UNIVERSITY OF CALIFORNIA

Santa Barbara

# Debugging and Verification of Infinite State Real-time Systems

A dissertation submitted in partial satisfaction

of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Zhe Dang

Committee in charge:

Professor Richard A. Kemmerer, Co-chairperson

Professor Tevfik Bultan, Co-chairperson

Professor Oscar H. Ibarra

August 2000

This dissertation of Zhe Dang is approved.

---

Oscar H. Ibarra

---

Tevfik Bultan, Committee Co-chairperson

---

Richard A. Kemmerer, Committee Co-chairperson

July 10, 2000

# Acknowledgments

First of all, thanks to the late professor Ronald V. Book, who was one of the pioneers in studying structural complexity theories, and who gave me a solid background in theoretical computer science during the first two years when I studied in the Mathematics Department at UCSB. Thanks also go to K. Wagner, H. Vollmer, T. Ebert, Y. G. Wang as well as other friends who work in complexity, from whom I learned many beautiful theories.

This dissertation simply does not exist without the guidance and support of my advisor, Professor R. A. Kemmerer. Many of the results presented here were from his original ideas. Besides sharp insights into both theory and applications, I also learned a serious attitude towards science and people from him. Many manuscripts with numerous red marks from him on my desk tell the attitude. I am lucky to have the best advisor for my dissertation research.

As any research combining both theory and application, this work can not be done without co-operation with people from different fields. Among them, I would like to mention T. Bultan, O. H. Ibarra, R. A. Kemmerer, P. Kolano, P. San Pietro and J. Su, with whom a number of papers were written.

Thanks also go to T. Bultan for being my Committee co-chair and to O. H. Ibarra for being a Committee member, patiently reading through the dissertation, discussing various topics in theory, and giving a lot of useful suggestions that significantly improve the presentation of this dissertation. In particular, they both pointed out a good way to write a technical proof by writing down ideas in English instead of in notations. I should also mention S. Stubblebine who gave me an opportunity to work in AT&T Labs Research during the summer of 1998. Anurag Acharya is appreciated for providing a fast machine for running all the mutation tests in this dissertation.

Finally, thanks to my wife for her love and cooking. This dissertation is dedicated to her.

# Vita

| | |
|---|---|
| 8/9/67 | Born in Anhui, China |
| 1987 | BS in Mathematical Logic, Nanjing University |
| 1990 | MS in Theoretical Computer Science,<br>Nankai Institute of Mathematics, Nankai University |
| 1990-94 | Assistant Professor, Department of Computer Science and Engineering,<br>Hebei Institute of Technology |
| 1994-96 | Teaching Assistant, Department of Mathematics,<br>University of California, Santa Barbara |
| 1998 | MS in Computer Science,<br>University of California, Santa Barbara |
| 1996-2000 | Research Assistant, Department of Computer Science,<br>University of California, Santa Barbara |

# Publications

Oscar H. Ibarra, Jianwen Su, Zhe Dang, Tevfik Bultan and Richard A. Kemmerer, "Counter Machines: Decidable Properties and Applications to Verification Problems," *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS 2000)*, Lecture Notes in Computer Science, Springer-Verlag (to appear)

Zhe Dang and Richard A. Kemmerer, "Three Approximation Techniques for ASTRAL Symbolic Model Checking of Infinite State Real-time Systems," *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 345-354, IEEE Press, 2000

Zhe Dang, Oscar H. Ibarra, Tevfik Bultan, Richard A. Kemmerer and Jianwen Su, "Binary Reachability Analysis of Discrete Pushdown Timed Automata, " *Proceedings of the International Conference on Computer Aided Verification (CAV 2000),* Lecture Notes in Computer Science 1855, pages 69-84, Springer-Verlag, 2000

Zhe Dang and Richard A. Kemmerer, "Using the ASTRAL Model Checker to Analyze Mobile IP," *Proceedings of the 21st International Conference on Software Engineering (ICSE 99),* pages 132-141, IEEE Press, 1999

Paul Z. Kolano, Zhe Dang and Richard A. Kemmerer, "The Design and Analysis of Real-Time Systems Using the ASTRAL Software Development Environment," *Annals of Software Engineering* 7(2): 177-210, 1999

Zhe Dang and Richard A. Kemmerer, "A Symbolic Model Checker for Testing ASTRAL Real-time Specifications," *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99),* pages 174-181, IEEE Press, 1999

Zhe Dang and Richard A. Kemmerer, "Using the ASTRAL Model Checker for Cryptographic Protocol Analysis," *Proceedings of DIMACS Workshop on Design and Formal Verification of Security Protocols,* 1997

# Abstract


Debugging and Verification of Infinite State Real-time Systems

Zhe Dang

The successes of the automatic verification of finite state systems describing protocols, hardware devices and reactive systems have lead to much research in automatic verification of infinite state systems. In this dissertation, we consider a special, yet very important, class of infinite-state systems, real-time systems. Real-time systems are widely regarded as a natural application area for formal methods, since the presence of a time variable makes the systems more difficult to specify, design and test. In this research, time is considered to be discrete. This makes it possible for us to use a number of existing results in automata theory. Instead of restricting our research to be purely theoretical, we tie the questions to an existing specification language, called ASTRAL, which has been used to specify a number of real-world systems.

Two subsets of the language, Small-ASTRAL and Mini-ASTRAL, are defined. They both preserve the most important timing features of ASTRAL. Though Small-ASTRAL is undecidable, it encourages us to investigate approximation techniques (partial image, random walk and dynamic environment generation) for debugging Small-ASTRAL specifications. Experimental results are presented and analyzed on a benchmark by using different approximation techniques as well different search strategies. One of the conclusions of this dissertation is that the proposed approximation techniques are effective for debugging a specification in a much shorter time than without using them. We also introduce a number of new proof techniques to show that both history-independent Mini-ASTRAL and the entire Mini-ASTRAL are decidable. These techniques allow us to further investigate other extensions of timed automata.

In theory, we introduce two new models for infinite state systems. The first model is a timed automaton coupled with a multi-queue automaton so that the resulting machine retains the decidability of a class of Presburger formulas. The second model is a class of multicounter machines with mixed types of counters. This model is capable of providing an alternative theoretical tool for analyzing various timed models.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

A basic formal verification problem is, given an implementation $\mathcal{I}$ and a specification or property $\mathcal{S}$ of a system show that $\mathcal{I}$ satisfies $\mathcal{S}$. For programs, the problem can be cast in Hoare's axiomatic framework [H69],

$$\{P\}Q\{R\},$$

which means, if the precondition $P$ holds before program $Q$ executes, then the postcondition $R$ holds after $Q$ completes. Hoare gives rules on how to compute the precondition from the postcondition, then the verification can (possibly) be carried out by computing $P$ from $R$. However, for systems that never terminate, like an air-traffic controller, which continuously interacts with its environment, the above correctness definition does not apply. Instead of formalizing correct *outputs* of the system, we need to characterize correct *behaviors* of the system.

Since Pnueli's landmark work [P77], people believe that temporal logics are a suitable formalism to specify behaviors of the above mentioned systems. For a finite state transition system, there are two views for describing behavior.

- A behavior is an (infinite) path of state transitions. A temporal logic that is interpreted on paths is called linear-time temporal logic (LTL) (see [E90] for a survey).

- A behavior is an (infinite) tree. Branches in a tree correspond to nondeterministic choices of the system. A temporal logic that is interpreted on these trees is called branching-time temporal logic such as computation tree logic (CTL) (see [E90] for a survey).

The above mentioned verification problem is therefore reformulated as the *model-checking* problem:

Given a finite state system $\mathcal{I}$ and a temporal logic formula $\mathcal{S}$, does $\mathcal{I}$ satisfy $\mathcal{S}$?

This problem is decidable. An easy way to see the decidability is as follows. For LTL, both $\mathcal{I}$ and $\mathcal{S}$ can be treated as Büchi automata – simply followed by the classical work of Büchi [B60] that relates S1S-definability with $\omega$-regularity (at a time *well before* temporal logics were introduced). The model-checking problem is therefore reiterated as the emptiness problem of the product automaton from $\mathcal{I}$ and the negation of $\mathcal{S}$ [W83]. Thus, it is decidable. In contrast, for CTL (as well as CTL* [EH86]) things are parallel to LTL, but another kind of automata called Rabin tree automata (instead of working on an $\omega$-word, these automata work on an $\omega$-tree.) are used. The establishment, by Rabin [R69], of the relation between S2S-definability and Rabin recognizability (again well before temporal logics) immediately gives the decidability of the model-checking problem. But these translations from a logic formula to an automaton are not efficient. Better ways (see [E90] for a survey, also [VW88, GPVW95, CVWY92]) are through Fisher-Lander closures of $\mathcal{S}$ (called the tableau method). It is known [SC85, W86, CES83, LP85] that the model-checking problem is PSPACE-complete for LTL and (deterministic) polynomial time for CTL. Notice that, the complexity is in the size of the transition system $\mathcal{I}$ and the size of the formula $\mathcal{S}$. Usually $\mathcal{I}$ is not small, and often its size is comparable to the whole state space. Thus, in a practical sense, a model-checker that checks $\mathcal{I}$ against $\mathcal{S}$ suffers from the state-explosion problem, since the size of the state space grows exponentially in the number of variables and concurrent components of the system.

The successes of the automatic verification of finite-state systems describing protocols, hardware devices, and reactive systems [CW96] have benefited mostly from the invention of symbolic model checking techniques [M92] (see also [BCM92, CGHJLMN93, CGL92]). For a large system, it is not practical to enumerate the states one by one through explicit-state exploration. Thus, it would be better if the exploration is carried out on a set of states.

A succinct representation technique like Binary Decision Diagrams (BDD) [B86, B92] of a finite set of states is naturally used. In this way, symbolic model-checking is accomplished through image computations. Here, an image refers to the symbolic representation (such as in BDDs) of a finite set of states (or even the transition system $\mathcal{I}$ and the specification $\mathcal{S}$ themselves). Efforts to reduce the state-explosion problem usually involve introducing new image reduction techniques that bring down the cost of image computations, such as symmetry reduction [ND96], abstract interpretations [CGL92, DGG97] and partial order [GW94].

For infinite state systems, the verification problem[Esp97] is much harder. There are at least two obvious difficulties. One is

what are the models for infinite state systems $\mathcal{I}$ ?

Unlike finite state systems, we simply do not have a universal model that is adequate to study the verification problem. Turing machines are considered as a universal model for computing, but here they are not interesting for addressing the decidability of the verification problem. The reason is that they are so powerful that the halting problem is undecidable. This immediately eliminates the possibility of finding a decidable procedure for verifying nontrivial properties with them. The second difficulty is

what are the formalisms for the specification $\mathcal{S}$ ?

Fortunately, for simpler infinite state systems, careful variations of temporal logics are still adequate to formulate $\mathcal{S}$, such as the ones in [AD94, ACD93, HNSY94, BEM97, FWW97, Wa97]. However, for complex infinite state systems, current research is still in the stage of investigating the possibility of model-checking specific kinds of systems against specific kinds of properties. The research in this dissertation is not an exception. Here, we consider a special, yet very important, class of infinite-state systems, real-time systems [JM86]. A real-time system can be defined as a system that performs its functions and responds to external events within a specified amount of time. Telephone switches, robot controllers and air-traffic controllers are examples of real-time systems. A real-time system must satisfy not only functional correctness requirements, but also timeliness requirements. Real-time systems are widely regarded as a natural application area for formal methods, since the presence of a time variable makes them more difficult to specify, design and test.

In this research, time is considered discrete. This makes it possible for us to use a number of existing results in automata theory. We distinguish two kinds of real-time systems: finite state real-time systems and infinite state real-time systems, depending on the number of states in a system when clocks are ignored. A typical example of finite state real-time systems is timed automata [AD94], where finite state machines are augmented with a number of clocks. An example of infinite state systems is pushdown timed automata [DIBKS00b], where pushdown machines are augmented with a number of clocks. In this dissertation, we consider both kinds of real-time systems. Further, instead of considering general temporal properties, we focus on a simpler but practically important class of properties called safety properties, which are satisfied by every reachable state. Verification of these properties is called safety analysis. We have noticed that there is a paradox in model-checking. Applications need a strong model so that a complex system can be specified. In contrast, a strong model (like a Minsky machine) capable of doing this usually does not have a decidable model-checking problem, even for safety properties. The research in this dissertation tries to give a number of answers to the following two questions:

(1). What kinds of infinite state real-time systems have a decidable safety analysis problem?

(2). For systems with an undecidable safety analysis problem, what can we do?

Instead of restricting our research to be purely theoretical, we couple the questions with an existing specification language, called ASTRAL [CGK97, CKM94, CPK94], which has been used to specify a number of real-world systems [BCF91, CKM94, BBKT95, CGK97, DK97, DK99a]. In this way, we believe the results presented here are made more practical.

The remainder of the dissertation is organized as follows.

Chapter 2 provides an overview of the real-time specification language ASTRAL. Instead of thoroughly giving the full features of this very expressive language, we focus on the difficulties that ASTRAL presents from the model-checking point of view. Two subsets of the language, Small-ASTRAL and Mini-ASTRAL, are defined. They both preserve the most important timing features of ASTRAL. Though Small-ASTRAL does not have a decidable safety analysis problem, it encourages us to investigate approximation techniques for debugging safety properties. In contrast, Mini-ASTRAL does have a decidable safety analysis problem. Thus, it allows us to fully verify a Mini-ASTRAL specification.

Chapter 3 presents three approximation techniques for debugging a Small-ASTRAL specification. The techniques are partial image, random walk and dynamic environment generation. A model-checker is implemented to carry out image calculations on a bounded-depth execution tree of a specification. Experimental results are presented and analyzed on a benchmark by using different approximation techniques as well different search strategies. One of the conclusions is that the approximation techniques proposed are effective for debugging a specification in a much shorter time than without using them.

Chapter 4 considers a special case for Mini-ASTRAL, i.e., history-independent Mini-ASTRAL. We show that a history-independent Mini-ASTRAL specification can be related to a timed automaton [AD94]. We also show that it has a decidable safety analysis problem, by showing a stronger result: the binary reachability of a (discrete time) timed automaton is Presburger. The technique used in the proof is more important than the result. It enables us to separate controls (i.e., enabling conditions of a transition) from clock progress. A number of extensions are also considered by introducing parameterized durations and non-region properties into the language. A number of decidable approximation techniques are proposed on these extensions.

Chapter 5 shows that the entire Mini-ASTRAL has a decidable safety analysis problem. The proof is based upon a new construction by eliminating history operators in Mini-ASTRAL and using the results in Chapter 4. In short, the main result in this chapter says that adding history operators in Mini-ASTRAL does not increase its expressive power.

Chapter 6 describes two theoretical results by looking at other infinite state systems. The first result deals with queues and clocks. Queues are useful in specifying application systems such as a scheduler. However, queues seem hopeless for verification, since a finite state machine augmented with one queue is as powerful as a Turing machine. We are, however, able to make some progress with this problem. In particular, a new real-time machine model is carefully defined containing multiple queues as well as clocks with certain restrictions. It is shown that the safety analysis problem is decidable for the machine by giving a characterization of binary reachability. This result allows us to specify and verify a number of interesting applications. The second result in this chapter discusses the verification problem of counter machines. In theory, counter machines have the same power as Turing machine. Thus, it is necessary to restrict the counter machine model in order to make the safety analysis problem decidable. We consider a new class of counter machines that include piecewise-monotonic counters, blind counters[G78] and reversal-bounded counters[I78]. The main result shows that the binary reachability of these machines is Presburger. This result will allow us to further investigate a number of non-standard timed systems in the future.

Chapter 7 concludes this dissertation research and proposes a number of future research directions.

# Chapter 2

# ASTRAL, Small-ASTRAL, Mini-ASTRAL and Challenges

ASTRAL [CGK97] is a high-level formal specification language for real-time systems. It is provided with structuring mechanisms that allow one to build modularized specifications of complex systems with layering. A real-time system is modeled by a collection of process specifications and a single global specification. Each process specification consists of a sequence of levels; each level is an abstract view of the process being specified. ASTRAL has been successfully used to specify a number of interesting real-time systems, including a CCITT system [CKM94], a complex wide-area phone system by composing several ASTRAL specifications [CGK97], a hardware description language [BCF91], a robot control system [BBKT95], cryptographic protocols [DK97], and Mobile IP [DK99a].

ASTRAL is a very expressive language. However, it is not theoretically possible to have a fully decidable verification or even semi-decidable verification of the entire ASTRAL language. Thus, we will focus on a subset of the ASTRAL specification language within which the most important features are preserved. The subset we consider is called Small-ASTRAL in which nontrivial approximation approaches for verification may exist. An even smaller subset, called Mini-ASTRAL, is also proposed for which an automatic verification procedure exists.

ASTRAL semantics are well-defined in [CPK94], where both axiomatic and model-theoretic semantics are given. However, it is normally convenient to use specific transition systems as the underlying computational models to study model-checking problems. Especially for real-time systems, timed automata [AD94] are considered as a standard model. Thus, in this chapter, we relate Small-ASTRAL (as well as Mini-ASTRAL) to timed automata and conclude that in order to study the verification problems of the two subsets, timed automata must be significantly extended and in fact, the extensions are not well studied in the literature.

This chapter is organized as follows. Section 2.1 gives a brief overview of the ASTRAL specification language. In Section 2.2, we will discuss from both a theoretical and a practical point of view what should not be included in Small-ASTRAL. In Section 2.3, we will discuss what must be included in Small-ASTRAL in order to preserve the most important features that are not covered by other real-time specification languages, but that are important and essential to building complex real-world specifications. The result is a subset of ASTRAL, called Small-ASTRAL and the definition of Small-ASTRAL is given in that section. In Section 2.4, we further restrict ourselves to a subclass of Small-ASTRAL, called Mini-ASTRAL, for which the model-checking procedure will be provided in the following chapters. Section 2.5 focuses on the challenges provided by the languages and points out some challenges that have not been well-studied in the literature, including many that have never been tried. In that section we also propose some new approaches to the problems.

## 2.1 ASTRAL Overview

A railroad crossing specification is used as a benchmark example throughout the remainder of this dissertation to illustrate various features of ASTRAL. The system description is taken from [HL94]. The system consists of a set of railroad tracks that intersect a street where cars may cross the tracks. A gate is located at the crossing to prevent cars from crossing the tracks when a train is near. A sensor on each track detects the arrival of trains on that track. The critical requirement of the system is that whenever a train is in the crossing the gate must be down, and when no train has been in between the sensors and the crossing for a reasonable amount of time, the gate must be up. The complete ASTRAL specification of the railroad crossing system can be found in Appendix A.

An ASTRAL specification includes a global specification and process specifications. The global specification contains declarations of process instances, global constants, nonprimitive types that may be shared by process types, and system level critical requirements. There is a process specification for each process type declared in the global specification. Each process specification consists of a sequence of levels; each level is an abstract view of the process being specified.

### 2.1.1 Processes, Constants, Variables, and Types

The global specification begins with a process type declaration:

```
PROCESSES
   the_gate: Gate,
   the_sensors: array [ 1..n_tracks ]  of Sensor.
```

This declaration indicates that there are `n_tracks` sensor instances of type `Sensor` in the system, where `n_tracks` is a global constant declared in the constant declaration part:

```
CONSTANT
   n_tracks: pos_integer.
```

In a global specification, the number of process instances must be fixed, but could be a parameterized constant, e.g. `n_tracks`. This feature allows a user to specify a system with a parameterized number of process instances. ASTRAL is a strongly typed language. Primitive types include `Integer, Real, Boolean, ID` and `Time`. Constructed types can be declared in the type declaration parts of the specification by using the `TYPEDEF` construct. For instance, `pos_integer` is defined as positive integers:

```
TYPE
   pos_integer: TYPEDEF i: integer ( i > 0 ).
```

The type `ID` is one of the primitive types of ASTRAL. Each process instance has a unique identifier. The ASTRAL specification function `IDTYPE(i)` returns the type of the process with the identifier `i`. The `IDTYPE` function is used in the global declaration to define `sensor_id`, which represents all identifiers of process instances of type `Sensor`:

```
TYPE
   sensor_id: TYPEDEF i: id ( IDTYPE  ( i )  = Sensor ).
```

Other type constructors in ASTRAL include `STRUCTURE`, `LIST` and `SET`. A `STRUCTURE` type is very similar to the one in a high-level programming language. It has a fixed number of fields with each again associated with an ASTRAL type. A `LIST` type is the set of (finite) sequences of data drawn from an ASTRAL type. A `SET` type indicates sets of data with an ASTRAL type, e.g., sets of integers. Data operations are the standard ones. For instance, they include almost all the Boolean and arithmetic operations.

In the railroad crossing specification there are two process specifications, `Gate` and `Sensor`. A process specification includes an interface section which specifies the imported variables, types, transitions and constants (from either the global specification, or exported by other processes) used by the process, and the variables and transitions exported by the process. ASTRAL does not have global variables. Therefore, variables should be declared in each process specification. The format used to declare variables is similar to that of the constant declaration above. Also, each process instance can have its own constants. Constants with the same names in distinct process instances do not necessarily share the same value. Since ASTRAL includes unbounded data types like `integer`, a variable or a constant is not necessarily bounded. Therefore, this language is able to specify infinite state systems. ASTRAL supports a modularized design principle: every variable is associated with a unique process instance, and changes to the variable can only be caused by the transitions specified inside that process instance. This is discussed further in the next subsection.

### 2.1.2 Transitions

The ASTRAL computation model is defined by the execution of state transitions. Transitions are only specified inside process specifications. Therefore, each transition in a process instance can only change the variables specified inside that process instance. The body of an ASTRAL transition includes pairs of entry and exit assertions with a nonzero duration indicated for each pair. The entry assertion must be satisfied at the time the transition starts, whereas the exit assertion will hold after the time indicated by the duration from when the transition fires. For example, in process `Gate`, the transition,

```
TRANSITION up
   ENTRY           [ TIME : up_dur ]
             position = raising
           & now - End ( raise )  >= raise_time
   EXIT
             position = raised,
```

specifies the gate being fully raised, after it has been rising for a reasonable amount of time (`raise_time`). The duration of this transition is indicated by the constant `up_dur`. In ASTRAL, `End(T,t)` is a predicate that is true if and only if the transition `T` ends at time `t` and there is no other time after `t` and before the current time (`now`) when `T` ends. `End(T)` is used to indicate the time `t` such that `End(T,t)` holds. `Start(T)` is defined similarly for the start time of `T`. A transition instance is fired if its entry assertion is satisfied and no other transition in the same process instance is executing. The execution of this transition instance is completed after the duration indicated in the transition specification. A transition can be exported. In this case, its entry assertion alone can not decide whether it is firable. According to the ASTRAL semantics, an exported transition must be called from the external environment in order to fire. `Call(T)` is used to indicate the time when a call to the exported transition `T` is made. ASTRAL broadcasts variable values instantaneously at the time that a transition finishes. Other process instances may refer to these variables as well as the start and end times of transitions under the assumption that these variables and the transitions are exported and the listening process instance properly imports them.

When it is the case that there is more than one transition instance that is enabled inside the same process instance, one of the enabled transitions is nondeterministically chosen to fire, assuming that there is no other transition executing at the moment. Inside a process instance executions of transitions are non-overlapping interleaved, while between process instances, maximal parallelism is supported. Thus, the execution of transition instances in different process instances is truly concurrent.

### 2.1.3 Assumptions and Critical Requirements

In addition to transitions, requirement descriptions are also included as a part of an ASTRAL specification. They comprise axioms, initial clauses, imported variable clauses, environmental assumptions and critical requirements. An axiom is usually used to specify a property about constants. An initial clause defines the possible system states at startup time. An imported variable clause defines the properties the imported variables should satisfy, for instance, patterns of changes to the values of imported variables and timing information about transitions exported from other processes. ASTRAL uses environmental assumptions to characterize the environment. An environment clause formalizes the assumptions that must hold on the behavior of the environment to guarantee some desired system properties. Typically, it describes the pattern of invocation (`Calls`) of exported transitions. The critical requirements include invariant clauses and schedule clauses. An invariant expresses the properties that must hold for every state of the system that is reachable from the initial state, no matter what the behavior of the external environment is. A schedule expresses the properties that must hold provided the external environment and the other processes in the system behave as assumed (i.e., as specified by the environmental assumptions and the imported variable clauses).

### 2.1.4 Use of Time

Since ASTRAL is intended to specify complex real-time systems, timing constructs are extensively used in a typical ASTRAL specification. As an example, below is the imported variable clause of process `Gate`.

```
FORALL s: sensor_id
   ( Change ( s.train_in_R, NOW )
     & ~s.train_in_R
  ->   0 <= NOW -  ( RIImax - response_time )
      & FORALL t: time
         ( NOW -  ( RIImax - response_time )  <= t
           & t < NOW
```

```
            ->   Past ( s.train_in_R, t )  )  )
```

It says that once a sensor reports a train, it will keep reporting a train at least as long as it takes the fastest train to exit the region. In the formula, `NOW` is the global clock indicating the current time. `Change(s.train_in_R, t)` is used to indicate the variable `s.train_in_R` changes at time `t`. `Past(s.train_in_R, t)` stands for the (past) value of the variable `s.train_in_R` at time `t`. In addition to these, a number of quantifications can be applied, e.g., `FORALL t: time`. Thus, timings are explicitly used in ASTRAL in the following ways:

- A transition has a positive duration that can be parameterized,
- Clocks, including `NOW` and `Start`, `End`, `Call` times of a transition, as well as change times of a variable, can be referenced,
- Histories of a variable can be referenced using `Past`,
- A number of quantifications over time can be applied to a formula,
- Clocks can be involved in an arithmetic expression containing parameterized constants. For instance, `NOW-(RIImax-response_time)` appears in the imported variable clause.

The complex use of timings allows ASTRAL to specify complex real-world systems. In a finite state system, properties can be specified by temporal logics such as CTL and LTL. However, timing is not explicit in these logics. In fact, the property is only able to characterize orderings of states without counting. With time explicitly used, timed automata [AD94] are able to describe systems with clocks and comparing a pair of clocks against a concrete constant. A later inclusion [AHV93] of parameterized constants makes it possible to specify systems with parameterized timing constraints. But these systems are history-independent, i.e., `Past` constructs can not be expressed.

### 2.1.5 Modularized Proof Theory

Modularization means the principle that a system specification can be broken into several loosely coupled functional modules. This greatly eases both the verification and the design work. Although most high level specification languages support modularization, each module in the specification is only a syntactical module. That is, those languages provide a way to write a specification as several modules; however, there is no way to verify the correctness of each process without looking at all the behaviors of all the other processes. In verifying a real-time system, it is extremely important to restrict the size of the system. In ASTRAL, a specification is composed of several process specifications. A process instance can be understood as a module, which provides an interface section including a declaration of the imported variables and transitions and an imported variable clause, as was mentioned before. This helps to develop a modular verification theory for real-time systems.

The ultimate goal of modularization is to partition a large system, both conceptually and functionally, into several small modules and to verify each small module instead of verifying the large system as a whole. This can be done in ASTRAL, according to the ASTRAL proof theory[CKM94], which is briefly introduced below. For example, the global invariant of an ASTRAL specification can be verified by using only the invariants for each process instance. It is not necessary to look at the details of each process instance's behavior. The global schedule can be verified by using only the global environment and the schedules in each process instance. In turn, verifying the schedule and the invariant of each process instance uses only the process's local assumptions and behaviors. For instance, verifying the local invariant uses only the behaviors of transitions of the process instance. Verifying the local schedule uses the process's local environment and imported variable clause, plus the behaviors of the process's transitions. The imported variable clause must be a correct assumption and needs to be verified by combining all the invariants from all the other process instances. Based upon this theory, this dissertation only discusses process-level model-checking for ASTRAL.

## 2.2 What should Not be Included in Small-ASTRAL

It would be wonderful if we could do model-checking for the entire ASTRAL language. But this section will show that this is theoretically impossible. We will discuss from a theoretical and a practical point of view what should not be included in Small-ASTRAL. The goal is that while keeping the most important features of ASTRAL, Small-ASTRAL is suitable for studying the possibility of automatic model-checking procedures either with or without approximations.

### 2.2.1 Primitive Data Types in ASTRAL

In ASTRAL, primitive data types include `Boolean, Integer, Real, ID` and `Time`. The only primitive type we choose not to include in Small-ASTRAL is `Real`. The reason for this choice is as follows.

- Classical automata theory focuses on discrete objects. Not including `Real` makes it possible to study the existence of decision procedures for various subsets and their extensions of Small-ASTRAL using a number of existing results in automata theory.

- We are not aware of a practical solver for first-order arithmetic formulas with mixed domains including real variables and integer variables. Although it is interesting to investigate model-checking problems for specifications on mixed domains, we prefer in this dissertation to only look at discrete transition systems augmented with complex timing constraints and discrete clock values.

In contrast, it is equally worthwhile to exclude `Integer` (and keep `Real`) in Small-ASTRAL. In the future, we would like to build a model-checker for a subset of Small-ASTRAL with continuous clocks using a real solver, which is similar to the one used in HyTech [HHW97], a model-checker for hybrid systems.

### 2.2.2 Constructed Data Types

In addition to primitive types, ASTRAL allows constructed data types including `Set`, `List`, `Struct`, *wff-type* (defined by an ASTRAL well-defined formula), `Array`, as well as enumerated types. It is obvious that we can not include `Set` nor `List` types. The reasons are as follows. In ASTRAL, a variable $x$ can be declared as a variable on the domain of the power set of integers, i.e., values of $x$ are subsets of integers:

```
TYPE
    x: SET OF integer.
```

Since the size of $x$ is not necessarily bounded, using the rich set of set operators provided by ASTRAL, which includes all the standard set operations, and an appropriate transition system, the whole Peano Arithmetic can be expressed, which includes all partial recursive functions. It follows immediately that such systems are undecidable for the domain emptiness of a function and the emptiness can be expressed as an ASTRAL invariant. The situation is similar for lists. In ASTRAL, `List` represents a list of data values; again the lengths are not necessarily uniformly bounded. For instance, a variable $x$ can be declared as a variable on the domain of all the lists (sequences) of integers:

```
TYPE
    x: LIST OF integer.
```

The `List` operators provided by ASTRAL are strong enough to define $x$ as a FIFO queue. However, it is known that a finite automaton equipped with a queue is enough to simulate a Turing machine. Thus, allowing `List` in Small-ASTRAL would make the language too expressive to lead to any interesting decidability results on model-checking. However, both `Set` and `List` are useful in a number of ASTRAL specifications. For instance, a queue makes it easy to specify a real-time process scheduler. If `Set` and `List` are required to have explicitly bounded sizes, then we can use a number of variables to denote its content. In this case, a specification can be manually translated into one without `Set` and `List` variables.

A `Struct`-typed variable in ASTRAL has finitely many component variables indicating the designated fields. Thus, a `Struct`-typed variable can be translated into a number of variables representing each field. Therefore, it is reasonable to keep this convenient construct.

A wff-type is defined through an ASTRAL well-defined formula. For instance, "positive integers" can be defined as a wff-type as follows:

```
TYPE
    pos_integer: TYPEDEF i: integer (i>0).
```

However, things could get unexpectedly complex if we do not restrict the form of wffs appearing after `TYPEDEF`. For instance, the set of all the solutions $x$ of a polynomial equation like $\exists i(i^2 - 3x^3 = 5)$ can be declared as a wff-type. Due to the well known undecidability of

Hilbert's Tenth Problem (HTP) [M70], it is undecidable to know whether even such a defined wff-type is empty or not. In addition to the theoretical difficulties, we have noticed that the intended purpose of including wff-types in ASTRAL is to allow a user to conveniently define some commonly used data types in the language, such as positive integers, the identifiers of all the process instances of some specific process type, etc. Thus, we will not exclude wff-types from Small-ASTRAL, but include several significant restrictions. We will make the restrictions clear later in Section 2.3.3.

Due to similar reasons as for `List`, arrays could be preserved as long as the size is bounded. Of course, the enumerated type can be preserved since an enumerated type variable is not different from a finite state variable.

### 2.2.3 Data Operations

Since `Set` and `List` are not contained in Small-ASTRAL, all the set and list operators are excluded. Nonlinear operators should be excluded, since they can cause undecidability in satisfiability due to the undecidability of HTP. Thus, only multiplication by an integer constant is allowed. Division is handled similarly.

Complex timing constraints make ASTRAL a powerful language for specifying real-time systems and they are also the major research area for real-time verification. Thus, we decided that all the time-related constructs should be included in Small-ASTRAL in order to inherit the most important timing features from ASTRAL. They include `Start, End, Call, Change, Past` as well as quantification over time. However, as we will see in the next section, we have to restrict their usage to a special form in order to make model-checking Small-ASTRAL possible.

## 2.3 Small-ASTRAL Grammar

Instead of adding new grammar constructs, in defining Small-ASTRAL, we only drop a number of grammar rules from the standard ASTRAL grammar [K99b] and apply more restrictions. This will make Small-ASTRAL compatible with the current ASTRAL SDE interface. As we mentioned earlier, the goal is that while keeping the most important features of ASTRAL, Small-ASTRAL is suitable for studying the possibility of automatic model-checking procedures either with or without approximations. A Small-ASTRAL specification consists of a single global specification and a number of process specifications. In the following subsections, we give the grammar rules to define the global specification and each process specification, along with the explanation of the reasons why we put specific restrictions on the grammar rules. The complete grammar for Small-ASTRAL is in Appendix B.

### 2.3.1 The Global Specification

Each Small-ASTRAL specification contains one global specification. Therefore, we do not consider compositions [CGK97]. A global specification is defined as follows.

```
global_spec:
    GLOBAL SPECIFICATION IDENTIFIER
    PROCESSES processes_decl_list
    type_clause
    axiom_clause
    constant_clause
    define_clause
    environment_clause
    invariant_clause
    schedule_clause
    END IDENTIFIER
```

A global specification starts from `processes_decl_list`, the process instance declarations, followed by global type declarations, global axiom clause, global constant declarations, global define declarations, global environment clause, global invariant clause, and global schedule clause. Each will be made clear in the following subsections.

**Process Instance Declarations**

This part indicates the process instances in the whole system with respect to each process type declared. For instance, a declaration

```
PROCESSES
   the_gate: Gate,
   the_sensors: array [ 1..n_tracks ]  of Sensor
```

indicates that there are `n_tracks` sensor instances of type `Sensor` in the system. The grammar rules for this part are self-evident:

```
processes_decl_list:
     processes_decl
     | processes_decl_list COMMA processes_decl
```

with

```
processes_decl:
     id_list COLON IDENTIFIER
     | id_list COLON ARRAY OPENSQUARE id_integer CLOSESQUARE
           OF IDENTIFIER
     | id_list COLON ARRAY OPENSQUARE id_integer DOT DOT
           id_integer CLOSESQUARE OF IDENTIFIER
```

where `id_list` is a sequence of `IDENTIFIER`'s and `id_integer` is either an `IDENTIFIER` or an `INTEGER_CONST`. `id_integer` is used to give the number of instances of a process type. We further require that it is either an integer or a parameterized constant of integer type.

**Global Type Declarations**

Global type declarations include a list of global type definitions, with each in the following form:

```
type_decl:
      IDENTIFIER colon_is OPENROUND id_list CLOSEROUND
```

```
| IDENTIFIER colon_is STRUCTURE OF OPENROUND
                id_type_list CLOSEROUND
| IDENTIFIER colon_is TYPEDEF IDENTIFIER colon_is any_type
     OPENROUND small_astral_wff CLOSEROUND.
```

The first line declares an enumerated type, the second line declares a `Struct`-type. The last line is a declaration of a wff-type. However, unlike the case for ASTRAL, we restrict the wff in the form to `small_astral_wff`, which will be defined in Section 2.3.3.

#### Global Assumption and Property Clauses

The `axiom_clause`, `environment_clause`, `invariant_clause` and `schedule_clause` constitute the global assumptions and the global properties. Each is in the form of `small_astral_wff`, which will be defined later.

#### Global Constant Declarations

Global constants are declared as a pair of a constant name and the data type of the constant. The name can be further appended with a number of parameters, and each should be of a bounded type (the bound, however, could be parameterized). The type is either one of the primitive types in Small-ASTRAL or a constructed type declared in the global type declaration part as above. We omit the rules here.

#### Global Define Clauses

Using `DEFINE`, a segment of an ASTRAL wff-formula can be abbreviated as an identifier. The rule used is:

```
define:
       IDENTIFIER COLON any_type EQEQ small_astral_wff
     | IDENTIFIER OPENROUND id_type_list CLOSEROUND COLON
            any_type EQEQ small_astral_wff
```

where `any_type` is one of the Small-ASTRAL primitive types and the constructed types. `id_type_list` is the parameter list of the identifier `IDENTIFIER`. The segment must be a small_astral_wff that will be defined in Section 2.3.3.

### 2.3.2 A Process Specification

In addition to the global specification, a Small-ASTRAL specification includes a number of process specifications. Each process specification is defined as in the following grammar.

```
process_spec:
    import_clause
    export_clause
    environment_clause
    impvar_clause
    type_clause
    axiom_clause
    variable_clause
    constant_clause
    define_clause
    initial_clause
    invariant_clause
    schedule_clause
    trans_decl_list
```

import_clause and export_clause are the same as in an ASTRAL process specification, listing all the imported variables used inside the process that are exported from other processes, and all the exported variables of the process that are used by other processes. Together with the imported variable clause impvar_clause, they constitute the interface of the process when considering the current process instance as a module. The local assumptions (including initial_clause, environment_clause, impvar_clause, and axiom_clause) and the local properties (including invariant_clause and schedule_clause) have the same grammar rules as the global assumptions and global properties. That is, they must be small_astral_wff. variable_clause and constant_clause declares the local variables and local constants of the process and have the same format as for global constant declarations. Also define_clause is similar to the global define declarations. Each process has a section trans_decl_list describing the local transition system, which includes a number of transition declarations trans_decl.

A transition declaration includes a heading, an entry assertion, and an exit assertion followed by a number of except-exit pairs:

```
trans_decl:
    TRANSITION trheading
    ENTRY OPENSQUARE TIME COLON duration CLOSESQUARE
         small_astral_wff
    EXIT small_astral_wff
    opt_except_list.
```

The heading provides the name of the transition followed by an optional list of parameters. We further restrict that each parameter should be within a finite domain, possibly bounded by a parameterized constant. Thus, for each transition name, there are at most finitely many parametric transition instances in the process. The entry assertion is a small_astral_wff with duration (either an integer constant or a declared constant) indicating the duration of this entry-exit pair. The exit assertion is also a small_astral_wff. A number of except-exit pairs could follow the entry-exit pair. Each pair would have a format similar to the entry-exit pair.

Probably the most significant difference between Small-ASTRAL and ASTRAL is the wff-formulas used. The following subsection defines Small-ASTRAL wffs.

### 2.3.3 Small-ASTRAL Well-formed Formulas

The major restriction put on ASTRAL well-formed formulas in order to define Small-ASTRAL wffs is that we only allow linear operations on integer-valued variables and restrict the use of timing-related constructs. As usual, all the Boolean operations are kept:

```
small_astral_wff:
      small_astral_wff IFF small_astral_wff
    | small_astral_wff NIFF small_astral_wff
```

```
| small_astral_wff IMPLIES small_astral_wff
| small_astral_wff NIMPLIES small_astral_wff
| small_astral_wff OR small_astral_wff
| small_astral_wff NOR small_astral_wff
| small_astral_wff AND small_astral_wff
| small_astral_wff NAND small_astral_wff
| NOT small_astral_wff
```

along with a number of comparisons (called atomic wffs) and arithmetic operations:

```
| small_astral_wff EQ small_astral_wff
| small_astral_wff NEQ small_astral_wff
| small_astral_wff LT small_astral_wff
| small_astral_wff NLT small_astral_wff
| small_astral_wff LTE small_astral_wff
| small_astral_wff NLTE small_astral_wff
| small_astral_wff GT small_astral_wff
| small_astral_wff NGT small_astral_wff
| small_astral_wff GTE small_astral_wff
| small_astral_wff NGTE small_astral_wff
| small_astral_wff PLUS small_astral_wff
| small_astral_wff MINUS small_astral_wff
| small_astral_wff TIMES INTEGER_CONST
| small_astral_wff DIVIDE INTEGER_CONST
| small_astral_wff MOD INTEGER_CONST
| MINUS small_astral_wff
| IDTYPE ( IDENTIFIER ).
```

In addition to constant values like `TRUE`, `FALSE` and `Self`, the global clock variable `NOW`, and the composite identifiers `id_combo` (like `P[i].x[address]` representing the `Struct`-typed local variable `x`'s field `address` of the process instance `P[i]`), a Small-ASTRAL formula can contain complex timing constraints and quantifications. These are discussed in the following subsections.

**Limited Uses of Timing constructs**

Timing constructs include `Start, End, Call, Change,` and `Past`. The limitations we put on them are as below.

- ASTRAL allows `Start, End, Call, Change` that trace up to any number of previous occurrences. For instance, `Start[n](T)` indicating the `n`-th last start time of the transition `T`. In Small-ASTRAL, we limit `n` to be either 1 or 2. In fact, having `n` more than 2 is rare in most of the existing ASTRAL specifications. In addition, by adding auxiliary clocks, `Start[n](T)` can be expressed in terms of the limited version for any constant `n`.

- `Past` can not be nested with other timing constructs. For instance,

  $$\texttt{Past}(\texttt{Start}[2](\texttt{T}), \texttt{t})$$

  is not allowed. The reason is that `Start[2](T)` has unbounded time values, it is impossible to record the whole history of `Start[2](T)` in order to resolve the value `Past(Start[2](T), t)` at some moment `t`.

The following rules, continued from the above grammar, summarize the limitations.

```
small_astral_wff:
    sec sec_times ( id_combo, IDENTIFIER )
    | sec sec_times ( id_combo )
    | NOCHANGE (id_combo)
    | PAST ( id_combo, IDENTIFIER)
```

where `sec` is `Start, End, Call,` or `Change`, `sec_times` is empty, `[1]`, or `[2]`.

