though* $[\![\neg\phi]\!]^a \cap \{1\} = \emptyset$ *we cannot conclude that the property holds. This result is trivial if* $w = 10$ *is never reached.*

ϕ	$[\![\neg\phi]\!]^a$	Time (secs.)	$[\![\neg\phi]\!]^a \cap I^a$
$AF(w \geq 10)$	$\emptyset$	0.015	$\emptyset$
$AG(0 \leq w \wedge w \leq 12)$	$\emptyset$	0.011	$\emptyset$
$AF(AG(1 \leq w \wedge w \leq 12))$	$\emptyset$	0.018	$\emptyset$
$AG(AG(AG(AG(AG(0 \leq w \wedge w \leq 12)))))$	$\emptyset$	0.021	$\emptyset$
$EF(w = 10)$	$\{1,2,3,4,5\}$	0.009	$\{1\}$
$AG(w = 10 \rightarrow AF(w < 10 \vee w > 10))$	$\emptyset$	0.020	$\emptyset$
$EU(w < 12, AU(w < 12, w \geq 12))$	$\{1,2,3,4,5\}$	0.020	$\{1\}$

Table 7.1: AMC Results for water-level monitor

7.2.4 Abstract Domain Refinement

With our abstract model checking, due to coarseness of the abstraction, it is very much possible that the correctness of a property cannot be established. An abstraction is deemed to be imprecise for a property ϕ if and only if $I \in \gamma([\![\phi]\!]^a \cap [\![\neg\phi]\!]^a)$ or $\alpha(I) \in [\![\phi]\!]^a \cap [\![\neg\phi]\!]^a$ where α and γ are the abstraction and concretisation functions defining the Galois connection.

In Example 42, due to the imprecise abstraction, the correctness of the formulas $EF(w = 10)$ and $EU(w < 12, AU(w < 12, w \geq 12))$ cannot concluded. To model check a formula of the form $EF(w = VALUE)$, it becomes necessary to refine the abstract state where w can take a value of $VALUE$ into three abstract states. The reason to do this refinement is that the transition relation we define does permit jumps between the states that are visited via delay transitions within the same location. Such jumps imply that it is possible to bypass the intermediate states that are actually visited in the original system. This is the loss of precision due to abstraction and our transition relation encoding. Therefore, though $w = 10$ is visited in the original system, in our encoding this state where $w = 10$ is bypassed. But by partitioning those abstract states where $w = 10$ and refining such states, our transition relation enforces that this state be visited. The refined abstraction, which is a refinement of the abstraction defined in Figure 7.8, is given in Figure 7.9.

Similarly the formula $EU(W < 12, AU(W < 12, W \geq 12))$ can be model checked by refining the original abstraction with respect to $W = 12$. With these refined abstractions, the AMC results for the formulas $EF(W = 10)$, $AG(W = 10 \rightarrow AF(W < 10 \vee W > 10))$ and $EU(W < 12, AU(W < 12, W \geq 12))$ are shown in Table 7.2. Since ${}^a[\![\neg\phi]\!] = \emptyset$, with the refined abstractions, these three properties are proved to hold.

```
version(v1_0,rstate__1(A,B,C,D),[10>B,D=4,-1*A> -10,1*A>=0,1*A+
-1*B=0,1*A+ -1*C=0]).
version(v2_0,rstate__2(A,B,C,D),[B=10,D=5,-1*C> -2,1*C>=0,1*A+
-1*C=0,1*B+ -1*C=10]).
version(v2_1,rstate__2(A,B,C,D),[B>10,D=5,-1*C> -2,1*C>=0,1*A+
-1*C=0,1*B+ -1*C=10]).
version(v3_0,rstate__3(A,B,C,D),[B=10,D=6,-2*C> -7,1*C>=0,1*A+
-1*C=2,1*B+2*C=12]).
version(v3_1,rstate__3(A,B,C,D),[B>10,D=6,-2*C> -7,1*C>=0,1*A+
-1*C=2,1*B+2*C=12]).
version(v3_2,rstate__3(A,B,C,D),[10>B,D=6,-2*C> -7,1*C>=0,1*A+
-1*C=2,1*B+2*C=12]).
version(v4_0,rstate__4(A,B,C,D),[10>B,D=7,-1*C> -2,1*C>=0,1*A+
-1*C=0,1*B+2*C=5]).
version(v5_0,rstate__1(A,B,C,D),[10>B,D=4,-1*C> -9,1*C>=0,1*A+
-1*C=2,1*B+ -1*C=1]).
```

Figure 7.9: Refinement of the abstraction of 7.6

.

7.3 Verification of other systems

7.3.1 Task scheduler

In this section, we verify the correctness of a task scheduler against three CTL formulas.

A task scheduler is that component of a multi-tasking real-time operating system which is responsible for scheduling the application tasks to meet their deadlines. We consider a task scheduler defined in [7]. This system has two tasks, namely *task*1 and *task*2, with the execution times of 4 and 8 seconds, respectively. These tasks are scheduled in response to the interrupts. There are two interrupts, namely *interrupt*1 and *interrupt*2. The *interrupt*1 might arrive *at most* every 10 seconds; while *interrupt*2 arrives *at most* every 20 seconds. The *interrupt*1 and

ϕ	$[\![\neg\phi]\!]^a$	Time (secs.)
$EF(W = 10)$	$\emptyset$	0.028
$AG(W = 10 \rightarrow AF(W < 10 \vee W > 10))$	$\emptyset$	0.032
$EU(W < 12, AU(W < 12, W \geq 12))$	$\emptyset$	0.041

Table 7.2: AMC Results for water-level monitor with refined abstractions

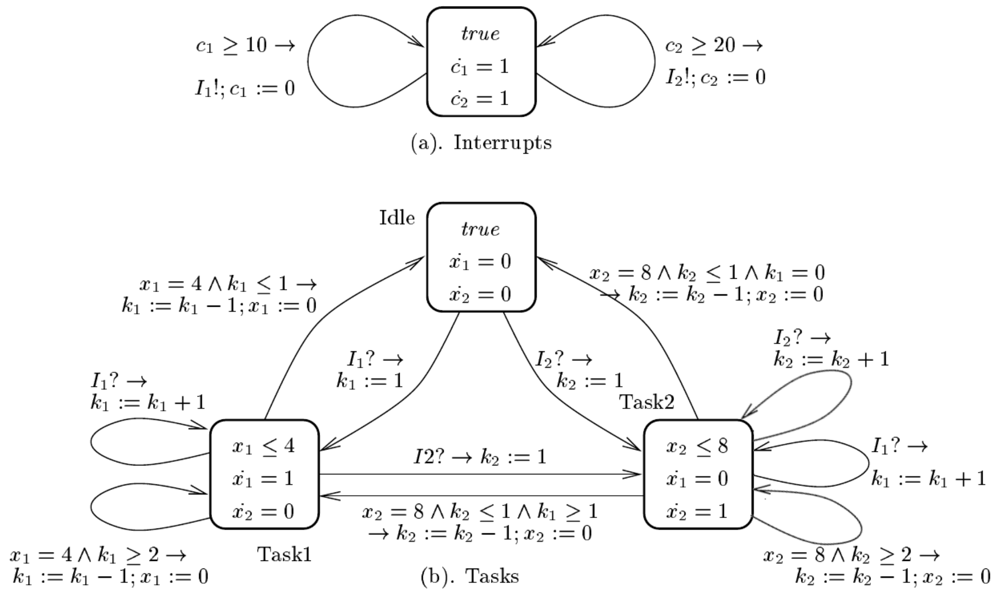
$c_1 \geq 10 \rightarrow$
$I_1!; c_1 := 0$
$true$
$\dot{c_1} = 1$
$\dot{c_2} = 1$
$c_2 \geq 20 \rightarrow$
$I_2!; c_2 := 0$
(a). Interrupts
Idle
$true$
$\dot{x_1} = 0$
$\dot{x_2} = 0$
$x_1 = 4 \wedge k_1 \leq 1 \rightarrow$
$k_1 := k_1 - 1; x_1 := 0$
$x_2 = 8 \wedge k_2 \leq 1 \wedge k_1 = 0$
$\rightarrow k_2 := k_2 - 1; x_2 := 0$
$I_1? \rightarrow$
$k_1 := 1$
$I_2? \rightarrow$
$k_2 := 1$
$I_2? \rightarrow$
$k_2 := k_2 + 1$
$I_1? \rightarrow$
$k_1 := k_1 + 1$
Task2
$x_1 \leq 4$
$\dot{x_1} = 1$
$\dot{x_2} = 0$
$I2? \rightarrow k_2 := 1$
$x_2 \leq 8$
$\dot{x_1} = 0$
$\dot{x_2} = 1$
$I_1? \rightarrow$
$k_1 := k_1 + 1$
$x_2 = 8 \wedge k_2 \leq 1 \wedge k_1 \geq 1$
$\rightarrow k_2 := k_2 - 1; x_2 := 0$
$x_1 = 4 \wedge k_1 \geq 2 \rightarrow$
$k_1 := k_1 - 1; x_1 := 0$
Task1
(b). Tasks
$x_2 = 8 \wedge k_2 \geq 2 \rightarrow$
$k_2 := k_2 - 1; x_2 := 0$

Figure 7.10: Tasks and Scheduler LHA models

*interrupt*2 are serviced with *task*1 and *task*2 respectively. When the high priority task *task*2 is requested while the low priority task *task*1 is running, *task*1 is pre-empted and *task*2 begins executing. Figure 7.10 shows the LHA model (taken from [57]) for the interrupts and the tasks.

In the LHA model, the variables:

1. $C1, C2$ model the clocks corresponding to the two interrupts *interrupt*1 and *interrupt*2, respectively;
2. $X1, X2$ model the clocks that keep track of the lapsed execution times corresponding to the tasks *task*1 and *task*2, respectively;
3. $K1, K2$ are the number of requests for the tasks *task*1 and *task*2, respectively.

We verify the following properties of this system:

1. $EF(K2 = 1)$: In future eventually the high priority task *task*2 is scheduled.
2. $AG(K2 > 0 \rightarrow AF(K2 = 0))$: Always the high priority task when scheduled will always complete its execution.
3. $AG(K2 \leq 1)$: Always globally the number of requests for the higher priority task never exceeds 1.

ϕ	$[\![\neg\phi]\!]^a$	Time (secs.)
$EF(K2 = 1)$	$\emptyset$	0.208
$AG(K2 \leq 1)$	$\emptyset$	0.031
$AG(K2 > 0 \rightarrow AF(K2 = 0))$	$\emptyset$	0.146

Table 7.3: AMC Results for the Scheduler properties

As illustrated in the verification of water-level system, the textual-LHA model $Sch_{textLHA}$ of this scheduler is first input to the LHA2CLP$_R$ compiler, which outputs both the CLP program and its annotation. These two outputs are fed into $LOGEN$ tool, which generates the specialised program. The TPCLP when input with this file generates: (i) the reachable states file; (ii) the abstract states file.

To do reachability analysis we load the minimal model $[\![Spec_{Sch}]\!]$ into SICStus and query for the properties to be checked; while to do model checking the AMC tool is input with the specialised program $Spec_{Sch}$ and $[\![Spec_{Sch}]\!]^a$. Here $[\![Spec_{Sch}]\!]^a$ stands for the abstraction derived from $[\![Spec_{Sch}]\!]$.

Reachability Analysis

The first property i.e. $EF(K2 = 1)$ can be checked following the recipe described in Section 5.3. As per this recipe, the minimal model is queried with the goal $\leftarrow$ `rstate`$(\_C1, \_C2, \_X1, \_X2, \_K1, K2, \_T, \_L), K2 = 1$. Since this goal succeeds, the property is proved to hold.

The third property i.e. $AG(K2 \leq 1)$ is verified following the recipe described in Section 5.2. According to this recipe, the minimal model is queried with the goal $\leftarrow$ `rstate`$(\_C1, \_C2, \_X1, \_X2, \_K1, K2, \_T, \_L), K2 > 1$. The constraint $K2 > 1$ is the negation of the safety proposition $K2 \leq 1$. This goal succeeds and thus the property is proved to hold.

Abstract model checking

The second property $AG(K2 >= 1) \rightarrow AF(K2 = 0)$ being a path property cannot be verified with reachability analysis alone. We use the AMC tool for this formula. The AMC tool when input with the appropriate files and the formula ϕ returns the set of abstract states $[\![\neg\phi]\!]^a$ as shown in Table 7.3. This table reports the model checking results for all the three properties.

7.3.2 Gas burner controller

In a gas stove or burner, the heat is generated by burning the gas. These gas stoves or burners, for safety, are equipped with a control system that make sure that no gas leaks beyond safety limits. This control system makes use of a thermostat. This system on detecting the absence of flame, while the gas valve still being in on position, should close the valve before the volume of gas leaked crosses a predetermined threshold. This control system was first formalised in [107] where a formal model in the language of Duration calculus is defined. The LHA model (shown in Figure 7.11) for this system is defined in [62].

In the LHA, the locations l_0 and l_1 correspond to the gas-leaking state no-leak states respectively. The variable x controls the time spent in each of the locations. The variable z records the total time spent in the leaking state; while variable y records the total time lapsed. The safety requirement on this system is that "in any interval of time of at least 60 seconds, the leaking time does not exceed 5 % of the total time". This requirement translates[2] to the safety property $AG(y \geq 60 \rightarrow 20 * z \leq y)$. The safety proposition is $\neg(y \geq 60) \vee (20 * z \leq y)$.

For this system, since the TPCLP tool does not terminate, the abstract minimal model is computed by using the CHA tool. This abstract minimal model is shown in Figure 7.12. This minimal is translated into a SICStus Prolog of the form shown in 7.13. This program is queried with the goal $\leftarrow \texttt{rstate}_i(x, y, z, t), (y \geq 60 \wedge 20 * z > y)$ (for i= 1 to 2). Here the constraint $(y \geq 60 \wedge 20 * z > y)$ is the negation of the safety proposition mentioned above. This query when expressed in SICStus Prolog takes the form of $\leftarrow \texttt{rstate}_i(X, Y, Z, T), Y \geq 60, 20 * Z > Y$. This goal fails, and hence the safety property is proved to hold.

7.3.3 Temperature controller

This is a control system that monitors the temperature of a nuclear reactor. The LHA model of this stem is shown in Figure 7.14 (taken from [62]). In this system,

[2]The requirement is when $y \geq 60$ z should be less than 5% of y i.e. $z \leq y/20$ or $20 * z \leq y$.

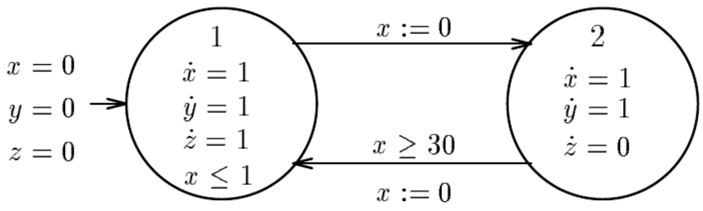

Figure 7.11: Gas burner LHA model

```
rstate__2(A,B,C,D) :- [-1*A+1*B+-1*C>=0,
                       -1*A+1*B+ -31*C>= -30,
                       1*C>=0, 1*A>=0,
                       1*A+ -1*D=0]
rstate__1(A,B,C,D) :- [1*C+ -1*D>=0, 1*D>=0,
                       1*B+ -31*C+30*D>=0, -1*D>= -1,
                       1*A+ -1*D=0]
```

Figure 7.12: Gas burner's abstract minimal model

```
:- use_module(library(clpq)).
rstate__2(A,B,C,D) :- {-1*A+1*B+ -1*C>=0,
                      -1*A+1*B+ -31*C>= -30,
                      1*C>=0, 1*A>=0, 1*A+ -1*D=0}.
rstate__1(A,B,C,D) :- {1*C+ -1*D>=0, 1*D>=0,
                      1*B+ -31*C+30*D>=0,
                      -1*D>= -1,1*A+ -1*D=0}.
```

Figure 7.13: Gas burner model in SICStus Prolog

the temperature is controlled by lowering one of the two cooling rods rod_1 and rod_2 into the reactor. There are three variables: x measuring the temperature and y_1, y_2 measuring the time lapsed after the previous instance when rod_1 and rod_2 were used. A cooling rod is put into use when the temperature hits 550. The constraint is that a cooling rod can only be deployed if it has been free for at least 20 seconds. When one cooling rod is not available the other cooling rod is used. So the unsafe condition that needs to be avoided is $x >= 550 \wedge y1 < 20 \wedge y2 < 20$ i.e. the temperature exceeding 550 while both rods have not been free for over 20 seconds.