**Limited Uses of Quantifications**

There are two types of quantifications in Small-ASTRAL. One is `FORALL` and the other is `EXISTS`. The rule representing them is, continued from the above grammar,

```
small_astral_wff:
      FORALL IDENTIFIER : any_type ( small_astral_wff )
    | EXISTS IDENTIFIER : any_type ( small_astral_wff ).
```

However, we only allow a quantified variable (`IDENTIFIER` in the above rules) to be either over a finite domain or of type `Time`. Thus, the following formula is not allowed: `EXISTS Y : Integer (Y-X>5)`. But `EXISTS Y: Time (Y-X>5)` is legal. `Start(T,t)` can be rewritten by `Start(T)=t`. `End, Call,` and `Change` are handled similarly. We call `sec sec_times ( id_combo )` in the grammar rules that appeared in the last subsection a SEC-clock. A clock variable is either a SEC-clock, the global clock `NOW`, or a quantified variable with type `Time`. We further require that each atomic wff can be re-arranged such that the left hand side contains either a clock or a difference between two clocks, and the right hand side does not contain a clock. For instance, `EXISTS Y: Time (Y>Start(T) -2)` can be rewritten into `EXISTS Y: Time (Y- Start(T)>-2)`. Thus, this is legal. However, `EXISTS Y: Time ( Y+Start(T)>-2)` is illegal. The reason for applying these restrictions is that, in practice, the sum of two clocks, as opposed to the difference of two clocks, is rarely used – this is also the reason that Timed Automata [AD94] became a standard model for real-time systems. Furthermore, allowing addition to clock constraints ($x-y\#c, x\#c$ with $c$ an integer and $x, y$ clocks, where # stands for $>, <$) makes the emptiness of Timed Automata undecidable [AD94]. In the following chapters, we shall see that Small-ASTRAL is much stronger than Timed Automata. Thus, it is reasonable that we eliminate other operations between clocks. For instance, in practice, timing restrictions like `Start(T1)-Start(T2)>Start(T3)-Start(T4)`, which is not a clock constraint, is still useful. This expression states that the interval between the (last) start times of transitions `T1` and `T2` is greater than the same interval for `T3` and `T4`. But it is known that Timed Automata augmented with these kinds of clock constraints are undecidable for emptiness [AD94, AHV93], since doing this gives the automata Turing computing power. Thus, standard clock constraints are the strongest clock relations we can use in a specification language that has a decidable verification procedure.

## 2.4 Mini-ASTRAL

Even though Small-ASTRAL, compared to ASTRAL, is rather restricted, a full verification procedure for it does not exist. We intentionally distinguish two sets of variables in a Small-ASTRAL process instance: one is all the local variables declared in the process and all the global constants, the other is all the clocks in the process instance. That is, we separate clocks from *(control) states* that are all the possible values of the variables. However, control states in Small-ASTRAL can be infinite, since integer-valued variables are allowed. Thus, a counter machine with two counters, which has Turing computing power, can be specified in Small-ASTRAL. This eliminates the possibility of automatically verifying a Small-ASTRAL process instance.

In this section, we propose a subset of Small-ASTRAL, called Mini-ASTRAL, for which a full verification procedure exists. We show the existence in the following chapters.

In order to achieve full verification, we need to further restrict the state space to be finite, while, in addition, keeping all the clocks unbounded. Mini-ASTRAL has the same grammar as Small-ASTRAL, except:

- The total number of process instances in a Mini-ASTRAL specification should be bounded by a natural number.
- Each global constant, local constant, or local variable is a finite state variable. Hence, each transition instance has a bounded duration.
- The number of parametric transition instances for a transition with a number of parameters must be bounded by a natural number.
- A quantified variable is either a variable of type `Time` or a finite state variable. Thus, quantifications on unbounded domains other than `Time` are not allowed.

One can immediately see that with the above restrictions, a Mini-ASTRAL process instance has only finite control states. However, because clocks can be unbounded, complex timing restrictions are preserved.

Mini-ASTRAL is probably the smallest subset of ASTRAL one can think of that still has the most important features of ASTRAL. However, this language touches a number of areas

$$\begin{array}{c} \boxed{\text{Mini-ASTRAL}} \\ \downarrow \\ \boxed{\text{History-independent timed Mini-ASTRAL}} \\ \downarrow \\ \boxed{\text{Untimed Mini-ASTRAL}} \end{array}$$

Figure 2.1: A hierarchy for Mini-ASTRAL

that previous model-checking theories never covered. Thus, we have to provide a new theory to investigate its verification problem. But before doing this, the next section discusses the challenges that Mini-ASTRAL, as well as Small-ASTRAL, present to the current real-time model-checking theories.

## 2.5 Challenges in Model-checking Mini-ASTRAL and Small-ASTRAL

This section presents a hierarchy for Mini-ASTRAL and shows which layer is already known to have a decidable verification procedure. Figure 2.1 shows the hierarchy for Mini-ASTRAL.

The smallest subset is Untimed Mini-ASTRAL, which contains all the Mini-ASTRAL specifications that do not use any clock variables (including the global clock `NOW`). By recalling that the control state space in a Mini-ASTRAL process instance is finite, it is easy to see that this layer covers exactly finite state machines. Also recall that ASTRAL only specifies safety properties and bounded liveness properties; therefore, the properties can be mapped to reachability problems for finite state systems. Thus, it follows immediately that by using the model-checking theories and techniques for finite state systems, as reviewed in Chapter 1, the model-checking problem is decidable for this layer. This area of finite state model-checking has been well-studied.

When we move up to the next layer, the subclass is history-independent timed Mini-ASTRAL. As stated earlier, a system is history-independent if the current state (including the current control state and the current clock values) only depends upon the last state.

In fact, this subclass of Mini-ASTRAL is exactly Mini-ASTRAL without `Past`. A system specified by this subclass of Mini-ASTRAL can be modeled by a timed automaton, as shown below.

A timed automaton can be regarded as a finite state machine with a number of real-valued clocks (or integer-valued clocks). All the clocks progress synchronously with rate 1, and a clock can be reset to 0 at some transition. Each transition also comes with an enabling condition in the form of clock constraints (i.e., Boolean combinations of $x\#c$ and $x-y\#c$ where $x$ and $y$ are clocks, $c$ is an integer constant, and $\#$ denotes $>, <$ or $=$). These constraints are called regions. We use discrete clocks in this section. A standard region technique [AD94] (and more recent techniques [BLPWY99] [W00]) can be used to analyze region reachability.

We use $x_{now}$ to denote a clock in a timed automaton that never resets. Thus, $x_{now}$ is the same as the global clock `NOW` in Mini-ASTRAL. But other clocks in Mini-ASTRAL, for instance `Start(T)`, behave differently from clocks in a timed automaton, since each time `T` starts, the clock `Start(T)` *jumps* to the value of `NOW`. A simple transformation changing each occurrence of `Start(T)` to `NOW-Start(T)` will map these jumps to clock resets in a timed automaton. `Start[2](T)` can be regarded as a new clock to store the previous start time when `T` starts. All the transforms apply to `End, Call, Change` by adding a Boolean variable to indicate that a corresponding event happens at the time a clock jumps. Notice that the transforms keep the clock constraints in the same format. Thus, the resulting automaton has enabling conditions that are clock constraints with quantifications (Recall that we allow quantification over time variables.). But quantifications can be eliminated for clock constraints, since clock regions are closed under projections. Thus, specifications in this subclass can be translated into a timed automaton. Therefore, the model-checking problem for this subclass is decidable. In the following chapters, we will give a new technique to verify specifications in this subclass. In fact, our results show that we can model-check an extended version of this subclass by not allowing the properties to be in the form of clock constraints, such as a Presburger formula. It should also be noted that, since the above transform can be reversed and a clock reset can be modeled by a "reset transition" being fired, this subset of Mini-ASTRAL is capable of modeling timed automata. Thus, this subclass is strong enough to cover most of the timed automata-related models currently being used to specify real-time systems.

The topmost layer in the hierarchy is Mini-ASTRAL itself. Since it uses the history-dependent operator `Past`, it presents new challenges to real-time verification. By allowing history-dependence, it is possible to

- specify more complex real-timed systems,
- modularize the system specification so that a modularized proof theory like the one for ASTRAL can be established.

The benefits are evident. However, a fundamental question is whether adding this history-dependent operator makes automatic verification impossible for Mini-ASTRAL. This problem has never been studied before. An intuitive approach is to use a number of variables to record the history of a local variable. But this is not feasible since time is unbounded, and we would thus use infinitely many variables to record the history. By bounding the depth of execution, we can, however, get an approximation of the reachable states, and this approximation can be used to debug the system. In Chapter 5, we will demonstrate a technique to automatically verify Mini-ASTRAL specifications, without bounding the depth. Thus, we completely solve this problem.

The biggest difference between Mini-ASTRAL and Small-ASTRAL is that the set of control states is infinite in a Small-ASTRAL specification. It makes automatically verifying Small-ASTRAL undecidable. Thus, for Small-ASTRAL, we focus on various approximation techniques to debug a specification. Similar to the hierarchy for Mini-ASTRAL, Figure 2.2 shows three layers of Small-ASTRAL.

The smallest subclass is untimed Small-ASTRAL in which no timing constructs and clock variables are used. Thus, the class covers all the infinite state transition systems with both entry assertions (enabling conditions) and exit assertions being Presburger[1]. This subclass covers most of the counter machine theoretic modeling and verification [CJ98, FS00]. In Chapter 6, we will briefly present some new reachability techniques and compare

[1]But exit assertions use both the current values and previous values of local variables in the process instance.

$$\boxed{\text{Small-ASTRAL}}$$
$$\downarrow$$
$$\boxed{\text{History-independent timed Small-ASTRAL}}$$
$$\downarrow$$
$$\boxed{\text{Untimed Small-ASTRAL}}$$

Figure 2.2: A hierarchy for Small-ASTRAL

the techniques with other work in this area.

The layer above untimed Small-ASTRAL covers all the history-independent timed Small-ASTRAL specifications. These specifications do not use `Past` but may contain all the other timing constructs. A system specified by this subclass of Small-ASTRAL can be modeled by a timed automaton with the following extensions:

- (infinite control states) The set of control states could be infinite.
- (parameterized tests) Clock constraints can be parameterized. For instance, $x-y > D$ where $x$ and $y$ are clocks, and $D$ is a parameterized constant.
- (parameterized durations) A transition may have a parameterized duration. Assigning a parameterized duration makes the verification, even in the sense of approximation, strictly harder, as we will show in a later chapter.

Several subsets of the subclass can be considered. For instance, [AHV93] considers a version of timed automata with finite control states, parameterized tests and unit durations. In the same paper, the authors showed that these timed automata are strong enough to simulate a counter machine with two counters (which has Turing computing power). Thus, the emptiness problem is undecidable. The authors proposed an approximation technique by bounding the depth of execution of the automata, and they showed that the set of all reachable clock configurations is expressible in the additive theory of reals (or alternately, when considering discrete clocks, is Presburger) by doing bounded symbolic execution. As far as we know, there is no work in the literature that considers more general parameterized timed automata by, for instance, allowing infinite control states or by considering parameterized durations. In a later chapter, we will demonstrate a number of approximation techniques that are stronger than the bounded execution approach. In our approaches, the depth of executions is not necessarily bounded.

When considering all of Small-ASTRAL, we currently have no better approximations than bounding the depth of execution. The approach is called Symbolic Bounded-Testing, which will be presented in the next chapter.

# Chapter 3

# Symbolic Bounded-Testing of Small-ASTRAL

In this chapter[1], we will look at bounded-testing approaches to debug a Small-ASTRAL specification. By bounded-testing, we mean that a pre-assigned bound is placed upon all the executions of the system. Thus, the set of all the reachable states within this bound is a subset of all the reachable states. The approaches therefore provide a lower approximation of the reachable set. Since, as we mentioned before, only safety and bounded liveness properties are specified in ASTRAL, these approaches can be used to debug, or more accurately, test a specification.

There are two kinds of approaches that can be considered. One is called *explicit-state bounded-testing* (EBT), and the other is called *symbolic bounded-testing* (SBT).

EBT is applied on a concrete instance of a Small-ASTRAL specification through a tool called the explicit-state model-checker for Small-ASTRAL [DK97, D97, KDK99, DK99a]. By a concrete instance, we mean that each parameterized constant is assigned a concrete value. Thus, the concrete instance has a bounded number of transitions and a bounded number of process instances in the whole system. However, local variables inside each process instance can still have an infinite number of values. A configuration of the instance

[1]Most of this chapter is evolved from the papers [DK97, DK99a, DK99b, KDK99, DK00a, DK00b].

is composed of the values of all the local variables and (global and local) constants. A specification can be simulated as follows. It starts from an initial configuration, traverses every branch of the execution tree (i.e., unfolding the transition system as a tree) up to a given depth. The depth is considered as the bound of the global clock `NOW`. The tree is built by considering the current concrete instance and the ASTRAL semantics. Since the local variables could have an infinite number of values, they can cause unbounded nondeterminism. That is, there are possibly unbounded many branches the model-checker can follow for a configuration. For instance, an `INITIAL` clause `x>5` will make the variable `x` have infinitely many choices of initial values. In order to exclude unbounded nondeterminism and make the exploration procedure feasible, we randomly pick a concrete value for `x` that satisfies the condition. Since Small-ASTRAL is also history-dependent, we have to store all the past values of a variable `x` if `Past(x,t)` is used in the specification. We use an array (with size equal to the pre-assigned depth) with each element storing the $i$-th change time (from `NOW=0`) and the value after the change. Quantifications on time variables are evaluated by looking at the contents of the array – noticing that each time variable is implicitly bounded by the pre-assigned depth. At each node, the model-checker checks the current configuration against the property. When an error is found, the model-checker provides a trace that leads to the error at the specification level. The details of this approach can be found in [DK97, D97].

In [DK99a], an improved version of the above explicit-state model-checker is presented. The new version generates customized C++ code for each specification. This code is actually a prototype implementation of the specified system and a control module to enumerate all the branches of execution of this implementation up to a system time bound set by the user. This code generator approach takes advantage of the fact that ASTRAL is a modularized specification language and that the mapping from an ASTRAL specification to C++ classes is quite natural. For example, a Small-ASTRAL process instance can be translated into a C++ class instance. Thus, the code generator makes it possible to model-check a realistic protocol within a reasonable amount of time. As a result, more realistic specifications can be validated.

Strictly speaking, the EBT-based model-checker is only a testing tool, which tests the specification for a set of given constant values. The constants must be carefully chosen to assure that something "interesting" actually occurs within the given system time bound. The model-checker is cheap and fast, and it gives some level of confidence when a specification survives the model-checking.

In contrast to the EBT, which traverses the states one by one, the SBT symbolically represents the current set of reachable states as a formula (more precisely, a Presburger formula). This chapter presents an SBT-based model-checker for Small-ASTRAL that tests the specification using the Omega library [P92]. The symbolic model-checker is implemented as a process level model-checker based upon the ASTRAL modularized proof theory. That is, the model-checker checks only a process instance's critical requirements.

This chapter is organized as follows. Section 3.1 presents how to translate a Small-ASTRAL specification into a transition system such that a formula in the specification is mapped to a Presburger formula. Section 3.2 shows a number of search algorithms to carry out the model-checking procedure. In Section 3.3 we introduce three approximation techniques that are used for large specification instances for which the model-checker fails to complete the search. As a case study, Section 3.4 presents and analyzes a number of experimental results on a benchmark example.

## 3.1 Building a Labeled Transition System

This section presents a procedure to translate a process instance's local requirements, assumptions and transition system into Presburger formulas, whenever possible. Presburger formulas [KK67] are arithmetic formulas over integer variables, which are built from logical connectives and quantifiers. The following grammar for generating Presburger formulas is adapted from [BGP97],

$$f ::= \; t \leq t \mid (f) \mid f \wedge f \mid \neg f \mid \exists \mathbf{intvar}(f),$$

$$t ::= \; (t) \mid t + t \mid t - t \mid \mathbf{intvar} \mid \mathbf{intcons},$$

where **intvar** and **intcons** are integer variables and integer constants, respectively. It is easy to see that one can use the grammar to represent a formula containing $<, \neq, \vee, \forall$ as well as multiplication by an integer constant. The complexity of solving Presburger formulas is extremely high ($O(2^{2^{2^n}})$) [O78]. The Omega library was developed by Pugh for manipulating integer tuple relations and sets that are characterized by Presburger formulas [P92]. Using the Omega library, the solution space of a Presburger formula can be compactly represented by an Omega set or relation that is a union of convex regions. Such a representation is defined as an *image* with its *size* given as the number of unions. Each convex region in an image is represented by a conjunction of linear constraints. In addition, the Omega library provides rich operations on Omega sets and relations, such as join, intersection and projection. For *practical* formulas, especially those with less alternations of quantifications, we found the solving time via the Omega library to be affordable in many cases. Bultan first used the Omega library in the area of model checking of an infinite state transition system, and he demonstrated that the Omega library is useful and efficient for handling a number of practical systems [BGP97, BGP99]. We used the Omega library for a more complex real-time specification language. This experience showed that the Omega library usually can comfortably handle up to 10 integer variables. However, for a formula with more than 20 variables and several quantifications, it is not unusual that it either takes hours to solve the formula or 256M of memory is quickly exhausted. The following subsections focus on the methods and techniques used in the implementation of the ASTRAL symbolic model-checker to encode a process instance using less variables.

### 3.1.1 Modeling

A process instance is modeled by a labeled transition system $\mathcal{T}$:

$$(Q, \mathbf{Init}, \rightarrow_{a\in\Sigma}, \mathbf{Assump}, \mathbf{Prop})$$

that consists of a set $Q$ of (infinitely many) states, a finite set of transitions $\rightarrow_a$ with name $a$ from $\Sigma$. Each $\rightarrow_a$ is a relation on $Q$, i.e.,

$$\rightarrow_a \subseteq Q \times Q.$$

$\mathbf{Init} \subseteq Q$ are the initial states. The assumption $\mathbf{Assump}$ and the property $\mathbf{Prop}$ of $\mathcal{T}$ are also subsets of states $Q$. As usual, for a set of states $R \subseteq Q$, we define the preimage,

$Pre_a(R)$, of a transition $\rightarrow_a$ as the set of all states from which a state in $R$ can be reached by this transition:

$$Pre_a(R) = \{q : \exists q' \in R \text{ s.t. } q \rightarrow_a q'\}.$$

Similarly, the postimage, $Post_a(R)$, of a transition $\rightarrow_a$ is the set of all states that are reachable from a state in $R$ by this transition:

$$Post_a(R) = \{q : \exists q' \in R \text{ s.t. } q' \rightarrow_a q\}.$$

The semantics of $\mathcal{T}$ is characterized by *runs*

$$q_0 a_1 q_1 a_2 \cdots$$

such that for all $i$,

$$q_i \rightarrow_{a_{i+1}} q_{i+1} \quad \text{and} \quad q_0 \in \textbf{Init}.$$

$\mathcal{T}$ is *correct* with respect to its specification, if for any run

$$q_0 a_1 q_1 a_2 \cdots$$

of $\mathcal{T}$, the following condition is satisfied for all $k$,

$$\{q_0, \cdots, q_k\} \subseteq \textbf{Assump} \quad \text{implies} \quad q_k \in \textbf{Prop}.$$

Since the Omega library is used to calculate the symbolic representation of a subset of the states, $\mathcal{T}$ is further restricted to have the form in which the components $Q$, **Init**, $\rightarrow_{a\in\Sigma}$, **Assump** and **Prop** can be expressed as Presburger formulas. The following subsections are devoted to the translation of a Small-ASTRAL process specification into a labeled transition system $\mathcal{T}$. The translation is carried out automatically in the ASTRAL symbolic model-checker.

### 3.1.2 Handling Constants and Local variables

A process instance may use a number of global and local constants, as well as local and imported (from other process instances) variables. In the current implementation, they are translated into integer variables. For example, in the benchmark specification, the local variable `position` of the process `Gate` is an enumerated type

`gate_position : (raised, raising, lowered, lowering)`.

This variable is represented by an integer variable

$$position$$

with the type assumption

$$position = 1 \vee position = 2 \vee position = 3 \vee position = 4$$

added to the **Assump** of $\mathcal{T}$.

Some constant values should be known in advance in order to carry out the symbolic model-checking procedure. There are two categories of these constants. One is the global constants that decide the number of process instances in the whole system. For example, the global constant `n_tracks` characterizing the number of `Sensor` process instances belongs to this category. Another category is the constants involved in multiplications. For example, for the term

$$\mathtt{min\_speed} * \mathtt{RIImin}$$

which appears in process `Sensor`'s local axiom clause, one of the two constants needs to be set. Otherwise, this local axiom is not in the proper form for a Presburger formula, since multiplications between two variables are not allowed. The model-checker automatically prompts the user for the constants that need to be set, together with a number of constants the user may want to set (for faster model-checking). It also recommends several constants for the user to set, like those scoped by a quantification, which is a very expensive operation in the Omega library.

### 3.1.3 Handling a Process Instance's Transition System

A process instance's transition system is characterized by a finite number of transitions. Each transition $T_i$ contains an entry assertion, an exit assertion and a duration. If both the entry assertion and the exit assertion can be translated into Presburger formulas, then $T_i$ can be easily translated into an Omega relation. For example, consider the transition

```
TRANSITION enter_R
  ENTRY           [ TIME : enter_dur ]
            ~train_in_R
  EXIT
            train_in_R = TRUE.
```

This transition can be translated as

$$\neg(ptrain\_in\_R = 1) \wedge train\_in\_R = 1 \wedge now = pnow + enter\_dur$$

where $ptrain\_in\_R$ and $pnow$ indicate the values before `enter_R` fires, while $train\_in\_R$ and $now$ indicate those after `enter_R` fires. (Note that the variable $train\_in\_R$ itself has a Boolean type assumption

$$train\_in\_R = 1 \vee train\_in\_R = 0$$

already added into the assumption **Assump** of $\mathcal{T}$.)

### 3.1.4 Handling Clocks

Clocks in a Small-ASTRAL process instance include

- the global clock `NOW`,
- `Start(T)`, `End(T)` and `Call(T)` of a transition instance `T` indicating the start, end, and call times. `T` can be either a local transition or an imported transition,
- `Change(X)` indicating the change time of a variable `X`. `X` can be either a local variable or an imported variable.

These are all translated into integer variables. But for imported transitions and exported transitions, we need more constraints to satisfy the ASTRAL semantics. For instance, if `T` is an exported transition, `T` must be called by the external environment in order to fire. Calls on `T` are totally controlled by the external environment, which is typically restricted by a process's local environment clause. Successive calls are not effective if the called transition has not fired in response to the first call. In order to characterize the behavior of an exported transition, it is necessary to introduce a new variable to indicate the last call time. For example, in process `Sensor`, the exported transition `enter_R`'s last call time would be indicated as $enter\_R\_lcl$. Constraints also need to be added to the translated Omega relation of `enter_R`:

$$penter\_R\_lcl > penter\_R\_lst.$$

The constraint says that before the transition fires, the last call time of `enter_R` must be greater than the last start time of `enter_R`. Furthermore, the variable $enter\_R\_lcl$ can be changed during a transition's execution and constraints about these changes are also added to a transition relation.

The interface sections of a process instance may also contain imported transitions (exported from other process instances) and their `Start`, `End` and `Call` times. When a process instance refers to one of these times, a new variable is introduced to indicate the last `Start` time of an imported transition. This variable's value can be changed during a transition's execution, and constraints are added to the transition relation. For example, one of the constraints is that, at the time that a transition `T` is completed, the new value of the last `Start` time of an imported transition can be either the same as the old value or should be greater than `T`'s start time and less than or equal to the current time.

### 3.1.5 Handling Histories of Variables

A process instance may refer to the history of an imported variable or a local variable through the past operator `past(X,t)`. The history-dependent features make model-checking the process instance more difficult. An approach to handle the history of an imported variable is to encode it as a series of variables to indicate all its times of change and the values at each change. The maximal number of such changes is bounded by a given *window-size*. For instance, under window-size 2,

$$\mathtt{past(s.train\_in\_R, t) = true}$$

can be translated into

$$(t \geq s\_train\_in\_R\_lc0 \rightarrow 1 = s\_train\_in\_R\_v\_0)$$

$$\wedge(t < s\_train\_in\_R\_lc0 \wedge t \geq s\_train\_in\_R\_lc1 \rightarrow 0 = s\_train\_in\_R\_v\_1)$$

$$\wedge(t < s\_train\_in\_R\_lc1 \rightarrow 1 = s\_train\_in\_R\_v\_2)$$

where variables $s\_train\_in\_R\_lc0$ and $s\_train\_in\_R\_lc1$ are the last two change times (with $s\_train\_in\_R\_lc0$ being the most recent), and the variables $s\_train\_in\_R\_v\_0$, $s\_train\_in\_R\_v\_1$, $s\_train\_in\_R\_v\_2$ are the values before and after each change. Since the type of `s.train_in_R` is Boolean, $s\_train\_in\_R\_v\_1$ and $s\_train\_in\_R\_v\_2$ are redundant. Therefore, the above translation can be further simplified by substituting $s\_train\_in\_R\_v\_1$ and $s\_train\_in\_R\_v\_2$ with $1 - s\_train\_in\_R\_v\_0$ and $s\_train\_in\_R\_v\_0$, respectively.

### 3.1.6 Handling Assumptions and Properties

If the goal is to check the local invariant, then ASTRAL proof theory requires that the only assumptions that can be used are the local and global axiom clauses and the local initial clause. Therefore, these clauses are added to **Assump**. The property **Prop** is the invariant clause. If the goal is to check the local schedule, then the local and global axiom clauses, the local initial clause, the local imported variable clause and the local environment clause can all be used as assumptions. Therefore, these clauses are added to **Assump**, and in this case the property **Prop** is the schedule clause. It is important to note, however, that the imported variable clause and the environment clause must hold during a transition's execution (not just at the times when it fires and ends). However, because for a transition with duration greater than 1, the process's environment could be changed during the duration, we must find a way to ensure that **Assump** holds consistently during a transition's execution. This will be discussed in the following subsections.

### 3.1.7 Strengthening a Transition Relation

A transition instance `T` in a Small-ASTRAL process instance is related to a transition $T$ in the labeled transition system $\mathcal{T}$. We have already seen that by simply combining the entry assertion and the exit assertion of `T`, a transition relation $T$ can be built. However, in order to obey the ASTRAL semantics, we need to further strengthen $T$ as below. What we missed from the above translation of `T` is

- how the environment changes during an execution of `T`,
- how to ensure that **Assump** holds consistently during the execution.

The easiest way is to constrain the environment from any changes during a transition's execution; i.e., the environment could change only at the end points of an execution. But our preliminary tests showed that although such a restriction is cheap, it is not effective in detecting bugs in a specification, since a lot of nontrivial errors can only be demonstrated when, for instance, an imported variable's value changes during the execution of a transition. Therefore, a more sophisticated class of assumptions are needed.

We propose two kinds of assumptions. One is called the *single-event* assumptions, the other is called the *multiple-event* assumptions. *Environment clocks* indicate the call time of a local exported transition and the start, end, or call times of an imported transition. By an event, we mean one of the followings behaviors:

- a change of the value of an imported variable,
- a change of an environment clock.

Remember that imported variables and environment clocks are all translated into integer variables. The single-event assumptions constrain the behaviors of the environment in such a way that during a transition's execution, each above mentioned integer variable changes at most once. In contrast, the multiple-event assumptions allow an unrestricted number of changes. Obviously, allowing multiple events during the execution makes the environment more accurate. However, this may cause the strengthened transition relation to become too complex to be calculated using the Omega Library.

Before we proceed to give the details of the single-event and multiple-event assumptions, some notation is needed. Let $T$ be a transition relation in the labeled transition system $\mathcal{T}$ translated from a Small-ASTRAL process instance $\mathcal{P}$. Recall that $T \subseteq Q \times Q$ consists of pairs of states in $Q$. $T$ itself is definable by a Presburger formula. Given a pair of states $\langle s, s' \rangle$ in $T$, we use $s(X)$ and $s'(X)$ to denote an imported variable $X$'s values before and after the execution of the transition. Similarly, $s(C)$ and $s'(C)$ denote the values of an environment clock $C$. Clearly, $s(X) \neq s'(X)$ (or $s(C) \neq s'(C)$) implies an event changing

$X$'s value (or $C$'s value) happens during the execution. $T$ itself has a duration $dur(T)$. Even though in the definition of $\mathcal{T}$, no time was mentioned, this fact is implicitly expressed in the transition relation $T$ as $s'(now) = s(now) + dur(T)$. But, since $T$ is a binary relation, a pair $\langle s, s' \rangle$ in $T$ only indicates that $T$ may transit from $s$ to $s'$ without mentioning what has happened during the time interval of the duration $dur(T)$. In order to strengthen $T$ such that **Assump** consistently holds in the interval, we must find an inexpensive way to recover the sequence of events that occurred. In the following subsections, we illustrate in detail the approach used.

### 3.1.8 The Single-event Assumptions

The single-event assumptions constrain the environment such that each imported variable and environment clock changes at most once during a transition. There are four levels, which are different in the accuracy of approximating the environment.

The first level bets that each "intermediate" state $\hat{s}$ satisfies **Assump** when $\hat{s}$ is obtained by setting the global clock $s'(now)$ in $s'$ to any value between $s(now)+1$ and $s'(now)$. Notice that $s(now) + dur(T) = s'(now)$. Thus, this requirement is independent of the choice of $s$. Formally, denote an image

$$A_1 = \{s' : \forall t \;\; 1 + s'(now) - dur(T) \leq t \leq s'(now), s'[now/t] \in \mathbf{Assump}\}$$

where $s'[now/t]$ denotes a state obtained by replacing $now$ with $t$. Thus, for each state $s'$ in $A_1$ **Assump** consistently holds on each intermediate state when the clock $now$ in $s'$ is rolled back from $s'(now)$ down to $1 + s'(now) - dur(T)$. It is easy to see that $A_1$ is definable by a Presburger formula. For this level, $T$ is strengthened as

$$T_{\text{level\_1}} = \{\langle s, s' \rangle \in T : s' \in A_1\}.$$

$T_{\text{level\_1}}$ is cheap to calculate, since we do not care when an event actually happened during the transition. However, it is not accurate. The reason is that the approach does not distinguish between the intermediate states before and after an event. Thus, although $T_{\text{level\_1}}$ is easy to calculate, this approach may cause false negatives. We will see how the model-checker handles them in the next section.

The second level allows at most one event to happen during the execution and checks **Assump** at the moment when that event occurs. More precisely, assume an environment clock $C_0$ changes during the execution; i.e., $s'(now) - s'(C_0) \leq dur(T) - 1$. Since this is the only event that occurs, for each environment clock $C \neq C_0$, $s'(now) - s'(C) \geq dur(T)$. $s'[now/s'(C_0)]$ represents the state that is rolled back from a state $s'$ to the time $s'(C_0)$ when the clock $C_0$ changes. The image $A_2^{C_0}$ is constructed such that $s'[now/s'(C_0)]$ satisfies **Assump**. That is,

$$A_2^{C_0} = \{s' : s'(now) - s'(C_0) \leq dur(T) - 1$$

$$\wedge \forall C \neq C_0 (s'(now) - s'(C) \geq dur(T))$$

$$\wedge s'[now/s'(C_0)] \in \textbf{Assump}\}.$$

The image $A_2$ is obtained by nondeterministically choosing which $C_0$ to change or having none of the environment clocks change. Thus, $A_2$ is defined by

$$A_2 = (\vee_{C_0} A_2^{C_0}) \vee \{s' : \forall C(s'(now) - s'(C) \geq dur(T))\}.$$

For this level, $T$ is strengthened as

$$T_{\text{level\_2}} = \{\langle s, s' \rangle \in T : s' \in A_2\}.$$

Again, $A_2$ is independent of the starting state $s$ of the transition. $T_{\text{level\_2}}$ is also cheap to calculate, but it is still not accurate. The reason is that $s'[now/s'(C_0)]$ is not exactly the state when $C_0$ changes since the values of the local variables and local clocks in $s'[now/s'(C_0)]$ are equal to those in $s'$, i.e., values after the transition $T$'s execution. In fact, according to the ASTRAL semantics, the values of local variables and clocks should have the same values as in the state $s$ when the transition started. Another source of inaccuracy is from the fact that in $A_2$ we do not check that **Assump** holds before and after the moment when $C_0$ changes. Thus, as was the case for the approach above, false negatives may occur. In order to faithfully rebuild the intermediate states, we have to take the starting state $s$ into account as follows.

The third level also allows at most one event during the execution, however, it checks **Assump** at the point when the event happens, as well as during the times before and after the event. Given a pair $\langle s, s' \rangle \in T$, the state $s[now/s'(C_0)]$ is exactly the state of $\mathcal{P}$ when $C_0$ changes, i.e., $s'(C_0) > s(now)$, during the transition. Since the change of $C_0$ is the only event, we have for each environment clock $C \neq C_0$, $s'(C) = s(C)$. Denote

$$A_3^{C_0} = \{\langle s, s' \rangle : s'(C_0) > s(now)$$

$$\wedge \forall C \neq C_0 (s'(C) = s(C))$$

$$\wedge s[now/s'(C_0)] \in \textbf{Assump}$$

$$\wedge \forall t \quad s(now) < t < s'(C_0)(s[now/t] \in \textbf{Assump})$$

$$\wedge \forall t \quad s'(C_0) < t < s'(now)(s[now/t, C_0/s'(C_0)] \in \textbf{Assump})\}.$$

Intuitively $A_3^{C_0}$ is the set of all the state pairs $\langle s, s' \rangle$ such that

- $C_0$ changes during the state transition from $s$ to $s'$,
- no other environment clock changes,
- at the change point $s'(now)$, **Assump** is satisfied by the intermediate state $s[now/s'(C_0)]$,
- before the change happens, i.e., for all $t$ with $s(now) < t < s'(C_0)$, **Assump** is satisfied,
- after the change happens, i.e., for all $t$ with $s'(C_0) < t < s'(now)$, **Assump** is satisfied. Notice that, in this case, the state at time $t$ is characterized by $s[now/t, C_0/s'(C_0)]$, obtained by substituting the global clock $now$ with $t$, and substituting the environment clock $C_0$ with $s'(C_0)$. It is incorrect if we use $s'[now/t]$ since the local variable values in $s'$ are not the same as those during the execution.

It is obvious that $A_3^{C_0}$ is definable by a Presburger formula. Also notice that $A_3^{C_0}$ does not depend on the choice of a transition $T$. There is the case when no $C_0$ changes, but **Assump** consistently holds during a transition's execution. This can be expressed by

$$A_3^{\emptyset} = \{\langle s, s' \rangle : \forall C (s(C) = s'(C))$$

$$\wedge \forall t \quad s(now) < t < s'(now)(s[now/t] \in \textbf{Assump})\}.$$

Allowing one to choose nondeterministically which clock $C_0$ to change and also including the case when no $C_0$ changes, we define

$$A_3 = (\vee_{C_0} A_3^{C_0}) \vee A_3^{\emptyset}.$$

Even though $A_3$ is independent of any choice of a transition, we can "tie" it with a specific transition $T$. That is, $T$ can be strengthened by

$$T_{\text{level\_3}} = T \wedge A_3.$$

The resulting transition relation $T_{\text{level\_3}}$ guarantees that **Assump** is satisfied during $T$'s execution.

The fourth level allows each environment clock to change at most once during the execution. To express the fact that **Assump** consistently holds during the execution, we enumerate all the subsets $\mathcal{C}$ of the environment clocks. For each subset $C$, we enumerate all the permutations $\tau_{\mathcal{C}}$. A permutation gives an ordering of the change times of clocks in $\mathcal{C}$ during the execution. Similar to the level 3 approach, the fact that **Assump** holds during an execution can be expressed as a set of state pairs $A_4^{\tau_{\mathcal{C}}}$ that is definable by a Presburger formula. We omit the technical details here. Finally, combining all the possible choices of the subsets $\mathcal{C}$ and the permutations, we define

$$A_4 = \vee_{\tau_{\mathcal{C}}} A_4^{\tau_{\mathcal{C}}}.$$

Again, $A_4$ is independent of any choice of a transition. For this level, $T$ can be strengthened by

$$T_{\text{level\_4}} = T \wedge A_4.$$

The resulting transition relation $T_{\text{level\_4}}$ ensures that **Assump** consistently holds during $T$'s execution, when each environment clock changes at most once.

### 3.1.9 The Multiple-event Assumptions

An environment clock could change multiple times during a transition's execution. But the fact that **Assump** holds during the execution in general is not expressible in a Presburger formula. That is, there is no way to strengthen $T$ as in the last subsection to ensure this fact. In order to allow multiple events, a transition relation $T$ is refined by splitting one big move of duration $dur(T)$ into $dur(T)$ moves of unit duration. To do this, a duration counter $i$ is used to indicate how many unit duration moves have been taken. There are three kinds of *micro-moves*. The first kind $T_1^\Delta$ is to start the transition $T$. Each tuple $\langle i, s, i', s' \rangle$ in $T_1^\Delta$ satisfies:

- $i = 0$ and $i' = 1$,
- $T_1^\Delta$ has unit duration. That is, $s'(now) = s(now) + 1$,
- $s$ satisfies the enabling condition of $T$,
- other components of $s'$ are exactly the same as $s$ except that each environment clock $C$ may change (i.e., $s'(C) = s'(now)$). Of course, when $C$ is a change clock of an imported variable, the value of variable changes imply the change of $C$.

The second kind $T_2^\Delta$ represents all the intermediate micro-moves before the duration is reached. Each tuple $\langle i, s, i', s' \rangle$ in $T_2^\Delta$ satisfies:

- $i' = i + 1 \wedge i' < dur(T)$,
- $T_2^\Delta$ has unit duration. That is, $s'(now) = s(now) + 1$,
- other components of $s'$ are exactly the same as $s$ except that each environment clock $C$ may change.

The third kind $T_3^\Delta$ leads to the end of the transition $T$. Each tuple $\langle i, s, i', s' \rangle$ in $T_3^\Delta$ satisfies:

- $i = dur(T) - 1 \wedge i' = 0$ (reset $i$ to zero),
- $T_3^\Delta$ has unit duration. That is, $s'(now) = s(now) + 1$,
- $\langle s, s' \rangle$ satisfies the exit assertion.
- each environment clock $C$ may change.

$T$ is simply replaced by a nondeterministic composition

$$T_1^\Delta \vee T_2^\Delta \vee T_3^\Delta.$$

From the above construction, after the transition $T_1^{\Delta}$ is taken, $T_2^{\Delta}$ is the only successive transition. $T_2^{\Delta}$ loops for $dur(T) - 1$ times and then triggers $T_3^{\Delta}$. Thus, the refined sequence of micro-moves of $T$ is unique. We do not claim that $T$ is equivalent to the composition; in fact, it is not. However, we shall see later that when model-checking $\mathcal{P}$ with the multiple-event assumptions, using this composition instead of $T$ is enough.

## 3.2 Symbolic Search Algorithms

The above section shows that a Small-ASTRAL process instance $\mathcal{P}$ can be translated into a labeled transition system $\mathcal{T}$. Using either the single-event assumptions or the multiple-event assumptions, a transition $T$ in $\mathcal{T}$ can be strengthened. The model-checker calculates the image of reachable states of $\mathcal{T}$, starting from the initial image **Init**. A naive way to do the model-checking is to calculate the Omega representation of a one_step transition defined by

$$T_{\text{one-step}} = \bigvee_i T_i$$

of the process instance by combining all the transition relations $T_i$. Next, calculate the iterations

$$\begin{aligned}
R_0 &= \mathbf{Init} \wedge \mathbf{Assump} \\
R_1 &= Post_{T_{\text{one-step}}}(R_0) \wedge \mathbf{Assump} \\
&\vdots \\
R_n &= Post_{T_{\text{one-step}}}(R_{n-1}) \wedge \mathbf{Assump} \\
&\vdots
\end{aligned}$$

and check for each $R_n$, whether

$$R_n \subseteq \mathbf{Prop}.$$

Our experiments showed, however, that this intuitive solution does not always work for a nontrivial ASTRAL specification, such as the railroad crossing benchmark. The reason is that $T_{\text{one-step}}$ is so large that its image can not be built using the Omega library, even when allowed to run for hours. A number of more sophisticated solutions are proposed in the following sections. The central idea is how does one make the image computations feasible when each $T_i$ has a large image.

### 3.2.1 Execution Graphs

In order to achieve better efficiency, the model-checker operates on the execution graph $G$ of a process instance (represented as a labeled transition system $\mathcal{T}$). The graph $G$ is a pair

$$\langle V_G, R_G \rangle$$

in which $V_G = \{initial, idle, T_1, \cdots, T_k\}$. *initial* indicates the initial transition which is defined as an identity transition on the initial states with zero duration. *idle* is a newly introduced transition, which has duration one. *idle* fires if every $T_i$ is not firable. *idle* will not change the values of local variables. $R_G \subseteq V_G \times V_G$ excludes all the pairs of transitions such that the second one is not immediately firable after the first one finishes. $G$ is automatically constructed by analyzing the initial conditions and checking if entry/exit pairs for different transitions are satisfiable using the Omega library[2]. Figure 3.1 is an example of the execution graph of the `Gate` process. A dashed arrow in the figure means zero or more *idle* transitions are executed to reach the next node.

Since, as will be discussed later, the use of the model-checker in the ASTRAL SDE is only for debugging purposes rather than full verification, there is no interest in calculating the least fixed point for the transition system. Therefore, a user needs to set the depth of the tree indicating the maximal number of iterations of transitions to check. Once this depth is set, the model-checker proceeds with the image computations on $G$ by using the following algorithms.

### 3.2.2 Depth-first Search under the Single-event Assumptions

In this subsection, as well as the following two subsections, we assume each transition in $\mathcal{P}$ is strengthened by using the single-event assumptions. The execution tree of $G$ is the the

[2]Note that the ASTRAL SDE has a tool called the sequence generator [K99a], which does this by using a theorem proving approach. The SDE sequence generator is also fully automated.

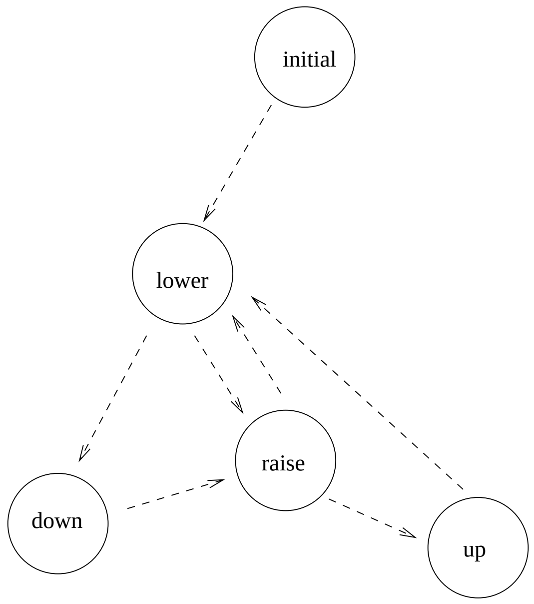


Figure 3.1: The execution graph of `Gate`

tree of all possible execution paths trimmed by the execution graph $G$. Figure 3.2 is a part of the execution tree of the process `Gate`. The depth-first search algorithm calculates the reachable image along each execution path in a depth-first order.

Figure 3.3 shows the recursive procedure. In the algorithm, $Post_A$, which was defined earlier, is the post image operator for the transition indicated by node $A$. Model-checking a node $A$ starts by calculating the preimage and postimage of it. If the postimage is not empty, which means that the transition indicated by $A$ is firable, then the preimage is checked with respect to the property, followed by checking every child node according to the execution graph. The model-checking procedure starts from the initial node *initial*,

$$\text{Check\_dfs}(initial, depth).$$

### 3.2.3 Breadth-first Search under Single-event Assumptions

Depth-first search may explore too many nodes before an error is detected or before all the nodes are traversed without finding any errors. Theoretically, the number of reachable

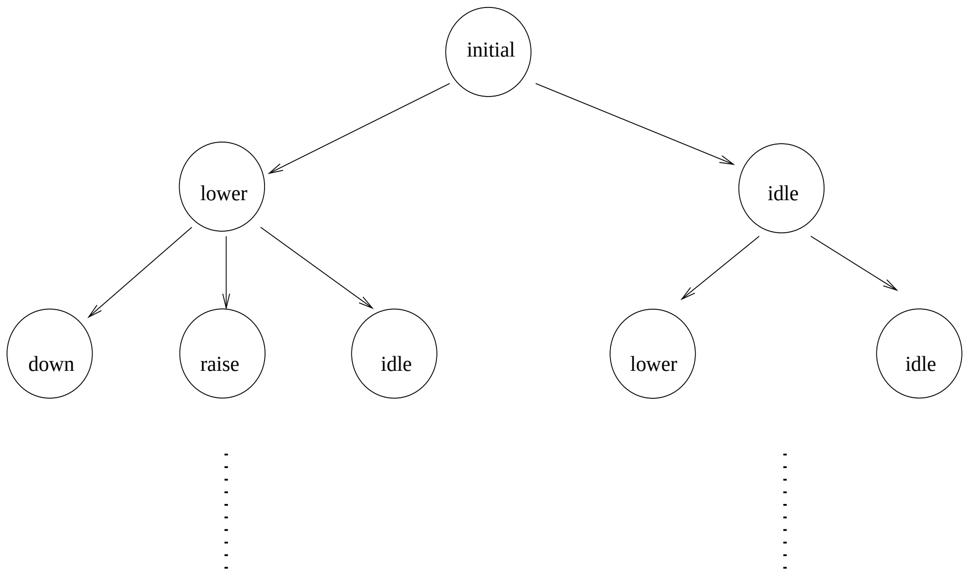


Figure 3.2: The execution tree of `Gate`

Boolean Check_dfs(Node $A$, int $depth$)

{

  if $A.layer = depth$ then return true;

  if $A.layer = 0$ then

    $A.postimage = \mathbf{Init} \wedge \mathbf{Assump}$;

  else

    $A.preimage = (A.parent).postimage$;

    $A.postimage = Post_A(A.preimage) \wedge \mathbf{Assump}$;

  if $A.postimage \neq \emptyset$ then

    if($A.preimage \not\subseteq \mathbf{Prop}$) then return false;

    else foreach $B$, $\langle A, B\rangle \in R_G$

      if($\neg$Check_dfs$(B, depth)$) return false;

  return true;

}

Figure 3.3: The depth-first search algorithm

nodes up to the pre-assigned depth is exponential in depth. Thus, it is natural to search in a breadth-first way. That is, all the nodes in the same layer are combined into one bigger "node". Thus, we only need to do at most *depth* number of iterations.

Figure 3.4 shows the algorithm. At each layer, the model-checker collects the union of all the postimages for each node $A$, i.e.,

$$postimage = \vee_A Post_A(preimage).$$

This postimage is used to check against the property, and propagate to the next layer to carry out the breadth-first search. The model-checker starts from Check_bfs($\mathbf{Init} \wedge \mathbf{Assump}, 0$).

Boolean Check_bfs(Image *preimage*,int *layer*)
{
    if $preimage = \emptyset$ then return true;
    if $layer = depth$ then return true;
    $postimage = \vee_A Post_A(preimage) \wedge \mathbf{Assump}$;
    if $postimage \not\subseteq \mathbf{Prop}$ then return false;
    if ¬Check_bfs($postimage, layer + 1$) then return false;
    return true;
}

Figure 3.4: The breadth-first search algorithm

### 3.2.4 Depth-Breadth Search under Single-event Assumptions

The breadth-first search algorithm works well for simple specifications. The reason is that the reachable image for each layer is not large and the image computations are affordable. In contrast, for complex specifications, this algorithm does not work since the size of the image for each layer grows dramatically fast. In that case, the depth-first search would be a better choice, since along an execution path, the reachable image usually is not large. But the load (i.e., the size of the reachable image) is not evenly distributed among paths. Obviously, it is wasteful to adopt depth-first search along a number of paths with extremely small load.

Thus, we need a dynamic approach to direct the model-checker to carry out either depth-first search or breadth-first search. The algorithm is called depth-breadth search as shown in Figure 3.5.

In the algorithm, the procedure Check_dbfs dynamically chooses a search approach. When the size of the reachable image $preimage.Size()$ is greater than a pre-assigned number IMAGE_SIZE, it calls the depth-first search procedure Check_dfs. Otherwise, it calls the breadth-first procedure Check_bfs. Both the procedures are adapted from the previous algorithms. It should be noted that a node $A$ could be $NULL$, which means it is a "node" after a round of breadth-first search. Thus, in Figure 3.5, when $A = NULL$, $\langle A, B \rangle \in R_G$ is always true, since after a breadth-first search, every branch in the graph could be taken. The depth-breadth search starts from the initial node and layer 0 with the initial image, i.e., Check_dbfs($\mathbf{Init}, 0, initial$).

### 3.2.5 Notes on the Algorithms

Recall that under single-event assumptions, a transition can be strengthened at four levels. For a given transition $T$, the strengthened transition at level 1 and level 2 can be expressed as

$$T_{\text{level\_1}} = \{\langle s, s' \rangle \in T : s' \in A_1\}$$

and

$$T_{\text{level\_2}} = \{\langle s, s' \rangle \in T : s' \in A_2\},$$

respectively. Each node, excluding the *idle* and *initial* nodes, corresponds to a strengthened transition. In the above algorithms, the image calculations

$$Post_{T_{\text{level\_1}}}(preimage)$$

and

$$Post_{T_{\text{level\_2}}}(preimage)$$

for a node representing $T_{\text{level\_1}}$ or $T_{\text{level\_2}}$ are directly applied on *preimage* and the strengthened transition. In fact, a more efficient way, which is used in the actual implementation,

```
Boolean Check_dbfs(Image preimage,int layer, Node A)
{
    preimage = preimage ∧ Assump;
    if preimage ⊈ Prop then return false;
    if layer = depth then return true;
    if preimage.Size() > IMAGE_SIZE then
        return Check_dfs(preimage,layer,A);
    else
        return Check_bfs(preimage,layer,A);
}
Boolean Check_bfs(Image preimage,int layer, Node A)
{
    Image postimage = ∨_B Post_B(preimage);
    if ¬Check_dbfs(postimage, layer + 1, NULL) then return false;
    return true;
}
Boolean Check_dfs(Image preimage,int layer, Node A)
{
    foreach B, ⟨A, B⟩ ∈ R_G
            Image postimage = Post_B(preimage);
            if ¬Check_dbfs(postimage, layer + 1, B) then return false;
    return true;
}
```

Figure 3.5: The depth-breadth search algorithm

is to calculate the intersections

$$Post_T(preimage) \wedge A_1$$

and

$$Post_T(preimage) \wedge A_2,$$

respectively. The reason is that constructing the transition relation $T_{\text{level\_1}}$ and $T_{\text{level\_2}}$ by restricting the range of $T$ to $A_1$ and $A_2$ respectively is very expensive, when $A_1$ and $A_2$ are large.

For the strengthened transition at level 3 and level 4, we have

$$T_{\text{level\_3}} = T \wedge A_3$$

and

$$T_{\text{level\_4}} = T \wedge A_4.$$

Instead of calculating the intersection of $T$ and $A_3$ (or $A_4$), it would be more efficient if the postimage calculations

$$Post_{T_{\text{level\_3}}}(preimage)$$

and

$$Post_{T_{\text{level\_4}}}(preimage)$$

can be split into

$$Post_T(preimage) \wedge Post_{A_3}(preimage)$$

and

$$Post_T(preimage) \wedge Post_{A_4}(preimage),$$

respectively. However, this is generally not true. Thus, if a user decides to use the approach with level 3 or level 4, the model-checker has to calculate the strengthened transition relation directly by intersecting $T$ with $A_3$ (or $A_4$). This calculation is expensive, as we show in section 3.4, which discusses the experiments.

### 3.2.6 Breadth-first Search under Multiple-event Assumptions

Recall that transitions are split into a number of micro-moves under multiple-event assumptions. The model-checking procedure proceeds by iterating a big micro-move by combining all the micro-moves. Thus, we only consider the breadth-first search strategy for multiple-event assumptions as follows.

Under multiple-event assumptions, a transition $T$, excluding the *idle* and *initial* transitions, is refined into a sequence of micro-moves $T_1^\Delta, T_2^\Delta$ and $T_3^\Delta$ as shown above. The labeled transition system $\mathcal{T}$ can be treated as a system with only one "big" transition

$$\bigvee_T T.$$

Iterations of the big transition demonstrate all the behaviors of $\mathcal{T}$. Recall that the construction of the micro-moves $T_1^\Delta, T_2^\Delta$ and $T_3^\Delta$ of $T$ ensures that the sequence of micro-moves uniquely corresponds to an execution of $T$. Thus, the big transition can be replaced by a big micro-move (i.e., a move with at most unit duration) as follows

$$\bigvee_{T \neq idle, initial} T_1^\Delta \vee T_2^\Delta \vee T_3^\Delta \bigvee initial \vee idle.$$

Thus, iterations of the big micro-move demonstrate refined traces of $\mathcal{T}$ with states at each time point presented. Thus, $\mathcal{T}$ is refined as a new labeled transition system with transitions

$$\cup_{T \neq idle, initial} \{T_1^\Delta, T_2^\Delta, T_3^\Delta\} \cup \{initial, idle\}.$$

The breadth-first search algorithm is exactly the same as the one under single-event assumptions, but for the new labeled transition system. [3]

## 3.3 Approximation Techniques

The symbolic model-checker presented in the last section is not feasible for large specifications due to high complexity. From our experiments, which we will discuss further later in

[3] We could consider depth-first search and depth-breadth search for the new transition system. But given the fact that the number of micro-moves (i.e., refined transitions) is large (triple the number of transitions in the original process instance), the execution tree of the new transition system is too expensive to explore. Therefore, these two search strategies have not been implemented.

this chapter, the high complexity of a single process instance can come from two sources: the local and global constants used in the instance and the local and imported variables that constitute the variable portion of the process instance. For example, in the `Gate` process there are 10 global constants and 6 local constants. These constants are used to parameterize the specified system, e.g., to specify a system containing a parameterized number of process instances as well as a system containing parameterized timing requirements. The local variables contribute to the local state and are changed by executing transitions. Though each process is modularized, each process instance does not stand alone. A process instance may also interact with other process instances through imported variables that are exported from the other process instances. Since a process's local properties are proved using only its local assumptions, the process instance must specify assumptions (environment clause and imported variable clause) that are strong enough to correctly characterize the environment and the behaviors of the imported variables. In order to guarantee the local properties, it is not unusual for an assumption to include complex timing requirements on the `call` patterns and the imported variables' change patterns. Thus, the second source of complexity primarily comes from history-dependency.

When it is not practical for the symbolic model-checker to complete the search procedure for a complex process instance, it is desirable to define approximation approaches to speed up the procedure by sacrificing coverage. Based upon the above analysis, four approaches can be used. The first approach is to assign concrete values to some of the constants before using the model checker. In [DK99b], it was shown that doing this will speed up the model checker and that it is still effective in finding bugs in some cases. There are, however, reasons for not using this approach. Most importantly, picking the right set of constant values to cause "interesting" things (especially potential errors) to happen is not trivial. Some choices will miss scenarios in which the specification would fail. In addition, even with a number of the constant values fixed, the model checking process is still expensive in some cases due to the complexity of the behavior of the local and imported variables. Experience shows that this approach, as well as using the explicit state model checker [DK99a], should be used in the earlier stages of debugging a specification, when errors are relatively easy to catch. The three remaining approaches, which are discussed in more detail in the following subsections, speed up the model checker by enforcing it to check either fewer nodes or smaller nodes. These approaches free the user from setting up constant values. Three different techniques are proposed. A random walk technique is used to allow the model-checker to randomly skip a number of branches when traversing the execution tree. A partial image technique considers only a subset of the image and uses this subset to calculate the postimage at each node. The dynamic environment generation technique generates different sequences of imported variable values for different execution paths. These three techniques are discussed in more detail in the following subsections.

### 3.3.1 Random walk

A path in the execution tree of an ASTRAL process is a sequence of transitions. Each node in the tree contains the image of all reachable states from the initial node along the path. Theoretically, the number of paths is exponential to the user-assigned search depth. Even though the symbolic model-checker itself adopts a number of trimming techniques, the time for a complete search for a large specification is unaffordable. It is our experience that, when a specification has a bug, this bug can usually be demonstrated by many different paths. The reasons are (1) The ordering of some transitions can be switched without affecting the result (though practically it is hard to detect this, due to history-dependency.[4]), (2) Most specifications contain a number of parameterized constants. When a specification has a bug, usually there are numerous scenarios and choices of parameterized constant values that reveal it, so these scenarios can be shown by many different paths.

Random walk is an approximation technique of searching only a portion of the reachable nodes on the execution tree. Though the technique is applicable for all four algorithms proposed in the previous section, without loss of generality, we illustrate the technique for the depth-breadth search algorithm in Figure 3.5. Figure 3.6 shows the recursive procedure which is similar to Figure 3.5 except that this algorithm includes a random choice (either skip or not) *toss* when the model-checker moves from one node to its children. The statement

$$postimage = \vee_{B,toss(A,layer)=tail} Post_B(preimage)$$

[4]This is significantly different from some standard techniques used in finite state model-checking, such as the partial order method [GW94]

in the procedure Check_bfs in Figure 3.6 only selects those nodes $B$ with the result of random tests $toss(A, layer)$ being $tail$. For the depth-first search procedure Check_dfs in the same figure, the statement

"foreach $B$, $\langle A, B\rangle \in R_G$ and $toss(A, layer) = tail$"

makes the model-checker skip the branches to $B$ when the random test fails. The random Boolean function $toss(A, layer)$ is not symmetric. The probability of result $tail$ is chosen as

$$(1 - \frac{layer}{depth}) + \frac{layer}{depth} \cdot \frac{1}{numChildren(A)}$$

where $numChildren(A)$ indicates the number of successors of node $A$ in the execution graph $G$, i.e., $numChildren(A) = |\{D : \langle A, D\rangle \in R_G\}|$,[5] and $layer$ indicates the layer that the model-checker is currently checking. The reason for this choice is to ensure the following:

- A short violation has less chance of being missed. When $layer$ is small, the probability of result $tail$ is large. When $layer$ is large, if $numChildren(A)$ is greater than 1, then the probability is small. Hence a longer path has a higher probability of being skipped.
- When $numChildren(A)$ is 1 [6], the probability of result $tail$ is 1. That is, a node with only one successor can not be skipped.

### 3.3.2 Partial image

In the Omega library, each image is represented by a union of convex linear constraints. The efficiency of an image calculation depends upon the number of variables and the number of constraints. Experience shows that, when a specification has a bug, there are usually numerous sets of parameterized constant and variable values that lead to the bug. These values usually satisfy many constraints in an image. Thus, considering only a part of the image will usually still let the model-checker find the bug. As reported in [DK99b], fixing a number of parameterized constant values increases the speed of the model-checker, since the number of variables in the image is decreased. This is a special case of the partial image technique. However, finding the right set of constant values leading to a potential bug is not

[5] Recall that when $A = NULL$, i.e., a breadth-first search is chosen, $\langle A, D\rangle \in R_G$ is always true.

[6] $numChildren(A)$ is always at least one, since each node has a successor through the $idle$ transition.

Boolean Check_dbfs(Image $preimage$,int $layer$, Node $A$)

{

  $preimage = preimage \wedge \textbf{Assump}$;

  if $preimage \not\subseteq \textbf{Prop}$ then return false;

  if $layer = depth$ then return true;

  if $preimage.Size() >$ IMAGE_SIZE then

    return Check_dfs($preimage$,$layer$,$A$);

  else

    return Check_bfs($preimage$,$layer$,$A$);

}

Boolean Check_bfs(Image $preimage$,int $layer$, Node $A$)

{

  Image $postimage = \vee_{B,toss(A,layer)=tail} Post_B(preimage)$;

  if $\neg$Check_dbfs($postimage, layer + 1, NULL$) then return false;

  return true;

}

Boolean Check_dfs(Image $preimage$,int $layer$, Node $A$)

{

  foreach $B$, $\langle A, B\rangle \in R_G$ and $toss(A, layer) = tail$

    Image $postimage = Post_B(preimage)$;

    if $\neg$Check_dbfs($postimage, layer + 1, B$) then return false;

  return true;

}

Figure 3.6: The depth-breadth search algorithm with random walk

easy for complex specifications; it usually requires a user that thoroughly understands the specification. The partial image technique presented here is used without fixing any constant values, by applying the $PartialImage()$ operator, which returns only half (randomly chosen) of the unions for the image, when the size of the image exceeds the pre-assigned "tolerance" IMAGE_SIZE. Again, the technique is applicable for all four algorithms proposed in the previous section. Without loss of generality, we illustrate the technique for the depth-breadth search algorithm in Figure 3.5. The algorithm, as shown in Figure 3.7, is essentially the same as Figure 3.5 except that the $PartialImage()$ operator is applied on each preimage when the procedure Check_dbfs is invoked. This reduced preimage is then used to calculate the postimage, as stated in the algorithm.

### 3.3.3 Dynamic environment generation

This approach reduces the number of variables involved in the image computations. Imported variables characterize the interface between the instance $\mathcal{P}$ and other process instances in the system. The environment of $\mathcal{P}$ is composed of the following elements,

- `Calls` of the transitions exported by $\mathcal{P}$,
- `Starts`, `Ends` and `Calls` of the imported transitions,
- the imported variables and their histories.

Except for the imported variables, each element needs only one variable to represent its value. Thus, the imported variables are essentially the bottleneck of the symbolic model-checker. We focus on how to approximate the imported variable part of the environment, and for the remainder of this section *environment* refers to the imported variable part. For other parts of the environment the technique can be used similarly. The approach used is to effectively generate a *reasonable* example of the environment for each execution path.

Before presenting the algorithm it is necessary to define some notation. Let *preimage* be an image representing the current reachable states. If all the imported variables and their histories have concrete values in *preimage*, *preimage* is called *having a concrete environment*. By using the `example` operator in the Omega library, a sample, in which

```
Boolean Check_dbfs(Image preimage,int layer, Node A)
{
    preimage = PartialImage(preimage);
    preimage = preimage ∧ Assump;
    if preimage ⊈ Prop then return false;
    if layer = depth then return true;
    if preimage.Size() > IMAGE_SIZE then
        return Check_dfs(preimage,layer,A);
    else
        return Check_bfs(preimage,layer,A);
}
Boolean Check_bfs(Image preimage,int layer, Node A)
{
    Image postimage = ∨_B Post_B(preimage);
    if ¬Check_dbfs(postimage, layer + 1, NULL) then return false;
    return true;
}
Boolean Check_dfs(Image preimage,int layer, Node A)
{
    foreach B, ⟨A, B⟩ ∈ R_G
            Image postimage = Post_B(preimage);
            if ¬Check_dbfs(postimage, layer + 1, B) then return false;
    return true;
}
```

Figure 3.7: The depth-breadth search algorithm with partial image

each variable has a concrete value, can be picked from $preimage$. The image $preimage$ can be made to have a concrete environment by replacing all the imported variables and their histories with the concrete values in the sample. The resulting image is denoted by $ConcreteEnv(preimage)$. In the same manner as for the partial image technique, the operator will not apply if the size of the preimage is less than the pre-assigned IMAGE_SIZE. Again, the technique is applicable to all four algorithms proposed in the previous section. Without loss of generality, we illustrate the technique for the depth-breadth search algorithm in Figure 3.5. The algorithm, as shown in Figure 3.8, is essentially the same as Figure 3.5 except that the $ConcreteEnv()$ operator is applied on each preimage when the procedure Check_dbfs is invoked. This reduced preimage is then used to calculate the postimage, as stated in the algorithm.

It should be noted that in the algorithm, whether or not to apply the $ConcreteEnv()$ operator depends upon the random test $toss(A, layer)$, which is defined in a previous section. As discussed for random walk, the random tests will ensure a reasonably large coverage of the execution paths, which is needed to detect a nontrivial bug in a specification.

## 3.4 Experimental Results

Experience shows that real-time specifications are hard to write and to read, especially when they involve complex timing constraints. A user can mutate a part of the specification where he or she believes that such a change should affect the behavior of the system. A similar technique, called mutation analysis [DLS78, H82], has been used in program testing for many years. The technique, by making a minor change to a program, can be used to show effectiveness of test data. But here, mutation testing is applied to specifications instead of programs. Mutation tests on a specification can help a user understand the specification, and they can test the strength of the specification. If the mutant is killed (i.e., a violation is found), then a specification level violation trace is demonstrated. Reading through the trace helps the user to quickly figure out where and how the syntax change affects the specification. If a mutation is created by weakening an assumption in the specification and the model-checker fails to find any violations, then a potential weakness is demonstrated in

```
Boolean Check_dbfs(Image preimage,int layer, Node A)
{
    if toss(A, layer) = tail then preimage = ConcreteEnv(preimage);
    preimage = preimage ∧ Assump;
    if preimage ⊈ Prop then return false;
    if layer = depth then return true;
    if preimage.Size() > IMAGE_SIZE then
        return Check_dfs(preimage,layer,A);
    else
        return Check_bfs(preimage,layer,A);
}
Boolean Check_bfs(Image preimage,int layer, Node A)
{
    Image postimage = ∨_B Post_B(preimage);
    if ¬Check_dbfs(postimage, layer + 1, NULL) then return false;
    return true;
}
Boolean Check_dfs(Image preimage,int layer, Node A)
{
    foreach B, ⟨A, B⟩ ∈ R_G
        Image postimage = Post_B(preimage);
        if ¬Check_dbfs(postimage, layer + 1, B) then return false;
    return true;
}
```

Figure 3.8: The depth-breadth search algorithm with dynamic environment generation

the original specification. There are two possibilities in this case. One is that the model-checker is not able to find the bug under this specific run with the specific setup. The other is that the mutation is equivalent to the original (correct) specification.

Since the use of the symbolic model-checker in the ASTRAL SDE is only for debugging purposes, its effectiveness for detecting a potential error in a specification is the major concern. The model-checker has been run on ten mutations of the `Gate` process from the railroad crossing specification. The reason that the `Gate` process specification was used is that it contains imported variables as well as their histories. These imported variables result in a large instance of the `Gate` process for which the symbolic model-checker failed to complete when not using approximation techniques. Each mutation contains a minor change to the original specification[7]. A detailed list of all the mutations can be found in Table 3.1.

| | |
|---|---|
| M1 | delete `raise_time` from the 1st conjunction of the axiom of `GATE` |
| M2 | replace `lower_dur` with `raise_dur` in the 3rd conjunction of the axiom of `GATE` |
| M3 | delete the 3rd conjunction from the axiom of `GATE` |
| M4 | delete `up_dur` from the 3rd conjunction of the axiom of `GATE` |
| M5 | delete `response_time` from the 3rd conjunction of the axiom of `GATE` |
| M6 | replace `response_time` with `lower_time` in the 1st conjunction of the schedule of `GATE` |
| M7 | delete `raise_dur` from the 2nd conjunction of the axiom of `GATE` |
| M8 | delete `now-End(lower)>=lower_time` from the entry assertion of transition `down` |
| M9 | delete `now-End(raise)>=raise_time` from the entry assertion of transition `up` |
| M10 | delete `position=raising` from the entry assertion of transition `raise` |

Table 3.1: Ten mutations of the railroad crossing specification

[7]The unmutated railroad crossing specification has been proved to be correct using the ASTRAL theorem prover, which is also part of the ASTRAL SDE [KDK99].

For all of the tests, the constants `min_speed` and `max_speed` were set to 15 and 20, respectively, the constant `n_tracks` was set to 2, and the window size was chosen as 3. There were no other user-assigned constants. This setting demonstrates the effectiveness of the model-checker on a large instance. All tests were performed on a Sun workstation with 4 CPUs and with 256M main memory and 512M swap memory. CPU time is measured in seconds and excludes the preprocessing time (normally 200 to 600 seconds, independent of whether an approximation technique is applied or not.). Our patience is set to 2000 CPU seconds. If the model-checker fails to complete the search within this limit, it will abort.

The design of the test cases tries to answer the following questions:

- Without using the approximation techniques, how well does each of the search algorithms perform in debugging the same specification. The algorithms include the three algorithms (the depth-first search, breadth-first search and the depth-breadth search) under the single-event assumptions with four different levels, and the breadth-first search algorithm under multiple-event assumptions.

- Using the approximation techniques, how much does the performance of each of the search algorithms improve in debugging a specification. The approximations are the three approximation techniques used separately and in combination.

Obviously, given the number of possible combinations of the choice of algorithms as well as the 10 mutations, it is not practical to run through all the tests. We decided to run the following two sets of tests. Firstly, we run M1 using each of the search algorithms without using the approximation techniques. This set of tests indicate how well the algorithms perform in debugging the same specification. Secondly, we run all ten mutations for the depth-first search algorithm by using the approximation techniques and their combinations. This set of tests indicate the effectiveness of the model-checker in debugging a specification by using the approximation techniques and their combinations. The details of the two sets of tests are as follows.

- Without using the approximation techniques, run M1 using each of the search algorithms:
  - df_single_1 (depth-first search under the single-event assumptions with level 1),

- bf_single_1 (breadth-first search under the single-event assumptions with level 1),
- db_single_1 (depth-breadth search under the single-event assumptions with level 1),
- bf_multiple (breadth-first search under the multiple-event assumptions).

Due to the randomness in a depth-first search where the model-checker chooses the order of branches to take, the first two algorithms df_single_1 and bf_single_1 are run three times. After this round of tests, we should have a good idea whether choosing a different search approach will greatly affect the performance for the model-checker in debugging the specification. Notice that under the single-event assumptions, level 2 is very similar to level 1 so it is not selected. Also levels 3 and 4 are not selected, since, as we mentioned before, the model-checker can not complete the preprocessing in the time-limit, due to the fact that the strengthened transition relations are extremely expensive to compute. When using the approximation techniques, we would also like to know whether they work for different algorithms. We chose to use the partial image technique on M1 for all of the above algorithms.

- Finally, and most importantly, we would like to know the effectiveness of the model-checker in detecting a bug in a specification with and without using the approximation techniques. We thoroughly ran through all ten mutations for the depth-first search algorithms under the single-event assumptions, i.e., df_single_1 and df_single_2 (of level 2), with all combinations of the approximation techniques:
  - `plain` (without using the approximation techniques),
  - `r.w.` (random walk)
  - `p.i.` (partial image)
  - `d.e.` (dynamic environment generation)
  - `r.w. + p.i.`
  - `r.w. + d.e.`
  - `p.i. + d.e.`

− r.w. + p.i. + d.e.

Again, due to the randomness in a depth-first search where the model-checker chooses the order of branches to take, each test is run three times.

### 3.4.1 Choosing among Various Algorithms

The tests in this subsection will clarify whether a choice of the search algorithms, i.e., df_single_1, bf_single_1, db_single_1 and bf_multiple will greatly influence the performance of the model-checker in detecting an error. We decided to choose M1 to run through all the algorithms three times and the results are shown in Table 3.2. Each test result is written as a pair. For instance, (429, $\times$) means the model-checker found a violation in 429 seconds. The status values are "$\times$" (the model-checker is able to detect a violation), "$\surd$" (the model-checker completes the search within the time-limit and reports no error), and "$\uparrow$" (the model-checker fails to finish within the time-limit). The time-limit, as mentioned before, is set to be 2000 CPU seconds. The search depth is 10 for df_single_1, bf_single_1 and db_single_1. Since bf_multiple uses micro-moves of unit duration, we set the depth to be 30 for it. IMAGE_SIZE is set to 10, which is used by the model-checker to decide, in db_single_1, when to shift from depth-first to breadth-first mode.

| algorithms | run 1 | run 2 | run 3 |
|---|---|---|---|
| df_single_1 | (429, $\times$) | (2000, $\uparrow$) | (1700, $\times$) |
| db_single_1 | (2000, $\uparrow$) | (1051, $\times$) | (283, $\times$) |
| bf_single_1 | (2000, $\uparrow$) | (2000, $\uparrow$) | (2000, $\uparrow$) |
| bf_multiple | (2000, $\uparrow$) | (2000, $\uparrow$) | (2000, $\uparrow$) |

Table 3.2: Different algorithms run on M1.

From the table, the depth-breadth search algorithm (db_single_1) performs marginally better than the depth-first search algorithm (df_single_1), although they both have one case out of 3 exceeding the time limit. Breadth-first algorithms, which carry a large reachable image for each layer and propagate this image into the next layer, fail to go deep enough to uncover a bug before the time-limit is reached. This is reflected in that both bf_single_1 and bf_multiple fail to kill M1 within the time-limit.

As we expected, breadth-first approaches are not good for debugging a large specification instance. In fact, the fast-growing sizes of the reachable images at each layer usually make it impossible for the model-checker to go deeper than a few layers (for bf_single_1, the model-checker only completed 5 layers in the time-limit.) Therefore, it is unlikely that a nontrivial bug will be uncovered.

Now we will look at whether an approximation technique, like the partial image technique, would make the model-checker faster and more effective in finding a bug for these algorithms. When an image is of size more than IMAGE_SIZE the partial image technique will randomly cut the image in half. For breadth-first algorithms, we should choose a larger IMAGE_SIZE than a depth-first or depth-breadth search algorithm, since otherwise for each layer too much information is lost after the cut. Therefore, we run the above tests with the partial image technique turned on and using 10 and 20 for IMAGE_SIZE when bf_single_1 and bf_multiple are applied. Other settings are the same as in Table 3.2.

The test results are presented in Table 3.3, where, for instance, "bf_multiple (20)" means the algorithm bf_multiple is used with IMAGE_SIZE=20. Using the approximation technique, both df_single_1 and db_single_1 are much faster in finding a bug. Under IMAGE_SIZE=10, both bf_single_1 and bf_multiple are able to complete the search but they are unable to kill M1. When IMAGE_SIZE is increased to 20, bf_single_1 kills M1 in each of the three runs, but bf_multiple is still unable to complete the search.

| algorithms | run 1 | run 2 | run 3 |
|---|---|---|---|
| df_single_1 | (49, $\times$) | (70, $\times$) | (143, $\times$) |
| db_single_1 | (254, $\times$) | (321, $\times$) | (225, $\times$) |
| bf_single_1 (10) | (393, $\surd$) | (395, $\surd$) | (394, $\surd$) |
| bf_single_1 (20) | (567, $\times$) | (567, $\times$) | (569, $\times$) |
| bf_multiple (10) | (1099, $\surd$) | (1098, $\surd$) | (1099, $\surd$) |
| bf_multiple (20) | (2000, $\uparrow$) | (2000, $\uparrow$) | (2000, $\uparrow$) |

Table 3.3: Different algorithms run on M1 using the partial image technique.

Our experience shows that the approximation techniques usually do not work well for breadth-first search algorithms, such as bf_single_1 and bf_multiple. The reason is that given a small IMAGE_SIZE, cutting the image in half loses too much information as compared to using the whole image for breadth-first search. However, for a large IMAGE_SIZE, the search procedure does not speed up much, since the image calculations are still expensive. However, for depth-first as well as depth-breadth search algorithms, the image size along a path is much smaller than the one for breadth-first search. Thus, even a small IMAGE_SIZE will make sure that a reasonably rich information source is preserved, even after an image cut.

### 3.4.2 Experiments with the Approximation Techniques

In this group of tests, we thoroughly run the two depth-first algorithms dfs_single_1 and dfs_single_2 with and without using the approximation techniques. We will see whether using the approximation techniques speeds up the search procedures while remaining effective in detecting violations in a mutation. Each of the two algorithms runs three times on a test case under each combination of the approximation techniques, including without using approximations. The search depth, i.e., the bound of the number of transitions in a path, is set to be ten. When the partial image technique is applied, the tolerance of image size, IMAGE_SIZE, is set to be ten. That is, as indicated in the algorithms before, if the current reachable image is larger than ten and the partial image technique is being used, the current image is (randomly) cut by a half. Each test result is written as a triple. For instance, (23, 1234, $\times$) means the model-checker found a violation after visiting 23 nodes in the execution tree taking 1234 seconds. The status values are "$\times$", "$\surd$" and "$\uparrow$" [8]. The time-limit, as mentioned before, is set to be 2000 CPU seconds.

Tables 3.4, 3.5, 3.6 and 3.7 show the test results. Among the ten mutations, M8 and M9 are both correct, though they are not strictly equivalent to the original specification. For instance, M8 deletes the conjunction

$$\texttt{now} - \texttt{End(lower)} >= \texttt{lower\_time}$$

from the entry assertion of the transition `down`. Doing this allows `down` to fire immediately

[8] Occasionally, the Omega library core-dumped on an image calculation for extremely large images. If this happens, $\uparrow$ status is possible even when time-limit is not reached.

| cases | plain | | | r.w. | | |
|---|---|---|---|---|---|---|
| | run 1 | run 2 | run 3 | run 1 | run 2 | run 3 |
| M1 | (28,429,×) | (48,2000,↑) | (58,1700,×) | (98,389,×) | (40,251,×) | (148,725,×) |
| M2 | (54,2000,↑) | (18,962,↑) | (61,2000,↑) | (95, 757, √) | (135, 816, √) | (148, 1875, √) |
| M3 | (78, 1820, ×) | (115, 2000, ↑) | (45, 2000, ↑) | (75, 2000, ↑) | (28, 457, ×) | (160, 728, √) |
| M4 | (95, 2000, ↑) | (74, 1784, ×) | (50, 2000, ↑) | ( 25, 466, ×) | (95, 1413, √) | (80, 403, √) |
| M5 | (100, 2000, ↑) | (90, 2000, ↑) | (65, 2000, ↑) | ( 130, 1327, √) | (45, 2000, ↑) | (83, 1080, ×) |
| M6 | (27, 502, ×) | (60, 2000, ↑) | (10, 71, ×) | ( 44, 175, ×) | (21, 131, ×) | (40, 249, ×) |
| M7 | (10, 97, ×) | (16, 151, ×) | (16, 254, ×) | ( 32, 218, ×) | (70, 2000, ↑) | (94, 633, ×) |
| M8 | (35, 2000, ↑) | (37, 2000, ↑) | (45, 2000, ↑) | ( 95, 2000, ↑) | (45, 2000, ↑) | (65, 481, √) |
| M9 | (95, 2000, ↑) | (70, 2000, ↑) | (82, 2000, ↑) | ( 150, 2000, ↑) | (40, 94, √) | (120, 622, √) |
| M10 | (115, 2000, ↑) | (80, 2000, ↑) | (90, 2000, ↑) | ( 87, 803, ×) | (125, 1243, √) | (129, 725, ×) |
| cases | p.i. | | | d.e. | | |
| | run 1 | run 2 | run 3 | run 1 | run 2 | run 3 |
| M1 | (13, 49,×) | (31, 70,×) | (143, 484,×) | (20, 40,×) | (100, 275,×) | (49, 104,×) |
| M2 | (175, 576, ×) | (148, 423, ×) | (317, 952, ×) | (258, 2000, ↑) | (661, 1482, ×) | (395, 2000, ↑) |
| M3 | (251, 700, ×) | (177, 492, ×) | (38, 120, ×) | (120, 2000, ↑) | (88, 399, ×) | (240, 2000, ↑) |
| M4 | (202, 558, ×) | (190, 547, ×) | (50, 137, ×) | ( 191, 1389, ×) | (250, 2000, ↑) | (28, 145, ×) |
| M5 | (18, 97, ×) | (28, 128, ×) | (41, 181, ×) | (95, 1509, ×) | (73, 419, ×) | (86, 1396, ×) |
| M7 | (24, 68, ×) | (24, 51, ×) | (77, 252, ×) | ( 68, 248, ×) | (23, 92, ×) | (51, 195, ×) |
| M6 | (96, 270, ×) | (75, 250, ×) | (38, 105, ×) | ( 12, 99, ×) | (25, 121, ×) | (61, 155, ×) |
| M8 | (265, 707, √) | (265, 708, √) | (265, 712, √) | ( 115, 2000, ↑) | (135, 2000, ↑) | (135, 2000, ↑) |
| M9 | (460, 1235, √) | (460, 1234, √) | (460, 1232, √) | ( 104, 1685, ↑) | (145, 2000, ↑) | (275, 2000, ↑) |
| M10 | (29, 157, ×) | (178, 548, ×) | (80, 230, ×) | ( 74, 385, ×) | (23, 130, ×) | (31, 259, , ×) |

Table 3.4: Experiments of dfs_single_1: the approximation techniques used separately

| cases | r.w. + p.i. | | | r.w. + d.e. | | |
|---|---|---|---|---|---|---|
| | run 1 | run 2 | run 3 | run 1 | run 2 | run 3 |
| M1 | (130, 305, √) | (130, 220, √) | (96, 185, ×) | (90, 1207,√) | (85, 323, √) | (130, 172,√) |
| M2 | (220, 426, √) | (316, 190, √) | (143, 314, ×) | (115, 500, √) | (155, 505, √) | (160, 316, √) |
| M3 | (80, 132, ×) | (45, 58, √) | (74, 132, ×) | (95, 400, ×) | (115, 397, √) | (220, 793, √) |
| M4 | (98, 205, ×) | (95, 136, √) | (130, 207, √) | ( 140, 1174, √) | (190, 822, √) | (95, 284, √) |
| M5 | (135, 259, √) | (210, 447, √) | (105, 171, √) | ( 70, 134, √) | (140, 445, ×) | (40, 112, √) |
| M6 | (18, 81, ×) | (58, 137, ×) | (71, 138, ×) | ( 79, 457, ×) | (14, 63, ×) | (68, 213, ×) |
| M7 | (75, 114, √) | (60, 104, ×) | (32, 59, ×) | ( 75, 231, ×) | (170, 328, √) | (58, 225, ×) |
| M8 | (75, 133, √) | (100, 164, √) | (85, 136, √) | ( 105, 442, √) | (80, 2000, ↑) | (110, 2000, ↑) |
| M9 | (55, 97, √) | (145, 251, √) | (115, 199, √) | ( 125, 542, √) | (185, 682, √) | (205, 456, √) |
| M10 | (210, 401, √) | (114, 255, ×) | (185, 390, ×) | ( 120, 312, √) | (150, 368, √) | (175, 735, √) |
| cases | p.i. + d.e. | | | r.w. + p.i. + d.e. | | |
| | run 1 | run 2 | run 3 | run 1 | run 2 | run 3 |
| M1 | (155, 374,×) | (26, 64,×) | (50, 128,×) | (130, 269,×) | (32, 78,×) | (18, 62,×) |
| M2 | (275, 693, √) | (385, 873, √) | (310, 772 √) | (105, 173, √) | (85, 137, √) | (90, 154,√) |
| M3 | (137, 290,×) | (41, 113, ×) | (23, 52, ×) | (105, 151, √) | (85, 106, √) | (105, 155, √) |
| M4 | (116, 244, ×) | (259, 547, ×) | (95, 208, ×) | ( 115, 148, √) | (65, 79, √) | (150, 204, √) |
| M5 | (48, 167, ×) | (177, 465, ×) | (224, 531, ×) | ( 115, 171, √) | (62, 164, ×) | (150, 232, √) |
| M6 | (124, 358, ×) | (102, 288, ×) | (18, 37, ×) | ( 87, 200, ×) | (75, 102, √) | (53, 108, ×) |
| M7 | (56, 117, ×) | (18, 63, ×) | (31, 68, ×) | (17, 38, ×) | (91, 161, ×) | (77, 157, ×) |
| M8 | (200, 421, √) | (240, 527, √) | (190, 410, √) | ( 110, 174, √) | (50, 64, √) | (70, 110, √) |
| M9 | (370, 825, √) | (310, 653, √) | (425, 1006, √) | ( 135, 192, √) | (145, 278, √) | (170, 303, √) |
| M10 | (27, 113, ×) | (240, 586, ×) | (27, 62, ×) | ( 98, 216, ×) | (85, 131, ×) | (129, 236, ×) |