This LHA has three locations: l_0, l_1 and l_2. In l_0, the temperature rises no more than 550 at the rates ranging in the interval $[1, 5]$. In l_1 (resp l_2), rod_1 (resp. rod_2) cools the temperature at the rates ranging in the interval $[-5, -1]$. In all

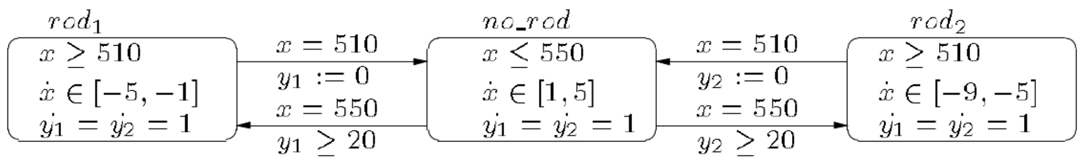

Figure 7.14: The Temperature Controller LHA

```
rstate__1(A,B,C,D) :- [1*A+ -1*D>=0,
                       2*A+51*C+ -53*D>=1020,1*D>=0,
                       2*A+51*B+ -53*D>=1020,-1*A+5*D>= -510,
                       -1*A>= -550]
rstate__2(A,B,C,D) :- [51*C+ -51*D>=344,
                       1*A>=510,1*A+5*D>=550,
                       -1*A+ -1*D>= -550,
                       1*B+ -1*D>=20]
rstate__3(A,B,C,D) :- [1*C+ -1*D>=20,
                       -1*A+ -5*D>= -550,51*B+ -51*D>=344,
                       1*A>=510,1*A+9*D>=550]
```

Figure 7.15: The Temperature Controller abstract minimal model

the locations, $y1$ and $y2$ change at the rate of $+1$.

For this system also the TPCLP does not terminate, hence the abstract minimal model is computed by using the CHA tool. This abstract minimal model is shown in Figure 7.15. As always, this minimal model is translated into a form accepted by SICStus Prolog. This program is queried with the goal $\leftarrow \texttt{rstate}_i(x, y1, y2, t), (x = 550 \wedge y1 < 20 \wedge y2 < 20$ (for i= 1 to 3). Here the constraint $(x = 550 \wedge y1 < 20 \wedge y2 < 20$ defines the unsafe state. This query when expressed in SICStus Prolog takes the form of:

$\leftarrow \texttt{rstate}_i(X, Y1, Y2, T), X = 550, Y1 < 20, Y2 < 20.$

This goal fails, and hence the safety property is proved to hold.

7.3.4 Train Gate Controller

This is a control system involving three components: (i) a train; (ii) a controller and (iii) a gate guarding the rail-road crossing. The LHA models for each of these components is shown in Figure 7.16. The job of this system is to close the gate when a train is approaching the crossing point. These three components interact by raising events. The complete system is a product of these three components.

The train signals its approach towards the cross when it is $1000m$ away from the cross by issuing the *approach* event; while it signals its exit from the critical zone when it is $100m$ past the cross by issuing the *exit* event. The variable x in the train LHA models the distance of train from/past the crossing point. The controller on receiving the *approach* event signals the gate to close by issuing the *lower* event; while on receiving the *exit* event signals the gate to open the gate by issuing a *raise* event. The maximum reaction time of the controller is 5 seconds i.e. it responds to the events from the train with in 5 seconds. The variable z in the controller LHA models this reaction time. The gate on receiving

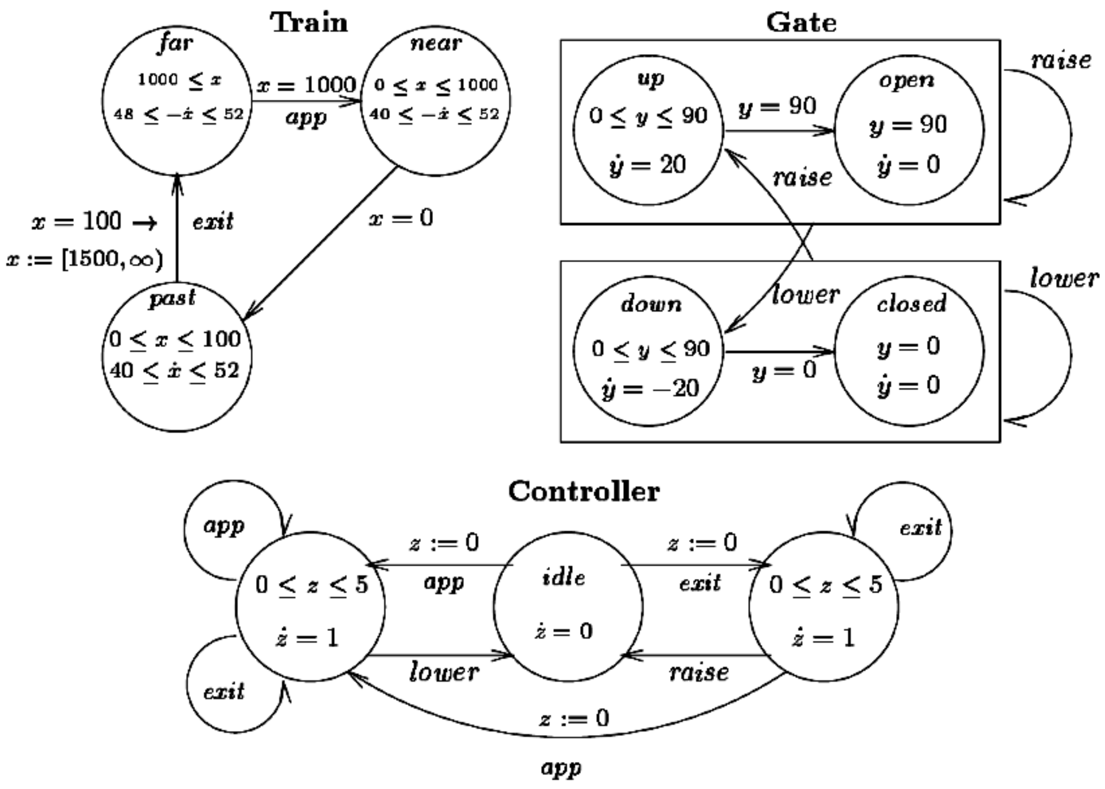

Figure 7.16: The Train gate controller LHA

the *raise* (resp. *lower*) event, raises (resp. lowers) the gate at the rate of 20 degrees/second. When the gate is at an angle of 0 (resp. 90) degrees it is in close (resp. open) state. The variable y models the gate angle. The speed of train varies as following: (a) $x \geq 1000 : -52 \leq \dot{x} \leq -48$; (b) $0 \leq x \leq 1000 : -52 \leq \dot{x} \leq -40$ and (c) $0 \leq x \leq 100 : 40 \leq \dot{x} \leq 52$. The sign of the speed signifies whether its approaching or receding from the cross.

Of this system, we wish to check the following CTL formulas:

1. $AG(x \leq 10 \rightarrow y = 0)$;

2. $AF(y = 0)$;

3. $AG(AF(y = 90))$;

4. $AG(y = 0 \rightarrow AF(y = 90))$;

5. $AG(y = 90 \rightarrow AF(y = 0))$;

ϕ	$[\![\neg\phi]\!]^a$	Time (secs.)
$AF(y = 0)$	$\emptyset$	0.307
$AGAF(y = 90)$	$\emptyset$	0.401
$AG(y = 0 \to AF(y = 90))$	$\emptyset$	0.409
$AG(y = 90 \to AF(y = 0))$	$\emptyset$	0.322

Table 7.4: AMC Results for Train Gate Controller

The first formula states the property that "whenever the train is within a $10m$ distance from the cross the gate is always closed". The second formula states the property that "always eventually the gate closes"; while the third formula states that "always globally always in future the gate opens". The fourth (resp. fifth) formula states the property that "a closed (resp. open) gate always eventually opens (resp. closes)".

We prove the first formula by reachability analysis, while the AMC is employed to verify the remaining. First, the product LHA of the three components is constructed whose minimal model is computed via LHA2CLP$_R$, LOGEN and TPCLP tool chain. This minimal model is queried with the safety violating goal. Since no such unsafe state exists, the first property holds. The AMC results for these formulas are reported in the Table 7.4.

7.3.5 Fischer Protocol

In multi-process systems, to assure mutually exclusive access to shared resources a protocol is followed. The Fischer protocol [83] is one such protocol. In [7], an LHA model for a two process Fischer protocol is defined.

Consider an asynchronous distributed system with two processes P_1 and P_2 each having their own local clocks. Each process's execution involves an atomic read and write operations on a shared memory. Thus there is a critical section in each process. The Fischer protocol ensures that at any time instance at most one of the two processes is in its critical section. The implementation of this protocol can be seen in the pseudo-code of the process P_i shown in Figure 7.17.

Here k is the shared variable between the two processes P_1 and P_2. The process P_i is allowed into critical section iff $k = i$. Each process has a local clock. The instruction `delay` b delays the process by b time units as measured by the process's local clock. The assignment instruction $k := i$ corresponds to a write access that takes a time units. In fact, a, b are parameters in this system. As is quite common in asynchronous systems, the local clocks may be inaccurate and might even have different rates of ticking. In this model, the clock of P_2 ticks at the rate of 0.9 to 1.1 times the ticking rate of the clock of P_1. [7] gives the two LHAs corresponding

```
repeat forever
    repeat until $k = i$
        await $k = 0$;
        $k := i$;
        delay $b$;
    Critical_section_i
    $k := i$;
```

Figure 7.17: The pseudo-code for Process P_i.

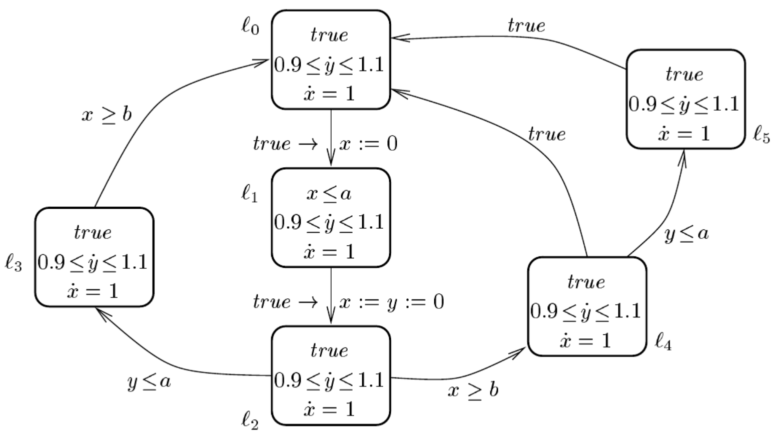

Figure 7.18: The Fischer protocol product LHA

to the two models. But we consider the product LHA as defined in [57]. This product LHA is shown in Figure 7.18. The variables x, y correspond to the rates of delays of P_1 and P_2 respectively. In location l_0, P_1 is idle; while in l_1 it has read $k = 0$. On the discrete transition l_0 to l_1, P_1 is supposed to set k to 1, so it is the last time P_2 can read $k = 0$. In l_2, P_1 waits for b time units, following which two transitions might occur: either b lapses and P_1 enters its critical section i.e. l_4 is entered; or on P_2 setting k to 2 i.e. l_3 is entered and thus P_1 is forbidden to enter its critical section.

The location l_4 corresponds to P_1 being in the critical section. If in this location, P_2 sets k to 2, it may also enter critical location. The location l_5 models this situation where both P_1 and P_2 are both in their critical sections. We need to prove that l_5 is never entered. To check this property, we compute the minimal model and see the constraints on the reachable states corresponding to this location l_5. The minimal model of the specialised CLP program of this LHA is shown in Figure 7.19. Recall that the reachable states with l_i as their location is defined

```
rstate__1(A,B,C,D,E,F) :-
       F=4,1*D>=0,1*C>=0,-10*B+11*E>=0,
       10*B+ -9*E>=0,1*A+ -1*E=0.
rstate__2(A,B,C,D,E,F) :-
       F=5,-10*B+11*E>=0,-1*A+1*E>=0,
       10*B+ -9*E>=0,1*A>=0,1*D>=0,-1*A+1*C>=0.
rstate__3(A,B,C,D,E,F) :-
       F=6,-1*A+1*E>=0,1*C>=0,1*D>=0,
       11*A+ -10*B>=0,-9*A+10*B>=0.
rstate__4(A,B,C,D,E,F) :-
       F=7,1*C>=0,-9*A+10*B>=0,
       11*A+ -10*B>=0,-1*A+1*E>=0,1*D>=0.
rstate__5(A,B,C,D,E,F) :-
       F=8,1*A+ -1*D>=0,11*A+ -10*B>=0,
       1*D>=0,-9*A+10*B>=0,-1*A+1*E>=0,1*C>=0.
rstate__1(A,B,C,D,E,F) :-
       F=4,1*A+ -1*D>=0,11*A+ -10*B>=0,
       1*D>=0,-9*A+10*B>=0,-1*A+1*E>=0,1*C>=0.
rstate__6(A,B,C,D,E,F) :-
       F=9,1*A+ -1*D>=0,11*A+ -10*B>=0,
       11*A+ -10*B+10*C+ -11*D>=0,10*C+ -9*D>=0,
       -9*A+10*B>=0,1*D>=0,-1*A+1*E>=0.
rstate__2(A,B,C,D,E,F) :-
       F=5,-10*B+11*E>=0,1*A>=0,
       -9*A+10*B+ -9*D>=0,-1*A+ -1*D+1*E>=0,1*D>=0,-1*A+1*C>=0.
```

Figure 7.19: The minimal model of the two process Fischer protocol system.

by the constrained fact rstate__$i+1(\bar{X})$. The first two elements A, B in the six-element list are the two clock variables x, y while the following two C, D are a, b and the remaining E, F are time variable t and location-index variable. So in rstate__6/6, we have a constraint $10 * C + -9 * D \geq 0$ i.e. the variables a and b are related by $10 * a \geq 9 * b$. Thus by choosing the parameters a, b such that $10 * a < 9 * b$ the location l_5 is never visited. Thus with TPCLP we can do such parametric analysis as well.

7.4 TPCLP and CHA computation times

In Table 7.5, we summarise the TPCLP results of computing a model for each of the six LHAs presented in this chapter. The number of locations in the automaton is Q and number of discrete transitions is Δ. The number of clauses in the trans-

Name	Q	Δ	$\|CLP\|$	TPCLP (secs.)
Fischer Protocol	6	8	9	0.023
Leaking Burner	2	2	5	∞
Scheduler	3	11	17	0.216
Train Gate Controller System	36	200	145	0.154
Temperature Controller	3	4	5	∞
Water Level	4	4	5	0.022

Table 7.5: TPCLP Results

lated CLP programs includes the clauses for the delay transitions. Timings are given in seconds and the symbol ∞ indicates failure to terminate within a time-out duration of 300 seconds. The symbol ∞ in Table 7.5 implies the non-termination of TPCLP.

Specialisation Vs. Efficiency Since the unspecialised versions of the CLP encodings of LHAs contain list arguments, TPCLP cannot compute minimal model of such CLP programs. Therefore it is not possible to measure the computational efficiency gained by specialisation.

Table 7.6 summarises the results of computing the approximate minimal models of the leaking burner and temperature controller systems.

Name	Q	Δ	$\|CLP\|$	CHA (secs.)
Leaking Burner	2	2	5	0.028
Temperature Controller	3	4	5	0.042

Table 7.6: CHA Results

Memory utilisation. The experiments were conducted on a computer with an Intel XEON CPU running at 2.66GHz and with 4GB RAM. In all of the above experiments, none of the tools (TPCLP, CHA or abstract model checker) utilised more than 2GB RAM.

Summary

In this chapter, we demonstrated how LHAs can be modelled and verified by applying the modelling and verification concepts defined in this dissertation. We only focussed on the CLP models that make use of the forwards reasoning driver.