Table 3.5: Experiments of dfs_single_1: the approximation techniques used in combination

| cases | plain | | | r.w. | | |
|---|---|---|---|---|---|---|
| | run 1 | run 2 | run 3 | run 1 | run 2 | run 3 |
| M1 | (20, 108, ×) | (24, 695, ×) | (54, 1250, ×) | (31, 151, ×) | (84, 867, ×) | (80, 329, ×) |
| M2 | (53, 2000, ↑) | (110, 2000, ↑) | (132, 2000, ↑) | (85, 926, √) | (105, 2000, ↑) | (152, 2000, ↑) |
| M3 | (29, 158, ×) | (100, 2000, ↑) | (34, 213, ×) | (65, 2000, ↑) | (100, 831, √) | (22, 149, ×) |
| M4 | (40, 1636, ×) | (210, 2000, ↑) | (200, 2000, ↑) | ( 80, 2000, ↑) | (85, 2000, ↑) | (50, 107, √) |
| M5 | (43, 168, ×) | (17, 134, ×) | (100, 2000, ↑) | (18, 151, ×) | (31, 173, ×) | (97, 768, ×) |
| M6 | (19, 76, ×) | (44, 968, ×) | (11, 71, ×) | ( 120, 1162, √) | (150, 2000, ↑) | (23, 86, ×) |
| M7 | (39, 506, ×) | (23, 256, ×) | (35, 502, ×) | ( 57, 258, ×) | (22, 92, ×) | (55, 225, ×) |
| M8 | (50, 2000, ↑) | (39, 2000, ↑) | (90, 2000, ↑) | ( 50, 2000, ↑) | (20, 27, √) | (145, 1633, √) |
| M9 | (90, 2000, ↑) | (205, 2000, ↑) | (105, 2000, ↑) | ( 225, 1556, √) | (125, 854, √) | (20, 24, √) |
| M10 | (184, 1860, ×) | (20, 1610, ×) | (19, 270, ×) | ( 133, 580, ×) | (59, 603, ×) | (95, 439, √) |
| cases | p.i. | | | d.e. | | |
| | run 1 | run 2 | run 3 | run 1 | run 2 | run 3 |
| M1 | (68, 31, ×) | (36, 87, ×) | (148, 360, ×) | (133, 306, ×) | (79, 298, ×) | (19, 103, ×) |
| M2 | (448, 1163, ×) | (439, 1129, ×) | (406, 1032, ×) | (314, 2000, ↑) | (267, 874, ×) | (290, 1581, √) |
| M3 | (102, 236, ×) | (266, 585, ×) | (89, 231, ×) | (34, 102, ×) | (44, 269, ×) | (279, 1581, ×) |
| M4 | (49, 131, ×) | (198, 453, ×) | (20, 48, ×) | (237, 759, ×) | (125, 2000, ↑) | (191, 1063, ×) |
| M5 | (241, 590, ×) | (266, 647, ×) | (22, 75, ×) | ( 182, 1171, ×) | (243, 946, ×) | (169, 1118, ×) |
| M6 | (96, 231, ×) | (74, 196, ×) | (49, 138, ×) | ( 36, 100, ×) | (87, 279, ×) | (25, 49, ×) |
| M7 | (18, 43, ×) | (22, 54, ×) | (25, 60, ×) | ( 118, 760, ×) | (14, 70, ×) | (16, 59, ×) |
| M8 | (265, 608, √) | (265, 605, √) | (265, 607, √) | ( 205, 1593, √) | (200, 2000, ↑) | (135, 2000, ↑) |
| M9 | (465, 1043, √) | (465, 1044, √) | (465, 1048, √) | ( 250, 2000, ↑) | (274, 2000, ↑) | (205, 2000, ↑) |
| M10 | (26, 75, ×) | (24, 72, ×) | (13, 54, ×) | ( 238, 1194, ×) | (220, 611, ×) | (19, 110, ×) |

Table 3.6: Experiments of dfs_single_2: the approximation techniques used separately

| cases | r.w. + p.i. | | | r.w. + d.e. | | |
|---|---|---|---|---|---|---|
| | run 1 | run 2 | run 3 | run 1 | run 2 | run 3 |
| M1 | (45, 85, ×) | (70, 104, √) | (99, 185, ×) | (100, 295, √) | (150, 298, √) | (110, 246, √) |
| M2 | (65, 100, √) | (115, 188, √) | (130, 227, √) | (89, 1611, ×) | (95, 243, , √) | (75, 650, √) |
| M3 | (110, 144, √) | (130, 183, √) | (215, 307, √) | (95, 268, √) | (70, 159, √) | (143, 366, √) |
| M4 | (26, 52, ×) | (39, 70, ×) | (19, 44, ×) | ( 145, 296, √) | (75, 148, √) | (130, 565, √) |
| M5 | (30, 37, √) | (115, 196, ×) | (25, 34, √) | (45, 76, √) | (120, 276, √) | (105, 233, √) |
| M6 | (88, 165, ×) | (14, 36, ×) | (16, 34, ×) | ( 69, 268, ×) | (42, 621, ×) | (12, 38, ×) |
| M7 | (103, 171, ×) | (80, 120, ×) | (19, 38, ×) | ( 21, 111, ×) | (26, 91, ×) | (21, 123, ×) |
| M8 | (100, 164, √) | (100, 172, √) | (45, 63, √) | ( 90, 706, √) | (60, 285, √) | (80, 431, √) |
| M9 | (215, 369, √) | (75, 121, √) | (150, 213, √) | ( 85, 228, √) | (155, 739, √) | (115, 283, √) |
| M10 | (150, 232, √) | (68, 107, ×) | (30, 65, ×) | ( 70, 134, √) | (75, 266, √) | (70, 102, √) |
| cases | p.i. + d.e. | | | r.w. + p.i. + d.e. | | |
| | run 1 | run 2 | run 3 | run 1 | run 2 | run 3 |
| M1 | (173, 386, ×) | (106, 211, ×) | (27, 86, ×) | (16, 41, ×) | (76, 133, ×) | (75, 147, √) |
| M2 | (410, 878, √) | (317, 695, ×) | (355, 706, √) | (175, 247, √) | (75, 99, √) | (165, 233, √) |
| M3 | (21, 42, ×) | (285, 511, ×) | (32, 63,×) | ( 60, 70,√) | (125, 166, √) | (120, 155, √) |
| M4 | (315, 603, √) | (39, 95, ×) | (209, 401, ×) | ( 148, 220, ×) | (130, 172, √) | (28, 66, ×) |
| M5 | (39, 108, ×) | (294, 605, ×) | (17, 50, ×) | ( 120, 174, √) | (96, 146, ×) | (191, 319, ×) |
| M6 | (52, 119, ×) | (36, 80, ×) | (28, 56, ×) | ( 74, 111, ×) | (80, 114, √) | (80, 108, √) |
| M7 | (15, 41, ×) | (178, 345, ×) | ( 16, 45, ×) | (73, 108, ×) | (36, 60, ×) | (57, 89, ×) |
| M8 | (100, 198, √) | (220, 430, √) | (210, 387, √) | ( 65, 105, √) | (140, 229, √) | (80, 111, √) |
| M9 | (350, 698, √) | (395, 803, √) | (380, 746, √) | ( 120, 183, √) | (185, 267, √) | (130, 169, √) |
| M10 | (86, 181, ×) | (36, 100, ×) | (87, 196, ×) | ( 100, 172, ×) | (65, 103, ×) | (90, 133, √) |

Table 3.7: Experiments of dfs_single_2: the approximation techniques used in combination

after the transition `lower`, while in the original specification there is a delay of `lower_time` between them. This won't cause any problems for the critical requirements of the `Gate` process, because eliminating the delay is essentially a special case of the original specification where `lower_time=0`. All the other mutations are flawed. Therefore the model-checker is expected to find violations.

The test results are evaluated by two criteria:

- On average, how much time is spent in finding an error when using a specific combination of the approximation techniques or without using them. This is called *cost per error*, denoted by CPE.
- The average chance that a run is able to kill a mutant. This is called *error finding ratio*, denoted by EFR.

Based upon the test data, we calculate CPE as follows. Excluding the live mutants ( M8 and M9, since both are correct), are the remaining eight mutations able to be killed? For instance, for the algorithm dfs_single_1 without using approximations (i.e., under the column "plain" in Table 3.4), CPE(dfs_single_1, plain) is the total run time of the eight mutations divided by the total number of runs where an error was found, i.e.,

$$39770/9 = 4418.$$

This indicates the average time to find a violation. All the CPEs are presented in Table 3.8 under the columns "cost per error".

We would also like to know whether aggressive approximations are effective in detecting violations. This is measured by the EFR. For instance, for the algorithm dfs_single_2 with the random walk technique, the three columns under "r.w." show that 13 runs out of 24 for the eight mutations successfully kill a mutant. Since each of the eight mutations are flawed, the expected number of successful runs is 24. Therefore, EFR(dfs_single_2, r.w.) is

$$13/24.$$

This indicates the chance that the model-checker is able to kill a mutant using the specific algorithm and the random walk technique. All the EFRs are presented in Table 3.8 under the columns "error finding ratio".

| approximations | dfs_single_1 | | dfs_single_2 | |
|---|---|---|---|---|
| | cost per error | error finding ratio | cost per error | error finding ratio |
| plain | 4418 | 9/24 | 1498 | 17/24 |
| r.w. | 1896 | 13/24 | 1530 | 13/24 |
| p.i. | 310 | 24/24 | 317 | 24/24 |
| d.e. | 991 | 19/24 | 777 | 22/24 |
| r.w. + p.i. | 406 | 12/24 | 224 | 13/24 |
| r.w. + d.e. | 1502 | 7/24 | 1017 | 7/24 |
| p.i. + d.e. | 344 | 21/24 | 288 | 21/24 |
| r.w. + p.i. + d.e. | 302 | 12/24 | 260 | 13/24 |

Table 3.8: Average cost of finding an error and the ratio of the number of errors found to the total number of errors for dfs_single_1 and dfs_single_2

A number of observations can be made from the results shown in Table 3.8.

- The difference of the two algorithms dfs_single_1 and dfs_single_2 is the different level in strengthening a transition, as previously introduced. dfs_single_2 is more accurate than dfs_single_1 in approximating the environment. Thus, it reduces the size of reachable images and makes the model-checker faster. Therefore, dfs_single_2 makes it possible for the model-checker to search more nodes within the time-limit, and it makes it more likely that the model-checker will find a bug. This is reflected in both the CPRs (4418 vs. 1498) and EFRs (9/24 vs. 17/24) for the two algorithms without using the approximation techniques. Thus, the data under the column "dfs_single_1" in Table 3.8 indicates that when the approximation techniques are not used the model-checker usually runs out of time before a violation is found. The data under the column "dfs_single_2" in Table 3.8 indicates that even without the approximation techniques the model-checker usually is able to find a bug marginally within the time-limit.

- The partial image technique is considered the most effective approach. For the algorithm dfs_single_1, it is 14 (4418/310) times faster than without using approximations. For dfs_single_2, it is almost 5 (1498/317) times faster. Notice that, for both algorithms, the EFRs with the partial image technique are 24/24, which means the technique is fast and extremely effective.

- The dynamic environment generation technique has high EFRs (19/24 and 22/24), even when it is used in combination with the partial image technique (21/24 and 21/24). For the two algorithms, it is 4 (4418/991) and 2 (1498/777) times faster than without using approximations, respectively. When used in combination, it is 12 (4418/344) and 5 (1498/288) times faster than without using approximations, respectively, while still keeping extremely high EFRs.

- Surprisingly, the random walk technique does not perform as well as expected. Used alone, it has low EFRs (13/24 and 13/24) and high costs per error (1896 and 1530). The reason is that even though a number of branches are skipped during the search procedure, the image computations along a single path are still expensive, and in some cases the model-checker is unable to hit the violation path before the time-limit is reached. In addition, skipping a branch means the whole subtree that starts from the branch is skipped by the model-checker. This approach sacrifices more coverage than the other two techniques; thus, this technique has a higher chance of skipping a violation path. This fact is indicated by its low EFRs including the cases when the technique is used in combination with the other two techniques. Thus, compared to the other two techniques, we do not consider random walk to be effective when used alone. But used in combination with the partial image technique (in both r.w.+p.i. and r.w.+p.i.+d.e.) it is acceptable. In this case, the results in the table point out that it has around 12/24 EFRs and on average is 6 times faster than without using the approximations.

- The techniques used in combination could speed up the search procedure, but, obviously since more coverage is sacrificed, this usage does not necessarily imply more effectiveness. For instance, the dynamic environment generation approach has very high EFRs (19/24 and 22/24). But when it is used with the other two techniques, the EFRs drop to 12/24 and 13/24, respectively. This is compensated by almost 3 times lower cost per error. For the partial image technique there is almost no compensation in CPRs at all when it is used with the other techniques.

As mentioned before, the two live mutations M8 and M9 are both correct. Runs on them can be used to analyze node coverage for each approximation technique. Node coverage (NC) is measured by the total number of nodes searched in both M8 and M9 for a specific algorithm. For instance, for dfs_single_1 with the dynamic environment generation technique used alone, the sum of nodes under columns "d.e." in Table 3.4 for M8 and M9 is 779. This number is NC(dfs_single_1, d.e.). All the NCs are presented in Table 3.9. The partial image technique and the dynamic environment generation technique, either used separately or in combination, generally have much higher node coverage (NC(dfs_single_1, d.e.)=779 is an exception) than all the others. This explains the fact that these techniques also have extremely high EFRs as shown in Table 3.8. In contrast, the random walk technique, whenever used separately or in combination with the other techniques, has significantly low node coverage. This is also consistent with the fact that it has low EFRs as shown in Table 3.8.

From the above analysis, one can conclude that the partial image technique and the

| approximations | dfs_single_1 | dfs_single_2 |
|---|---|---|
| plain | 364 | 579 |
| r.w. | 810 | 585 |
| p.i. | 2175 | 2190 |
| d.e. | 779 | 1269 |
| r.w. + p.i. | 575 | 685 |
| r.w. + d.e. | 810 | 585 |
| p.i. + d.e. | 1735 | 1655 |
| r.w. + p.i. + d.e. | 680 | 720 |

Table 3.9: Node coverage (NC) for the M8 and M9

dynamic environment generation technique are both effective, either when used separately or in combination, – they are able to kill all the mutants in almost every run and in a much shorter time. The random walk technique, when used alone, is not considered effective, due to the fact that it is not significantly faster in finding an error and it suffers from a low EFR. The situation is improved somewhat when it is used in combination with the other two techniques.

## 3.5 Related Work

The underlying computation model for Small-ASTRAL and the model checker is based upon labeled transition systems. The model was widely used in various contexts, from concurrent programs [S93] to timed automata [AD94]. The experience gained from our work, therefore, will not be limited to the ASTRAL environment. It should be applicable to debugging specifications written in many other expressive languages as long as safety properties are concerned.

The symbolic model checker in this chapter carries out symbolic search up to a pre-assigned depth. This type of a *bounded model checking* approach was used in [AHV93] for a form of parameterized timed automata. But the image computations in [AHV93] are essentially breadth-first search. That is, the symbolic representation of all the reachable states is calculated at each iteration. Our experience shows that breadth-first search does not work well for large specifications. The reason is that the image size at each iteration step grows dramatically and can be easily out of control for the model checker. Thus, the model checker works on the execution tree of the transition system. In addition to the breadth-first search, we also propose the depth-first search and the depth-breadth search strategies, which, as far as we know, are not covered by previous studies in symbolic verification/debugging infinite state systems, though both depth-first search and breadth-first search are adopted in a number of explicit finite state model checkers [H97, CGJ98, D96]. Furthermore, the discrete clocks model that we consider is stronger than parameterized timed automata [AHV93]. This is because we allow Presburger formulas over clocks and data variables. Our method of image computations on an execution tree can be traced back to the earlier symbolic execution techniques for programs [C76, K76, KE85] as well as for specifications [DK94]. However, the major difference is that our procedure is fully automatic and applicable to real-time systems.

An environment is tied to an ASTRAL transition at different levels of approximation: from the cheapest single-event assumptions (i.e., assuming, during the transition's execution, there is at most one event, such as a change of an environmental clock occurring) to the most expensive multiple-event assumptions (i.e., assuming that multiple events could happen). Since the model checker is essentially a debugger, a user need not go for a higher level if using a lower level approximation of the environment successfully detects a violation. This is influenced by the idea of building layered specifications. That is, for the purpose of both better design and better verification, specifications may be first written at the most abstract layer then refined down to a layer close to the real implementation, as shown in the work of Gerber and Lee [GL92] and HMS [GF91], as well as in ASTRAL [CGK97]. But here we deal with one layer of ASTRAL specification and the purpose of introducing various levels of approximations on the environment is to decrease the cost of image computations. We were unable to find similar techniques for restricting the behaviors of an environment for cheaper symbolic model checking of real-time systems in the literature.

The three approximation techniques for image computations(i.e., random walk, partial image and dynamic environment generation) are influenced by advocates for using lightweight formal methods [JW96]. We prefer to consider the three techniques to be natural instead of original, though we are not not aware of their use in symbolic model checking/debugging of infinite state real-time systems. The name "random walk" is borrowed from the theory of stochastic processes. In particular, the *random simulation* technique has been used in protocol testing for many years [H91], by exercising a random path of a protocol. But random simulation is essentially explicit state based, unlike our use of it for symbolic model checking on the execution tree of a specification. The partial image technique was inspired by traditional techniques in program testing, such as sampling and random testing methods [DW80, DN81] as well as domain testing [WC81]. However, instead of picking a single or several samples from the domain, the partial image technique selects a subset of the preimage and uses this subset to calculate the postimage at each node. Dill and Wong-Toi [DW95] considered the *underapproximation* technique for reachable sets of timed automata. These sets are pairs of control states and clock regions. Our partial image technique, however, is built on the representation of Presburger formulas via the Omega library. Thus, it is applicable to a wider range of transition systems, including infinite-state real-time systems. In addition, the work of Dill and Wong-Toi does not consider the depth-first and the depth-breadth search strategies. The dynamic environment generation technique is similar to the idea of Colby, Godefroid and Jagadeesan [CGJ98] in that both address the problem of automatically closing an open system, in which some of the components are not present. Their approach targets concurrent programs, instead of real-time specifications. Unlike doing static analysis of a concurrent program, our technique dynamically selects a reasonable environment according to the imported variable clause of the ASTRAL specification. With concrete values for the environment as well as the history, the cost of later image calculations can be greatly decreased.

Bultan [BGP97, BGL98] used the Omega library as a tool to symbolically represent a set of states that is characterized by a Presburger formula. He also investigated partitions and approximations in order to calculate fixed points. As in the work presented in this chapter, Bultan worked with infinite state systems. However, the systems Bultan considered are "simple" in the following sense: (1) quantifications are only limited to a very small number, (2) the transition system is a straightforward history-independent transition system; i.e., the current state only depends upon the last state, and other history references are not allowed, (3) the transition system itself is not a real-time system in the sense that no duration is attached to a transition and the start and end times are not allowed to be referenced. Unfortunately, a typical ASTRAL specification, such as the benchmark considered in this chapter, is not "simple". For these complex systems, a fixed point (i.e., the set of all reachable states) may not be computable. However, because the ASTRAL symbolic model checker is primarily intended to be used as a debugger instead of a verifier, calculating the fixed point of a transition system is not an important issue. Therefore, Bultan's approaches, as well as other approximation techniques, such as abstraction [CC99, CGL92, DGG97] and over-approximation [DW95], can be considered to be orthogonal to the approaches presented in this chapter.

The model checker considered in this chapter is modularized; one need only check one process instance for each process type declared, without looking at the transition behaviors of other process instances. The STeP system also uses a modularized approach [Bjorner et al. 96, BMSU97]. However, STeP primarily uses a theorem prover to validate a property while the approach presented here uses a fully automatic model checker. Comparisons between ASTRAL and other specification languages on a wide range of language design issues can be found in the ASTRAL overview paper [CGK97].

# Chapter 4

# History-independent Mini-ASTRAL and Its Extensions

In the previous chapter, we presented bounded-testing tools for Small-ASTRAL specifications. A significant restriction of the tools is that before the model-checking procedure is carried out, a pre-assigned depth must be set. However, in practice, especially in the final stage of design, an error, if any, is relatively hard to detect and usually a longer trace is needed to reveal it. In this chapter, we investigate a subset of Mini-ASTRAL that is history-independent, and we show that it is decidable to verify any specification in the subset, without restricting the depth. History-independent Mini-ASTRAL has very limited expressive power. In this chapter, we also discuss a number of possible further extensions of history-independent Mini-ASTRAL such that the extensions also lead to a class of systems that can be automatically verified or can be effectively approximated.

This chapter is organized as follows. In Section 4.1, we formally define a discrete timed automaton, which is a variation of the one in [AD94]. In Section 4.2, we propose a machine model called clock-jump machines. This model is essentially equivalent to discrete timed automata, but clock-jump machines are more convenient for describing clock behaviors in ASTRAL. In Section 4.3, we show that a history-independent Mini-ASTRAL process $\mathcal{P}$ can be translated into a clock-jump machine. In Section 4.4, we consider a number of extensions of history-independent Mini-ASTRAL, by allowing non-region properties, by allowing parameterized durations, and by allowing parameterized constants. In Section 4.5, a proof is given showing that binary reachability of discrete timed automata is Presburger. This result makes it possible to automatically verify history-independent Mini-ASTRAL with non-region properties. In Section 4.6, related work is addressed.

## 4.1 Discrete Timed Automata

A timed automaton [AD94] is a finite state machine augmented with a number of real-valued clocks. All the clocks progress synchronously with rate 1, and a clock can be reset to 0 by some transition. Let $\mathbf{Z}$ be the set of integers with $\mathbf{Z}^+$ representing the nonnegative integers. Here, we consider clocks in $\mathbf{Z}$. A *clock constraint* is a Boolean combination of *atomic clock constraints* in the following form:

$$x\#c, x - y\#c$$

where $\#$ denotes $\leq, \geq, <, >$, or $=$, $c$ is an integer, and $x$ and $y$ are integer-valued clocks. Let $\mathcal{L}_X$ be the set of all clock constraints on clocks $X$.

Formally, a *discrete timed automaton* (DTA) $\mathcal{A}$ is a tuple

$$\langle S, X, E\rangle$$

where $S$ is a finite set of *(control) states.* $X$ is a finite set of *clocks* with values in $\mathbf{Z}^+$. $E \subseteq S \times 2^X \times \mathcal{L}_X \times S$ is a finite set of *edges* or *transitions.* Each edge $\langle s, \lambda, l, s'\rangle$ denotes a transition from state $s$ to state $s'$ with *enabling condition* $l \in \mathcal{L}_X$ and a set of clock resets $\lambda \subseteq X$. Note that $\lambda$ may be empty. Also note that since each pair of states may have more than one edge between them, $\mathcal{A}$ is, in general, nondeterministic.

The semantics is defined as follows. $\alpha \in S \times (\mathbf{Z}^+)^{|X|}$ is called a *configuration* with $\alpha_x$ being the value of clock $x$ and $\alpha_q$ being the state under this configuration. $\alpha \rightarrow^{\langle s,\lambda,l,s'\rangle} \alpha'$ denotes a one-step transition along an edge $\langle s, \lambda, l, s'\rangle$ in $\mathcal{A}$ satisfying

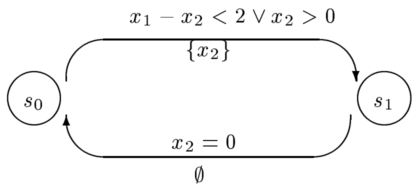


Figure 4.1: An example of a DTA.

- The state $s$ is set to a new location $s'$, i.e., $\alpha_{\mathbf{q}} = s, \alpha'_{\mathbf{q}} = s'$.

- Each clock changes according to the edge given. If there are no clock resets on the edge, i.e., $\lambda = \emptyset$, then clocks progress by one time unit, i.e., for each $x \in X$, $\alpha'_x = \alpha_x + 1$. If $\lambda \neq \emptyset$, then for each $x \in \lambda$, $\alpha'_x = 0$ while for each $x \notin \lambda$, $\alpha'_x = \alpha_x$.

- The enabling condition is satisfied, that is, $l(\alpha)$ is true.

We simply write $\alpha \to \alpha'$ if $\alpha$ can reach $\alpha'$ by a one-step transition. A *path* $\alpha_0 \cdots \alpha_k$ satisfies $\alpha_i \to \alpha_{i+1}$ for each $i$. Also write $\alpha \leadsto^{\mathcal{A}} \beta$ if $\alpha$ reaches $\beta$ through a path.

Figure 4.1 shows an example of a DTA with two clocks $x_1$ and $x_2$. The following sequence of configurations is a path: $\langle s_0, x_1 = 0, x_2 = 0\rangle, \langle s_1, x_1 = 0, x_2 = 0\rangle, \langle s_0, x_1 = 1, x_2 = 1\rangle, \langle s_1, x_1 = 1, x_2 = 0\rangle$.

The above defined $\mathcal{A}$ is a little different from the standard (discrete) timed automaton given in [AD94]. In that model, each state is assigned with a clock constraint called an *invariant* in which $\mathcal{A}$ can remain in the same control state while all the clocks synchronously progress with rate 1 as long as the invariant is satisfied. It is easy to see that, when integer-valued clocks are considered, $\mathcal{A}$'s remaining in a state can be replaced by a self-looping transition with the invariant as the enabling condition and without clock resets. Each execution of such a transition causes all the clocks to progress by one time unit. Another difference is that in a standard DTA a state transition takes no time, even if the transition has no clock resets. In order to translate a standard DTA to our definition, we introduce a dummy clock. Thus, for each state transition $t$ in a standard DTA the translated transition $t'$ is exactly the same as $t$ but the dummy clock is reset in $t'$. This ensures that all clock values remain the same when $t$ has no clock resets. Thus, a standard DTA can be easily transformed into a DTA.

One of the most important results about discrete timed automata is that the region reachability problem is decidable. Let $\mathcal{A}$ be a discrete timed automaton. Given two control states $s_1$ and $s_2$, and two clock constraints $P_1$ and $P_2$ (called *regions*). Are there two configurations $\alpha^1$ and $\alpha^2$ of $\mathcal{A}$ such that

- $\alpha^1 \leadsto^{\mathcal{A}} \alpha^2$,
- $\alpha^1_{\mathbf{q}} = s_1, \alpha^2_{\mathbf{q}} = s_2$,
- clock values in $\alpha^1$ (resp $\alpha^2$) satisfy $P_1$ (resp $P_2$) ?

That is, find configurations $\alpha^1$ and $\alpha^2$ such that $\alpha^1$ at control state $s_1$ in region $P_1$ can reach $\alpha^2$ at control state $s_2$ in region $P_2$. This problem is called *the region reachability problem.* By constructing a finite bisimulation, this problem can be reduced to a reachability problem of finite automata, as shown in [AD94].

**Theorem 1** *The region reachability problem for (discrete) timed automata is decidable [AD94].*

In the next section, we will propose a variation of discrete timed automata called clock-jump machines. These machines also have a decidable region reachability problem. Later we will show how to translate a history-independent mini-ASTRAL process instance into a clock-jump machine and reduce the verification problem of history-independent mini-ASTRAL to the region reachability problem of these machines. Therefore, history-independent mini-ASTRAL can be automatically verified.

## 4.2 Clock-Jump Machines

In an ASTRAL process instance, a clock such as the start time of a transition `T`, `Start(T)`, behaves as follows. `Start(T)` stays unchanged until `T` starts. At the moment when `T` starts, `Start(T)` jumps to the current time `NOW`. Thus, the concept of clock resets in a discrete timed automaton defined in the previous section is not convenient for describing clock behaviors in ASTRAL. In this section, we introduce a variation, called clock-jump machines, of discrete timed automata that use clock jumps instead of clock resets. In addition, the machines also use extended clock constraints defined as follows.

An *extended clock constraint* is a Boolean combination of *extended atomic clock constraints* in the following form:

$$f ::= t_{clock} < t_{finite} \mid t_{clock} = t_{finite} \mid t_{clock} > t_{finite}$$

where clock terms are

$$t_{clock} ::= x \mid x - y$$

and finite-variable terms are

$$t_{finite} ::= v \mid c \mid t_{finite} + t_{finite} \mid t_{finite} - t_{finite}$$

in which $x, y$ are clocks and $v$ stands for a finite state variable and $c$ is an integer constant. Denote $\mathcal{E}_{X,V}$ as the set of extended clock constraints on a finite set of clocks $X$ and a finite set of finite state variables $V$. An extended clock constraint can be rewritten as a clock constraint if all the finite state variables are explicitly enumerated. For each finite state variable $v \in V$, we use $dom(v)$ to denote its domain, and use $dom(V)$ to denote the cartesian product $\Pi_{v \in V} dom(v)$.

Formally, a *clock-jump machine* $\mathcal{J}$ is a tuple

$$\langle Q, E, L, X, V, \textbf{Init}, \textbf{Assump}, \textbf{Prop} \rangle$$

that consists of a finite set $Q$ of *control locations* with a special location $q_0$ being the initial location, a finite set of edges

$$E \subseteq Q \times Q.$$

$X$ is a finite set of clocks (including a special clock *now* representing the current time), and $V$ is a finite set of finite state variables. **Init**, **Assump** and **Prop** are extended clock constraints in $\mathcal{E}_{X,V}$, which indicate the initial condition, the assumptions (that must be satisfied at each transition), and the (safety) property that needs to be verified. Each edge $e$ is associated with an entry condition $entry_e \in \mathcal{E}_{X,V}$ and an exit condition $exit_e \in \mathcal{E}_{X,V \cup V'}$. A *label* is a subset of clocks. $L$ labels each edge with a finite number of labels $\lambda$. Each label is either an *idle label* ($\lambda = X$) or a *non-idle label* ($now \notin \lambda$).

Assume $x_0 = now, x_1, \cdots, x_k$ are clocks in $X$. We use $\mathbf{X}$ and $\mathbf{V}$ to denote a vector of the clock values and a vector of finite state values. We use a primed notation to indicate the next value after a transition. For instance $\mathbf{X}' = \mathbf{X}$ indicates clocks do not change. Let $\lambda$ be a label. Thus, either $now \notin \lambda$ or $\lambda = X$. The notation $[\mathbf{X}, \mathbf{X}']^\lambda$ is an abbreviation of the following condition:

- If $\lambda = \emptyset$, then only the now-clock *now* progress by one time unit, all the other clocks do not change. That is, $\mathbf{X}'[0] = \mathbf{X}[0] + 1$, and $\mathbf{X}'[i] = \mathbf{X}[i]$ for each $1 \le i \le k$.

- If $X \ne \lambda \ne \emptyset$, then all the clocks in $\lambda$ jump to *now*, and all the other clocks do not change. That is, for each $0 \le i \le k$ with $x_i \in \lambda$, $\mathbf{X}'[i] = \mathbf{X}[0]$, and for each $0 \le i \le k$ with $x_i \notin \lambda$, $\mathbf{X}'[i] = \mathbf{X}[i]$.

- If $X = \lambda$, then all the clocks do not change. That is, for each $0 \le i \le k$, $\mathbf{X}'[i] = \mathbf{X}[i]$.

The semantics of $\mathcal{J}$ is defined as follows. A *configuration* $\alpha$ is a tuple $(\alpha_\mathbf{q}, \mathbf{X}, \mathbf{V})$ where $\alpha_\mathbf{q}$ is the control location, $\mathbf{X}$ is an array of the clock values, and $\mathbf{V}$ is an array of the values for finite state variables under the configuration. We use $\alpha_{X,V}$ to denote $(\mathbf{X}, \mathbf{V})$ in $\alpha$.

$$\alpha \rightarrow^{e, entry_e, exit_e, \lambda} \beta$$

denotes a *one-step transition* along an edge $e$ under a label $\lambda \in L(e)$ satisfying

- $e = (\alpha_\mathbf{q}, \beta_\mathbf{q})$. That is, the edge $e$ connects the location $\alpha_\mathbf{q}$ to the location $\beta_\mathbf{q}$.

- Clocks change according to $\lambda$. In addition, the assumption **Assump**, the entry assertion $entry_e$ and the exit assertion $exit_e$ must be satisfied. Assume $\alpha = (\alpha_\mathbf{q}, \mathbf{X}, \mathbf{V})$ and $\beta = (\beta_\mathbf{q}, \mathbf{X}', \mathbf{V}')$. Formally, $\mathbf{X}, \mathbf{V}, \mathbf{X}'$ and $\mathbf{V}'$ satisfy

$$\mathbf{Assump}(\mathbf{X}, \mathbf{V}) \wedge entry_e(\mathbf{X}, \mathbf{V}) \wedge exit_e(\mathbf{X}', \mathbf{V}, \mathbf{V}') \wedge [\mathbf{X}, \mathbf{X}']^{\lambda_e}.$$

We simply write $\alpha \rightarrow \beta$ if $\alpha$ can reach $\beta$ by a one-step transition. A *path*

$$\alpha_0 \cdots \alpha_k$$

satisfies

$$\alpha_i \rightarrow \alpha_{i+1}$$

for each $i$. Also write $\alpha \rightsquigarrow^{\mathcal{J}} \beta$ if $\alpha$ reaches $\beta$ through a path. $\alpha$ is called an *initial configuration* if $\alpha_{\mathbf{q}} = q_0$ and $\alpha_{X,V} \in \mathbf{Init}$.

The *verification problem* of a clock-jump machine $\mathcal{J}$ is whether starting from an initial configuration $\alpha$, $\mathcal{J}$ can only reach a configuration satisfying **Prop**. Essentially, a clock-jump machine is equivalent to a discrete timed automaton by translating each clock $x_i$ in the clock-jump machine into a clock $y_i = now - x_i$ in the discrete timed automaton (thus, a clock jump of $x_i$ corresponds to a clock reset of $y_i$.) and building finite state variables in $V$ of the clock-jump machine into the control states for the discrete timed automaton. Thus, the verification problem of clock-jump machines can be reduced to the region reachability problem of discrete timed automata. Hence, it is decidable from Theorem 1.

**Theorem 2** *The verification problem of clock-jump machines is decidable.*

In the next section, we will show that a history-independent Mini-ASTRAL process instance can be translated into a clock-jump machine.

## 4.3 Relating a History-independent Mini-ASTRAL Process Instance to a Clock-jump Machine

In this section, we use $\mathcal{P}$ to denote a history-independent Mini-ASTRAL process instance, and use $\mathcal{J}$ to denote a clock-jump machine. We start by looking at whether each component of $\mathcal{P}$ can be properly translated into those in $\mathcal{J}$.

Local variables in $\mathcal{P}$, by definition, are of finite states. Thus, they naturally correspond to finite state variables $V$ in $\mathcal{J}$.

`NOW` can be directly mapped to *now* in $\mathcal{J}$ indicating the current time. Clocks in $\mathcal{P}$ include the (last) start time `Start(T)`, the (last) end time `End(T)`, the (last) call time `Call(T)`, of a transition `T` (which is either imported or local), and the (last) change time `Change(x)` of a variable `x` (which is either local or imported). Therefore, it is natural to relate a clock in $\mathcal{P}$ directly to a clock in $\mathcal{J}$. For instance, whenever a transition fires, there is a jump on the clock `Start(T)` that brings it to *now*.

By definition, initial conditions, assumptions and properties in $\mathcal{P}$ are in the form of extended clock constraints. Thus, they are directly mapped into **Init**, **Assump**, and **Prop** in $\mathcal{J}$. Given an instance of a transition, the entry assertion and the exit assertion, being in the form of clock constraints by definition, are directly mapped into the entry condition and the exit condition of an edge in $\mathcal{J}$.

Up to now, we have shown that the clocks and formulas in $\mathcal{P}$ are consistent with those required by $\mathcal{J}$, but we still do not know how to construct the machine $\mathcal{J}$. We will discuss this further below.

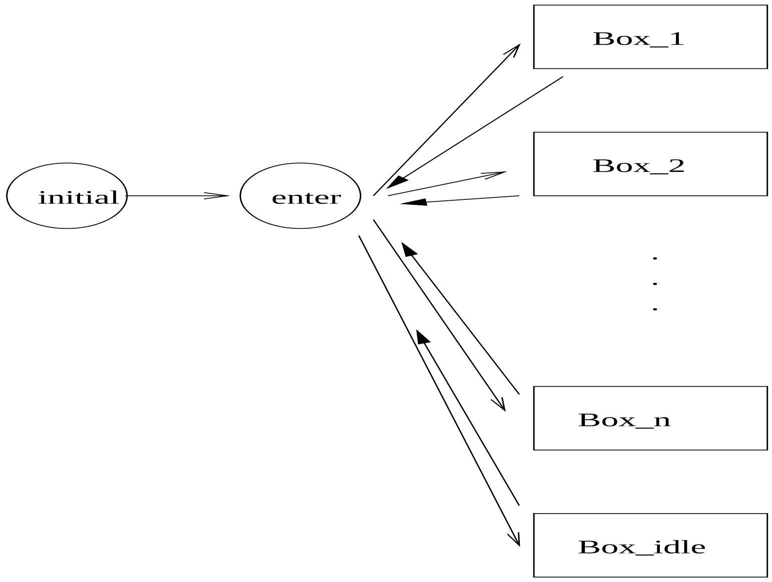


Figure 4.2: Graph representation of the clock-jump machine $\mathcal{J}$

Figure 4.2 shows the graphical presentation of the clock-jump machine $\mathcal{J}$ with a number of uninterpreted components `Box_1` $\cdots$ `Box_n` and `Box_idle`. $\mathcal{J}$ starts from the initial location `initial` as shown in the figure, then moves to the location `enter`. At this time, it chooses (nondeterministically) one of the branches to execute. After finishing the execution of one of the components it moves back to the location `enter`. **Init** in $\mathcal{J}$ is the initial clause in $\mathcal{P}$ plus a number of other initial conditions on the clocks. For instance, all the clocks

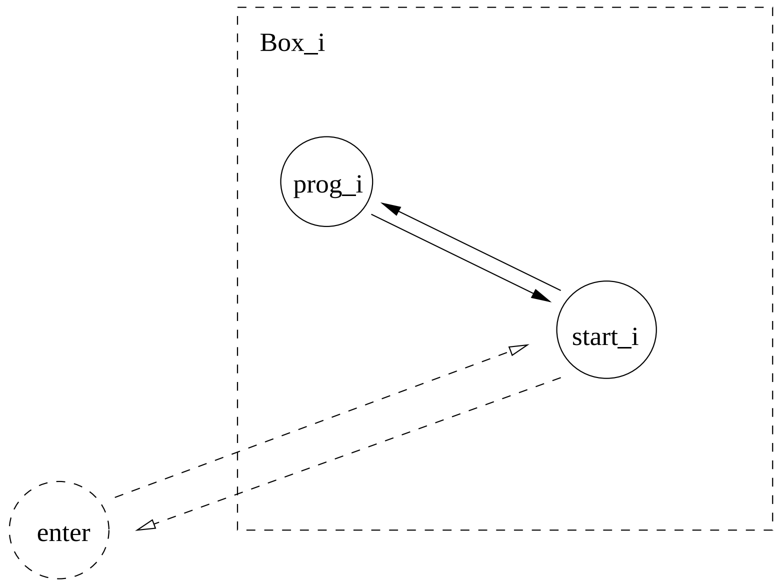


Figure 4.3: Details of `Box_i` of $\mathcal{J}$

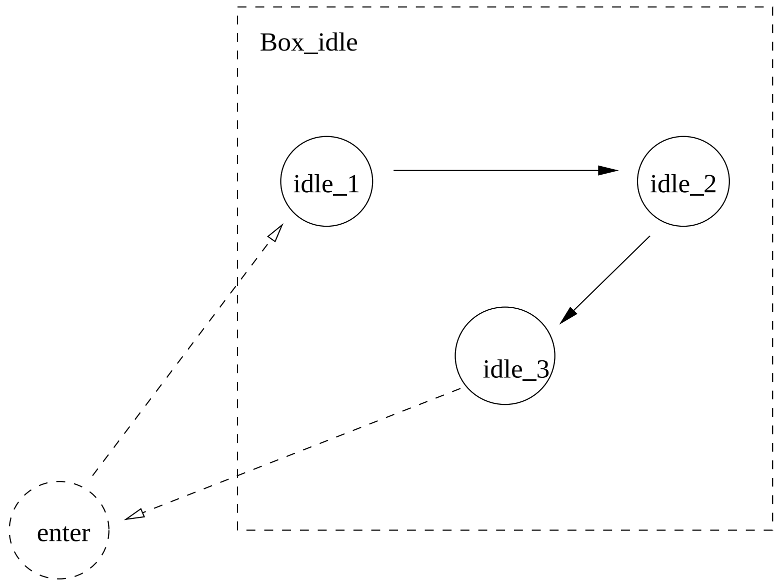


Figure 4.4: Details of `Box_idle` of $\mathcal{J}$

start from 0.[1]

The edge `<initial,enter>` in Figure 4.2 is with the entry condition being *true*, the exit condition keeps all the variables (both local variables and imported variables) unchanged (called *identity*). Labels on this edge are either an idle label or any nonempty subset of clocks corresponding to `Call(T)` (with `T` either local exported or imported) or to `Start(T)` (with `T` imported). It should be noted that *now* does not progress on this edge. Therefore, at this moment, a number of local exported transitions and a number of imported transitions could be called, and a number of imported transitions start. **Assump** is the axiom clause of $\mathcal{P}$. If **Prop** is to check the schedule clause instead of the invariant clause of $\mathcal{P}$, the environment clause and the the imported variable clause are conjoined to **Assump**. A number of axioms to ensure the correct semantics are also added to **Assump**. These include, for instance, a requirement that an exported transition must be called in order for it to start and the transition can not be called if after the previous call the transition has not been fired. According to the semantics of $\mathcal{J}$, **Assump** is used as a global environment to restrict the behavior of $\mathcal{J}$: at each edge, only the values of clocks and finite state variables that satisfy **Assump** can be fired on this edge. Therefore, the graph does not necessary present a total transition system – this is acceptable, since we only have safety properties in **Prop**.

Assume $\mathcal{P}$ has $n$ transition instances $T_1, \cdots, T_n$, each with a single entry-exit pair (A transition with more than one entry-exit pair can be split.). Each $T_i$ is represented as a `Box_i` component in Figure 4.2. According to the ASTRAL semantics, when all the transitions are not firable and none is currently executing the system will idle with *now* progressing by one time unit. This transition, which is not included in the specification, is added as $T_{idle}$, represented by the `Box_idle` component in the figure. Details of each box are presented in Figure 4.3 and Figure 4.4.