The next chapter presents the existing work related to the CLP-based techniques for modelling and verification of hybrid systems.

Chapter 8

Related work

The idea of modelling and verifying transition systems of various kinds as CLP programs goes back many years. Several works also exist where infinite state systems' verification is pursued by applying abstract interpretation. In this chapter, we review existing works on CLP-based verification. These works cover systems with both discrete and continuous state variables and consider both safety and liveness properties. However no comprehensive approach exists to verify arbitrary temporal properties for both finite and infinite systems. There are frameworks for abstract model checking that are not based on CLP; we argue that our approach has advantages over previous work and gives a direct, elegant and practical approach for combining abstract interpretation and model checking.

We survey the related work from three perspectives, namely, (i) the modelling of embedded systems in CLP; (ii) the verification of safety and liveness properties by computing minimal models and (iii) abstract model checking, which are the three main contributions of this dissertation.

CLP-based modelling and verification of embedded systems

Gupta and Pontelli [55] are the first to propose a CLP-based framework to verify hybrid systems specified in the language of Timed Automata (TA). Here the TA model of a system is represented by a CLP program, which is executed to verify properties. The meaning of a timed automaton is given by the sequence of events generated by it. So to verify a property, the executable CLP programs are queried with goals, which specify event sequences (characterising a property) with constraints.

This work presents two variants of system modelling: (i) the syntax of timed automata is modelled with DCG clauses, while the semantics is modelled with constraint clauses; (ii) syntax and semantics are both modelled with CLP clauses. However, the translation scheme from TA to CLP is not formally defined. When a system comprises more than one automaton, a driver interfaces the individual components.

Since the CLP program is executed to verify a property, this approach is based on the procedural semantics, it has obvious limitations as a proof procedure for infinite behaviours. Besides this limitation, they use a non-standard property specification language. In this language, it might not be possible to state equivalents of nested CTL properties.

Jaffar *et al* [70] also propose a CLP based framework to verify TA models. This framework defines a systematic scheme for translating the language of TA into CLP. They define a new assertion language for specifying system properties. Here also the CLP program is executed to verify a property. The proof method relies on the techniques of tabling [115] and co-induction [113].

Our work differs from the above two works [70] [55] in two aspects: (i) we model the LHA specifications and (ii) we consider the temporal logic of CTL that is a more standard property specification language in literature. Though comparing various formalisms is not the aim of our work, it is noteworthy that LHAs are *more expressive* than other formalisms such as Timed Automata (TA) [6] or other finite automata as discussed in [21]. Thus we cover a wider range of systems than them. Since they do not define the correspondence of their property specification language with standard temporal logics, it is not straightforward to identify the class of CTL properties that are not taken care by them. However, it seems that specifying nested CTL properties in their language is not possible.

Hickey and Witternberg [65] model hybrid systems in a new CLP language, called Analytic Constraint Logic Programming (ACLP) [63], a higher order constraint language, to model hybrid systems. In particular, the hybrid systems are modelled in CLP(F) [64], a kind of ACLP, having the capability to encode (arbitrary) ordinary differential equations. However, though they can model non-linear hybrid systems, only safety properties could be verified in this work.

Falaschi and Villanueva [42] propose a model checking methodology to verify reactive systems modelled in the language of timed concurrent constraint programming (`tccp`) language. In this work: (i) a `tccp` model is first compiled into a `tccp structure`, which is a variant of Kripke structure; (ii) the reactive properties are specified in a linear temporal language (analogous to Linear Temporal Logic (LTL)) suitable for reasoning about `tccp` models. The developed model checker proves a property by checking that there is no path in the `tccp` structure along which the negation of the property holds. Since: (i) the `tccp` models can be expressed in the language of linear TA (LTA); and (ii) the language of LHA being more powerful than that of TA, it is not clear whether `tccp` is powerful enough to specify LHAs, particularly, their multi-rate dynamics. Furthermore, they consider properties specified in a linear-time temporal language, while we consider the CTL language, which is a branching-time temporal language.

Other CLP transition system models

Delzanno and Podelski [35] develop techniques for modelling and verifying *discrete* transition systems. Here a systematic scheme to translate the *guarded-command specification language* extended with shared variables [112] into CLP programs is presented; while the properties are specified in CTL. They define $[\![\phi]\!]$ in terms of the CLP fixed point semantics of a constraint logic program, which is a composition of CLP program encoding the system and a CLP program encoding the CTL property. Only the properties of the form *AFp*, *EFp*, *AGp*, *EGp* can be verified in this work.

CLP interpreters for temporal formulas

A somewhat different but related CLP modelling and proof approach is followed in [19, 88, 96, 36, 43, 99, 100]. This is to encode a proof procedure for a modal logic such as CTL, μ-calculus or related languages as a logic program, and then prove formulas in the language by running the interpreter (usually with tabling to ensure termination). The approach is of great interest but adapting it for abstraction of infinite state systems seems difficult since the proof procedures themselves are complex programs and include negation. The programs usually contain a predicate encoding the transitions of the system in which properties are to be proved, and thus could in principle be coupled to our translation.

In almost all of the above cited works, they consider the minimal fragment of a temporal logic. It is minimal in the sense that any formula (in the consider temporal language) could be written using this minimal fragment and negation. They define a CLP interpreter based around this minimal fragment. The CLP program corresponding to a formula (not from the minimum fragment) often involves negation. The execution of such CLP programs requires a CLP implementation with sound negation (like *constructive negation* [114]). But such CLP implementations are still being researched. Furthermore, for the same reason and negation being antitone, it is difficult to approximate negation. Consequently verification based on abstract interpretation requires that the solution set be under-approximated, which is not easy. For these reasons, the cited works are restricted to a subset of CTL.

In our approach, since we consider full CTL, any arbitrary formula can be reduced to a NNF formula. Consequently the limitations implied by the negation are totally bypassed. Besides this, in these cited works, the full prover is run for each formula to be proved, whereas in our case, a minimal model is computed once and the properties of the form AGp and EFp are proved by querying this minimal model.

Direct analysis of Hybrid systems

Another direction we could have taken is to develop a direct translation of LHA semantics into CLP, without the intermediate representation as a transition system. For example a "forward collecting semantics" for LHAs is given in [56], in the form of a recursive equation defining the set of reachable states for each location. It would be straightforward to represent this equation as a number of CLP clauses, whose least model was equivalent to the solution of the equation. A technique similar to our method of deriving the `d` predicate for the constraints on the derivatives could be used to model the operator $S \nearrow D$ in [56] representing the change of state S with respect to derivative constraints D. The approach using transition systems is a little more cumbersome but gives the added flexibility of reasoning forwards or backwards, adding traces, dependencies and so on as described in Chapter 5. The clauses we obtain for the forward transition system (after partial evaluating the `transition` predicate) are essentially what would be obtained from a direct translation of the recursive equation in [56].

Other Model checkers

Uppaal [14] is a CTL model checker. The system to be verified is input as a TA. As mentioned earlier, the language of LHA is more expressive than the language of TA. Consequently, from the modelling perspective, our framework has two advantages over the Uppaal [14] model checker or any other TA model checkers. Firstly, we can directly handle LHAs having *multi-rate dynamics*[1] , whereas Uppaal mandates that an LHA specification be compiled down into a TA specification before being verified. Secondly, Uppaal restricts the clock variables to be compared only with natural numbers, i.e. guards such as $x > 1.1$, $10 * x > 11$ or $x > y$ are not permitted [75]. Besides this, Uppaal cannot verify nested CTL formulas excepting those of the form $AG(EX\,\texttt{true})$ and $AG(p \rightarrow AFq)$ where p, q are propositions.

HYTECH [61] is a model checker for the systems specified in the language of LHA. Here the properties are specified in the language of ICTL, which is an extension of CTL. They use standard model checking algorithms. As with many of the other approaches, the HYTECH implementation does not handle the full language[2].

[1]A timed automaton has variables called *clocks* that vary at a single rate i.e. $\dot{x} = 1, \dot{y} = 1$, while an LHA can have variables that vary at different rates i.e. $\dot{x} = 1, \dot{y} = 2$.

[2]Personal communication with Wong-Toi.

PHAVer [45] is yet another tool to verify hybrid systems. It can only verify safety properties.

Abstract model checking

The topic of model-checking infinite state systems using some form of abstraction has been already widely studied. Abstract model checking is described by Clarke *et al.* [24, 73]. In this approach a state-based abstraction is defined where an abstract state is a set of concrete states. A state abstraction together with a concrete transition relation Δ induces an *abstract transition relation* Δ_{abs}. Specifically, if X_1, X_2 are abstract states, $(X_1, X_2) \in \Delta_{abs}$ iff $\exists x_1 \in X_1, x_2 \in X_2$ such that $(x_1, x_2) \in \Delta$. From this basis an abstract Kripke structure can be built; the initial states of the abstract Kripke structure are the abstract states that contain a concrete initial state, and the property labelling function of the abstract Kripke structure is induced straightforwardly as well. Model checking CTL properties over the abstract Kripke structure is correct for *universal* temporal formulas (ACTL), that is, formulas that do not contain operators EX, EF, EG or EU. Intuitively, the set of paths in the abstract Kripke structure represents a superset of the paths of the concrete Kripke structure. Hence, any property that holds for all paths of the abstract Kripke structure also holds in the concrete structure. If there is a finite number of abstract states, then the abstract transition relation is also finite and thus a standard (finite-state) model checker can be used to perform model-checking of ACTL properties. Checking properties containing existential path quantifiers is not sound in such an approach.

This technique for abstract model checking can be reproduced in our approach, although we do not explicitly use an abstract Kripke structure. Checking an ACTL formula is done by negating the formula and transforming it to negation normal form, yielding an *existential* temporal formula (ECTL formula). Checking such a formula using our semantic function makes use of the $\mathsf{pred}_\exists$ function but not the $\mathsf{pred}_\forall$ function. For this kind of abstraction the relation on abstract states $s \longrightarrow s'$ defined as $s \in (\alpha \circ \mathsf{pred}_\exists \circ \gamma)(\{s'\})$ is identical to the abstract transition relation defined by Clarke *et al.* Note that whereas abstract model checking the ACTL formula with an abstract Kripke structure yields an under-approximation of the set of states where the formula holds, our approach yields the complement, namely an over-approximation of the set of states where the negation of the formula holds.

There have been different techniques proposed in order to overcome the restriction to universal formulas. Dams *et al.* [31] present a framework for constructing abstract interpretations for μ-calculus properties in transition systems. This involves constructing a *mixed transition system* containing two kinds of transition relations, the so-called free and constrained transitions. Godefroid *et al.* [52] proposed the use of *modal transition systems* [85] which consist of two components, namely *must*-transitions and *may*-transitions. In both [31] and [52], given an abstraction together with a concrete transition system, a mixed transition system, or an (abstract) modal transition system respectively, is automatically generated. Following this, a modified model-checking algorithm is defined in which any formula can be checked with respect to the dual transition relations. Our approach by contrast is based on the standard semantics of the μ-calculus. The *may*-transitions and the *must*-transitions of [52] could be obtained from the functions $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)$ and $(\alpha \circ \mathsf{pred}_\forall \circ \gamma)$ respectively. For the case of an abstraction given by a partition $A = \{d_1, \ldots, d_n\}$ it seems that an abstract modal transition system could be constructed with set of states A such that there is a *may*-transition $d_i \rightarrow d_j$ iff $d_i \in (\alpha \circ \mathsf{pred}_\exists \circ \gamma)(\{d'_j\}$ and a *must*-transition $d_i \rightarrow d_j$ iff $d_i \in (\alpha \circ \mathsf{pred}_\forall \circ \gamma)(\{d_j\}$. However the two approaches are not interchangeable; in [52] a concrete modal transition system has the same set of *must*-transitions and *may*-transitions, but applying the above constructions to the concrete state-space (with α and γ as the identity function) does not yield the same sets of *must*- and *may*-transitions (unless the transition system is deterministic). We have shown that the construction of abstract transition systems as in [24, 73], and abstract modal transition systems in particular [31, 52] is an avoidable complication in abstraction. Probably the main motivation for the definition of abstract transition systems is to re-use existing model checkers, as remarked by Cousot and Cousot [29] (though this argument does not apply to modal or mixed transition systems in any case).

The application of the theory of abstract interpretation to temporal logic, including abstract model checking, is thoroughly discussed by Cousot and Cousot [28, 29]. Our abstract semantics is inspired by these works, in that we also proceed by direct abstraction of a concrete semantic function using a Galois connection, without constructing any abstract transition relations. The technique of constructing abstract functions based on the pattern $(\alpha \circ f \circ \gamma)$, while completely standard in abstract interpretation [27], is not discussed explicitly in the temporal logic context. We focus only on state-based abstractions (Section 9 of [29]) and we ignore abstraction of traces. Our contribution compared to these works is to work out the abstract semantics for a specific class of constraint-based abstractions, and point the way to effective abstract model checking implementations using SMT solvers. Kelb [76] develops a related abstract model checking algorithm based on abstraction of universal and existential predecessor functions.

Giacobazzi and Quintarelli [51] discuss abstraction of temporal logic and their refinement, but deal only with checking universal properties. Saïdi and Shankar [110] also develop an abstract model checking algorithm integrated with a theorem proving system for handling property-based abstractions. Their approach also uses abstract interpretation but develops a framework that uses both over- and under-approximations for handling different kinds of formula.

Summary

In this chapter, we presented some of the existing works related to modelling and verification of hybrid systems. We only focussed on the CLP-based approaches and abstraction-based approaches. The next chapter concludes this dissertation by presenting a summary of contributions and some directions for further work.

Chapter 9

Conclusion

The thesis of this dissertation was that *general purpose program analyses meant for constraint logic programs can be applied to the formal verification of high-level specifications of embedded systems*. This dissertation's investigation into the stated thesis can be broadly categorised into three parts: (a) modelling of embedded systems in CLP; (b) verification of embedded system with CLP analyses alone and (c) verification of embedded systems with abstract model checking in which constraints play an important role. We chose the LHA language as the high level specification language in this investigation.

Modelling in CLP

To analyse an LHA model with CLP analysis tools, first the LHA model is modelled as a constraint logic program. We presented (in Chapter 3) a standard scheme to translate LHA models into constraint logic programs. This translation is mechanised with a compiler, whose source language is the text LHA and target language is the CLP. This compiler is also implemented in Ciao Prolog, which is a CLP language. Since the language of LHA is graphical, a simple textual language named text-LHA is defined. This part of the dissertation focussed on modelling of high level models in CLP.

This part of the dissertation contributes to the area of CLP-based system modelling. On the one hand we add to the literature on CLP modelling techniques in showing that certain aspects (sufficient enough to prove CTL properties) of the hybrid dynamics of LHAs can be captured in CLP programs. On the other we add to the existing literature showing that both forwards and backwards reasoning models can be generated from a single system specification.

Verification with CLP analyses

The CLP program encoding an LHA is then specialised using a general purpose partial evaluator for logic programs. Such a specialised program is then subjected to static analysis to compute its minimal model, which results in a set of reachable states (or reaching states) that form the basis for proving arbitrary CTL formulas. For a certain class of LHAs, as defined in [62], the concrete minimal model computation will not terminate in which case we apply the theory of abstract interpretation and compute the abstract minimal model. In Chapter 5, two methods were presented to verify simple universal safety and simple existential liveness properties. These methods were exclusively based on CLP analysis alone.

This parts of the dissertation contributes to the area of CLP-based proof techniques. We add to the existing literature showing that effective reasoning can be carried out on CLP programs, and display a family of different reasoning styles (based on forwards and backwards drivers) using a single set of CLP analysis tools.

Abstract model checking

Chapter 6 defined the technique of abstract model checking that can verify arbitrary CTL formulas. Here we presented a systematic application of the theory of abstract interpretation to the algorithmic verification technique of model checking. In this part of the dissertation, we have demonstrated a practical approach to abstract model checking, by constructing an abstract semantic function for the CTL language based on a Galois connection. Much previous work on abstract model checking is restricted to verifying *universal properties* and requires the construction of an abstract transition system. In other approaches in which arbitrary properties can be checked [52, 31], a dual abstract transition system is constructed. Like Cousot and Cousot [29] we do not find it necessary to construct any abstract transition system, but rather abstract the concrete semantic function systematically. Using abstract domains based on constraints we are able to implement the semantics directly. The use of an SMT solver adds greatly to the effectiveness of the approach.