Let us first look at the edge `<enter, start_i>` from the location `enter` in Figure 4.2 to the location `start_i` in `Box_i` in Figure 4.3. The entry condition of this edge is the entry

[1]In fact, clocks, other than `NOW`, start from a negative number. The reasons are as follows. `Change(x)` starting from 0 means that the variable `x` changes initially – this violates the ASTRAL semantics. Thus, in this case, we let it start from -1. Another example is `Start(T)`. At the location `initial` in $\mathcal{J}$, it is unknown whether `T` starts at 0 or not. Thus, we temporarily assign it as -1. Sometimes `Start[2](T)` is involved in $\mathcal{J}$. In this case, `Start[2](T)` is always negative (treated as undefined) before the second time `T` starts.

assertion of $T_i$. The exit condition is identity. This edge has only one label $\lambda$: if Start($T_i$) is never used in $\mathcal{P}$, $\lambda$ is an idle label, otherwise $\lambda$ is {Start($T_i$)}. Thus, *now* does not progress on the edge. It is similar for the edge <enter, idle_1> from the location enter in Figure 4.2 to the location idle_1 in Box_idle in Figure 4.4. The entry condition is the entry assertion of $T_{idle}$ (constructed by negating all the entry assertions of other transitions $T_i$.), the exit condition is an identity. The idle label is the only label on this edge. Thus, *now* does not progress on the edge.

When $\mathcal{J}$ enters Box_i as in Figure 4.3, it executes the loop between the location start_i and the location prog_i for $dur(T_i)$ times ($dur(T_i)$ is the duration of $T_i$). The edge <start_i,prog_i> simply make the now-clock *now* progress, without changing other clocks and variables. However, if it is the last loop, local variables are changed according to the exit assertion of $T_i$. The edge <prog_i,start_i> does not make *now* progress, however, it may (nondeterministically) choose one or more of the following clocks to jump (this is similar to the ideas of multiple-event assumptions in Section 3.1.9.):

- Call(T) with T being either a local exported transition or an imported transition,
- Start(T), End(T) with T being an imported transition,
- Change(x) with x being an imported variable.
- End($T_i$), Change(x) (with x being a local variable) if the current loop is the last loop.

These labels will ensure that the environment can nondeterministically change even while $T_i$ is executing. The loop is controlled by a finite state counter $c$ bounded by the duration of $T_i$. The edge <start_i,enter> that goes back to enter simply does not change any clock nor variable value except to reset the counter $c$ back to 0.

The edges in Box_idle are very similar to those in Box_i. The edge <idle_1,idle_2> simply progress the now-clock. <idle_2,idle_3> allows the environment to nondeterministically change as above, but without any changes to local variables and local transition's Start and End times. The reason is that $T_{idle}$ is not a transition in the specification. The edge <idle_3,enter> does not change any clock nor variable value.

So far, a history-independent Mini-ASTRAL process instance $\mathcal{P}$ is translated into a clock-jump machine $\mathcal{J}$. From Theorem 2, it is decidable to verify a history-independent Mini-ASTRAL process instance.

## 4.4 Extensions of History-independent Mini-ASTRAL

History-independent Mini-ASTRAL has very limited expressive power. As shown in the previous section, verifying a history-independent Mini-ASTRAL process instance is essentially equivalent to verifying a discrete timed automaton against a region property. In this section, we will discuss a number of possible further extensions of history-independent Mini-ASTRAL such that the extensions also lead to a class of systems that can be automatically verified or can be effectively approximated.

### 4.4.1 Allowing non-region properties

In ASTRAL, a property **Prop** may not be in the form of clock regions. For instance, the following property

$$\mathtt{Start}(\mathtt{T}_1) - \mathtt{Call}(\mathtt{T}_1) > \mathtt{Start}(\mathtt{T}_2) - \mathtt{Call}(\mathtt{T}_2) + 2$$

is a Presburger formula over clocks. It says that the delay (the difference between the last start time of a transition and the last call time of the same transition) of $\mathtt{T}_1$ is always greater than that of $\mathtt{T}_2$ by at least 2 time units. By extending history-independent Mini-ASTRAL with Presburger properties over clocks, is it still decidable to verify history-independent Mini-ASTRAL process instances? This problem can be reduced to the Presburger safety analysis problem (defined below) for discrete timed automata. Given a discrete timed automaton $\mathcal{A}$ and two sets of configurations $I$ and $P$. If, starting from a configuration in $I$, $\mathcal{A}$ can only reach configurations in $P$, then $P$ is called a *safety property* with respect to the initial condition $I$. The *Presburger safety analysis problem* is whether $P$ is a safety property with respect to the initial condition $I$, given $P$ and $I$ being definable by Presburger formulas over clocks. The following theorem asserts that the Presburger safety analysis problem is decidable for discrete timed automata.

**Theorem 3** *The Presburger safety analysis problem is decidable for discrete timed automata.*

The proof of the theorem can be found in Section 4.5. Thus, history-independent Mini-ASTRAL with Presburger properties over clocks can be automatically verified.

### 4.4.2 Allowing parameterized durations

We can extend history-independent Mini-ASTRAL such that the duration of transition instances can be parameterized (integer) constants. Thus, $\mathcal{P}$ can be parameterized. We use $d_i$ to denote the parameterized durations. However, the use of $d_i$ is restricted. The format of the clock-jump machine $\mathcal{J}$ translated from $\mathcal{P}$ under this extension requires that:

- **Init**, **Assump** and each entry condition of a transition instance are Boolean combinations of extended clock constraints in $\mathcal{E}_{X,V}$ and extended clock constraints in $\mathcal{E}_{V,D}$. Thus, $d_i$ can be used in an entry assertion like $x - y > 4 \wedge d_1 + d_2 < d_3$, but not $x - y > d_1$.

- Each exit assertion of a transition instance is an extended clock constraint in $\mathcal{E}_{V \cup V',D}$.

- **Prop** is a linear relation over $X, V, D$. Thus, a property like $x - y + z > d_1 - d_2$ is allowed.

- Each edge is associated with a parameterized duration $d_i$. When $\lambda$, one of the labels on an edge, is empty, the now-clock progresses by $d_i$.

- Each $d_i$ is positive. This fact is built into **Assump**.

But with such an extension, it is currently open whether the verification problem of $\mathcal{J}$ is decidable or not, since this problem is related to the open problem of whether the emptiness problem of 2NCM(1,r) (nondeterministic counter machine with one reversal-bounded counter and a two-way input tape) is decidable [IJTW95]. But if $\mathcal{J}$ is deterministic, then the verification problem is decidable. The proof [DIBKS00a] uses a 2DCM(1,r) (deterministic counter machine with one reversal-bounded counter and a two-way input tape, which is known to have a decidable emptiness problem [IJTW95]) to simulate $\mathcal{J}$ and a padding technique to check the property.

This result requires that $\mathcal{J}$ be deterministic. However, most nontrivial ASTRAL specifications are nondeterministic. The proof in [DIBKS00a] also gives a technique to create a deterministic approximation of $\mathcal{J}$.

### 4.4.3 Allowing parameterized constants

In a history-independent Mini-ASTRAL process $\mathcal{P}$, parameterized constants are not allowed. In fact, if parameterized constants $d_i$ in $D$ are included in a clock constraint like

$$x - y > d_i$$

then the region reachability problem is undecidable [AHV93] for a DTA $\mathcal{A}$. In fact, such an $\mathcal{A}$ is able to simulate a Turing Machine. Clearly, symbolic bounded-testing techniques can be used to debug such an $\mathcal{A}$ up to a pre-assigned execution depth. This technique is also used in [AHV93] and in the HyTech model-checker [HHW97], which is for hybrid systems.

It is theoretically interesting to determine whether a nontrivial lower approximation approach, other than bounding the depth of executions, exists to debug such an $\mathcal{A}$. In the following, we propose a technique that restricts clock behaviors and thus may go to an infinite depth.

The format of the clock-jump machine $\mathcal{J}$ translated from $\mathcal{P}$ under this extension requires that **Init**, **Assump**, **Prop**, and each entry condition of a transition instance are linear relations over $X, V$, and $D$. Note that durations are always 1, as in the original definition of clock-jump machines. Thus, under this extension, a transition with the entry assertion

$$\mathtt{Start(T_1)} - \mathtt{Call(T_1)} > \mathtt{d_1} + \mathtt{Change(v_1)}$$

is allowed. Recall that in a clock-jump machine, a clock $x$, other than *now*, can either *jump* (i.e., $x := now$) or remain the same on any edge. Given a configuration $\alpha$ of $\mathcal{J}$, we use $\alpha(x)$ to indicate the value of clock $x$.

The first approximation is to restrict the number of clock jumps in a path. Let $r$ be a fixed positive integer. A path

$$\alpha_0\alpha_1\cdots\alpha_k$$

is called an $r$*-jump* path if for each $x \in X$, there are at most $r$ many $i$ such that $\alpha_i(x) = \alpha_i(now)$. Write $\alpha \leadsto^{r,\mathcal{J}} \beta$ if $\alpha \leadsto^{\mathcal{J}} \beta$ through an $r$*-jump* path. The $r$-jump reachability problem is to determine whether there is a configuration $\beta$ satisfying $\neg$**Prop** such that $\alpha \leadsto^{r,\mathcal{J}} \beta$ for some initial configuration $\alpha$.

The second approximation is to restrict the delay between two consecutive jumps of the same clock. Given a positive integer $B$. A path

$$\alpha_0\alpha_1 \cdots \alpha_k$$

is called $B$*-bounded* if for each $j < k$, each $x \in X$,

$$|\alpha_j(x) - \alpha_{j+1}(x)| < B.$$

Write $\alpha \leadsto^{B,\mathcal{J}} \beta$ if $\alpha \leadsto^{\mathcal{J}} \beta$ through a $B$-bounded path. The $B$-bounded reachability problem is whether there is a configuration $\beta$ satisfying $\neg$**Prop** such that $\alpha \leadsto^{B,\mathcal{J}} \beta$ for some initial configuration $\alpha$.

The third approximation combines the above approximations. Given positive integers $B$ and $r$. A path

$$\alpha_0\alpha_1 \cdots \alpha_k$$

is called $\langle B, r\rangle$*-finite-crossing* if for each $x \in X$ there are at most $r$ many $j$ such that

$$|\alpha_j(x) - \alpha_{j+1}(x)| \geq B.$$

Write $\alpha \leadsto^{B,r,\mathcal{J}} \beta$ if $\alpha \leadsto^{\mathcal{J}} \beta$ through a $\langle B, r\rangle$-finite-crossing path. The $\langle B, r\rangle$-finite-crossing reachability problem is whether there is a configuration $\beta$ satisfying $\neg$**Prop** such that $\alpha \leadsto^{B,r,\mathcal{J}} \beta$ for some initial configuration $\alpha$.

Notice that, $\leadsto^{r,\mathcal{J}}, \leadsto^{B,\mathcal{J}}$ and $\leadsto^{B,r,\mathcal{J}}$ are all subsets of $\leadsto^{\mathcal{J}}$. Thus, they are lower approximations of $\mathcal{A}$. Therefore, they can be used to debug **Prop**. It is clear that none of the three approximations restrict the depth of executions to be finite. The importance of the three approximations is that:

- The reachability problem under each of them is decidable. The proof is a direct application of our recent result on a class of generalized counter machines with strongly reversal-bounded counters [ISDBK00] (see also [DIBKS00a].).

- The parameterized clock constraints presented here are more general than those in [AHV93]. In that paper, a (parametric) clock constraint is in the form of comparing the difference of two clocks against a parameterized constant. But here, we allow general linear relations on clocks as clock constraints.

- We are not aware of an approximation technique for parameterized timed automata that could extend a symbolic search procedure to an infinite depth. The techniques presented in this subsection probably are the only ones that do not use the bounded-testing technique for parameterized timed automata verification.

## 4.5 Decidability of Verification for Presburger Safety Properties

This sections gives the proof of Theorem 3. The proof is organized as follows. Given a discrete timed automaton, we first show that the binary reachability $\rightsquigarrow^{\mathcal{A}}$ is definable by a Presburger formula. Then, we will show that the Presburger safety analysis problem is decidable, i.e., Theorem 3. Before we proceed to show the results, some further definitions are needed.

### 4.5.1 Preliminaries

A *nondeterministic multicounter machine (NCM)* $M$ is a nondeterministic machine with a finite set of *(control) states* $Q = \{1, 2, \cdots, |Q|\}$, and a finite number of counters $x_1, \cdots, x_k$ with integer counter values. Each counter can add 1, subtract 1, or stay unchanged. Those counter assignments are called *standard assignments*. $M$ can also test whether a counter is equal to, greater than, or less than an integer constant. Those tests are called *standard tests*. It is well-known that counter machines with two counters have undecidable halting problem. Thus, we have to restrict the behaviors of the counters. One such restriction is to limit the number of reversals a counter can make. A counter is $n$*-reversal-bounded* if it changes mode between nondecreasing and nonincreasing at most $n$ times. For instance, the following sequence of a counter values: $0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 3, 2, 1, 1, 1, 1, \cdots$ demonstrates only one counter reversal. A counter is *reversal-bounded* if it is *$n$-reversal-bounded* for some $n$. We note that a reversal-bounded $M$ (i.e., each counter in $M$ is reversal-bounded) does not necessarily limit the number of moves or the number of reachable configurations to be finite.

Let $(j, v_1, \cdots, v_k)$ denote the *configuration* of $M$ when it is in state $j \in Q$ and counter $x_i$ has value $v_i \in \mathbf{Z}$ for $i = 1, 2, \cdots, k$. Each integer counter value $v_i$ can be represented by a unary string $0^{v_i}$ ($1^{v_i}$) when $v_i$ positive (negative). Thus, a configuration $(j, v_1, \cdots, v_k)$ can be represented as a string by concatenating the unary representations of each $j$ as well as $v_1, \cdots, v_k$ with a separator #. For instance, (1,2,-2) can be represented by $0^1\#0^2\#1^2$. Similarly, an integer tuple $(v_1, \cdots, v_k)$ can also be represented by a string. Thus, in this way, a set of configurations and a set of integer tuples can be treated as sets of strings, i.e., a language. It is noticed that a configuration $\alpha$ of a discrete timed automaton $\mathcal{A}$ can be similarly encoded as a string $[\alpha]$.

Note that the above defined $M$ does not have an input tape; in this case it is used as a system specification rather than a language recognizer, in which we are more interested in the behaviors that $M$ generates. When a NCM $M$ is used as a language recognizer, we attach a separate one-way read-only input tape to the machine and assign a state in $Q$ as the final state. $M$ *accepts* an input iff it can reach the final state. When $M$ is reversal-bounded, the emptiness problem, i.e., whether $M$ accepts some input, is known to be decidable,

**Theorem 4** *The emptiness problem for reversal-bounded nondeterministic multicounter machines with a one-way input tape is decidable [I78].*

Actually, Theorem 4 can be strengthened for integer tuples.

**Theorem 5** *A set of $n$-tuples of integers is definable by a Presburger formula iff it (with the tuples represented as strings) can be accepted by a reversal-bounded nondeterministic multicounter machine [I78].*

### 4.5.2 A Binary Reachability Characterization

Let $\mathcal{A}$ be a discrete timed automaton with clocks $x_1, \cdots, x_k$. The binary reachability $\leadsto^{\mathcal{A}}$ can be treated as a language $\{[\alpha]\#[\beta] : \alpha \leadsto^{\mathcal{A}} \beta\}$ where $[\alpha]$ (resp $[\beta]$) is the string encoding of configuration $\alpha$ (resp $[\beta]$). The two encodings are separated by a delimiter "#". The main result in this subsection claims that the binary reachability $\leadsto^{\mathcal{A}}$ can be accepted by a reversal-bounded NCM using standard tests and assignments. $\mathcal{A}$ itself can be regarded as an NCM, when we refer to a clock as a counter. However, tests in $\mathcal{A}$ as clock constraints are not standard tests. Furthermore, $\mathcal{A}$ is not reversal-bounded since clocks can be reset for an unbounded number of times.

The proof of the main result proceeds as follows. We first show that $\leadsto^{\mathcal{A}}$ can be accepted by a reversal-bounded NCM using nonstandard tests and assignments. Then, we show that these nonstandard tests can be made standard. Finally, these nonstandard assignments can be simulated by standard ones. Throughout the two simulations the counters remain reversal-bounded.

First, we show that clocks $x_1, \cdots, x_k$ in $\mathcal{A}$ can be translated into reversal-bounded ones. Let $y_0, y_1, \cdots, y_k$ be another set of clocks such that $x_i = y_0 - y_i$ $(1 \leq i \leq k)$. Let $\mathcal{A}'$ be a discrete timed automaton that is exactly the same as $\mathcal{A}$, except

- $\mathcal{A}'$ has clock $y_0$ that never resets. Intuitively, the *now-clock* $y_0$ denotes current time.
- Each $y_i$ with $1 \leq i \leq k$ denotes the (last) time when a reset of clock $x_i$ happens. Thus, each reset of clock $x_i$ on an edge is replaced by updating $y_i$ to the current time, i.e., $y_i := y_0$. If $x_i$ does not reset on an edge, the value of $y_i$ is unchanged. Also, only if there is no clock reset on an edge, add an assignment $y_0 := y_0 + 1$ to the edge to indicate that the now-clock progresses with one time unit. Only these assignments can change $y_0$.
- the enabling condition on each edge of $\mathcal{A}$ is replaced by substituting $x_i$ with $y_0 - y_i$. Note that the enabling conditions $x_i\#c$ and $x_i - x_j\#c$ become $y_0 - y_i\#c$ and $y_j - y_i\#c$, respectively. Thus, the resulting enabling conditions are Boolean combinations of $y_i - y_j\#c$ with $0 \leq i, j \leq k$ and $c$ being an integer constant.

Counters $y_0, y_1, \cdots, y_k$ in $\mathcal{A}'$ do not reverse. The reason is that assignments that change the counter values are only in the form of: $y_0 := y_0 + 1$ and $y_i := y_0$ for $1 \leq i \leq k$, and there is no way that a counter $y_i$ decreases. For a configuration $\alpha$ of $\mathcal{A}$ and $u \in \mathbf{Z}^+$, write $\alpha^u$ to be a configuration of $\mathcal{A}'$ such that $\alpha^u_{y_0} = u$ ($y_0$'s value is $u$), and for each $1 \leq i \leq k$,

$\alpha^u_{y_i} = u - \alpha_{x_i}$ ($y_i$ is the translation of $x_i$). Also write $max_\alpha$ to be the maximal value of clocks $\alpha_{x_i}$ in $\alpha$ (note that, each $\alpha_{x_i}$ is nonnegative by definition). Thus, $\alpha^{max_\alpha}$ is the configuration $\alpha^u$ of $\mathcal{A}'$ with $y_0$'s value being $max_\alpha$. It follows directly, by induction on the length of a path, that the binary reachability of $\mathcal{A}$ can be characterized by that of $\mathcal{A}'$ as follows.

**Theorem 6** *For any pair of configurations $\alpha$ and $\beta$ of a discrete timed automaton $\mathcal{A}$, the following holds,*

$$\alpha \rightsquigarrow^{\mathcal{A}} \beta \text{ iff there exist } v \in \mathbf{Z}^+ \text{ with } v \geq max_\alpha \text{ such that } \alpha^{max_\alpha} \rightsquigarrow^{\mathcal{A}'} \beta^v.$$

From the above theorem, it suffices for us to investigate the binary reachability of $\mathcal{A}'$. As mentioned above, $\mathcal{A}'$ is an NCM with reversal-bounded counters $y_0, y_1, \cdots, y_k$. However, instead of standard tests, $\mathcal{A}'$ has tests that check an enabling condition by comparing the difference of two counters against an integer constant. Also the assignments include only $y_0 := y_0 + 1$ and $y_i := y_0$ for $1 \leq i \leq k$ in $\mathcal{A}'$, which are not standard assignments. The following theorem says the nonstandard tests can be made standard.

**Theorem 7** *The binary reachability $\rightsquigarrow^{\mathcal{A}'}$ of $\mathcal{A}'$ can be accepted by a reversal-bounded NCM using standard tests and nonstandard assignments that are of the form $y_0 := y_0 + 1$ and $y_i := y_0$ with $1 \leq i \leq k$.*

*Proof.* We construct the reversal-bounded NCM as required. Given a pair of string encodings of configurations $\alpha^{\mathcal{A}'}$ and $\beta^{\mathcal{A}'}$ of $\mathcal{A}'$ on $M$'s one-way input tape, $M$ first copies $\alpha^{\mathcal{A}'}$ into its $k+1$ counters $y_0, y_1, \cdots, y_k$. Thus, $M$'s input head stops at the beginning of $\beta^{\mathcal{A}'}$. $M$ starts simulating $\mathcal{A}'$ as follows. Tests in $\mathcal{A}'$ are Boolean combinations of $y_i - y_j \# c$ for $0 \leq i, j \leq k$. Using only standard tests, $M$ cannot directly compare the difference of two counter values against an integer $c$ by storing $y_i - y_j$ in another counter, since each time this "storing" is done it will cause at least a counter reversal, and we don't have a bound on the number of such tests. In the following, we provide a technique to avoid such nonstandard tests. Assume $m$ is one plus the maximal absolute value of all the integer constants that appear in the tests in $\mathcal{A}'$. Denote the finite set $[m] =_{def} \{-m, \cdots, 0, \cdots, m\}$. $M$ uses its finite control to build a finite table. For each pair of counters $y_i$ and $y_j$ with $0 \leq i, j \leq k$, there is a pair of entries $a_{ij}$ and $b_{ij}$. Each entry can be regarded as finite state control variable with states in $[m]$. Intuitively, $a_{ij}$ is used to record the difference between the values of two counters $y_i$ and $y_j$. $b_{ij}$ is used to record the "future" value of the difference when a clock assignment $y_i := y_0$ occurs in the future. During the computation of $\mathcal{A}'$, when the difference goes beyond $m$ or below $-m$, $a_{ij}$ stays the same as $m$ or $-m$. $M$ uses $a_{ij}\#c$ to do a test $y_i - y_j\#c$. Doing this is always valid, as we will show later. Thus, $M$ only uses standard tests. Below, "ADD 1" means adding one if the result does not exceed $m$, otherwise it keeps the same value. "SUBTRACT 1" means subtracting one if the result is not less than $-m$, otherwise it keeps the same value. In the following, we show how to construct the table. When assignment $y_0 := y_0 + 1$ is being executed by $\mathcal{A}'$, $M$ updates the table as follows, for each $0 \leq i, j \leq k$:

- $a_{ij}$ stays the same if $i > 0$ and $j > 0$. That is, the now-clock's progressing does not affect the difference between two non-now-clocks,
- $a_{ij}$ ADD 1 if $i = 0$ and $j > 0$, noticing that $y_i$ is the now-clock and $y_j$ is a non-now-clock (thus it remains unchanged),
- $a_{ij}$ SUBTRACT 1 if $i > 0$ and $j = 0$, noticing that $y_j$ is the now-clock and $y_i$ is a non-now-clock (thus it remains unchanged),
- $a_{ij}$ is always 0 if $i = 0$ and $j = 0$. The difference between two identical now-clocks is always 0.

After updating all $a_{ij}$, entries $b_{ij}$ are updated as below, for each $0 \leq i, j \leq k$,

- $b_{ij} := a_{0j}$. Thus $b_{ij}$ is the value of $y_i - y_j$ assuming currently there is a jump $y_i := y_0$.

It is noticed that an edge in $\mathcal{A}'$ cannot contain two forms of assignment, i.e., both $y_0 := y_0 + 1$ and $y_i := y_0$. Let $\tau \subseteq \{y_1, \cdots, y_k\}$ denote assignments $y_i := y_0$ for $i \in \tau$ on an edge being executed by $\mathcal{A}'$. $M$ updates the table as follows, for each $0 \leq i, j \leq k$:

- $a_{ij} := 0$ if $i, j \in \tau$, noticing that both $y_i$ and $y_j$ are currently the same value as the now-clock $y_0$,
- $a_{ij} := b_{ij}$ if $i \in \tau$ and $j \notin \tau$, noticing that $y_i$ currently is the same value of the now-clock $y_0$ and the difference $y_i - y_j$ is prestored as $b_{ij}$,

- $a_{ij} := -b_{ji}$ if $i \notin \tau$ and $j \in \tau$, noticing that $y_i - y_j = -(y_j - y_i)$,

- $a_{ij}$ stays the same if $i \notin \tau$ and $j \notin \tau$, since clocks outside $\tau$ are not changed.

After updating all $a_{ij}$, entries $b_{ij}$ are updated as follows, for each $0 \le i, j \le k$:

- $b_{ij} := 0$ if $i, j \in \tau$, noticing that both $y_i$ and $y_j$ are currently the same value as the now-clock $y_0$,

- $b_{ij} := a_{0j}$ if $i \in \tau$ and $j \notin \tau$, noticing that $y_i$ currently is the same value as the now-clock $y_0$,

- $b_{ij} := -a_{0i}$ if $i \notin \tau$ and $j \in \tau$, noticing that $y_j$ currently is set to the now-clock $y_0$,

- $b_{ij}$ stays the same if $i \notin \tau$ and $j \notin \tau$, noticing that $b_{ij}$ represents $y_0 - y_j$ and in fact the two clocks $y_0$ and $y_j$ are unchanged after the transition.

The initial values of $a_{ij}$ and $b_{ij}$ can be constructed directly from $\alpha^{\mathcal{A}'}$ as follows, for each $0 \le i, j \le k$:

- $a_{ij} := \alpha^{\mathcal{A}'}_{y_i} - \alpha^{\mathcal{A}'}_{y_j}$ if $|\alpha^{\mathcal{A}'}_{y_i} - \alpha^{\mathcal{A}'}_{y_j}| \le m$,

- $a_{ij} := m$ if $\alpha^{\mathcal{A}'}_{y_i} - \alpha^{\mathcal{A}'}_{y_j} > m$,

- $a_{ij} := -m$ if $\alpha^{\mathcal{A}'}_{y_i} - \alpha^{\mathcal{A}'}_{y_j} < -m$,

and for each $0 \le i, j \le k$: $b_{ij} := a_{0j}$.

$M$ then simulates $\mathcal{A}'$ exactly except using $a_{ij}\#c$ for a test $y_i - y_j\#c$ in $\mathcal{A}'$, with $-m < c < m$. Then, we claim that doing this by $M$ is valid,

**Claim.** Each time after $M$ updates the table by executing a transition, $y_i - y_j\#c$ iff $a_{ij}\#c$, and $y_0 - y_j\#c$ iff $b_{ij}\#c$, for all $0 \le i, j \le k$ and for each integer $c \in [m-1]$.

**Proof of the Claim**. We prove it by induction. Obviously, the Claim holds for the initial values of all clocks $y_i$ (in configuration $\alpha^{\mathcal{A}'}$) and the corresponding entries $a_{ij}$ and $b_{ij}$, by the choice of $m$. Suppose that $\mathcal{A}'$ is currently at configuration $\gamma$ and the Claim holds. Thus, for all $0 \le i, j \le k$ and for each integer $c \in [m-1]$, $\gamma_{y_i} - \gamma_{y_j}\#c$ iff $a_{ij}\#c$ and $\gamma_{y_0} - \gamma_{y_j}\#c$ iff $b_{ij}\#c$ hold. Therefore, $\gamma$ satisfies an enabling condition in $\mathcal{A}'$ iff the entries $a_{ij}$ satisfy the same enabling condition by replacing $y_i - y_j$ with $a_{ij}$, noticing that $m$ is chosen such that it is greater than the absolute value of any constant in all the enabling conditions in $\mathcal{A}'$. Assume $\gamma$ satisfies the enabling condition on an edge $e$ and $\mathcal{A}'$ will execute $e$ next. Thus, $M$, using the entries $a_{ij}$ to test the enabling condition, will also execute the same edge. We use $\gamma'$ to denote the configuration after executing the edge, and use $a'_{ij}$ and $b'_{ij}$ to denote the table entries after executing the edge. We need to show, for all $0 \leq i, j \leq k$ and for each integer $c \in [m-1]$,

(*) $\quad \gamma'_{y_i} - \gamma'_{y_j} \# c$ iff $a'_{ij} \# c$

and

(**) $\quad \gamma'_{y_0} - \gamma'_{y_j} \# c$ iff $b'_{ij} \# c$

hold. There are two cases to be considered according to the form of the assignment. Suppose the assignment on $e$ is a clock progress $y_0 := y_0 + 1$. After this assignment, $\gamma'_{y_0} = \gamma_{y_0} + 1$ and $\gamma'_{y_i} = \gamma_{y_i}$ for each $1 \leq i \leq k$. On the other hand, according to the updating algorithm above, $a'_{ij}$ are updated for each $0 \leq i, j \leq k$ as follows, depending on the case. There are four subcases:

- If $i > 0$ and $j > 0$, then $\gamma'_{y_i} = \gamma_{y_i}$, $\gamma'_{y_j} = \gamma_{y_j}$, $a'_{ij} = a_{ij}$. The claim (*) holds trivially.
- If $i = 0$ and $j > 0$, then $\gamma'_{y_i} = \gamma_{y_i} + 1$, $\gamma'_{y_j} = \gamma_{y_j}$, $a'_{ij} = a_{ij}$ ADD 1. Since $y_0$ is the only now-clock, all $\gamma_{y_i} - \gamma_{y_j}$, $\gamma'_{y_i} - \gamma'_{y_j}$, $a'_{ij}$ and $a_{ij}$ are nonnegative. It suffices to show for any $c \geq 0, c \in [m-1]$, the claim holds. In fact, $\gamma'_{y_i} - \gamma'_{y_j} \# c$ iff $\gamma_{y_i} - \gamma_{y_j} \# c - 1$ iff $a_{ij} \# c - 1$ iff $a_{ij} + 1 \# c$. Also, $a_{ij} + 1 \# c$ iff $a'_{ij} \# c$, by separating the cases for $a_{ij} = m$ and $a_{ij} < m$, and noticing that $c < m$. Thus, (*) holds, i.e., $\gamma'_{y_i} - \gamma'_{y_j} \# c$ iff $a'_{ij} \# c$.
- If $i > 0$ and $j = 0$, similar as above.
- If $i = 0$ and $j = 0$, the Claim (*) holds trivially.

Noticing that under the assignment $y_0 := y_0 + 1$, $b'_{ij} := a'_{0j}$. Thus, (**) can be shown using (*).

When the assignment is in the form of $y_i := y_0$ for $y_i \in \tau \subseteq \{y_1, \cdots, y_k\}$, (note that in this case, the now-clock does not progress, i.e., $\gamma'_{y_0} = \gamma_{y_0}$) there are four cases to consider in order to show (*) for all $0 \leq i, j \leq k$,

- If $i, j \in \tau$, then $\gamma'_{y_i} = \gamma'_{y_j} = \gamma_{y_0}$, $a'_{ij} = 0$ and therefore $\gamma'_{y_i} - \gamma'_{y_j} = a'_{ij} = 0$. Thus, the Claim (*) trivially holds.

- If $i \in \tau$ and $j \notin \tau$, then, $\gamma'_{y_i} = \gamma_{y_0}$, $\gamma'_{y_j} = \gamma_{y_j}$, $a'_{ij} = b_{ij}$. Thus, for each $c \in [m-1]$, $\gamma'_{y_i} - \gamma'_{y_j} \# c$ iff $\gamma_{y_0} - \gamma_{y_j} \# c$ iff (induction hypothesis) $b_{ij} \# c$ iff $a'_{ij} \# c$. The Claim (*) holds.

- If $i \notin \tau$ and $j \in \tau$, similar as above.

- If $i \notin \tau$ and $j \notin \tau$, then, $\gamma'_{y_i} = \gamma_{y_i}$, $\gamma'_{y_j} = \gamma_{y_j}$, and $a'_{ij} = a_{ij}$. Thus, the Claim (*) holds trivially.

Now we prove Claim (**) under this assignment for $\tau$. Again, there are four cases to consider:

- If $i, j \in \tau$, then $b'_{ij} = 0$, noticing that $\gamma'_{y_j} = \gamma_{y_0}$ and $\gamma'_{y_0} = \gamma_{y_0}$, Claim (**) holds.

- If $i \in \tau$ and $j \notin \tau$, then, $b'_{ij} = a'_{0j}$. Claim (**) holds directly from Claim (*).

- If $i \notin \tau$ and $j \in \tau$, similar as above.

- If $i \notin \tau$ and $j \notin \tau$, then, $b'_{ij} = b_{ij}$. In fact, $\gamma'_{y_0} = \gamma_{y_0}$, $\gamma'_{y_j} = \gamma_{y_j}$. Thus, Claim (**) holds directly from the induction hypothesis.

This ends the proof of the Claim. Thus, it is valid for $M$ to use $a_{ij} \# c$ to do each test $y_i - y_j \# c$. At some point, $M$ guesses that it has reached the configuration $\beta^{\mathcal{A}'}$ by comparing the counter values with $\beta^{\mathcal{A}'}$ through reading the rest of the input tape. $M$ accepts iff such a comparison succeeds. Clearly $M$ accepts $\rightsquigarrow^{\mathcal{A}'}$. ∎

Assignments in $M$ constructed in the above proof, in the form of, $y_0 := y_0 + 1$ and $y_i := y_0$ with $1 \leq i \leq k$, are still not standard. We will now show that these assignments can be made standard, while the machine is still reversal-bounded. Let $M'$ be an NCM that is exactly the same as $M$. $M'$ simulates $M$'s computation from the configuration $\alpha^{\mathcal{A}'}$. Initially, each $y_i := \alpha^{\mathcal{A}'}_{y_i}$ as we indicated in the above proof. However, each time that $M$ executes an assignment $y_0 := y_0 + 1$, $M'$ increases **all** the counters by 1, i.e., $y_i := y_i + 1$ for each $0 \leq i \leq k$. When $M$ executes an assignment $y_i := y_0$, $M'$ does nothing. For each $1 \leq i \leq k$, at some point, either initially or at the moment $y_i := y_0$ is being executed by $M$, $M'$ guesses (only once for each $i$) that $y_i$ has already reached the value given in $\beta^{\mathcal{A}'}$. After such a guess for $i$, an execution of $y_0 := y_0 + 1$ will not cause $y_i := y_i + 1$ as indicated above (i.e., $y_i$ will no longer be incremented). However, after such a guess for $i$, a later execution of $y_i := y_0$ in $M$ will cause $M'$ to abort abnormally (without accepting the input). At some point after **all** $1 \leq i \leq k$ have been guessed, $M'$ guesses that it has reached the configuration $\beta^{\mathcal{A}'}$. Then, $M'$ compares its current configuration with the one on the rest of the input tape $\beta^{\mathcal{A}'}$. $M'$ accepts iff such a comparison succeeds. Clearly, $M'$ uses only assignments $y_0 := y_0 + 1$ and $y_i := y_i + 1$ for $1 \leq i \leq k$. Thus, $M'$ is also reversal-bounded and accepts $\leadsto^{\mathcal{A}'}$. Therefore,

**Theorem 8** *The binary reachability* $\leadsto^{\mathcal{A}'}$ *of* $\mathcal{A}'$ *can be accepted by a reversal-bounded NCM using standard tests and assignments.*

Combining the above theorem with Theorem 6 and noticing that $v$ in Theorem 6 can be guessed, it follows immediately that,

**Theorem 9** *The binary reachability of a discrete timed automaton can be accepted by a reversal-bounded multicounter machine using standard tests and assignments. From Theorem 5, it is Presburger over clocks.*

### 4.5.3 Proof of Theorem 3

Now we are ready to show Theorem 3.

Theorem 3. The Presburger safety analysis problem is decidable for discrete timed automata.

*Proof.* Recall that the Presburger safety analysis problem can be formulated as, given $I$ and $P$ definable by Presburger formulas, deciding whether starting from a configuration in $I$, $\mathcal{A}$ can only reach configurations in $P$. That is, whether the following closed formula (i.e., without free variables)

$$\forall\alpha\forall\beta(\alpha \leadsto^{\mathcal{A}} \beta \wedge \alpha \in I \rightarrow \beta \in P)$$

holds. From Theorem 9, $\leadsto^{\mathcal{A}}$ is Presburger. Noticing that $I$ and $P$ are Presburger and the fact that Presburger formulas are closed under quantification. Thus, the above closed formula is Presburger. Since testing the truth value of a closed Presburger formula is decidable, the Presburger safety analysis problem is thus decidable. ∎

## 4.6 Related Work

Timed automata were originally proposed in [AD94] and quickly became a standard model for real-time systems. The main advantage of the model is that the region reachability problem is decidable [AD94]. The technique, which is adapted from [DIBKS00b], used in this chapter to eliminate clock comparisons allows us to verify a broader range of properties that are not covered by timed temporal logics [AH94, ACD93, HNSY94], such as a safety property represented by a Presburger formula over clocks as shown in Theorem 3. In fact, in [DIBKS00b] we are able to show that even for a stronger model (called a discrete timed pushdown automaton, which is a discrete timed automaton augmented with a pushdown stack) binary reachability can be characterized by a machine with a decidable emptiness problem. Using this characterization, a safety property represented as a Presburger formula can be verified for discrete timed pushdown automata. Recently, Comon et. al. [CJ99, CJ98] showed that the binary reachability of timed automata (with real-valued clocks) is expressible in the additive theory of reals. They show that a timed automaton with real-valued clocks can be flattened into one without nested cycles. Their technique also works for discrete timed automata. Our proof technique does not use the region technique [AD94] nor the flattening technique [CJ99]. In fact, the flattening technique can not be used to show the binary reachability of discrete timed pushdown automata, since by flattening the sequence of stack operations cannot be maintained.

The expressive power of timed automata is still limited, since many real-time systems are simply not finite-state, even when time is ignored. One way of extending the finite state underlying model of a timed automaton is to build timing information into (or even couple a timed automaton with) an infinite state system model that has been well studied in the verification literature. Among others, there are many timed extensions of Petri Nets (such as Timed Petri Nets [M76]), timed pushdown systems [BER95, DIBKS00b], and many versions of timed process algebra (such as TCCS [MT89]).

In addition, there are practical needs to augment timed automata with more complex clock constraints and clock behaviors. But, unfortunately, most of the results are frustrating. For instance, the reachability problem is undecidable, when one allows the addition operator in a clock constraint [AD94], or when one allows parameterized constants in a clock constraint (even if there are only three clocks) [AHV93]. This extended form of timed automata is useful in practice, as shown by the real-world experience of Berard and Fribourg [BF99] in the verification of the ABR conformance protocol. However, the undecidability result makes it impossible to perform automatic verification even for safety properties. To this end, decidable approximation techniques are needed such that an algorithmic procedure exists to analyze the systems under the given approximation. The techniques would provide a way to help a user gain confidence in a specification or to debug a specification. One of the most direct techniques is to bound the number of transitions to a fixed number and calculate the (symbolic representation of the) reachable sets up to that bound [AHV93]. Other applicable approaches include investigating the behaviors of a cycle in the system [CJ98, BW94]. The approximations proposed in Section 4.4.2 and Section 4.4.3 on a class of generalized discrete timed automata restrict clock behaviors but do not necessarily bound the number of transition iterations to be finite.

# Chapter 5

# Automatic Verification of Mini-ASTRAL

In the previous chapter, we have shown that a history-independent Mini-ASTRAL process instance can be automatically verified. Being a nontrivial subset of ASTRAL, Mini-ASTRAL inherits the modularization feature of ASTRAL. That is, partitioning a large system, both conceptually and functionally, into several small modules (ASTRAL processes) and verifying each small module instead of verifying the large system as a whole. Therefore, we can verify the correctness of each module without looking at the behaviors of the other modules. As we stated before, a module must be provided with an interface section that is a first-order formula to abstract its environment. It is not unusual for these formulas to include complex timing requirements on the variables that refer to their past values. By including the history-dependent operator `Past`, the history-independent Mini-ASTRAL becomes the entire Mini-ASTRAL. A natural question is: do we have an effective procedure to automatically verify a Mini-ASTRAL process instance $\mathcal{P}$? The answer is yes, as we will show in this chapter.

This chapter is organized as follows. In Section 5.1, the intuition behind the construction in this chapter is sketched by investigating an example. In Section 5.2, a class of history-dependent formulas called past formulas are defined and a procedure to show that the truth values of closed past formulas can be recursively calculated is presented. In Section 5.3 and 5.4, two classes of past machines are discussed and it is shown that both have decidable safety properties. In Section 5.5, a procedure is given such that, in theory, a Mini-ASTRAL process instance can be translated to a past timed automaton. Thus, it is theoretically possible to automatically verify the instance. A number of technical proofs are shown in Section 5.6. Section 5.7 addresses some related work.

## 5.1 An Example

As we discussed in 2.5, allowing history-dependency (i.e., including the `Past` operator) makes it possible to specify complex real-timed systems in a modularized way. However, the fundamental question of whether after adding this history-dependent operator an automatic verification procedure still exists has never been answered before. Before we give a positive answer to this question, we first illustrate the intuition behind the constructions in the following sections by using the `Gate` process in the railroad specification (see Appendix A) discussed in 2.1 as an example.

We look at an instance of the `Gate` process by considering the specification with one railroad track (i.e., `n_track`=1, and therefore there is only one `Sensor` process instance.) and assigning concrete values to parameterized constants as follows:

`raise_dur`=1,
`up_dur`=1,
`lower_dur`=1,
`down_dur`=1,
`raise_time`=1,
`lower_time`=1,
`response_time`=1,
`wait_time`=3,
`RImax`=5,
`RIImax`=6.

The transition system of the process instance of `Gate` can be represented as the timed

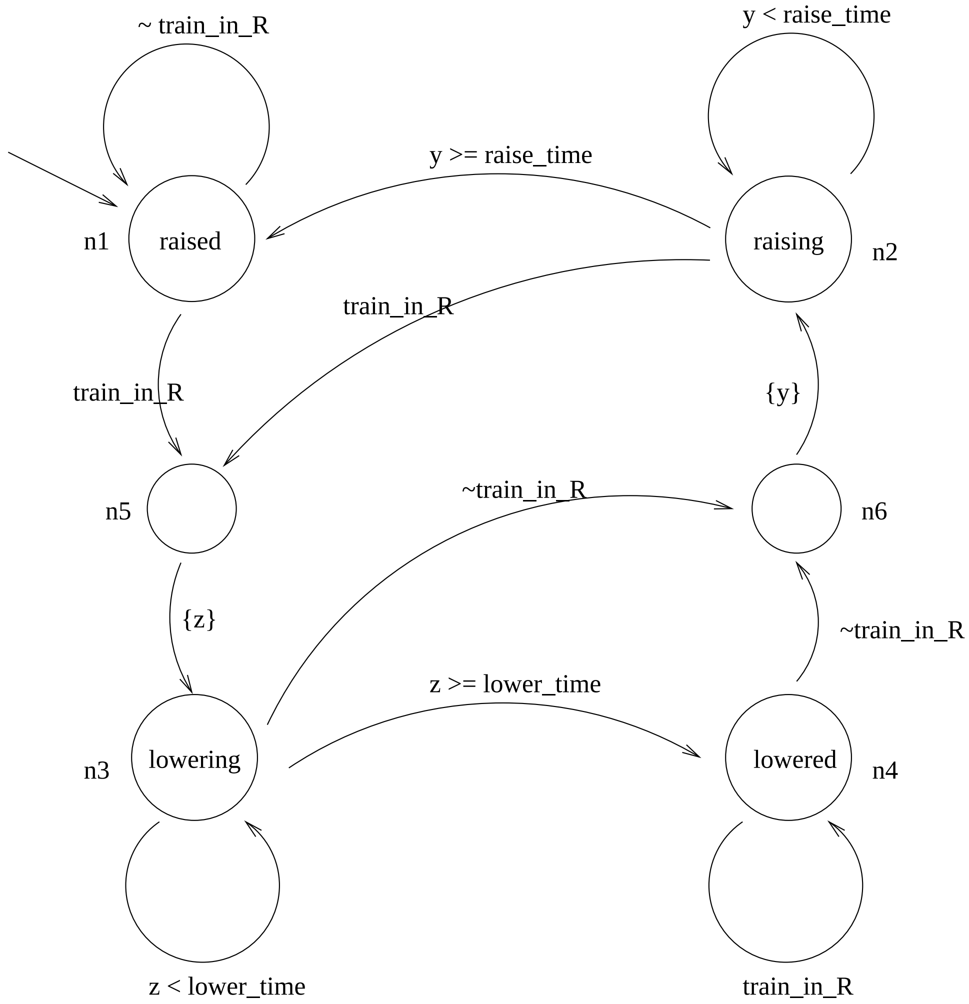


Figure 5.1: The transition system of a `Gate` instance represented as a timed automaton

automaton shown in Figure 5.1. The local variable `position` in `Gate` has four possible values. They are `raised`, `raising`, `lowering` and `lowered`, which are represented by nodes $n_1, n_2, n_3$ and $n_4$ in the figure, respectively. There are two dummy nodes $n_5$ and $n_6$ in the graph, which will be made clear in a moment. The initial node is $n_1$. That is, the initial position of the gate is `raised`.

The transitions `lower, down, raise` and `up` in `Gate` are represented in the figure as follows. The transition `lower`,

```
TRANSITION lower
ENTRY           [ TIME : lower_dur ]
          ~ ( position = lowering
           |  position = lowered )
        & EXISTS s: sensor_id
                   ( s.train_in_R )
EXIT
          position = lowering,
```

corresponds to the edges $\langle n_1, n_5 \rangle$ and $\langle n_5, n_3 \rangle$, or the edges $\langle n_2, n_5 \rangle$ and $\langle n_5, n_3 \rangle$. The clock $z$ is used to indicate

$$\texttt{now} - \texttt{End(lower)}$$

used in transition `down`. Whenever the transition `lower` completes, $z$ resets to 0. Thus, a dummy node $n_5$ is introduced such that $z$ is reset on the edge $\langle n_5, n_3 \rangle$ to indicate the end of the transition `lower`. On an edge without clock resets (such as $\langle n_1, n_5 \rangle$ and $\langle n_2, n_5 \rangle$), *now* progresses by one time unit. Thus, the two edges $\langle n_1, n_5 \rangle$ and $\langle n_2, n_5 \rangle$ indicate the duration `lower_dur` of the transition `lower` (recall the parameterized constant `lower_dur` was set to be 1.).

Similar to `lower`, transition `raise`,

```
TRANSITION raise
  ENTRY         [ TIME : raise_dur ]
          ~ ( position = raising
           |  position = raised )
```

```
            & FORALL s: sensor_id
                      ( ~s.train_in_R )
     EXIT
              position = raising
```

corresponds to the edges $\langle n_3, n_6\rangle$ and $\langle n_6, n_2\rangle$, or the edges $\langle n_4, n_6\rangle$ and $\langle n_6, n_2\rangle$. The clock $y$ is used to indicate

$$\texttt{now} - \texttt{End}(\texttt{raise})$$

used in transition `up`. A dummy node $n_6$ is introduced such that $y$ is reset on the edge $\langle n_6, n_2\rangle$ to indicate the end of the transition `raise`.

The other two transitions `down` and `up` corresponds to the edges $\langle n_3, n_4\rangle$ and $\langle n_2, n_1\rangle$, respectively. Recall idle transitions are necessary to be added indicating the behavior of the process when no transition is enabled and is executing (cf. Section 3.2.1). They are represented by self-loops on nodes $n_1, n_2, n_3$ and $n_4$ in the figure.

Besides the local variable `position`, `Gate` has an imported variable `train_in_R` to indicate an arrival of a train (notice that we have only one `Sensor` process instance. Thus, the imported variable `s.train_in_R` in the specification is simply written as `train_in_R`). `Gate` has no control over the imported variable. That is, `train_in_R` can be either true or false at any given time. The following sequence of tuples of variables $n$ (ranging over nodes), `position, train_in_R, now`, $z$ and $y$ is an execution of the automaton in Figure 5.1:

$\langle n = n_1, \texttt{position} = \texttt{raised}, \texttt{train\_in\_R} = \texttt{false}, now = 0, z = 0, y = 0\rangle$ (this is the initial configuration) $\overset{\langle n_1,n_1\rangle}{\longrightarrow}$

$\langle n = n_1, \texttt{position} = \texttt{raised}, \texttt{train\_in\_R} = \texttt{false}, now = 1, z = 1, y = 1\rangle \overset{\langle n_1,n_1\rangle}{\longrightarrow}$

$\langle n = n_1, \texttt{position} = \texttt{raised}, \texttt{train\_in\_R} = \texttt{true}, now = 2, z = 2, y = 2\rangle \overset{\langle n_1,n_5\rangle}{\longrightarrow}$

$\langle n = n_5, \texttt{position} = \texttt{raised}, \texttt{train\_in\_R} = \texttt{false}, now = 3, z = 3, y = 3\rangle \overset{\langle n_5,n_3\rangle}{\longrightarrow}$ (after this edge is fired, $z$ resets and, since clock resets take no time, `train_in_R` does not change.)

$\langle n = n_3, \texttt{position} = \texttt{lowering}, \texttt{train\_in\_R} = \texttt{false}, now = 3, z = 0, y = 3\rangle$.

The automaton walks along the path $\langle n_1, n_1\rangle, \langle n_1, n_1\rangle, \langle n_1, n_5\rangle$, and $\langle n_5, n_3\rangle$ while the imported variable `train_in_R` demonstrates the value $false$ at $now = 0, 1,$ and $3$, and value $true$ at $now = 2$. But this sequence is not an intended execution of the `Gate` process. The reason is as follows. `train_in_R` has value *true* at $now = 2$. At $now = 3$, it changes to $false$. This change is too fast, since the gate `position` at $now = 3$ is `lowering` when the change happens. At $now = 3$, the train had already crossed the intersection. This is bad, since the gate was not in the fully lowered position `lowered`. Thus, the imported variable clause is needed to place extra requirements on the behaviors of the imported variable. The requirement essentially states that once the sensor reports a train's arrival, it will keep reporting a train at least as long as it takes the fastest train to exit the region. By substituting the parameterized constants and noticing that there is only one sensor in the system, the imported variable clause (cf. Appendix A) can be written as

$$
\begin{array}{l}
\quad now \geq 1 \wedge \\
\quad \texttt{past}(\texttt{train\_in\_R}, now - 1) = true \wedge \\
\quad \texttt{train\_in\_R} = false \\
\rightarrow \\
\quad now \geq 5 \wedge \\
\quad \forall t (t \geq now - 5 \wedge t < now \ \rightarrow \ \texttt{past}(\texttt{train\_in\_R}, t) = true).
\end{array}
$$

We use $f$ to denote this clause. Figure 5.1 can be modified by adding $f$ to the enabling condition of each edge. The result is shown in Figure 5.2.

Now, let us check whether the automaton in Figure 5.2 rules out the execution sequence shown above. We extend a configuration for the automaton in Figure 5.1 by including the value for $f$. Initially at node $n_1$, $f$ is trivially satisfied. It is also satisfied following the path $\langle n_1, n_1 \rangle, \langle n_1, n_1 \rangle, \langle n_1, n_5 \rangle$. At $n_5$, the history values of `train_in_R` are $false$ (at $now = 0, 1$) and *true* (at $now = 2$). At $now = 3$, `train_in_R` changes to $false$ again. It is easy to check that $f$ is not satisfied at $now = 3$. Therefore, the automaton in Figure 5.2 can not walk from $n_5$ to $n_3$.[1]

The automaton shown in Figure 5.2 is different from a timed automaton, since the enabling conditions contain history-dependent constructs, such as `past(train_in_R, t)` in

[1] With this modeling, the configuration $\langle n = n_5, \texttt{position} = \texttt{raised}, \texttt{train\_in\_R} = \texttt{false}, now = 3, z = 3, y = 3 \rangle$ is reachable even though it violates the imported variable clause $f$. However, it is a deadlock configuration; once it is reached, the automaton can not go to any other configuration. A slight change to the automaton will make sure that the configuration is not reachable. But this is irrelevant to the following discussion.

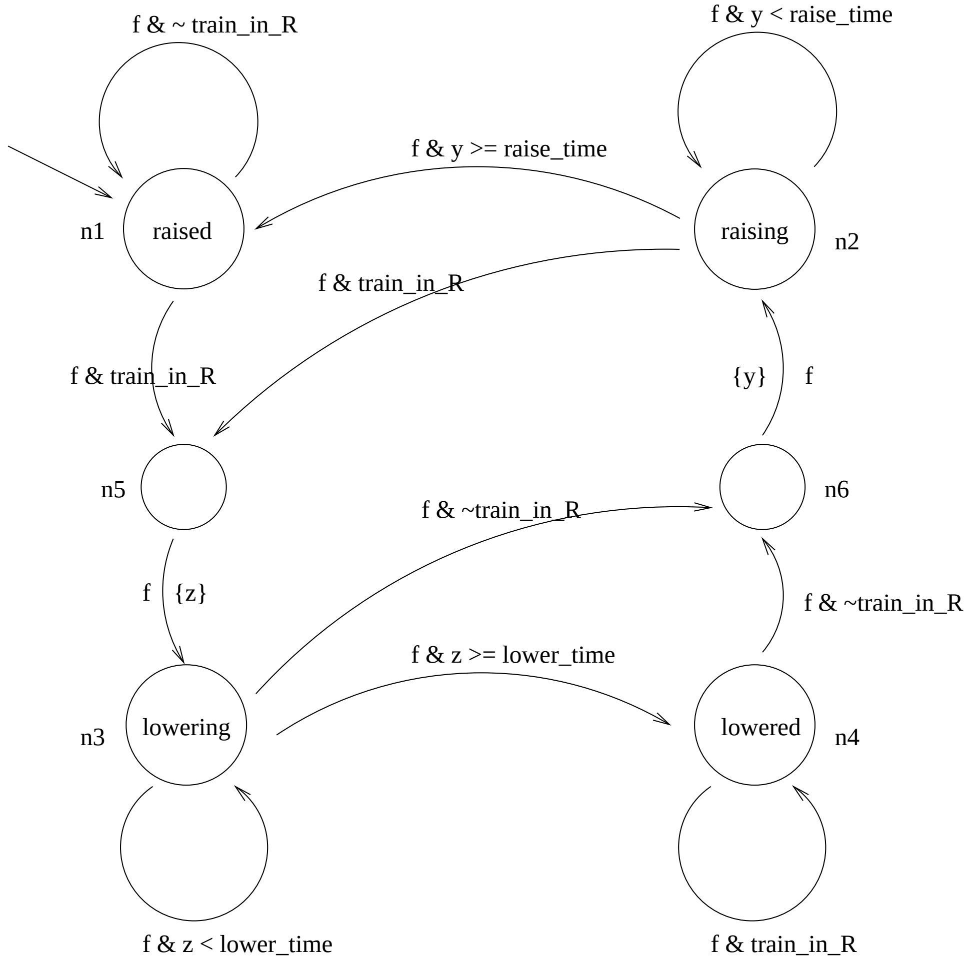


Figure 5.2: The transition system in Figure 5.1 under the imported variable clause $f$ of `Gate`

$f$. Executing the automaton involves looking back the history of variables that appear in the history-dependent constructs. A history is essentially unbounded, and obviously finite state variables are not enough to represent it.

This chapter shows a procedure to eliminate $f$ in Figure 5.2 when it is in a Mini-ASTRAL formula (called a past formula in the next section). Now, we take $f$ as an example to show the ideas in the procedure.

$f$ can be considered simply as a Boolean variable. But when the automaton in Figure 5.2 walks from one node to another (say, from $n_1$ to $n_5$) with $now$ progressing by one time unit (i.e., at $n_5$ the time is $now + 1$), $f$ may change its truth value. The imported variable `train_in_R` under the history-dependent constructs in $f$ has value either $true$ or $false$ at $n_5$. We use $f^+$ to denote the truth value of $f$ at $n_5$. There are two cases to consider.

Case 1. If the value of `train_in_R` is $true$ at $n_5$, $f^+$ can be written as (by replacing $now$ with $now + 1$, and understanding $\texttt{past}(\texttt{train\_in\_R}, now + 1)$ to be the current value ($true$) of `train_in_R`),

$$
\begin{array}{l}
\quad now + 1 \geq 1 \wedge \\
\quad \texttt{past}(\texttt{train\_in\_R}, now + 1 - 1) = true \wedge \\
\quad true = false \\
\rightarrow \\
\quad now + 1 \geq 5 \wedge \\
\quad \forall t(t \geq now + 1 - 5 \wedge t < now + 1 \ \rightarrow\ \texttt{past}(\texttt{train\_in\_R}, t) = true).
\end{array}
$$

In this case, $f^+$ is $true$ by noticing the subformula $true = false$.

Case 2. If the value of `train_in_R` is $false$, $f^+$ can be written as

$$
\begin{array}{l}
\quad now + 1 \geq 1 \wedge \\
\quad \texttt{past}(\texttt{train\_in\_R}, now + 1 - 1) = true \wedge \\
\quad false = false \\
\rightarrow \\
\quad now + 1 \geq 5 \wedge \\
\quad \forall t(t \geq now + 1 - 5 \wedge t < now + 1 \ \rightarrow\ \texttt{past}(\texttt{train\_in\_R}, t) = true).
\end{array}
$$

We can simplify $f^+$ as (note that $now \geq 0$ is always true.)

$$\texttt{past}(\texttt{train\_in\_R}, now) = true$$

$\rightarrow$

$now \geq 4 \wedge$

$\forall t(t \geq now - 4 \wedge t < now + 1 \ \rightarrow$ `past(train_in_R,` $t) = true)$.

The quantified subformula can be expanded, by splitting $t < now + 1$ into $t < now$ and $t = now$, into

$\forall t(t \geq now - 4 \wedge t < now \ \rightarrow$ `past(train_in_R,` $t) = true)$

conjuncted with

`past(train_in_R,` $now) = true$.

By putting them together, $f^+$ is

`past(train_in_R,` $now) = true$

$\rightarrow$

$now \geq 4 \wedge$

`past(train_in_R,` $now) = true) \wedge$

$\forall t(t \geq now - 4 \wedge t < now \ \rightarrow$ `past(train_in_R,` $t) = true)$.

That is, $f^+$ is

`past(train_in_R,` $now) = true$

$\rightarrow$

$now \geq 4 \wedge$

$\forall t(t \geq now - 4 \wedge t < now \ \rightarrow$ `past(train_in_R,` $t) = true)$.

Denote this formula as $g_1$.

Combining the two cases, $f^+$ can be calculated as

$$\texttt{train\_in\_R} = true \vee (\texttt{train\_in\_R} = false \wedge g_1)$$

where `train_in_R` is the value at $now+1$. That is, the "next" value $f^+$ of $f$ can be calculated by using the next value of `train_in_R` and the old value of $g_1$.

But, how can we calculate $g_1^+$? Again, we try to find a way to calculate $g_1^+$ without using the entire history. Similar to $f^+$, there are two cases to consider in order to calculate $g_1^+$.

Case 1. If the value of `train_in_R` is $true$ at $n_5$, $g_1^+$ can be written as

`past(train_in_R,` $now + 1) = true$

$\rightarrow$

$now + 1 \geq 4 \wedge$

$\forall t(t \geq now + 1 - 4 \wedge t < now + 1 \ \rightarrow \ \texttt{past}(\texttt{train\_in\_R}, t) = true).$

Notice that $\texttt{past}(\texttt{train\_in\_R}, now + 1)$ is the value of $\texttt{train\_in\_R}$ at $now + 1$ (which is *true* as given), $g_1^+$ can be simplified as (also expanding the quantified subformula similarly),

$now \geq 3 \wedge$

$\texttt{past}(\texttt{train\_in\_R}, now) = true \wedge$

$\forall t(t \geq now - 3 \wedge t < now \ \rightarrow \ \texttt{past}(\texttt{train\_in\_R}, t) = true).$

Denote this formula as $g_2$.

Case 2. If the value of $\texttt{train\_in\_R}$ is $false$, $g_1^+$ is *true* by noticing that

$$\texttt{past}(\texttt{train\_in\_R}, now + 1) = false.$$

Combine the two cases, $g_1^+$ is

$$(\texttt{train\_in\_R} = true \wedge g_2) \vee \texttt{train\_in\_R} = false.$$

Then, how about $g_2^+$? We can continue the process similarly:

Case 1. If the value of $\texttt{train\_in\_R}$ is *true* at $n_5$, $g_2^+$ can be written and simplified as

$now \geq 2 \wedge$

$\texttt{past}(\texttt{train\_in\_R}, now) = true \wedge$

$\forall t(t \geq now - 2 \wedge t < now \ \rightarrow \ \texttt{past}(\texttt{train\_in\_R}, t) = true).$

Denote this formula as $g_3$.

Case 2. If the value of $\texttt{train\_in\_R}$ is $false$ at $n_5$, $g_2^+$ can be written and simplified as $false$.

Combine the two cases, $g_2^+$ is

$$(\texttt{train\_in\_R} = true \wedge g_3).$$

Continuing this process, we have $g_3^+$ as

$$(\texttt{train\_in\_R} = true \wedge g_4)$$

where $g_4$ is the formula

$now \geq 1 \wedge$

$\texttt{past}(\texttt{train\_in\_R}, \mathit{now}) = \mathit{true} \wedge$

$\forall t(t \geq \mathit{now} - 1 \wedge t < \mathit{now} \ \rightarrow \ \texttt{past}(\texttt{train\_in\_R}, t) = \mathit{true})$.

Again, $g_4^+$ is

$$(\texttt{train\_in\_R} = \mathit{true} \wedge g_5)$$

where $g_5$ is the formula

$\mathit{now} \geq 0 \wedge$

$\texttt{past}(\texttt{train\_in\_R}, \mathit{now}) = \mathit{true} \wedge$

$\forall t(t \geq \mathit{now} \wedge t < \mathit{now} \ \rightarrow \ \texttt{past}(\texttt{train\_in\_R}, t) = \mathit{true})$.

$g_5$ can be further simplified as $\texttt{past}(\texttt{train\_in\_R}, \mathit{now}) = \mathit{true}$.

Finally, $g_5^+$ can be written as

$$\texttt{train\_in\_R} = \mathit{true}.$$

Now, let's summarize. Up to now, we have the following equations:

$f^+ = \texttt{train\_in\_R} \vee (\neg\texttt{train\_in\_R} \wedge g_1)$,

$g_1^+ = (\texttt{train\_in\_R} \wedge g_2) \vee \neg\texttt{train\_in\_R}$,

$g_2^+ = \texttt{train\_in\_R} \wedge g_3$,

$g_3^+ = \texttt{train\_in\_R} \wedge g_4$,

$g_4^+ = \texttt{train\_in\_R} \wedge g_5$,

$g_5^+ = \texttt{train\_in\_R}$.

In the equations, train_in_R is the value at $\mathit{now}+1$. Thus, the truth value of $f$ at $\mathit{now}+1$ can be calculated by using the old values of a number of Boolean variables $g_1, \cdots, g_5$ at time $\mathit{now}$, together with the value of train_in_R at $\mathit{now} + 1$. Again the next value of these Boolean variables at $\mathit{now} + 1$ can be calculated by using their old values together with train_in_R at $\mathit{now} + 1$. By looking at the formulas of $f, g_1, \cdots, g_5$, their initial values can be easily calculated. The automaton in Figure 5.2 can thus be modified into one without past formulas $f$, by building the finite state transitions using Boolean variables as shown in the above equations. Therefore, the past formula is eliminated from the automaton. The resulting automaton is simply a timed automaton.

The equations demonstrate a recursive procedure to calculate a past formula's truth value. A natural question would be: for any past formula, do we always have a recursive calculation procedure like above? We will answer this question positively in the following sections.

## 5.2 Past Formulas

Let $A$ be a finite set of finite state variables, for instance, a bounded range of integers. We use $a, b \cdots$ to denote them. In the following we assume they are Boolean variables. Later we will show a way to extend the procedure to a bounded range of integers. All clocks are discrete as before. Let *now* be the clock representing the current time. Let $X$ be a finite set of integer-valued variables. *Past-formulas* are defined as

$$f ::= a(y) \mid y < n \mid y < z + n \mid \forall_x f[0, now] \mid f \vee f \mid \neg f$$

where $a \in A$, $y$ and $z$ are in $X \cup \{now\}$, $x \in X$, and $n$ is an integer. Intuitively, $a(y)$ is the variable $a$'s value at time $y$, i.e., $\texttt{Past}(a, y)$. *Quantification* $\forall_x f[0, now]$, with $x \neq now$ (i.e., *now* can not be quantified), means, for all $x$ from 0 to *now*, $f$ holds. An appearance of $x$ in $\forall_x f[0, now]$ is called *bounded*. We assume any $x$ is bounded by at most one $\forall_x$. $x$ is *free* in $f$ if $x$ is not bounded in $f$. $f$ is *closed* if *now* is the only free variable. Past-formulas are interpreted on a history of Boolean variables. A history consists of a sequence of boolean values for each variable $a \in A$. The length of all the sequences is $\mathbf{n}$ that is the value of *now*. Formally, a *history* $H$ is a pair

$$\langle \{\|a\|\}_{a \in A}, \mathbf{n} \rangle$$

where $\mathbf{n} \in \mathbf{Z}^+$ is a nonnegative integer representing the value of *now*, and for each $a \in A$, the mapping

$$\|a\| : 0..\mathbf{n} \to \{0, 1\}$$

gives the Boolean value of $a$ at each time point from 0 to $\mathbf{n}$. Let

$$\mathcal{B} : X \to \mathbf{Z}^+$$

be a *valuation* for variables in $X$. Thus $\mathcal{B}(x) \in \mathbf{Z}^+$ denotes the value of $x \in X$ under this valuation $\mathcal{B}$. We use $\mathcal{B}(n/x)$ to denote substituting $x$'s value in the valuation by a non-negative integer $n$. Given a history $H$ and a valuation $\mathcal{B}$, the interpretations of past-formulas are as follows, for each $y, z \in X \cup \{now\}$ and each $x \in X$,

$\|x\|_{H,\mathcal{B}} = \mathcal{B}(x)$,

$\|now\|_{H,\mathcal{B}} = \mathbf{n}$,

$\|a(y)\|_{H,\mathcal{B}} \iff \|a\|(\|y\|_{H,\mathcal{B}})$,

$\|y < n\|_{H,\mathcal{B}} \iff \|y\|_{H,\mathcal{B}} < n$,

$\|y < z + n\|_{H,\mathcal{B}} \iff \|y\|_{H,\mathcal{B}} < \|z\|_{H,\mathcal{B}} + n$,

$\|\forall_x f[0, now]\|_{H,\mathcal{B}} \iff$ for all $k$ with $0 \leq k \leq \mathbf{n}$, $\|f\|_{H,\mathcal{B}(k/x)}$,

$\|f_1 \vee f_2\|_{H,\mathcal{B}} \iff \|f_1\|_{H,\mathcal{B}}$ or $\|f_2\|_{H,\mathcal{B}}$,

$\|\neg f\|_{H,\mathcal{B}} \iff$ not $\|f\|_{H,\mathcal{B}}$.

Two past-formulas $f_1$ and $f_2$ are equivalent, i.e.,

$$f_1 \sim f_2$$

if

$$\|f_1\|_{H,\mathcal{B}} \text{ iff } \|f_2\|_{H,\mathcal{B}}$$

for all $H$ and $\mathcal{B}$. When $f$ is a closed formula, we write $\|f\|_H$ instead of, for all $\mathcal{B}$, $\|f\|_{H,\mathcal{B}}$. We use $\exists_x$ to denote $\neg\forall_x\neg$.

A history $H$ gives values for each Boolean variable $a \in A$ at each past time from 0 to $\mathbf{n}$. Thus, the history can be regarded as a sequence of snapshots $S_0, \cdots, S_{\mathbf{n}}$ such that each snapshot gives a value for each $a \in A$. Now with $now$ progressing from $\mathbf{n}$ to $\mathbf{n}+1$, history $H$ is updated to a new history $H'$ by adding a new snapshot $S_{\mathbf{n}+1}$ to history $H$. This newly added snapshot represents the new values of $a \in A$ at the new current time $\mathbf{n}+1$. A closed past formula may have different truth values when interpreted under $H$ and under $H'$. Is there any way to calculate the truth value of the formula under $H'$ by using the new snapshot $S_{\mathbf{n}+1}$ and the truth value of the formula under $H$? If this can be done, the truth value of the formula can be updated along with the history's update from $\mathbf{n}$ to $\mathbf{n}+1$, without looking back at the old snapshots $S_0, \cdots, S_{\mathbf{n}}$. The rest of this section is devoted to showing that this can be done. That is, the truth value of each closed past formula can be recursively computed. We start by showing that $n$-level quantified past formulas have at most finite many equivalence classes with respect to the equivalence relation $\sim$ on past formulas.

Given a past-formula $f$, denote $\Gamma_f$ to be the set of all variables (including bounded variables, free variables and *now*) appearing in $f$. Since bounded variables in a past formula can be properly renamed without affecting the semantics, we can assume that $f$ uses exactly $n_f$ bounded variables, where $n_f$ is the levels of quantification in $f$ – that is the maximal depth of quantification of each appearance of $x \in \Gamma_f$. Without loss of generality, assume

$$\Gamma_f = \{x_1, \cdots, x_{|\Gamma_f|}\}$$

with the first $n_f$ variables,

$$B^{n_f} = \{x_1, \cdots, x_{n_f}\},$$

being the bounded variables, and

$$U^{n_f} = \{x_{n_f+1}, \cdots, x_{|\Gamma_f|}\}$$

being the free variables, with the last one, $x_{|\Gamma_f|}$, being *now*.

Now we fix any past formula $f$. $f$ may contain atomic formulas like $x < y + k$ or $x < k$ with $x, y \in \Gamma_f$. Let $m$ be the maximal absolute value of all such integers $k$. Denote

$$[m] = \{-m, \cdots, 0, \cdots, m\}.$$

Denote $(\Sigma \cup T)^n$, for some $n \leq n_f$, to be the set of all past-formulas that can be constructed from

- atomic formulas for clock constraints

$$T = \{x < y + k : k \in [m], x, y \in \Gamma_f\} \cup \{x < k : k \in [m], x \in \Gamma_f\},$$

- atomic formulas for past values

$$\Sigma = \{a(x) : x \in \Gamma_f, a \in A\},$$

using exactly $n$ levels of quantification ($\forall$), and each formula in $(\Sigma \cup T)^n$ has bounded variables

$$B^n = \{x_1, \cdots, x_n\},$$

and has free variables

$$U^n = \{x_{n+1}, \cdots, x_{|\Gamma_f|}\}.$$

By definition, $f \in (\Sigma \cup T)^{n_f}$. In particular, when $n = 0$, $(\Sigma \cup T)^0$ means all Boolean combinations of the atomic formulas in $\Sigma$ and $T$.

It is noted that $\sim$ is an equivalence relation on $(\Sigma \cup T)^n$. For two formulas $g$ and $h$, we also say that formula $g$ *can be written as* formula $h$ if $g \sim h$. Given $P, Q \subseteq (\Sigma \cup T)^n$, two subsets of formulas in $(\Sigma \cup T)^n$, and a formula $g \in (\Sigma \cup T)^n$, we write $g \overset{\in}{\sim} P$ if there is a formula $h \in P$ such that $g$ can be written as $h$; i.e., $g \sim h$. We write $P \overset{\subseteq}{\sim} Q$ if every formula in $P$ can be written as a formula in $Q$; i.e., for all $g \in P$, $g \overset{\in}{\sim} Q$. Notice that $\overset{\subseteq}{\sim}$ is transitive. Using induction on $n$, and noticing the fact that there are at most finite different Boolean functions for a given arity, we can easily show,

**Lemma 1** *$(\Sigma \cup T)^n / \sim$ is finite for a given $n$ and $f$. Furthermore, for any $P \subseteq (\Sigma \cup T)^n$, $P/\sim$ is also finite.*

For a formula $h \in (\Sigma \cup T)^n$, we write $h^+$ to be the result of replacing each appearance of *now* with *now* $+ 1$ in $h$. For instance, assume that $h$ is a past formula

$$\forall_x (a(now) \vee b(x))[0, now].$$

Then $h^+$ is

$$\forall_x (a(now+1) \vee b(x))[0, now+1].$$

We use $(\Sigma \cup T)^{n+}$ to denote the set of all $h^+$ with $h$ in $(\Sigma \cup T)^n$. We also use $\Sigma^+$ and $T^+$ to denote the results of substituting each appearance of *now* with *now* $+ 1$ in each atomic formulas in $\Sigma$ and $T$, respectively. However, $h^+$, such as the example shown above, is not strictly in the form of past formulas. The reason is that $h^+$ may contain subformulas like $a(now + 1)$ and quantification over $[0, now + 1]$ instead of $[0, now]$. We can similarly define the semantics for formulas $h^+$ in $(\Sigma \cup T)^{n+}$ by interpreting $h^+$ over a "history"

$$\langle \{\|a\|\}_{a \in A}, \mathbf{n} + 1 \rangle$$

where $\mathbf{n} \in \mathbf{Z}^+$ is the value for *now* (thus, *now* $+ 1$ has value $\mathbf{n} + 1$), and for each $a \in A$, the mapping

$$\|a\| : 0..\mathbf{n} + 1 \rightarrow \{0, 1\}$$

giving the Boolean value of $a$ at each time point from 0 to $\mathbf{n}+1$ (Recall in the semantics definition of a past formula, the domain of $\|a\|$ is $0..\mathbf{n}$.). Thus, the "history" here extends a history for a past formula by including the next values for each $a \in A$. Interpretations of $h^+$ are completely analogous to those for past formulas and are omitted here. Just like the same notations used for formulas in $(\Sigma \cup T)^n$, for formulas in $(\Sigma \cup T)^{n+}$, we also use notations like $\sim$, $\stackrel{\in}{\sim}$ and $\stackrel{\subseteq}{\sim}$ that can be analogously defined.

Note that each bounded variable in $B^n$ has the range $[0, now]$. Define

$$[U^n] = \wedge_{x \in U^n}(0 \leq x \leq now).$$

Thus, $[U^n]$ restricts each free variable also to the range $[0, now]$. For a formula in $(\Sigma \cup T)^n$, we would expect that $h^+$ (by replacing each appearance of $now$ with $now+1$) is in $(\Sigma^+ \cup T^+)^n$. But when free variables in $h^+$ are restricted within the range from 0 to $now$ (i.e., $[U^n]$ holds.), $h^+$ is equivalent to a formula in $(\Sigma^+ \cup T)^n$ as shown below.

**Lemma 2** *For each $h \in (\Sigma \cup T)^n$, $[U^n] \wedge h^+$ is equivalent ($\sim$) to a formula in $(\Sigma^+ \cup T)^n$. That is, $[U^n] \wedge h^+ \stackrel{\in}{\sim} (\Sigma^+ \cup T)^n$.*

*Proof.* See Section 5.6.1. ∎

A special case is when $f$ is a closed formula. If $n$ is the levels of quantification in $f$, then the $(\Sigma \cup T)^n$ are all closed formulas with the same level of quantification as $f$. Thus, in this case, $U^n = \{now\}$. Notice that $[U^n]$ is always true and $f \in (\Sigma \cup T)^n$; therefore, from Lemma 2, we have

**Lemma 3** *If $f$ is a closed formula and $n$ is the levels of quantification in $f$, then for each $h \in (\Sigma \cup T)^n$, $h^+$ is equivalent to a formula in $(\Sigma^+ \cup T)^n$; i.e., $(\Sigma \cup T)^{n+} \stackrel{\subseteq}{\sim} (\Sigma^+ \cup T)^n$. In particular, since $f \in (\Sigma \cup T)^n$, we have $f^+$ is equivalent to a formula in $(\Sigma^+ \cup T)^n$; i.e., $f^+ \stackrel{\in}{\sim} (\Sigma^+ \cup T)^n$.*

From now on, we only consider $f$ as being closed and $n$ as the levels of quantification in $f$. Atomic formulas in $\Sigma$ can be separated into two parts:

$$\Sigma_1 = \{a(now) : a \in A\}$$

and

$$\Sigma_2 = \{a(x) : a \in A, x \in \Gamma_f, x \neq \mathit{now}\}.$$

Notice that $\Sigma_2^+ = \Sigma_2$, since *now* does not appear in any element of $\Sigma_2$. Thus,

$$(\Sigma^+ \cup T)^n = (\Sigma_1^+ \cup \Sigma_2 \cup T)^n.$$

Is there any way to move $\Sigma_1^+$ out of the parenthesis at the right hand side? For instance, what if we can show that

$$(\Sigma_1^+ \cup \Sigma_2 \cup T)^n \overset{\subseteq}{\sim} (\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0 \ ?$$

That is, each formula in $(\Sigma_1^+ \cup \Sigma_2 \cup T)^n$ is equivalent to a formula in $(\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0$. In other word, any formula in $(\Sigma_1^+ \cup \Sigma_2 \cup T)^n$ (and hence in $(\Sigma^+ \cup T)^n$ from the above equation) can be expressed as a Boolean combination of formulas in $\Sigma_1^+$ and formulas in $(\Sigma_2 \cup T)^n$. Recall that, from Lemma 3, $f^+$ is equivalent to a formula in $(\Sigma^+ \cup T)^n$. Therefore, if the above result can be shown, we can conclude that $f^+$ is equivalent to a formula in $(\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0$. That is, the "next" value $f^+$ of $f$ can be represented as a Boolean combination of

- formulas in $\Sigma_1^+$ – they are the "next" value of each $a \in A$, and
- formulas in $(\Sigma_2 \cup T)^n$ – they are of current values.

The following technique will show that $\Sigma_1^+$ can be moved outside.

Each element $a(\mathit{now}+1) \in \Sigma_1^+$, $a \in A$, has a truth value of either 0 or 1. There are $2^{|A|}$ many choices. Let $\tau$ be any one such choice for each $a(\mathit{now}+1)$. Given a $g \in (\Sigma_1^+ \cup \Sigma_2 \cup T)^n$, we can replace each appearance of $a(\mathit{now}+1)$ in $\Sigma_1^+$ by the corresponding value of $a(\mathit{now}+1)$ under the choice $\tau$. Denote the result as $g_\tau$. Notice that $g_\tau \in (\Sigma_2 \cup T)^n$, since all atomic formulas in $\Sigma_1^+$ are replaced. Also denote $\bar{\tau}$ as the conjunction of $a(\mathit{now}+1) = \tau(a)$ for $a \in A$ where $\tau(a)$ is the truth value of $a(\mathit{now}+1)$ under the choice $\tau$. Then, each $g$ in $(\Sigma^+ \cup T)^n$ can be written in the form:

$$\bigvee_{\tau} (\bar{\tau} \wedge g_\tau).$$

Notice that the above disjunctive formula is in $(\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0$; i.e., it is a Boolean combination of atomic formulas in $\Sigma_1^+$ and formulas in $(\Sigma_2 \cup T)^n$. Hence, any formula in $(\Sigma^+ \cup T)^n$ is equivalent to a formula in $(\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0$. That is,

$$(\Sigma^+ \cup T)^n \overset{\subset}{\sim} (\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0.$$

Notice that from Lemma 3,

$$(\Sigma \cup T)^{n+} \overset{\subset}{\sim} (\Sigma^+ \cup T)^n.$$

Hence

$$(\Sigma \cup T)^{n+} \overset{\subset}{\sim} (\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0.$$

In particular, $(\Sigma_2 \cup T)^n \overset{\subset}{\sim} (\Sigma \cup T)^n$. Thus,

$$(\Sigma_2 \cup T)^{n+} \overset{\subset}{\sim} (\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0.$$

Therefore, we have,

**Lemma 4** *If $f$ is a closed formula and $n$ is the levels of quantification in $f$, then*

$$(\Sigma \cup T)^{n+} \overset{\subset}{\sim} (\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0$$

*and*

$$(\Sigma_2 \cup T)^{n+} \overset{\subset}{\sim} (\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0.$$

Before we proceed further to demonstrate a procedure to calculate $f \in (\Sigma_2 \cup T)^n$, we interpret the intuitive meaning behind Lemma 4. Notice that from Lemma 1, $(\Sigma \cup T)^n/\sim$ is finite. Hence, $(\Sigma_2 \cup T)^n/\sim$ is also finite. Assume there are totally $k$ equivalent classes in $(\Sigma_2 \cup T)^n/\sim$. We pick an element $g_i$ ($g_i$ is a formula in $(\Sigma_2 \cup T)^n$) from each equivalent class in $(\Sigma_2 \cup T)^n/\sim$, $1 \leq i \leq k$. Recall $f \in (\Sigma \cup T)^n$. From the first part of the lemma,

$$(\Sigma \cup T)^{n+} \overset{\subset}{\sim} (\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0,$$

we have $f^+ \overset{\in}{\sim} (\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0$. That is, the "next" value $f^+$ of $f$ can be represented as a Boolean combination of the "next" values of formulas in $\Sigma_1$ and the "current" values of formulas in $(\Sigma_2 \cup T)^n$. Thus, $f^+$ can be written as a Boolean combination of formulas in $\Sigma_1^+$

and formulas $g_1, \cdots, g_k$. Then, how to calculate the "next" values $g_1^+, \cdots, g_k^+$ of formulas $g_1, \cdots, g_k$? The second part,

$$(\Sigma_2 \cup T)^{n+} \stackrel{\subseteq}{\sim} (\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0,$$

of the lemma gives the answer. From the second part, $g_i^+ \stackrel{\in}{\sim} (\Sigma_1^+ \cup (\Sigma_2 \cup T)^n)^0$ for each $1 \leq i \leq k$. That is, the "next" value $g_i^+$ of $g_i$ can be written as a Boolean combination of the "next" values of formulas in $\Sigma_1$ and the "current" values of formulas $g_1, \cdots, g_k$. Recall

$$\Sigma_1 = \{a(now) : a \in A\}$$

is finite since $A$ is finite. Thus, once we know the "next" values of each variable $a \in A$, we can calculate the truth value of $f^+$ by using those "next" values together with the "current" values of formulas $g_1, \cdots, g_k$. Once more, the "next" values of formulas $g_1, \cdots, g_k$ can be recursively calculated by using the "next" values of each variable $a \in A$, together with the "current" values of formulas $g_1, \cdots, g_k$.

Before we summarize the above discussion, we give some notation. A Boolean function is a mapping $\mathbf{Z}^+ \to \{0, 1\}$ from nonnegative integers to Booleans. A Boolean predicate is a mapping $\{0, 1\}^m \to \{0, 1\}$ for some $m$. Each closed past formula can be regarded as a Boolean function (with the only free variable *now*). We use $u$ to denote the Boolean function representing closed formula $f$. We use $v_1, \cdots, v_{|A|}$ to denote the Boolean functions representing elements (also closed formulas) in $\Sigma_1$. We use $u_1, \cdots, u_k$ to denote the Boolean function representing closed formula $g_1, \cdots, g_k$. To summarize the above discussions on Lemma 4, we have,

**Theorem 10** *When a closed past formula $f$ is treated as a Boolean function $u$, and Boolean functions $v_1, \cdots, v_{|A|}$ and $u_1, \cdots, u_k$ are defined as above, there are Boolean predicates $O, O_1, \cdots, O_k$ such that for all $t$ in $\mathbf{Z}^+$,*

$$u(t+1) = O(v_1(t+1), \cdots, v_{|A|}(t+1), u_1(t), \cdots, u_k(t))$$

*and for each $i$, $1 \leq i \leq k$,*

$$u_i(t+1) = O_i(v_1(t+1), \cdots, v_{|A|}(t+1), u_1(t), \cdots, u_k(t)).$$

Therefore, $u(t+1)$, as well as each $u_i(t+1)$, can be calculated by using the values of $v_1, \cdots, v_{|A|}$ at $t+1$, and values of $u_1, \cdots, u_k$ at $t$.

To conclude this section, we point out that once the functions $v_1, \cdots, v_{|A|}$ representing each $a(now)$ for $a \in A$ are known, each closed past formula can be recursively calculated as in Theorem 10. In the next two sections, we will build past formulas into a transition system. We first consider a simpler system, called a past machine, which involves only one clock *now*. Later, we study a more complex system called a past timed automaton, which contains a number of clocks and past formulas (not necessarily closed) as enabling conditions. The latter model is strong enough to model any Mini-ASTRAL process instances.