Further work

Whenever the computation of concrete minimal model becomes impossible, we are forced to compute an abstract model. Since abstraction introduces loss of precision, in abstraction-based techniques the challenge is to find precise abstractions. The choice of precision depends on the chosen abstractions. Currently we use two abstractions: an abstraction defined in [35] and a classical abstract interpretation based on convex polyhedral hulls with widening and narrowing operators [30].

More complex and precise abstractions, such as those based on the power-set domain consisting of finite sets of convex polyhedra could be used [10]. The resulting abstract model can be represented as a set of constrained facts, as in the concrete model (since a convex polyhedron can be represented as a linear constraint).

Similarly, in the abstract model checking part, to make the implementation of abstract model checker simpler, we considered the abstractions that partition the reachable state space. However, we do not always get disjoint partitions from the TPCLP or *CHA* tools. When the abstraction is not disjoint, we make it disjoint by defining a new state that is a union of the overlapping abstract states. Instead of making a union, which induces more loss in precision, the overlapping abstract states could be partitioned into finer (disjoint) abstract states. But such refinement, in the worst case, might produce an exponentially large state space. Nevertheless, in such cases (where we achieve partition by abstraction), in which AMC concludes neither the correctness of a CTL formula ϕ nor the correctness of the negated formula $\neg\phi$ such a refinement becomes the only reasonable way to proceed. Besides these specific possible improvements, the approach to the technique of abstract model checking proposed in this dissertation being new – it is new in the sense of being much simpler than the current approaches – can be further developed along the following generic lines:

1. Experimenting with a wider range of modelling, programming or specification languages.
2. Checking the scalability of the proposed tool chain to verify complex systems.
3. Since our AMC is based on arbitrary Galois connections, richer abstractions can be pursued. Examples for such abstractions could be power-set domains, symbolic abstractions to handle richer data structures (arrays, stacks, etc.), etc.
4. Abstractions not limited to partitions alone can be pursued.
5. Property-based abstraction refinements can be pursued.

Bibliography

[1] *Formal methods specification and verification guidebook for software and computer systems*, volume II: A Practitioner's Companion. Office of Safety and Mission Assurance, NASA, National Aeronautics and Space Administration, Washington, DC 20546, USA, 1997.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[3] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[4] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

[5] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer, 1992.

[6] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[7] R. Alur, T. A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Trans. Software Eng.*, 22(3):181–201, 1996.

[8] R. Alur, T. A. Henzinger, G. Lafferriere, and G. J. Pappas. Discrete abstractions of hybrid systems. In *Proceedings of the IEEE*, pages 971–984, 2000.

[9] S. N. Artemov, J. Davoren, and A. Nerode. Modal logics and topological semantics for hybrid systems. Technical report, Cornell University, 1997.

[10] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. *STTT*, 9(3-4):413–414, 2007.

[11] R. Bagnara, P. M. Hill, and E. Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[12] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[13] G. Banda and J. P. Gallagher. Analysis of linear hybrid systems in CLP. In M. Hanus, editor, *LOPSTR*, volume 5438 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2008.

[14] G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *SFM-RT 2004*, number 3185 in Lecture Notes in Computer Science, pages 200–236. Springer, September 2004.

[15] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[16] J. Bhadra, M. S. Abadir, L.-C. Wang, and S. Ray. A survey of hybrid techniques for functional verification. *IEEE Design and Test of Computers*, 24:112–122, 2007.

[17] D. Boulanger and M. Bruynooghe. A systematic construction of abstract domains. In B. L. Charlier, editor, *SAS*, volume 864 of *Lecture Notes in Computer Science*, pages 61–77. Springer, 1994.

[18] D. Boulanger, M. Bruynooghe, and M. Denecker. Abstracting s-semantics using a model-theoretic approach. In M. V. Hermenegildo and J. Penjam, editors, *PLILP*, volume 844 of *Lecture Notes in Computer Science*, pages 432–446. Springer, 1994.

[19] C. Brzoska. Temporal logic programming in dense time. In *ILPS*, pages 303–317. MIT Press, 1995.

[20] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. Lo'pez-Garc'ıa, and G. Puebla. The Ciao Prolog system. reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), Aug 1997. Available from http://www.clip.dia.fi.upm.es/.

[21] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, 1(1/2), 2006.

[22] K. Clark. Predicate Logic as A Computational Formalism. Technical Report DOC 79/59, Imperial College, London, Department of Computing, 1979.

[23] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[24] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *POPL*, pages 342–354, 1992.

[25] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[26] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[27] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[28] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Autom. Softw. Eng.*, 6(1):69–95, 1999.

[29] P. Cousot and R. Cousot. Temporal abstract interpretation. In *POPL'2000*, pages 12–25, 2000.

[30] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[31] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[32] R. David and H. Alla. On hybrid Petri nets. *Discrete Event Dynamic Systems*, 11(1-2):9–40, 2001.

[33] M. Davis. Mathematical procedures for decision problems. Technical report, Institute for Advanced Study, 1954.

[34] J. M. Davoren. *Modal Logics for Continuous Dynamics.* PhD thesis, CALIFORNIA UNIV BERKELEY, Nov 1997.

[35] G. Delzanno and A. Podelski. Model checking in CLP. In *TACAS*, pages 223–239, 1999.

[36] X. Du, C. R. Ramakrishnan, and S. A. Smolka. Real-time verification techniques for untimed systems. *Electr. Notes Theor. Comput. Sci.*, 39(3), 2000.

[37] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(t). In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[38] E. A. Emerson. Temporal and modal logic. pages 995–1072, 1990.

[39] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

[40] O. Ernst-Rüdiger and C. V. Ossietzky. *Real-Time Systems: Formal Specification and Automatic Verification.* Cambridge University Press, Cambridge University Press, 32 Avenue of the Americas, New York, NY 10013-2473, USA, 2008.

[41] D. A. Fahrland. Combined discrete event continuous systems simulation. *Simulation*, 14(2):61–72, 1970.

[42] M. Falaschi and A. Villanueva. Automatic verification of timed concurrent constraint programs. *TPLP*, 6(3):265–300, 2006.

[43] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite-state systems by specializing constraint logic programs. In M. Leuschel, A. Podelski, C. Ramakrishnan, and U. Ultes-Nitsche, editors, *Proceedings of the Second International Workshop on Verification and Computational Logic (VCL'2001)*, pages 85–96. Tech. Report DSSE-TR-2001-3, University of Southampton, 2001.

[44] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science.*

[45] G. Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In M. Morari and L. Thiele, editors, *HSCC*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 2005.

[46] M. Gabbrielli and G. Gupta, editors. *Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005, Proceedings*, volume 3668 of *Lecture Notes in Computer Science*. Springer, 2005.

[47] J. Gallagher. A bottom-up analysis toolkit. Technical Report CSTR-95-016, Department of Computer Science, University of Bristol, July 1995.

[48] J. P. Gallagher, D. Boulanger, and H. Saglam. Practical model-based static analysis for definite logic programs. In *ILPS*, pages 351–365, 1995.

[49] J. P. Gallagher and K. S. Henriksen. Abstract domains based on regular types. In B. Demoen and V. Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 27–42. Springer, 2004.

[50] J. P. Gallagher, K. S. Henriksen, and G. Banda. Techniques for scaling up analyses based on pre-interpretations. In Gabbrielli and Gupta [46], pages 280–296.

[51] R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples, and refinements in abstract model-checking. In P. Cousot, editor, *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*, volume 2126 of *Lecture Notes in Computer Science*, pages 356–373. Springer, 2001.

[52] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In K. G. Larsen and M. Nielsen, editors, *CONCUR 2001*, volume 2154 of *Lecture Notes in Computer Science*, pages 426–440. Springer, 2001.

[53] R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer, 1993.

[54] A. Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2/3):151–238, 1992.

[55] G. Gupta and E. Pontelli. A constraint-based approach for specification and verification of real-time systems. In *IEEE Real-Time Systems Symposium*, pages 230–239. IEEE Computer Society, 1997.