## 5.3 Past Machines: A Simpler Case

A *past-machine* $M$ is a tuple

$$\langle S, A, E, now \rangle$$

where $S$ is a finite set of *(control) states*. $A$ is a finite set of Boolean variables, and *now* is the only clock in $M$. *now* is used to indicate the current time. $E$ is a finite set of *edges* or *transitions*. Each edge

$$\langle s, \lambda, l, s' \rangle$$

denotes a transition from state $s$ to state $s'$ with *enabling condition* $l$ and *assignments* $\lambda$ to the Boolean variables in $A$. $l$ is a closed past-formula. $\lambda : A \to \{0, 1\}$ denotes the new value $\lambda(a)$ of each variable $a \in A$ after an execution of the transition. Execution of a transition causes the clock *now* to progress by 1 time unit. A *configuration* $\alpha$ of $M$ is a pair $\langle \alpha_{\mathbf{q}}, \alpha_H \rangle$ where $\alpha_{\mathbf{q}}$ is a state and $\alpha_H$ is a history,

$$\alpha_H = \langle \{||a||^{\alpha_H}\}_{a \in A}, \mathbf{n}^{\alpha_H} \rangle.$$

$$\alpha \to^{\langle s, \lambda, l, s' \rangle} \beta$$

denotes a one-step transition along edge $\langle s, \lambda, l, s' \rangle$ in $M$ satisfying:

- The state $s$ is set to a new location, i.e., $\alpha_{\mathbf{q}} = s, \beta_{\mathbf{q}} = s'$.
- The enabling condition is satisfied, i.e., $||l||_{\alpha_H}$ holds under the history $\alpha_H$.

- The clock *now* progresses by one time unit, i.e., $\mathbf{n}^{\beta_H} = \mathbf{n}^{\alpha_H} + 1$.

- The history $\alpha_H$ is extended to $\beta_H$ by adding the resulting values (given by the assignment $\lambda$) of the Boolean variables after the transition. That is, for all $a \in A$, for all $t$, $0 \le t \le \mathbf{n}^{\alpha_H}$, history $\beta_H$ is consistent with history $\alpha_H$; i.e., $\|a\|^{\beta_H}(t) = \|a\|^{\alpha_H}(t)$. In addition, $\beta_H$ extends $\alpha_H$; i.e., for each $a \in A$, $\|a\|^{\beta_H}(\mathbf{n}^{\beta_H}) = \lambda(a)$.

Simply write $\alpha \to \beta$ if $\alpha$ can reach $\beta$ by a one-step transition. A *path*

$$\alpha_0 \cdots \alpha_k$$

satisfies

$$\alpha_i \to \alpha_{i+1}$$

for each $i$. Write $\alpha \rightsquigarrow \beta$ if $\alpha$ reaches $\beta$ through a path. $\alpha$ is *initial* if $\mathbf{n}^{\alpha_H} = 0$. There are only finitely many initial configurations. $\beta$ is *admissible* if $\alpha \rightsquigarrow \beta$ for some initial configuration $\alpha$. Denote $R = \{\langle \alpha, \beta \rangle : \alpha \text{ is initial}, \alpha \rightsquigarrow \beta\}$.

Given a past machine $M$ specified as above. $M$, starting from an initial configuration $\alpha$ (i.e., with $now = 0$) can be simulated by a counter machine with reversal-bounded counters. In the following, we will show the construction. Each enabling condition $l$ on an edge $e \in E$ of $M$ is a closed past-formula. From Theorem 10, each $l$ can be associated with a number of Boolean functions $O_l, O_{1,l}, \cdots, O_{k,l}$, and a number of Boolean variables $u^l_1, \cdots, u^l_k$. Each $u^l_i$ represents an element in $(\Sigma_2 + T)^n$, in which $\Sigma_2$ and $T$ are constructed from $l$ with $n$ being the level of quantification of $l$. $l$ itself can be considered as a Boolean variable $u^l$. We use a primed form to indicate the previous value of a variable – here, a variable changes with time progressing. Thus, from Theorem 10,[2] these variables are updated as,

$$u^l := O_l(A, {u^l_1}', \cdots, {u^l_k}')$$

and for all $u^l_i$

$$u^l_i := O_{i,l}(A, {u^l_1}', \cdots, {u^l_k}').$$

Thus, $M$ can be simulated by a counter machine $M'$ as follows. $M'$ is exactly the same as $M$ except that each test of an enabling condition of $l$ in $M$ is replaced by a test of a

[2]We simply use $A$ to indicate the current value of $a \in A$ supposing the "current time" can be figured out from the context.

Boolean variable $u^l$ in $M'$. Further, whenever $M$ executes a transition, $M'$ does the following (sequentially):

- increase the counter *now* by 1,
- change the values of Boolean variables $a \in A$ according to the assignment given in the transition in $M$,
- for each enabling condition of $l$ in $M$, $M'$ has Boolean variables $u^l, u_1^l, \cdots, u_k^l$. $M'$ updates (given as above) $u_1^l, \cdots, u_k^l$ and $u^l$ for **each** $l$. Of course, during the process, the new values of $a \in A$ will be used, which were already updated in the above.

The initial value of Boolean variables $a$, $u^l$ and $u_j^l$ can be assigned using the initial value of $a$ in $\alpha$.[3] $M'$ contains only one counter *now*, which never reverses. Essentially $M'$ is a finite state machine augmented by one reversal-bounded counter. It is obvious that $M'$ faithfully simulates $M$. A configuration $\beta$ can be encoded as a string composed of the control state $\beta_q$, the current time $\mathbf{n}^{\beta_H}$ (as a unary string), and the history concatenated by the values of $a \in A$ at time $0 \le t \le \mathbf{n}^{\beta_H}$. All components are separated by a delimiter "#" as follows:

$$1^{\beta_q} \# \pi_0 \# \cdots \# \pi_{\mathbf{n}^{\beta_H}} \# 1^{\mathbf{n}^{\beta_H}}$$

where $\pi_t$ is a binary string with length $|A|$ indicating the values of all $a \in A$ at $t$. Thus, in this way, a set of configurations can be considered as a language. Denote $R_\alpha$ to be the set of configurations $\beta$ with $\alpha \rightsquigarrow \beta$. Then,

**Theorem 11** *$R_\alpha$ can be accepted by a reversal-bounded nondeterministic multi-counter machine.*

*Proof.* Given $\beta$ on the input tape. We construct another counter machine $M''$ to simulate $M'$ ($M'$ is the machine given as above). $M''$ starts from the initial configuration $\alpha$, During the simulation, $M''$ will check that the history encoding at the current time is consistent with the values of all $a \in A$ at current time by moving the input head from left to right.

[3] Each $u_j^l$ by definition corresponds to a past-formula. The initial value of $u_j^l$ can be calculated by replacing *now* in the formula with 0. The initial value, therefore, only depends upon the initial value of $a \in A$.

At some moment, $M''$ guesses $M$ has reached configuration $\beta$. Then $M''$ checks that the current values of *now* and variables $a \in A$ are consistent with those given on the input tape. $M''$ accepts if all the checks are consistent. Note that $M''$ uses a one-way input tape and only reversal-bounded counters. Thus, $R_\alpha$ can be accepted by a reversal-bounded multi-counter machine. ∎

Since there are only finitely many initial configurations,

$$R = \{\langle \alpha, \beta \rangle : \alpha \text{ initial } , \beta \in R_\alpha\}$$

can be accepted by a reversal-bounded nondeterministic multi-counter machine.

**Theorem 12** *$R$ can be accepted by a reversal-bounded nondeterministic multi-counter machine.*

If we don't consider the history in a configuration $\beta$ and instead we only consider the current values of all variables $a \in A$, then we write a *shortened configuration* $[\beta]$ to represent a tuple with a control state, a valuation of the current values of $a \in A$, and the current time. Use $[R]$ to denote all $\langle [\alpha], [\beta] \rangle$ with $\langle \alpha, \beta \rangle \in R$. Note that in the above proofs the history on the input tape is only used to check the consistency of the current values of variables $a \in A$ with time progressing. Thus, the same result holds without making these checks. Combining Theorems 5 and 12, we have,

**Theorem 13** *$[R]$ is Presburger.*

So far, a safety property $P$ like

$$now > 5 \rightarrow a \vee b$$

can be automatically verified for a past machine $M$. The reason is that this property is Presburger, and according to Theorems 13, 4, and 5, the emptiness of

$$\{\langle [\alpha], [\beta] \rangle \in [R] : [\beta] \text{ satisfies } \neg P\}$$

(this set is not empty if and only if $P$ is violated) is decidable.

But past machines are not strong enough to characterize a Mini-ASTRAL process instance. The reason is that past machines contain only closed past formulas. Thus, we have to extend past machines by allowing a number of clock variables in the system, as shown in the next section.

## 5.4 Past Timed Automata: A More Complex Case

Past machines can be extended by allowing extra free variables, in addition to *now*, in an enabling condition. We use $Z = \{z_1, \cdots, z_k\} \subseteq X$ to denote the variables other than *now*. A *past timed automaton* $M$ is a tuple

$$\langle S, A, Z, E, now \rangle$$

where $S$ and $A$ are the same as those for a past machine. $Z$ is a finite set of clocks. *now* is a clock that never resets indicating the current time; $now \notin Z$. Each edge, from state $s$ to state $s'$, in $E$ is denoted by

$$\langle s, \delta, \lambda, l, s' \rangle.$$

$l$ and $\lambda$ have the same meaning as in a past machine, though the enabling condition $l$ may contain, in addition to *now*, free (clock) variables in $Z$. $\delta \subseteq Z$ denotes a set of clock jumps. $\delta$ may be empty. A configuration $\alpha$ of $M$ is a tuple $\langle \alpha_{\mathbf{q}}, \alpha_H, \alpha_Z \rangle$ where $\alpha_{\mathbf{q}}$ is a state, $\alpha_H$ is a history as defined in a past machine, and $\alpha_Z \in (\mathbf{Z}^+)^{|Z|}$ is a valuation of clock variables $Z$. We use $\alpha_z$ to denote the value of $z \in Z$ under this configuration.

Similarly,

$$\alpha \rightarrow^{\langle s, \delta, \lambda, l, s' \rangle} \beta$$

denotes a one-step transition along edge $\langle s, \delta, \lambda, l, s' \rangle$ in $M$ satisfying:

- The state $s$ is set to a new location, i.e., $\alpha_{\mathbf{q}} = s, \beta_{\mathbf{q}} = s'$.
- The enabling condition is satisfied, i.e., $||l||_{\alpha_H, \mathcal{B}(\alpha_Z/Z)}$ holds for any $\mathcal{B}$. That is, $l$ is evaluated under the history $\alpha_H$ and replacing each free clock variable $z \in Z$ by the value $\alpha_z$ in the configuration $\alpha$.
- Each clock changes according to the edge given.

– If $\delta = \emptyset$, i.e., there are no clock jumps on the edge, then the now-clock progresses by one time unit. That is, $\mathbf{n}^{\beta_H} = \mathbf{n}^{\alpha_H} + 1$. All the other clocks do not change; i.e., for each $z \in Z$, $\beta_z = \alpha_z$.

– If $\delta \neq \emptyset$, then all the clocks in $\delta$ jump to *now*, and the other clocks do not change. That is, for each $z \in \delta$, $\beta_z = \mathbf{n}^{\alpha_H}$. In addition, for each $z \notin \delta$ , $\beta_z = \alpha_z$. In addition, the clock *now* does not progress, i.e., $\mathbf{n}^{\beta_H} = \mathbf{n}^{\alpha_H}$.

- The history is updated similarly as for past machines. That is,

  – If $\delta = \emptyset$, then *now* progresses, for all $a \in A$, for all $t$, $0 \leq t \leq \mathbf{n}^{\alpha_H}$, $||a||^{\beta_H}(t) = ||a||^{\alpha_H}(t)$, and $||a||^{\beta_H}(\mathbf{n}^{\beta_H}) = \lambda(a)$.

  – If $\delta \neq \emptyset$, then *now* does not progress, for all $a \in A$, for all $t$, $0 \leq t \leq \mathbf{n}^{\alpha_H} - 1$, $||a||^{\beta_H}(t) = ||a||^{\alpha_H}(t)$, and $||a||^{\beta_H}(\mathbf{n}^{\beta_H}) = \lambda(a)$. Thus, even though the now-clock does not progress, the current values of variables $a \in A$ may change according to the assignment $\lambda$.

$\alpha$ is initial if all clocks including *now* are 0. We can similarly define $\alpha \leadsto \beta$. We also adopt similar notation for $R$ and $[R]$. That is, $R = \{\langle \alpha, \beta \rangle : \alpha \text{ is initial}, \alpha \leadsto \beta\}$. We use $[\beta]$ to denote the shortened version of configuration $\beta$; i.e., containing only the current control state, the values of $a \in A$, and the clock values. Accordingly, $[R] = \{\langle [\alpha], [\beta] \rangle : \langle \alpha, \beta \rangle \in R\}$. A (shortened) configuration can be encoded, as before, into a string using unary string representation of clock values concatenated by the control state and (the history) of Boolean variables. Thus, both $R$ and $[R]$ can be considered as languages by concatenating the two encodings of (shortened) configurations with delimiter "#".

The main result in this section is that $R$ and $[R]$ can be accepted by a reversal-bounded NCM. The major difference between a past machine and a past timed automaton is that the enabling condition on an edge in the past timed automaton is not a closed past formula. The proof will show that an enabling condition $l$ with free variables in $Z$ can be made closed.

**Theorem 14** *$R$ for a past timed automaton can be accepted by a reversal-bounded NCM using standard tests and assignments. Thus, so can $[R]$.*

*Proof.* See Section 5.6.2. ∎

Notice that $[R]$ is a tuple language; therefore, from Theorem 5,

**Theorem 15** *$[R]$ for a past timed automaton is Presburger.*

The importance of the automata-theoretic characterization of $[R]$ is that the satisfiability of Presburger formulas is decidable. We now formulate properties that can be automatically verified for a past timed automaton, which is the theoretical foundation for verification of Mini-ASTRAL in the next section.

We first need some notation. From now on, a configuration, unless specified otherwise, contains only the control state, the values of Boolean variables and the values of all clocks. Thus, a configuration is without the history. Previously we used $[\alpha]$ to represent it, but now we simply use $\alpha$. We use *a configuration with a history* to indicate a long form of a configuration. For a pair of configurations $\alpha$ and $\beta$, we use $\alpha \leadsto \beta$ to represent $\langle \alpha, \beta \rangle \in [R]$. Here, $\alpha$ is implicitly an initial configuration. We use $\alpha, \beta \cdots$ to denote variables ranging over configurations. We use $\mathbf{q}, \mathbf{a}, \mathbf{x}$ to denote variables ranging over control states, Boolean values, and clock values. Of course, *now* is one of the clocks. Note that $\alpha_{x_i}$, $\alpha_q$, $\alpha_a$ are still used to denote the values of clock $x_i$, the control state, and the value of Boolean variable $a$. An NCM-term $t$ is defined as follows:

$$t \ ::= \ n \mid \mathbf{q} \mid \mathbf{x} \mid \alpha_{x_i} \mid \alpha_q \mid t - t \mid t + t$$

where $n$ is an integer. An NCM-formula $f$ is defined as follows:

$$f \ ::= \ \alpha_a \mid t > 0 \mid \neg f \mid f \vee f \mid \alpha \leadsto \beta \mid \forall \alpha(f) \mid \forall \mathbf{x}(f) \mid \forall \mathbf{q}(f).$$

$f$ is a Presburger formula over control state variables, clock value variables and configuration variables, noticing that from Theorem 15 $\leadsto$ is Presburger. Thus, if $f$ is closed (i.e., without free variables), then the truth value of $f$ is decidable. Thus, a property formulated as a closed NCM-formula can be verified.

In the next section, we will show that properties expressed in a Mini-ASTRAL process instance are covered by closed NCM-formulas, and, in fact, are much simpler than closed NCM-formulas. This is the reason why Mini-ASTRAL can be automatically verified, as shown in the next section.

## 5.5 Relating a Mini-ASTRAL Process Instance to a Past Timed Automaton

In this section we will show that a Mini-ASTRAL Process Instance $\mathcal{P}$ can be translated into a past timed automaton and the properties in $\mathcal{P}$ can be, theoretically, automatically verified. A past timed automaton extends a clock-jump machine, which was discussed in the last chapter. The clock behaviors in a past timed automaton are exactly the same as those in a clock-jump machine, except past values of Boolean variables can be referenced in a clock constraint, as in a past formula. In the following, we will show that there are no technical obstacles in translating $\mathcal{P}$ into a past timed automaton $M$, by considering each component of $\mathcal{P}$.

### 5.5.1 Local Variables

Local variables in $\mathcal{P}$, by definition, are of finite states. A past timed automaton as defined in this chapter, allows Boolean variables as state variables. Theoretically, we can use a number of Boolean variables to denote the domain of a finite state variable. For instance, assume $a$ with domain $\{1, 2, 3\}$ instead of a Boolean variable. We use three Boolean variables $a_1, a_2, a_3$ to represent $a$. $a_i$ stands for $a = i$ with $i = 1, 2, 3$. Thus, in this way, A past timed automaton with variables in finite domains can be translated into one with only Boolean variables.

### 5.5.2 Clocks

Clocks in $\mathcal{P}$, including `NOW`, the (last) start time `Start(T)`, the (last) end time `End(T)`, the (last) call time `Call(T)` of a transition `T`, and the (last) change time `Change(x)` of a variable `x` are directly related to clocks in $M$.

### 5.5.3 Formulas

Formulas in $\mathcal{P}$, including initial conditions, entry-exit assertions, assumptions and properties are all in the form of past formulas in which the only free variables are clocks. Thus, they are all in the form allowed by $M$.

As in Chapter 4, the control structure of $M$ is exactly as the clock-jump machine translated from a history-independent Mini-ASTRAL process instance, except that the enabling conditions, instead of extended clock constraints, are past formulas. Thus, theoretically, $\mathcal{P}$ can be translated into $M$, and it can be automatically verified.

## 5.6 Technical Proofs

### 5.6.1 Proof of Lemma 2

Lemma 2. For each $h \in (\Sigma \cup T)^n$, $[U^n] \wedge h^+$ is equivalent ($\sim$) to a formula in $(\Sigma^+ \cup T)^n$. That is, $[U^n] \wedge h^+ \overset{\in}{\sim} (\Sigma^+ \cup T)^n$.

*Proof.* We prove the lemma by induction on $n$. We first prove the lemma holds for $n = 0$. That is, $h \in (\Sigma \cup T)^0$ without quantification. Thus, $U^0 = \Gamma_f$. We have the following cases to consider:

Case 1. $h = a(x)$ with $a \in A$ and $x \in \Gamma_f$. Since $h \in \Sigma$, we have $h^+ \in \Sigma^+$. The lemma holds trivially.

Case 2. $h = x < y + k$ with $x, y \in \Gamma_f$ and $k \in [m]$. Note that by definition, $h \in T$. There are four subcases to consider:

Subcase 1. $x \neq \mathit{now}$ and $y \neq \mathit{now}$. Thus, $h^+ = h \in T$. The lemma holds trivially.

Subcase 2. $x = \mathit{now}$ but $y \neq \mathit{now}$. Thus, $h^+$ is $\mathit{now} + 1 < y + k$ with $k \in [m]$. In this case, if $k > 0$, then $k - 1 \in [m]$. Thus, $\mathit{now} < y + (k-1) \in T$. If $k \leq 0$, then since $y \in U^0$ is restricted in $[0, \mathit{now}]$, $(\mathit{now} + 1) < y + k$ can be replaced by false ($0 > 0$). Thus, the lemma holds.

Subcase 3. $x \neq \mathit{now}$ and $y = \mathit{now}$. Thus, $h^+$ is $x < (\mathit{now} + 1) + k$. In this case, if $k < 0$, then $k + 1 \in [m]$. Thus, $x < (\mathit{now} + 1) + k \in T$. If $k \geq 0$, then since $x \in U^0$ is restricted in $[0, \mathit{now}]$, $x < (\mathit{now} + 1) + k$ can be replaced by true (0=0). Thus, the lemma holds.

Subcase 4. $x = y = \mathit{now}$. Thus, $h^+$ is $(\mathit{now} + 1) < (\mathit{now} + 1) + k$. Thus, $h^+$ can be replaced by the truth value of $0 < k$. Thus, the lemma holds.

Hence, the lemma holds for Case 2.

Case 3. $h = f_1 \vee f_2$ or $h = \neg f_1$ with $f_1$ and $f_2$ satisfying the lemma. The lemma holds for $h$ noticing that $(\Sigma^+ \cup T)^n$ is closed for Boolean operations.

Thus, for $n = 0$, the lemma holds.

Assume the lemma holds for all $n' \leq n-1$ with $n \geq 1$. Now, consider the case for $n$.

The following is the only nontrivial case. Let $h = \forall_x g[0, now] \in (\Sigma \cup T)^n$. Without loss of generality, assume $x$ is the last element $x_n$ in $B^n$. So, $h^+ = \forall_x g^+[0, now+1]$ and it can be further split into two parts:

$$\forall_x g^+[0, now] \wedge g^+(now + 1/x)$$

where $g^+(now+1/x)$ means replacing $x$ with $now + 1$ in $g^+$. Now consider the two parts $\forall_x g^+[0, now]$ and $g^+(now + 1/x)$, respectively.

- Since $g \in (\Sigma \cup T)^{n-1}$, by the induction hypothesis, $[U^{n-1}] \wedge g^+$ is equivalent to a formula in $(\Sigma^+ \cup T)^{n-1}$. Also notice that (in the right hand side of the following equation, the inner $[U^{n-1}]$ is subsumed by the outer $[U^n]$.)

  $$[U^n] \wedge \forall_x g^+[0, now] \text{ is equivalent to } [U^n] \wedge \forall_x([U^{n-1}] \wedge g^+)[0, now].$$

  Thus, $[U^n] \wedge \forall_x g^+[0, now]$ is equivalent to a formula in $(\Sigma^+ \cup T)^n$.

- Now consider $[U^{n-1}] \wedge g^+(now+1/x)$. Free variables appearing in $g^+(now+1/x)$ are in $U^{n-1}$, since the free variable $x$ is already replaced. Denote $g' = g(now/x)$. Thus, $x$ doesn't appear in $g'$. It is observed that

  $$g' \in (\Sigma \cup T)^{n-1}$$

  and

  $$g^+(now + 1/x) = g'^+.$$

  Thus, $[U^{n-1}] \wedge g^+(now+1/x) = [U^{n-1}] \wedge g'^+$. By the induction hypothesis, $[U^{n-1}] \wedge g'^+$ is equivalent to a formula in $(\Sigma^+ \cup T)^{n-1}$. Therefore, since $x$ does not appear in $U^{n-1}$ and $g^+(now+1/x)$, $[U^{n-1}] \wedge g^+(now+1/x)$ is equivalent to $\forall_x([U^{n-1}] \wedge g^+(now+1/x))$. Thus, $[U^{n-1}] \wedge g^+(now + 1/x)$ is equivalent to a formula in $(\Sigma^+ \cup T)^n$.

Thus, combining the two parts, $[U^n] \wedge h^+$ is therefore equivalent to a formula in $(\Sigma^+ \cup T)^n$.
∎

### 5.6.2 Proof of Theorem 14

Theorem 14. $R$ for a past timed automaton can be accepted by a reversal-bounded NCM using standard tests and assignments. Thus, so can $[R]$.

*Proof.* Let $M$ be a past timed automaton specified as

$$\langle S, A, Z, E, now \rangle.$$

Recall that intuitively a clock $z_i$ stands for the (last) time a jump "$z_i = now$" happens. Thus, for each $z_i$ we introduce a Boolean variable $b_i \notin A$. Denote all $b_i$ as $B$.

We construct a machine $M'$ that is exactly the same as $M$ except:

- $M'$ has Boolean variables $b_i$ for each $z_i \in Z$. For each edge $e$ in $M$, if $e$ has some jump $z_i := now$, then add $b_i := true$ as an assignment on this edge. For other $z_i$ that do not jump on this edge, $b_i$ is unchanged. If $e$ does not have any jumps, then $b_i = false$ for all $i$.

- Since the initial value of clocks $z_i$ are 0, therefore, initially, $b_i = true$ for all $i$.

- Denote $Q_i$ as

  $$b_i(z_i) \wedge \forall_{x_i}(z_i < x_i \leq now \to \neg b_i(x_i))[0, now])$$

  that means $z_i$ is the last jump time of clock $z_i$. The enabling condition $l$ on an edge is replaced by

  $$\forall 0 \leq z_1, \cdots, z_k \leq now((Q_1 \wedge \cdots \wedge Q_k) \to l).$$

  Note that the above is a closed past-formula.

It is easy to check that $M'$ simulates $M$. In the proof of Theorem 11, we know that a closed enabling condition can be replaced by a Boolean variable while firing a transition will accordingly update a number of extra Boolean variables. But there is a slight problem in doing the updates in $M'$. The reason is that $now$ behaves differently in $M'$ – in a transition where there is a jump, $now$ does not progress. However, in a past machine, whenever a transition fires, $now$ progresses by one time unit. In the following, we will give the updating procedure for $M'$. In $M'$, each enabling condition $l$ is in the closed form. Note that besides Boolean variables in $A$, $M'$ also has Boolean variables $b_i$ in $B$. An assignment in a transition in $M'$ includes an assignment to clocks (jumps) and an assignment $\tau$ to all Boolean variables in $\hat{A} = A \cup B$. We use $\tau(a) \in \{0, 1\}$ to denote the value of $a \in \hat{A}$ under $\tau$. For each edge $e \in E$ of $M'$, the enabling condition is $l_e$ and the assignment to Boolean variables is $\tau_e$. From the proof of Theorem 11, we know that by representing $l_e$ by a Boolean variable $u^{l_e}$, we need extra Boolean variables $u_1^{l_e}, \cdots, u_{k_e}^{l_e}$, and Boolean functions $O^{l_e}, O_1^{l_e}, \cdots, O_{k_e}^{l_e}$, as well as Boolean variables $a \in \hat{A}$. The idea is as follows. When $e$ does not have a jump, i.e., after firing $e$ *now* will progress, we update these Boolean variables normally, i.e., as given in the proof of Theorem 11. However, if afterwards the next transition has a jump, i.e., *now* does not progress, we have to precompute the updates at the previous transition. Doing these precomputations, we need multiple copies of extra Boolean variables $u^{l_e,\tau}, u_1^{l_e,\tau}, \cdots, u_{k_e}^{l_e,\tau}, a^\tau$ for each $a \in \hat{A}$ and for each possible assignment $\tau$ for Boolean variables $\hat{A}$. Note that we have at most $2^{|\hat{A}|}$ choices of $\tau$. Whenever $M$ fires a transition $e_0$, $M'$ updates all the Boolean variables as follows (sequentially), recalling that a primed form of a variable is used to indicate the previous value before the transition fired.

- change the values of Boolean variables $a \in \hat{A}$ according to the assignment $\tau_{e_0}$ given on the edge $e_0$,
- if $e_0$ does not have a jump,
  - $now := now + 1$,
  - for each $\tau$ and for each $a \in \hat{A}$, $a^\tau := \tau(a)$. Denote all $a^\tau$ as $A^\tau$.
  - For **all** edges $e$, and for each $\tau$ and for each $1 \le i \le k_e$,

    $$u_i^{l_e,\tau} := O_i^{l_e}(A^\tau, (u_1^{l_e})', \cdots, (u_{k_e}^{l_e})')$$

    and $u^{l_e,\tau} := O^{l_e}(A^\tau, (u_1^{l_e})', \cdots, (u_{k_e}^{l_e})')$.
  - For **all** edges $e$, $u^{l_e}$ is assigned as the precomputed value according to the assignment $\tau_{e_0}$, i.e., $u^{l_e} := u^{l_e,\tau_{e_0}}$. Also, $u_i^{l_e} := u_i^{l_e,\tau_{e_0}}$ for $1 \le i \le k_e$.
- if $e_0$ has at least one jump,
  - execute all the jumps $z_i := now$ on $e_0$. But *now* does not progress.

– For **all** edges $e$, $u^{l_e}$ is assigned as the precomputed value according to the assignment $\tau_{e0}$; i.e., $u^{l_e} := (u^{l_e,\tau_{e0}})'$, and $u_i^{l_e} := (u_i^{l_e,\tau_{e0}})'$ for $1 \leq i \leq k_e$. All other Boolean variables are unchanged.

The initial values of Boolean variables $u^{l_e}, u_j^{l_e}$ and $u^{l_e,\tau}, u_j^{l_e,\tau}$ can be assigned using the initial value of $a$ and $\tau(a)$. All updates to Boolean variables and tests on them can be left to the finite control of $M'$. Each enabling condition $l_e$ in $M'$ is replaced by $u^{l_e}$. Thus, $M'$ becomes an NCM $M''$ after doing this. Tests in $M''$ (by looking at the procedure given above) are tests of Boolean variables; thus, they are standard. But clock assignments in $M''$ in the form of

$$now := now + 1$$

and

$$x_i := now$$

are not standard. Recall the semantics that $now := now + 1$ happens if and only if there is no clock jump $x_i := now$ happening on an edge. We will now show that these assignments can be made standard, while the machine is still reversal-bounded, by showing that $R$ can be accepted by a reversal-bounded counter machine. Given (the string encodings) two configurations $\alpha$ and $\beta$ with $\alpha$ being initial on the input tape, we will construct a machine $\hat{M}$. Let $\hat{M}$ be an NCM that is exactly the same as $M''$ except for the following. $\hat{M}$ simulates $M''$'s computation from the initial configuration $\alpha$. Initially, each counter $x_i$ is 0 as we indicated before. However, each time that $M''$ executes an assignment $now := now + 1$, $\hat{M}$ increases all the counters by 1, i.e., $x_i := x_i + 1$ for each $0 \leq i \leq k$. When $M''$ executes a jump assignment $x_i := now$, $\hat{M}$ does nothing to the clocks. Boolean variable updates in $M''$ are faithfully simulated by $\hat{M}$. For each clock $x_i$, at some point, either initially or at the moment $x_i := now$ is being executed by $M''$, $\hat{M}$ guesses (only once for each $i$) that $x_i$ has already reached the value given in $\beta$. After such a guess for $i$, an execution of $now := now + 1$ will not cause $x_i := x_i + 1$ as indicated above (i.e., $x_i$ will no longer be incremented). However, after such a guess for $i$, a later execution of a jump $x_i := now$ in $M''$ will cause $\hat{M}$ to abort abnormally (without accepting the input). At some point after all $x_i$ have been guessed, $\hat{M}$ guesses that it has reached the configuration $\beta$. Then, $\hat{M}$

compares its current configuration with the one on the rest of the input tape $\beta$. Also, notice that during the simulation $\hat{M}$ also checks the consistency of Boolean variables in $A$ with the ones encoded in the histories in $\beta$, as in Theorem 11. $\hat{M}$ accepts iff such a comparison succeeds. Clearly, $\hat{M}$ uses only assignments $now := now + 1$ and $x_i := x_i + 1$ for $1 \leq i \leq k$. Thus, $\hat{M}$ is also reversal-bounded and accepts $R$. ∎

## 5.7 Related Work

Past formulas are not new. In fact, they can be expressed in TPTL [AH94]. It has been shown [AH94] that validity checking for TPTL (and hence closed past formulas) is decidable. However, when dense clocks are considered, the decidability does not remain [AH94]. Results in [AH94] show that discrete timed automata can be model-checked against TPTL, in which explicit time references to the value of a finite state variable are allowed. But, here, we put a past formula into the enabling condition of a transition in a generalized timed system (strictly stronger than discrete timed automata). This makes it possible to model a real-time machine that is history-dependent. The tableau technique in proving Lemma 2 is a modification of the technique proposed in [AH94].

If a past-machine $M$ does not contain any past terms and quantification on its enabling conditions, $M$ is essentially equivalent to the standard timed automaton given in [AD94] when integer-valued clocks are considered.

The construction technique discussed in this chapter actually works on a broader class of machines, called *past pushdown timed automata* [DIBKS00c], which is a past timed automaton, as shown in this chapter, that is further augmented with a pushdown stack. In [DIBKS00c], a richer form of properties is shown to have decidable verification procedures. The results, theoretically, can be used to analyze pushdown processes [BEM97, FWW97] further augmented with clocks and histories.

Pierluigi San Pietro points out that a past formula without an explicit appearance of *now* (i.e., *now* is used only for bounds for quantifications.) is expressible in WS1S (weak second-order monadic logic with one successor). In fact, they are exactly LTL formulas; i.e., they define start free $\omega$-regular languages [W83]. Inclusions of the free variable *now* may cause a problem, since this takes the formulas out of WS1S (therefore, also out of LTL). But since the properties are safety properties, we may construct another Buchi automaton with its behaviors restricted by an LTL formula and eliminate past formulas from the past timed automaton. Intersecting the two automata will give the results in this chapter.

Due to the result that a Presburger formula on clocks can be verified for a past timed automaton as shown in Section 5.4, Mini-ASTRAL can be extended such that the properties are in the form of Presburger formulas over clocks and still keep decidability for the verification problem.

# Chapter 6

# Theoretical Results

As stated in Chapter 1, another research issue in this dissertation is to investigate, in theory, what kind of models and properties of infinite systems have a decidable model-checking problem. This chapter[1] tries to look into this issue by presenting two theoretical results.

Previous work on infinite-state systems has concentrated on only a limited number of models, such as Petri Nets ($PN$), Pushdown Automata ($PA$) and Timed Automata ($TA$), and has been used to study the decidability and complexity of model-checking various temporal and modal logics. A timed automaton [AD94] is basically a finite-state automaton with a certain number of unbounded clocks that can be tested and reset. Since their introduction and the definition of appropriate model checking algorithms [ACD93, HNSY94], timed automata have become a standard model to investigate the verification of real-time systems and have been extensively studied (see [A99, Y98] for surveys). The expressive power of $TA$ has many limitations in modeling, since many real-time systems are simply not finite-state, even when time is ignored. Other infinite-state models for which forms of automatic verification are possible are based on $PN$ [G98], on $PA$ [BEM97, FWW97], or on process calculi [M98, BS97]; however, at least in their basic versions, they do not consider timing requirements and are thus not amenable for modeling real-time systems. Among the infinite-state models that consider time, there are many timed extensions of Petri Nets (such

[1] The work in this chapter is adapted from work [PD00a, PD00d] co-authored with Pierluigi San Pietro in Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italia.

as Timed Petri Nets [M76]), but these models usually have undecidable binary reachability if the net is unbounded (i.e., if it is not finite state).

Very recently, we have proposed Timed Pushdown Automata ($TPA$) by extending pushdown processes with unbounded discrete clocks, and we have shown that safety and binary reachability analysis are still decidable [DIBKS00b]. Pushdown automata have had an enormous impact in the theory and application of compiler construction, but the natural model of most systems is not a pushdown stack. *Queues*, not stacks, are a good model for many interesting systems, such as schedulers, for which automatic verification has rarely been attempted.

Queues are usually regarded as hopeless for verification, since it is well known that a finite-state automaton equipped with one unbounded queue can simulate a Turing Machine. There are, however, restricted models with queues for which reachability is decidable(e.g., [CF97, WB98]). In this dissertation, we consider the Generalized Context-free Grammars (GCG) of [BCCC92], which use both queues and stacks with suitable constraints in order to generate only semilinear languages, and which are powerful enough to model most scheduling policies [BCCC99]. However, automatic verification of GCG has never been investigated, and GCG does not consider time. In the first part of this chapter, we study how to couple a timed automaton with a multi-queue automaton (inspired by the GCG model) so that the resulting machine can be effectively used for modeling, while retaining the decidability of a class of Presburger formulas over the binary reachability set, with control-state variables, clock value variables and count variables. Therefore, these machines are amenable to the modeling and automatic verification of many infinite-state real-time systems, such as real-time process schedulers.

Another field of research is on counter machines, i.e., automata whose memory tapes can only assume numeric values. Various models of counter machines have been studied in the past (e.g., one-counter machines and their variants [G75], blind [G78] or reversal-bounded [BB74, I78] multi-counter machines). The recent interest in counter models [CJ98, FS00, ISDBK00] is not motivated by investigations of their formal language properties but by their applications to model checking or safety analysis of infinite-state systems. Many models of infinite state systems are based on some kind of counters. For instance, Petri Nets can be seen as particular counter machines, called Vector Addition Systems. The timed models, such as Timed Automata [AD94], usually allow numerical clocks, which progress with time and can be tested, compared or reset. These clocks, whose values can be either discrete or dense, can basically be regarded as some sort of counters (e.g., they can be incremented, reset, or transferred [Y98], but they are never decremented). Other more complicated models that are ultimately based on counters are Rectangular Automata [HKPV95], which can be regarded as a timed model with skewed clocks. That is, the clocks can change with a rate included in some given (real) interval.

Unfortunately, the counters studied in traditional automata theory are not always adequate for these new models, even when considering the discrete case. This is an obstacle to the theoretical investigation of model-checking and safety analysis. Various new types of counter machines have been studied [CJ98, FS00], but the subject is far from being exhausted.

Our aim in the second part of this chapter is to contribute to the theory of multi-counter machines, by introducing and studying a new type of integer counter, namely *piecewise-monotonic* counters. These counters are combined with blind counters and reversal-bounded counters in order to provide a model, the MMMCM (multi-reversal bounded multi-blind multi-piecewise-monotonic counter machine), for analyzing various infinite-state machines. A counter is piecewise-monotonic if between two consecutive resets either there is no increase or there is no decrease (i.e., the counter is monotonic in the interval). A counter is reversal-bounded [I78] if it changes mode between nondecreasing and nonincreasing at most a bounded number of times. A counter is blind [G78] if it is not allowed to test against 0. In an MMMCM, each counter can be incremented, decremented and reset. However, only piecewise-monotonic counters and reversal-bounded counters may be tested against 0. In our model, we allow one unrestricted counter, which is not reversal-bounded and which is able to be incremented, decremented, tested, and reset. We also allow transfer operations (called *append*) from the unrestricted counter or from a piecewise-monotonic counter to a blind counter, but with some restrictions.

We are mainly interested in those properties of MMMCMs that could be useful for verifying the decidability of infinite-state models, rather than in pure language-theoretic properties. Therefore, we concentrate on showing that MMMCMs allow decidability proofs of safety analysis or model-checking for some 'interesting' infinite state models. In fact, MMMCMs only accept effective semilinear languages (therefore, they have decidable emptiness), and the binary reachability is also effectively semilinear. Moreover, they are effectively closed under intersection with regular languages.

We believe that the MMMCM model is capable of providing an alternative theoretical tool to analyze various timed models. For instance, if the reachability set of a particular infinite model $\mathcal{M}$ can be accepted by an MMMCM, then it is also decidable to verify if a state of $\mathcal{M}$ is reachable. Also, (non)closure properties can be used to prove that certain kinds of logic are (un)decidable for model-checking over $\mathcal{M}$.

In this chapter, we actually introduce a more general model than a counter machine, namely the reversal-bounded multi-counter multi-pushdown machine, extending the multi-pushdown model of [CBCC96] with reversal-bounded counters. This model, with suitable restrictions on the counters and the pushdowns, is effectively semilinear and it may be used to show that the MMMCM model is also effectively semilinear.

This chapter is structured as follows. The first part includes five subsections. Section 6.1 recalls a number of definitions on GCG grammars. Section 6.2 introduces multi-queue-stack machines and proves the effective semilinearity of the model, by using a GCG. Section 6.3 defines multi-queue discrete timed automata by carefully coupling a discrete timed automaton with a multi-queue machine (i.e., a multi-queue-stack machine without pushdown stacks). Section 6.4 proves the main result for this part of the chapter, i.e., the effective semilinearity of binary reachability for multi-queue discrete timed automata by proving that the binary reachability can be accepted by multi-queue-stack machines. Section 6.5 proves the decidability of a class of Presburger formulas over the binary reachability.

The second part of this chapter includes two subsections. Section 6.6 introduces and studies the reversal-bounded multi-counter multi-pushdown machines. Section 6.7 defines the MMMCMs and shows that they are effectively semilinear.

Technical proofs can be found in Section 6.8.

## 6.1 Generalized Context-free Grammars

To help the reader, this section briefly gives the basic definitions of Generalized Context-free Grammars (GCG), also known as multidepth-breadth grammars [BCCC92, BCCC99], which are used to prove that a Multi-queue Discrete Timed Automaton has a computable semilinear binary reachability set. For readers that are not familiar with semilinear languages, Section 6.8.1 recalls the definitions.

A *list* of degree $k \geq 0$ (also called a $k$-list) over two finite alphabets $V_N$ and $\Sigma$ is a string $\gamma$ of the form: $\gamma_0(\gamma_1)_1(\gamma_2)_2 \ldots (\gamma_k)_k$ where $\gamma_0 \in \Sigma^*, \gamma_j \in V_N^*$, for $1 \leq j \leq k$, and '(', ')' $\notin \Sigma \bigcup V_N$. Each $\gamma_j$, $j \geq 1$, is called the $j$-th component of $\gamma$, while $\gamma_0$ is called the terminal part of $\gamma$. By convention, an element $(\gamma_i)_i$ of a $k$-list is omitted whenever $\gamma_i = \epsilon$. The null $k$-list $\epsilon(\epsilon)_1(\epsilon)_2 \ldots (\epsilon)_k$ is denoted by $\epsilon$.

Let $S, Q$ be two symbols ($S$ stands for stack and $Q$ stands for queue). Let $z \in \{S, Q\}^+$, $k = |z|$.

A *generalized context-free grammar* (GCG) $G$ *with rewriting disposition* $z$ is a 5-tuple

$$(V_N, \Sigma, P, A, z)$$

where $V_N$ and $\Sigma$ are two finite alphabets, called the nonterminal and terminal alphabets, respectively, $A \in V_N$ is the axiom, and $P$ is a set of elements (called *productions*) of the form: $X \to \gamma$, where $X \in V_N$ and $\gamma$ is a $k$-list over $V_N$ and $\Sigma$.

For every $k \geq 1$, a derivation is a relation between $k$-lists, which is defined as follows. Let $X \to \alpha_0(\alpha_1)_1 \ldots (\alpha_k)_k$ be a production. If $\gamma = x(X\gamma_i)_i(\gamma_{i+1})_{i+1} \ldots (\gamma_k)_k$ is a $k$-list, then $\gamma \Rightarrow x\alpha_0(\alpha_1)_1 \ldots (\alpha_{i-1})_{i-1}(\delta_i)_i \ldots (\delta_k)_k$ is a derivation step where: for every $j$, $i \leq j \leq k$, if the $j$-th element of $z$ is $S$ then $\delta_j = \alpha_j\gamma_j$, else $\delta_j = \gamma_j\alpha_j$.