[56] N. Halbwachs, Y. E. Proy, and P. Raymound. Verification of linear hybrid systems by means of convex approximations. In *SAS'94*, volume 864 of *Lecture Notes in Computer Science*, pages 223–237, 1994.

[57] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[58] K. S. Henriksen. *A Logic Programming Based Approach to Applying Abstract Interpretation to Embedded Software.* PhD thesis, Roskilde University, Roskilde, Denmark, October 2007.

[59] T. A. Henzinger. The theory of hybrid automata. In *LICS*, pages 278–292, 1996.

[60] T. A. Henzinger and P.-H. Ho. Algorithmic analysis of nonlinear hybrid systems. In P. Wolper, editor, *CAV*, volume 939 of *Lecture Notes in Computer Science*, pages 225–238. Springer, 1995.

[61] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 460–463. Springer, 1997.

[62] T. A. Henzinger and V. Rusu. Reachability verification for hybrid automata. In T. A. Henzinger and S. Sastry, editors, *HSCC*, volume 1386 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 1998.

[63] T. J. Hickey. Analytic constraint solving and interval arithmetic. In *POPL*, pages 338–351, 2000.

[64] T. J. Hickey. Clip: A CLP(Intervals) dialect for metalevel constraint solving. In E. Pontelli and V. S. Costa, editors, *PADL*, volume 1753 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2000.

[65] T. J. Hickey and D. K. Wittenberg. Using analytic CLP to model and analyze hybrid systems. In V. Barr and Z. Markov, editors, *FLAIRS Conference*. AAAI Press, 2004.

[66] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[67] C. A. R. Hoare. A model for communicating sequential processes. In *On the Construction of Programs*, pages 229–254. 1980.

[68] C. Huizing and R. Gerth. Semantics of reactive systems in abstract time. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 291–314, London, UK, 1992. Springer-Verlag.

[69] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL*, pages 111–119, 1987.

[70] J. Jaffar, A. E. Santosa, and R. Voicu. Modeling systems in CLP. In Gabbrielli and Gupta [46], pages 412–413.

[71] N. D. Jones. The essence of program transformation by partial evaluation and driving. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Ershov Memorial Conference*, volume 1755 of *Lecture Notes in Computer Science*, pages 62–79. Springer, 1999.

[72] N. D. Jones. Transformation by interpreter specialisation. *Sci. Comput. Program.*, 52:307–339, 2004.

[73] E. M. C. Jr., O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.

[74] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[75] J.-P. Katoen. Concepts, algorithms, and tools for model checking. *A lecture notes of the course "Mechanised Validation of Parallel Systems" for 1998/99 at Friedrich-Alexander Universitat,Erlangen-Nurnberg*, page 195, 1999.

[76] P. Kelb. Model checking and abstraction: A framework preserving both truth and failure information. Technical report, Carl yon Ossietzky Univ. of Oldenburg, Oldenburg, Germany, 1994.

[77] R. M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.

[78] J. C. Kelly and K. Kemp. *Formal methods specification and verification guidebook for software and computer systems*, volume I: Planning and technology insertion. Office of Safety and Mission Assurance, NASA, National Aeronautics and Space Administration, Washington, DC 20546, USA, July 1995.

[79] R. A. Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, 1979.

[80] G. Labinaz, M. M. Bayoumi, and K. Rudie. A survey of modeling and control of hybrid systems. *Annual Reviews of Control*, 21:79–92, 1997.

[81] S. S. Lam and A. U. Shankar. A relational notation for state transition systems. *IEEE Trans. Software Eng.*, 16(7):755–775, 1990.

[82] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[83] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[84] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[85] K. G. Larsen and B. Thomsen. A modal process logic. In *Proceedings, Third Annual Symposium on Logic in Computer Science, 5-8 July 1988, Edinburgh, Scotland, UK*, pages 203–210. IEEE Computer Society, 1988.

[86] L. Lavagno, A. Sangiovanni-Vincentelli, and E. Sentovich. Models of computation for embedded system design. pages 45–102, 1999.

[87] M. Leuschel and J. Jørgensen. Efficient specialisation in Prolog using the hand-written compiler generator logen. *Electr. Notes Theor. Comput. Sci.*, 30(2), 1999.

[88] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'99)*, volume 1817, pages 63–82, April 2000.

[89] J. W. Lloyd. *Foundations of logic programming; (2nd extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[90] N. A. Lynch, R. Segala, F. W. Vaandrager, and H. B. Weinberg. Hybrid I/O automata. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 496–510. Springer, 1995.

[91] Z. Manna and A. Pnueli. Verification of concurrent programs, part i: The temporal framework. Technical report, Stanford, CA, USA, 1981.

[92] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[93] S. Narain and J. Rothenberg. A logic for simulating discontinous systems. In E. A. MacNair, K. J. Musselman, and P. Heidelberger, editors, *Winter Simulation Conference*, pages 692–701. ACM Press, 1989.

[94] S. Narain and J. Rothenberg. Proving temporal properties of hybrid systems. In O. Balci, editor, *Winter Simulation Conference*, pages 250–256. IEEE, 1990.

[95] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[96] U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In *Computational Logic*, volume 1861 of *LNCS*, pages 384–398, 2000.

[97] U. Nilsson and J. Maluszynski. *Logic, Programming, and PROLOG.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[98] J. S. Ostroff. Formal methods for the specification and design of real-time safety critical systems. *J. Syst. Softw.*, 18(1):33–60, 1992.

[99] G. Pemmasani, C. R. Ramakrishnan, and I. V. Ramakrishnan. Efficient real-time model checking using tabled logic programming and constraints. In *ICLP*, volume 2401 of *Lecture Notes in Computer Science*, pages 100–114, 2002.

[100] J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation, 12th International Workshop, LOPSTR 2002, Madrid, Spain, September 17-20,2002, Revised Selected Papers*, pages 90–108, 2002.

[101] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition:, New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Pages: Suppl. 1, English translation by Clifford F. Greene, Jr.

[102] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[103] A. Pnueli. Responding to the panel questions Schloss Ringberg Seminar: Model checking and program analysis. Email response hosted at http://www.mpi-inf.mpg.de/ podelski/Ringberg/panel.htm, Feb. 2000. On 17 Feb 2000 11:36:40 +0200 (IST) Prior to the Schloss Ringberg Seminar on Model Checking and Program Analysis (scheduled for Sunday, 20.2.2000).

[104] M. R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005.

[105] G. Puebla. Analysis and Specialization for Pervasive Systems: An EU FET Project IST-2001-38059. http://clip.dia.fi.upm.es/ clip/Projects/ASAP, Aug. 2007. ASAP Project.

[106] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[107] A. P. Ravn, H. Rischel, and K. M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Trans. Software Eng.*, 19(1):41–55, 1993.

[108] G. A. Robinson, L. T. Wos, and D. F. Carson. Some theorem proving strategies and their implementation. Technical report, Argonne National Laboratory, 1964.

[109] W. C. Rounds. A spatial logic for the hybrid π-calculus. In R. Alur and G. J. Pappas, editors, *HSCC*, volume 2993 of *Lecture Notes in Computer Science*, pages 508–522. Springer, 2004.

[110] H. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454. Springer, 1999.

[111] D. Sannella. A survey of formal software development methods. In R. Thayer and A. McGettrick, editors, *Software Engineering: A European Perspective*, pages 281–297. IEEE Computer Society Press, 1993. Originally published as report ECS-LFCS-88-56, Dept. of Computer Science, Univ. of Edinburgh, 1988.

[112] A. U. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Comput. Surv.*, 25(3):225–262, 1993.

[113] L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In S. Etalle and M. Truszczynski, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 330–345. Springer, 2006.

[114] P. J. Stuckey. Constructive negation for constraint logic programming. In *LICS*, pages 328–339. IEEE Computer Society, 1991.

[115] H. Tamaki and T. Sato. Old resolution with tabulation. In E. Y. Shapiro, editor, *ICLP*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1986.

[116] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pac. J. Math.*, 5:285–309, 1955.

[117] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.

[118] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344. IEEE Computer Society, 1986.

[119] F. Wang. Formal verification of timed systems: A survey and perspective. In *Proceedings of the IEEE*, volume 92, pages 1283–1307, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[120] J. M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–23, 1990.