Notice that only *leftmost* derivations are defined: at any step of a derivation the leftmost nonterminal symbol is eliminated from the $k$-list, and each component of a production is written into the corresponding component of the original $k$-list. The idea is that each component can be either a pushdown stack ($S$) or a FIFO queue ($Q$). In the former case, the corresponding component of the production is rewritten to the left of the original component; otherwise, it is rewritten to the right. Since the content of the components is read from left to right, the former case corresponds to the adoption of a stack policy and the latter case to the adoption of a queue policy. The terminal part is always written at the right of the terminal part of the original $k$-list. A consequence of the definition is that the symbols in the $i+1$-th component, $i \geq 1$, can be rewritten only when the first $i$ components are empty.

A derivation always starts with the marked string $(A)_1$. A string $x \in \Sigma^*$ is *derivable* from $A$ if $(A)_1 \Rightarrow_G^* x$. The language generated by G is $L(G) = \{x \in \Sigma^* \mid (A)_1 \Rightarrow_G^* x\}$.

The following is an example of a GCG $G = (\{A, B, C, D\}, \{a, b, c, d\}, P, A, SQ)$, where $P$ is

$$\{A \to a(AB)_1(C)_2, A \to a(B)_1(C)_2, B \to b(D)_2, C \to c, D \to d\}.$$

$G$ generates the non context-free language $\{a^n b^n c^n d^n \mid n \geq 1\}$. A derivation of $aabbccdd$ is $(A)_1 \Rightarrow_G a(AB)_1(C)_2 \Rightarrow_G aa(BB)_1(CC)_2 \Rightarrow_G aab(B)_1(CCD)_2 \Rightarrow_G aabb(CCDD)_2 \Rightarrow_G aabbc(CDD)_2 \Rightarrow_G aabbcc(DD)_2 \Rightarrow_G aabbccd(D)_2 \Rightarrow_G aabbccdd$.

Many properties of GCG grammars have been studied in the past. For each grammar, it is possible to define an equivalent accepting machine, called mono-static multi-stack-queue automaton [BCCC92]. The languages generated by GCG are semilinear (this is obvious, since the language generated by a GCG is a set of suitable permutations of words of a context-free language, which is because the effect of multiple stacks and queues is to only change the context-free derivation order). Closure properties and parsing complexity have also been extensively studied: for every $k > 0$, the class of languages accepted by a GCG with rewriting disposition $S^k$ is a full abstract family of languages (AFL) [CBCC96], and the word problem can be deterministically solved in polynomial-time [CS96]. However, when the rewriting disposition includes a queue, most closure properties do not hold. For instance, there is no closure under intersection with regular languages. Also, GCGs have been compared with other formalisms [CS00], such as tree adjoining grammars.

## 6.2 Multi-queue-stack Machines

In this section, we propose a machine model, called multi-queue-stack machines, which are equipped with a number of queues and stacks. Notice that mono-static multi-stack-queue automata [BCCC92] can be considered as multi-queue-stack machines with one state. We will show that languages accepted by multi-queue-stack machines are semilinear. This result will be used in Section 6.3 to show the main result of the first part of this chapter.

A finite state machine augmented with a number of queues and stacks can achieve Turing computing power if we do not apply significant restrictions on the queue and stack behaviors. In order to ensure that multi-queue-stack machines accept only semilinear languages, two restrictions are added to the machine model:

- When a pop from either a queue (or a stack) occurs, all the queues and stacks before the queue (or the stack) must be empty,
- A pop from either a queue (or a stack) can occur only when the machine is in a specific state (called the final state, but note that the machine does not necessarily terminate at this state.). This condition essentially requires that a pop is stateless.

In the rest of this section, we first formally define a multi-queue-stack machine. Then, we show that languages accepted by multi-queue-stack machines are semilinear.

A *Multi-queue-stack Machine (MQSM)* $M$ with $n$ (FIFO) queues, $m$ (LIFO) stacks and a one-way input tape is a tuple

$$\langle S, \Sigma, \Gamma, \Theta, s_0, s_f, \delta, Q_1, \cdots, Q_n, C_1, \cdots, C_m \rangle,$$

where $S$ is a finite set of states with the *initial state* $s_0 \in S$ and the *final state* $s_f \in S$, $\Sigma$ is the input alphabet, $\Gamma$ and $\Theta$ are two disjoint alphabets for the queues $Q_1, \cdots, Q_n$ and the stacks $C_1, \cdots, C_m$ respectively. The queues and the stacks are arranged so that the queues $Q_1, \cdots, Q_n$ are followed by the stacks $C_1, \cdots, C_m$. $\delta$ is a finite set of *transitions*. We distinguish three kinds of transitions. They are push-transitions (pushing symbols to stacks and queues), pop-queue-transitions (popping the top symbol from the first nonempty queue), and pop-stack-transitions (popping the top symbol from the first nonempty stack, and all the queues, preceding all the stacks, must be empty). Both pop-queue-transitions and pop-stack-transitions happen only at the final state $s_f$. Formally the three kinds of transitions are defined as below.

A *push-transition* has the form $\langle s, \sigma, (\eta_1, \cdots, \eta_n, \xi_1, \cdots, \xi_m), s' \rangle$. That is, from state $s \in S$, $M$ moves its input head to the right and reads an input symbol $\sigma \in \Sigma$ (if $\sigma = \epsilon$, however, $M$'s input head does not move, i.e., $M$ executes an $\epsilon$-move), puts $\eta_1, \cdots, \eta_n \in \Gamma^*$ at the end of queues $Q_1, \cdots, Q_n$, and pushes $\xi_1, \cdots, \xi_m \in \Theta^*$ onto the stacks $C_1, \cdots, C_m$.

A *pop-queue-transition* has the form of $\langle s_f, \sigma, \gamma, s' \rangle$. That is, from the final state $s_f$, on the input symbol $\sigma \in \Sigma \cup \{\epsilon\}$, $M$ pops the top of the first nonempty queue and transits to state $s' \in S$.

A *pop-stack-transition* has the form $\langle s_f, \sigma, \theta, s' \rangle$. That is, from the final state $s_f \in S$, on the input symbol $\sigma \in \Sigma \cup \{\epsilon\}$, $M$ pops the top of the first nonempty stack (at this moment, all the queues must be empty.) and transits to state $s' \in S$.

A *configuration* of $M$ is a tuple $\langle s, w; \gamma_1, \cdots, \gamma_n; c_1, \cdots, c_m \rangle$, where $s \in S$ is the state, $w \in \Sigma^*$ is the input word, $\gamma_1, \cdots, \gamma_n \in \Gamma^*$ are the contents of the queues (with the leftmost character being the head and the rightmost character being the tail), $c_1, \cdots, c_m \in \Theta^*$ are the contents of the stacks (with the leftmost character being the top and the rightmost character being the bottom). The one-step transition $\Rightarrow_M$ of $M$ is a binary relation over configurations, which can be defined in an obvious way.

The *transition relation* $\Rightarrow_M^*$ is the transitive closure of the binary relation $\Rightarrow_M$ over configurations. A string $w \in \Sigma^*$ is *accepted* by $M$ if $M$ reaches $s_f$ and has read the entire input word with all the stacks and queues empty; i.e.,

$$\langle s_0, w; \gamma_0, \epsilon, \cdots, \epsilon; \epsilon, \cdots, \epsilon \rangle \Rightarrow_M^* \langle s_f, \epsilon; \epsilon, \cdots, \epsilon; \epsilon, \cdots, \epsilon \rangle.$$

The following theorem states that languages accepted by multi-queue-stack machines are semilinear. The proof of this result is based on the fact that the language accepted by MQSM may be generated by a Generalized Context-free Grammar. In fact, an MQSM is actually a GCG in disguise. Since GCGs only generate suitable permutations of context-free languages, and hence their languages are semilinear and the semilinear sets are effectively constructible. The actual grammar used to simulate an MQSM is given in the proof.

**Theorem 16** *Languages accepted by multi-queue-stack machines are semilinear.*

*Proof.* See Section 6.8.2. ∎

This result will be used later to show that the binary reachability of a multi-queue discrete timed automaton is semilinear by constructing a multi-queue discrete timed automaton to accept the binary reachability.

## 6.3 Multi-queue Discrete Timed Automata

In this section, we propose multi-queue discrete timed automata, a variant of discrete timed automata augmented with a number of queues. A possible form of a multi-queue discrete timed automaton is to couple a discrete timed automaton with a multi-queue automaton (that is a multi-queue-stack machine without pushdown stacks). However, the resulting machine has Turing computing power. The reason is that a pop is no longer stateless since the pop may depend on the result of clock comparisons in an enabling condition of the discrete timed automaton. In order to ensure the binary reachability of a multi-queue discrete timed automaton is semilinear, we further require that a pop is stateless. That is, a pop only happens at the final state and after the pop all the state information (including clock values) before the pop is lost. Therefore, we introduce a restart transition that fires only at the final state and resets all the clocks to 0. A restart transition, by popping the top symbol of the first nonempty queue, transits from the final state to a state only depending on the top symbol. The dependency is given as the restart set in the discussion below. The formal definition is as follows.

A *Multi-queue Discrete Timed Automaton (MQDTA)* with $n \geq 0$ FIFO queues is a tuple

$$\langle S, X, \Gamma, s_f, \mathbf{R}, E, Q_1, \cdots, Q_n \rangle$$

where $S$ is a finite set of *(control) states,* $X$ is a finite set of *clocks* with values in $\mathbf{Z}^+$, $\Gamma$ is the queue alphabet, $s_f \in S$ is the *final state*, $\mathbf{R} \subseteq \Gamma \times S$ is the *restart set*, $E$ is a finite set of edges, and $Q_1, \cdots, Q_n$ are queues. Intuitively, for a pair $(\gamma, s) \in \mathbf{R}$, $s$ will be the next start state of $\mathcal{A}$ after popping the top symbol $\gamma$ of the first nonempty queue at the final state $s_f$. An edge in $E$ is exactly an edge in a discrete timed automaton augmented with a number of words that will be pushed at the end of the queues. That is, each edge $e \in E$ is in the form of $\langle s, \lambda, (\eta_1, \cdots, \eta_n), l, s' \rangle$ where $s, s' \in S$ with $s \neq s_f$ (the final state $s_f$ does not have a successor), $\lambda \subseteq X$ is the set of clock resets, $l \in \mathcal{L}_X$ is the enabling condition. The queue operation is characterized by a tuple $(\eta_1, \cdots, \eta_n) \in (\Gamma^*)^k$ to denote that each $\eta_i$ is put at the end of the queue $Q_i$, $1 \leq i \leq n$.

The semantics is defined as follows. A *configuration* $\alpha$ of $\mathcal{A}$ is a tuple $\langle s, \pi_1, \cdots, \pi_n, c_1, \cdots, c_k \rangle$ where $s \in S, c_1, \cdots, c_k \in \mathbf{Z}^+$ being the state and the clock values, respectively. $\pi_1, \cdots, \pi_n \in \Gamma^*$ are the contents of each queue, with the leftmost character being the head and the rightmost character being the tail. We use $\alpha_{Q_i}$ to denote the content of queue $\pi_i$ in $\alpha$, with $\alpha_{\mathbf{q}}$, $\alpha_{x_1}, \cdots, \alpha_{x_k}$ to denote the state $s$, the clock values $c_1, \ldots, c_k$, respectively.

Let $\alpha \overset{\langle s,\lambda,(\eta_1,\cdots,\eta_n),l,s'\rangle}{\longrightarrow} \alpha'$ denote a one-step transition along an edge $\langle s, \lambda, (\eta_1, \cdots, \eta_n), l, s'\rangle$ in $\mathcal{A}$. The effect of this one-step transition changes clock values exactly as in a discrete timed automaton. In addition, the transition also pushes each word $\eta_i$ into queue $Q_i$, $1 \leq i \leq n$. Formally, the one-step transition satisfies the following conditions:

- The state $s$ is set to a new location $s'$, i.e., $\alpha_{\mathbf{q}} = s, \alpha'_{\mathbf{q}} = s'$.
- Each clock changes according to $\lambda$. If there are no clock resets on the edge, i.e., $\lambda = \emptyset$, then all clocks progress by one time unit, i.e., for each $x \in X$, $\alpha'_x = \alpha_x + 1$. If $\lambda \neq \emptyset$, then for each $x \in \lambda$, $\alpha'_x = 0$ while for each $x \notin \lambda$, $\alpha'_x = \alpha_x$.
- The enabling condition is satisfied, i.e., $l(\alpha)$ is true.
- Each word $\eta_i$ is put at the end of queue $Q_i$; i.e., $\alpha'_{Q_i} = \alpha_{Q_i}\eta_i$ for each $1 \leq i \leq n$.

In addition to the above defined one-step transition, an MQDTA $\mathcal{A}$ can fire a restart transition when it is in the final state $s_f$. A restart transition pops the first nonempty queue and resets all the clocks to zero. Formally, a *restart transition* $\alpha \overset{restart}{\longrightarrow} \alpha'$ in $\mathcal{A}$ satisfies the following conditions:

- $\alpha_{\mathbf{q}} = s_f$, i.e., this restart transition only fires at the final state.
- Let $\gamma$ be the head of the first nonempty queue (assume the queue is $Q_j$ for some $1 \leq j \leq n$.). Assume $\alpha_{Q_j} = \gamma\pi$ for some $\pi \in \Gamma^*$. Then, $\alpha'_{Q_j} = \pi$, and for all $1 \leq i \leq n$ with $i \neq j$, $\alpha'_{Q_i} = \alpha_{Q_i}$. That is, the head $\gamma$ must be removed from the queue, while the other queues are not modified.
- The next state $\alpha'_{\mathbf{q}}$, depending on the symbol $\gamma$, should be indicated in the restart set $\mathbf{R}$. That is, $(\gamma, \alpha'_{\mathbf{q}}) \in \mathbf{R}$.
- Clocks are reset, i.e., $\alpha'_x = 0$ for all $x \in X$.

We simply write $\alpha \to_{\mathcal{A}} \alpha'$ if $\alpha$ can reach $\alpha'$ by either a one-step transition or a restart transition. The *binary reachability* $\leadsto^{\mathcal{A}}$ is the reflexive and transitive closure of $\to_{\mathcal{A}}$.

From now on, $\mathcal{A}$ is an MQDTA specified as above. A configuration $\alpha = \langle s, \pi_1, \cdots, \pi_n, c_1, \cdots, c_k\rangle$ can be encoded as a string $[\alpha]$ by concatenating the symbol representation of the state $s$, the strings of the queue contents $\pi_1, \cdots, \pi_n$, and the (unary) string representation of the clock values $c_1, \cdots, c_k$ with a delimiter "\$". The binary reachability $\leadsto^{\mathcal{A}}$ can be considered as the language: $\{[\alpha]\$[\beta] : \alpha \leadsto^{\mathcal{A}} \beta\}$. The main result of this part of the chapter is to show that $\leadsto^{\mathcal{A}}$ is a semilinear language, which is done in the next section.

## 6.4 Semilinearity Results

Let $\mathcal{A} = \langle S, \{x_1, \cdots, x_k\}, \Gamma, s_f, \mathbf{R}, E, Q_1, \cdots, Q_n\rangle$ be an MQDTA with clocks $x_1, \cdots, x_k$ and queues $Q_1, \cdots, Q_n$. The enabling conditions on edges, are Boolean combinations of $x_i \# c$, $x_i - x_j \# c$ with $c$ an integer. In Section 4.5, a technique is used to eliminate these clock comparisons in a discrete timed automaton based on the definition of a finite table lookup $a_{ij} \# c$ and $b_i \# c$ to replace tests $x_i - x_j \# c$ and $x_i \# c$. It can easily be seen that exactly the same technique can be applied to the multi-queue discrete timed automaton $\mathcal{A}$. A little attention must be paid to the final state $s_f$ in $\mathcal{A}$. However, since after a restart transition fires at $s_f$ all the clocks reset to 0, all the finite state variables $a_{ij}$ and $b_i$ (representing $x_i - x_j$ and $x_i$) are set to 0. By replacing the tests in $\mathcal{A}$, the resulting automaton is denoted as $\mathcal{A}'$. It is noted that $\mathcal{A}'$ is a multi-queue discrete timed automaton but without clock comparisons. A completely analogous argument can be established that $\mathcal{A}'$ simulates $\mathcal{A}$ faithfully.

Finite state variables $a_{ij}$ and $b_i$ can be built into the states of $\mathcal{A}'$, by expanding the number of states in $\mathcal{A}'$. However, since firing a restart transition at the final state $s_f$ does not depend on the actual clock values (hence, neither on $a_{ij}$ nor on $b_i$), the final state $s_f$ will not be expanded. Thus, $\mathcal{A}'$, after expanding the states, is a multi-queue discrete timed automaton with all the enabling conditions being simply *true*. Such an $\mathcal{A}'$ is called a *static* multi-queue discrete timed automaton.

Thus, in order to prove that the binary reachability of multi-queue discrete timed automata is semilinear, it suffices to show that the binary reachability of static multi-queue discrete timed automata is semilinear.

**Theorem 17** *Given a static multi-queue discrete timed automaton $\mathcal{A}$. The language $\leadsto^{\mathcal{A}}$ is semilinear.*

*Proof.* See Section 6.8.3. ∎

Since static multi-queue discrete timed automata are able to simulate multi-queue discrete timed automata as mentioned before, we can establish the following main theorem.

**Theorem 18** $\leadsto^{\mathcal{A}}$ *is a semilinear language for any multi-queue discrete timed automaton $\mathcal{A}$.*

An MQDTA $\mathcal{A}$ has no input tape, i.e., there are no event labels on edges. However, if each edge is labeled, we can extend the states of $\mathcal{A}$ by combining a state with a label. In this case, a configuration contains only the current event label instead of the whole input word consumed. This may make applications more convenient to deal with, though all results still hold.

## 6.5 Verification Results

In this section, we formulate properties that can be verified for an MQDTA. Given an MQDTA $\mathcal{A}$, let $\alpha, \beta \cdots$ denote variables ranging over configurations. Let $\alpha_{\mathbf{q}}$ (state variables), $\alpha_{x_i}$ (clock value variables) and $\alpha_{Q_j}$ (queue content variables) denote the state, the clock $x_i$'s value and the content of the queue $Q_j$ of $\alpha$, respectively, for $1 \leq i \leq k, 1 \leq j \leq n$. We use a count variable $\#_\gamma(\alpha_{Q_j})$ to denote the number of occurrences of a character $\gamma \in \Gamma$ in the content of the queue $Q_j$ in $\alpha$, $1 \leq j \leq n$. An MQDTA-term $t$ is defined as follows:

$$t \ ::= \ n \mid \alpha_{\mathbf{q}} \mid \alpha_{x_i} \mid \#_\gamma(\alpha_{Q_j}) \mid t - t \mid t + t$$

where $n$ is an integer, $\gamma \in \Gamma$, $1 \leq i \leq k, 1 \leq j \leq n$. An MQDTA-formula $f$ is defined as follows:

$$f \ ::= \ t > 0 \mid t \bmod n = 0 \mid \neg f \mid f \vee f$$

where $n \neq 0$ is an integer. Thus, $f$ is a quantifier-free Presburger formula over control state variables, clock value variables and count variables. For $m \geq 1$, let $F$ be a formula in the following format:

$$\bigvee_{1 \leq i \leq m} (f_i \wedge \ \alpha^i \leadsto^{\mathcal{A}} \beta^i)$$

where each $f_i$ is an MQDTA-formula and all $\alpha^i$ and $\beta^i$ are configuration variables. Let $\exists F$ be a closed formula such that each free variable in $F$ is existentially quantified. Then, the property $\exists F$ can be verified.

**Theorem 19** *The truth value of $\exists F$ with respect to a multi-queue discrete timed automaton $\mathcal{A}$ is decidable for any MQDTA-formula $F$.*

*Proof.* See Section 6.8.4. ∎

For instance, the following property: "for all configurations $\alpha$ and $\beta$ with $\alpha \leadsto^{\mathcal{A}} \beta$, clock $x_2$ in $\beta$ is the sum of clocks $x_1$ and $x_2$ in $\alpha$, and symbol $\gamma_1$ appears in the first queue $Q_1$ in $\beta$ twice as many times as symbol $\gamma_2$ does in the second queue $Q_2$ in $\alpha$." can be expressed as, $\forall\alpha\forall\beta(\alpha \leadsto^{\mathcal{A}} \beta \rightarrow (\beta_{x_2} = \alpha_{x_1} + \alpha_{x_2} \wedge \#_{\gamma_1}(\beta_{Q_1}) = 2\#_{\gamma_2}(\alpha_{Q_2})))$. The negation of this property is equivalent to $\exists F$ for some MQDTA-formula $F$. Thus, it can be verified.

## 6.6 Reversal-bounded Multi-counter Multi-pushdown Machines

A $k$-multi-counter $m$-multi-pushdown machine is a nondeterministic machine with a one-way (read-only) input tape, a finite control, $k$ counters (each stores non-negative integers), and $m$ pushdown stacks. The stacks are ordered from left to right: they are denoted by $S_1, \cdots, S_m$, where each $S_i$ is called the $i$-th stack. The machine performs the following actions with one move:

- move the head one cell right (read an input symbol), or stay in the same position ($\epsilon$-move),
- switch the internal state of the finite control,

- test a counter against 0,
- increment a counter by 0, -1, +1,
- update pushdown stacks by reading (popping) the top symbol from the first (from $S_1$ to $S_m$) nonempty stack and pushing in parallel $m$ stack words (possibly empty) to stacks.

The formal definition below is an extension of the multi-pushdown automaton studied in [CBCC96]. This automaton was conceived to overcome the well-known fact that an automaton with two pushdown stacks can simulate a Turing Machine. In particular, the pushdown stacks are ordered from left to right and it is possible to read (pop) only the top of the leftmost nonempty pushdown stack. With this restriction, the languages accepted by multi-pushdown machines are always semilinear. Adding reversal-bounded counters actually increases the languages accepted by a multi-pushdown machine.

Formally, a *$k$-multi-counter $m$-multi-pushdown machine* (MCPM) $M$ is a tuple

$$\langle Q, \Sigma, \Gamma, q_0, q_f, Z_0, \delta, k, m \rangle$$

where $Q$ is a finite non-empty set of internal states, $\Sigma$ is a finite input alphabet, $\Gamma$ is a finite stack alphabet, $q_0, q_f \in Q$ are the initial state and the final state respectively, $Z_0 \in \Gamma$ is the initial stack symbol, and $\delta$ is a partial transition mapping

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times \{0,1\}^k \to \mathcal{P}_F(Q \times \{-1, 0, +1\}^k \times (\Gamma^*)^m)$$

where $\mathcal{P}_F(E)$ is the set of the finite subsets of a set $E$.

We use $\mathbf{N}$ to indicate non-negative integers. A configuration $\alpha$ of $M$ is a tuple

$$\langle q, \sigma; c_1, \cdots, c_k; \gamma_1, \cdots, \gamma_m \rangle$$

where $q \in Q$, $\sigma \in \Sigma^*$, $c_1, \cdots, c_k \in \mathbf{N}$, $\gamma_1, \cdots, \gamma_m \in \Gamma^*$. We use $\alpha_{\mathbf{q}}$, $\alpha_{x_i}$, $\alpha_{S_j}$ to indicate the state $q$, the counter value $c_i$ of counter $x_i$, and the stack word $\gamma_j$ of the stack $S_j$, $1 \le i \le k, 1 \le j \le m$. A one-step transition $\to_M$ between configurations is defined, for some $1 \le i \le m$,

$$\langle q, a\sigma; c_1, \cdots, c_k; \epsilon, \cdots, \epsilon, A\gamma_i, \cdots, \gamma_m \rangle \to_M \langle q', \sigma; c'_1, \cdots, c'_k; \tau_1, \cdots, \tau_{i-1}, \tau_i\gamma_i, \cdots, \tau_m\gamma_m \rangle$$

if $(q', d_1, \cdots, d_k, \tau_1, \cdots, \tau_m) \in \delta(q, a, A, e_1, \cdots, e_k)$ satisfying:

- $a \in \Sigma \cup \{\epsilon\}$,
- for each $1 \le j \le k$, $e_j = 1$ if $c_j = 0$, and $e_j = 0$ if $c_j \ne 0$,
- for each $1 \le j \le k$, $c'_j = c_j + d_j$. Notice that each $d_j \in \{-1, 0, +1\}$.

Thus, the above transition does the following:

- reads a symbol $a$ (could be $\epsilon$) from the input,
- reads the top symbol $A$ from the first non-empty stack $S_i$,
- tests each counter $x_j$ against 0 and the result should be consistent with $e_j$, $1 \le j \le k$,
- increments the counters $x_j$ by $d_j$, $1 \le j \le k$,
- writes (push) $\tau_1, \cdots, \tau_m$ to the stacks.

Denote with $\leadsto^M$ the transitive and reflexive closure of $\rightarrow_M$. A string $\sigma \in \Sigma^*$ is *accepted* by $M$ iff

$$\langle q_0, \sigma; 0, \cdots, 0; Z_0, \epsilon, \cdots, \epsilon \rangle \leadsto^M \langle q_f, \epsilon; 0, \cdots, 0; \gamma_1, \cdots, \gamma_m \rangle.$$

Thus, we require that all the counters go back to 0 on acceptance. The language accepted by $M$ is denoted by $L(M)$.

A counter is *$r$-reversal-bounded* if it changes mode between nondecreasing and nonincreasing at most $r$ times. A $k$-multi-counter $m$-multi-pushdown machine $M$ can be effectively made reversal-bounded, since $M$ always "knows" (i.e., it stores in the finite control) when a counter reverses by detecting the increment changes from -1 to +1 or vice versa.

From now on in this section, $M$ is a reversal-bounded $k$-multi-counter $m$-multi-pushdown machine specified as above. With one pushdown stack, $M$ is a reversal-bounded counter pushdown machine, and it is known that the emptiness problem is decidable [I78]. In contrast, $M$ without counters is exactly a multi-pushdown automaton [CBCC96], and it is known that it accepts a semilinear language.

**Theorem 20** *Languages accepted by multi-counter multi-pushdown machines (MCPMs) without counters are effective semilinear languages. Therefore, the emptiness problem for MCPMs without counters is decidable [CBCC96].*

The following theorem states that the decidability of emptiness also holds when $M$ has both reversal-bounded counters and multi-pushdown stacks, by showing that $L(M)$ is an effective semilinear language.

**Theorem 21** *Given a reversal-bounded $k$-multi-counter $m$-multi-pushdown machine $M$, $L(M)$ is an effective semilinear language. Thus, the emptiness problem for reversal-bounded $k$-multi-counter $m$-multi-pushdown machines is decidable.*

*Proof.* See Section 6.8.5. ∎

The following corollaries state a few properties of the class of languages accepted by reversal-bounded MCPMs.

**Corollary 1** *For every reversal-bounded $k$-multi-counter $m$-multi-pushdown machine $M$ and for every regular language $R$, a reversal-bounded multi-counter multi-pushdown machine $\hat{M}$ can be constructed such that $L(\hat{M}) = L(M) \cap R$.*

*Proof.* See Section 6.8.6. ∎

**Corollary 2** *Let $\mathcal{L}$ be the class of languages accepted by reversal-bounded multi-counter multi-pushdown machines (MCPMs). Then*

1. *$\mathcal{L}$ is the smallest class of languages such that: a) it contains the languages accepted by MCPMs without counters; b) it is closed under homomorphism and intersection with semilinear commutative languages.*

2. *$\mathcal{L}$ is closed under permutation.*

*Proof.* See Section 6.8.7. ∎

If $M$ does not have an input tape, i.e., $\Sigma = \emptyset$, we are more interested in the reachability between configurations. In this case, $\rightsquigarrow^M$ is called the *binary reachability* of $M$ without input tape. A configuration of this $M$, $\alpha = \langle q, \epsilon; c_1, \cdots, c_k; \gamma_1, \cdots, \gamma_m \rangle$ can be encoded as a string $[\alpha]$ by concatenating the symbol representation of $q$, the (unary) string representation of $c_1, \cdots, c_k$, and the strings $\gamma_1, \cdots, \gamma_m$ with a delimiter "\$". The binary reachability $\leadsto$ can be considered as the language: $\{[\alpha]^r\$[\beta] : \alpha \leadsto \beta\}$, where $[\alpha]^r$ is the encoding of $[\alpha]$ with each stack word reversed.

**Theorem 22** *Given a reversal-bounded multi-counter multi-pushdown machine (MCPM) $M$ without input, $\leadsto^M$ can be accepted by a reversal-bounded MCPM with an input tape.*

*Proof.* See Section 6.8.8. ∎

Since the construction of $M'$ in the proof of Theorem 22 is effective, it follows immediately, from Theorem 21, that:

**Corollary 3** *Given a reversal-bounded multi-counter multi-pushdown machine $M$ without input, $\leadsto^M$ is an effective semilinear language.*

## 6.7 Piecewise-monotonic Counters

The results in the previous section also hold when counters are integer-valued instead of nonnegative integer valued, since the sign can be built into the finite control. Thus, from now on, we consider integer-valued counters.

A *multi-reversal-bounded multi-piecewise-monotonic multi-blind counter machine (MMMCM)* $M$ is a tuple

$$\langle Q, \Sigma, \delta, q_0, q_f, D, k, m \rangle$$

where

- $M$ has a set of $k$ reversal-bounded counters (denoted by $x_i, 1 \le i \le k$), and an array of counters (denoted by $y_j$, $1 \le j \le m$), each $y_j$ is either a piecewise-monotonic counter or a blind counter (we will make this clear later) plus an unrestricted counter $y_0$. A disposition function $D$ is used to indicate the status of each $y_j$, $1 \le j \le m$; that is, $D(j) = 1$ iff $y_j$ is a piecewise-monotonic counter. We further require that piecewise-monotonic counters precede blind counters in the ordering: for all $1 \le j_1 < j_2 \le m$, if $D(j_1) \ne 1$ then $D(j_2) \ne 1$.

- $Q$ is the finite (non-empty) set of states.
- $M$ has a one-way input tape with $\Sigma$ being the input alphabet.
- $q_0, q_f \in Q$ are the initial state and the final state, respectively.
- $\delta$ is a finite subset of transitions. Each transition transits from state $q$ to state $q'$ when reading symbol $a \in \Sigma \cup \{\epsilon\}$ on the input tape, and takes one of the following forms with $1 \leq i \leq k, 0 \leq j \leq m, p \in \{-1, 0, +1\}, b \in \{0, 1\}$,
  - an *increment* transition, $\mathbf{Incr}(q, q', a, x_i, p)$ for reversal-bounded counter $x_i$ or $\mathbf{Incr}(q, q', a, y_j, p)$ for piecewise-monotonic counter (or blind counter) $y_j$. The effect is to increment the counter by $p \in \{-1, 0, +1\}$,
  - a *test* transition, $\mathbf{Test}(q, q', a, x_i, b)$ for reversal-bounded counter $x_i$ or $\mathbf{Test}(q, q', a, y_j, b)$ with either $j = 0$ or $D(j) = 1$ (i.e., $y_j$ must be a piecewise-monotonic counter. Thus, a blind counter can not be tested against 0). $b$ is the result of the counter value tested against 0,
  - an *append* transition, $\mathbf{Append}(q, q', a, y_{j_1}, y_{j_2})$ with $0 \leq j_1 < j_2 \leq m$. $y_{j_1}$ is not a blind counter, $y_{j_2}$ is a blind counter. The effect is: $y_{j_2} := y_{j_2} + y_{j_1}; y_{j_1} := 0$,
  - a *reset* transition, $\mathbf{Reset}(q, q', a, y_j)$ for piecewise-monotonic counter (or blind counter) $y_j$. The effect is $y_j := 0$.

A *configuration* $\alpha$ of $M$ is a tuple

$$\langle q, x; c_1, \cdots, c_k; d_0, d_1, \cdots, d_m \rangle$$

where $q \in Q, x \in \Sigma^*$, $c_1, \cdots, c_k$ are counter values for $x_1, \cdots, x_k$, and $d_0, d_1, \cdots, d_m$ are counter values for $y_0, y_1, \cdots, y_m$. We use $\alpha_{\mathbf{q}}, \alpha_{x_i}, \alpha_{y_j}$ to denote $q, c_i, d_j$, respectively, $1 \leq i \leq k$, $0 \leq j \leq m$. A one-step transition $\rightarrow$ is a binary relation over configurations, which can be defined in an obvious way. The transitive and reflexive closure of $\rightarrow$ is denoted by $\rightsquigarrow^M$. In any computation, we further require that:

- Each reversal-bounded counter is $r$-reversal-bounded where $r \geq 0$ is independent of the computation.

- Each piecewise-monotonic counter is piecewise-monotonic, i.e., it does not reverse before going back to 0 (by a reset transition, or after an append transition).

$M$ can be effectively made to fulfill both of the above requirements by adding a number of Boolean variables to detect whether a counter reverses. Thus, we assume, in this section, that all computations of $M$ already guarantee the requirements. In this case, $\rightsquigarrow^M$ characterize exactly the desired computations.

A word $x \in \Sigma^*$ is *accepted* by $M$ if

$$\langle q_0, x; 0, \cdots, 0; 0, \cdots, 0\rangle \rightsquigarrow^M \langle q_f, \epsilon; c_1, \cdots, c_k; d_0, d_1 \cdots, d_m\rangle.$$

Denote with $L(M)$ the language accepted by $M$. The languages accepted by MMMCMs are semilinear as shown in the following theorem.

**Theorem 23** *Given a multi-reversal-bounded multi-piecewise-monotonic multi-blind counter machine (MMMCM) $M$ specified as above, $L(M)$ is an effective semilinear language. Thus, the emptiness problem for MMMCMs is decidable.*

*Proof.* See Section 6.8.9. ∎

As before, we may consider $M$ without an input tape; i.e., $\Sigma = \emptyset$. Without the input tape content in the configurations, $\rightsquigarrow^M$ is a tuple language; i.e., it consists of integer tuples. Thus, $\rightsquigarrow^M$ is a semilinear language iff $\rightsquigarrow^M$ is definable as a Presburger formula. The following theorem states that $\rightsquigarrow^M$ is actually a semilinear language.

**Theorem 24** *The binary reachability $\rightsquigarrow^M$ for a multi-reversal-bounded multi-piecewise-monotonic multi-blind counter machine $M$ without an input tape is an effective semilinear language; i.e., it is definable as a Presburger formula which can be effectively computed.*

*Proof.* See Section 6.8.10. ∎

Let $S$ be a set of configurations of $M$ (without an input tape) that is definable by a Presburger formula. The postimage $Post^*_M(S)$ of $S$ with respect to $M$ is defined as

$$\{\beta : \alpha \rightsquigarrow^M \beta, \alpha \in S\}.$$

The preimage $Pre^*_M(S)$ of $S$ with respect to $M$ is defined as

$$\{\alpha : \alpha \leadsto^M \beta, \beta \in S\}.$$

Both $Post^*_M$ and $Pre^*_M$ are important for the verification of safety properties of infinite state systems. Intuitively, $Post^*_M(S)$ is the set of all reachable configurations of $M$ from a configuration in $S$. In contrast, $Pre^*_M(S)$ is the set of configurations from which $M$ can reach a configuration in $S$. The following theorem gives a characterization of $Post^*_M(S)$ and $Pre^*_M(S)$.

**Theorem 25** *Let $S$ be a set of configurations of a multi-reversal-bounded multi-piecewise-monotonic multi-blind counter machine $M$ without an input tape that is definable by a Presburger formula. Then both $Post^*_M(S)$ and $Pre^*_M(S)$ are effectively definable by Presburger formulas.*

*Proof.* See Section 6.8.11. ∎

Theorem 25 already shows that verifying safety properties that are definable by Presburger formulas is decidable. To see this, let $I$ and $S$ be two sets of configurations that are definable by Presburger formulas. The *safety analysis problem* is whether starting from a configuration in $I$, $M$ always reaches a configuration in $S$. The problem is equivalent to testing the emptiness of the set

$$I \cap Pre^*_M(\neg S).$$

From Theorem 25, $Pre^*_M(\neg S)$ is definable by a Presburger formula; therefore, $I \cap Pre^*_M(\neg S)$ is also definable by a Presburger formula. Thus, this test is decidable.

## 6.8 Technical Proofs

### 6.8.1 Semilinear Sets and Languages

To help the reader, this section briefly summarizes the basic definitions of semilinear sets.

Let $\Sigma = \{a_1, \cdots, a_k\}$ be a finite alphabet. We fix an ordering of the elements, for instance the order $a_1, \cdots, a_k$. For $w \in \Sigma^*$, $p(w) = (\#_{a_1}(w), \cdots, \#_{a_k}(w))$ is a $k$-ary vector of natural numbers where $\#_{a_i}(w)$ is the number of occurrences of $a_i$ in $w$. For a language

$L \subseteq \Sigma^*$, $p(L) = \{p(w) : w \in L\}$ is the *Parikh set* of $L$, and the mapping $p$ is a *Parikh map* of $L$ [P66]. A subset of $\mathbf{Z}^r$ is called a linear set if there exists $\mathbf{v}_0, \mathbf{v}_1, \cdots, \mathbf{v}_m$ in $\mathbf{Z}^r$ such that $Q = \{\mathbf{v} : \mathbf{v} = \mathbf{v}_0 + k_1\mathbf{v}_1 + \cdots + k_m\mathbf{v}_m, k_i \in \mathbf{Z}\}$. A finite union of linear sets is called a *semilinear set*. Semilinear sets are precisely the sets definable by Presburger formulas. $L$ is a *semilinear language* if $p(L)$ is a semilinear set. The emptiness problem for semilinear languages is decidable.

A *Parikh transform* $P$ translates each $w \in \Sigma^*$ into $a_1^{\#_{a_1}(w)} \cdots a_k^{\#_{a_k}(w)}$. Let $\$ \notin \Sigma$ be a symbol. For every $n \leq 1$, a language $L$ is *segmented* if every word of $L$ has the form $w_1\$w_2 \cdots \$w_n$, where each $w_i \in \Sigma^*$. $P$ is extended to every word $\mathbf{w} = w_1\$ \cdots \$w_n$ of a segmented language $L$ in such a way that $P(\mathbf{w}) = P(w_1)\$ \cdots \$P(w_n)$, and $P(L) = \{P(\mathbf{w}) : \mathbf{w} \in L\}$. A segmented language $L$ is *locally commutative* if $\mathbf{w} \in L$ iff $P(\mathbf{w}) \in P(L)$.

**Lemma 5** *For all languages $L_1$ and $L_2$, with $L_2$ segmented and locally commutative, the following hold:*

1. *$P(L_1)$ is a semilinear language iff $L_1$ is.*
2. *If $L_1$ and $L_2$ are semilinear languages then so is $L_1 \cap L_2$.*

*Proof. 1.* Let $p$ be the Parikh mapping. Simply notice that $p(P(L_1)) = p(L_1)$.

*2.* Suppose that each word $L_2$ has $n \geq 1$ occurrences of \$, and let $C$ be the language of all the words in $(\Sigma \cup \{\$\})^*$ with exactly $n$ occurrences of \$. Let $L_3$ be the segmented language $L_1 \cap C$, which is semilinear because $L_1$ is semilinear and $C$ is obviously semilinear and commutative. Since $L_2 \subseteq C$, then $L_1 \cap L_2 = L_1 \cap (C \cap L_2) = L_3 \cap L_2$. Hence, $P(L_1 \cap L_2) = P(L_3 \cap L_2) = P(L_3) \cap P(L_2)$, since $L_2$ is locally commutative and $L_3$ is segmented. Then, from the proof of part (1) above, $p(L_1 \cap L_2) = p(P(L_1 \cap L_2)) = p(P(L_3) \cap P(L_2))$. Since elements in $P(L_3)$ and $P(L_2)$ are made of tuples, and from part (1), $P(L_3)$ and $P(L_2)$ are semilinear languages, and $P(L_3) \cap P(L_2)$ is also a semilinear language [GS64]. The result follows from part (1). ∎

### 6.8.2 Proof of Theorem 16

Theorem 16. Languages accepted by multi-queue-stack machines are semilinear.

*Proof.* Let $M = \langle S, \Sigma, \Gamma, \Theta, s_0, s_f, \delta, Q_1, \ldots, Q_n, C_1, \ldots, C_m \rangle$ be an MQSM machine with $n \geq 0$ queues and $m \geq 0$ stacks. Assume that $S, \Gamma, \Theta$ are pair-wise disjoint and that $s_0 \neq s_f$ (otherwise $M$ accepts only the empty string $\epsilon$). Let $h : S \to S - \{s_f\}$ be the homomorphism defined by $h(s) = s$ for every $s \in S, s \neq s_f$, and $h(s_f) = \epsilon$.

Let $G = (V_N, \Sigma, P, s_0, SQ^n S^m)$ be a generalized context-free grammar where $V_N = \{A\} \cup S \cup \Gamma \cup \Theta$ and $P$ is defined by the following clauses:

1. For all $a \in \Sigma \cup \{\epsilon\}$, for all $s, s' \in S$, for all $\gamma_1, \ldots, \gamma_n \in \Gamma^*$, and for all $\theta_1 \ldots \theta_m \in \Theta^*$, if $(s, a, \gamma_1, \ldots, \gamma_n, \theta_1, \ldots, \theta_m, s') \in \delta$ then in $P$ there is the production:
   $s \to a(h(s'))_1(\gamma_1)_2 \ldots (\gamma_n)_{n+1}(\theta_1)_{n+2} \ldots (\theta_m)_{n+m+2}$
2. For all $a \in \Sigma \cup \{\epsilon\}$, for all $\gamma \in \Gamma$, and for all $s \in S$, if $(s_f, a, \gamma, s) \in \delta$, then the production $\gamma \to a(h(s))_1$ is in $P$.
3. For all $a \in \Sigma \cup \{\epsilon\}$, for all $\theta \in \Theta$, and for all $s \in S$, if $(s_f, a, \theta, s) \in \delta$, then the production $\theta \to a(h(s))_1$ is in $P$.
4. No other production is in $P$.

In $G$, the first additional stack is introduced in order to simulate the state of $M$, which is always saved as the only symbol on the stack. When $M$ enters the final state $s_f$, then $s_f$ itself is not put on the stack, but the stack is left empty: hence, it is possible to execute a pop move from the other tapes. We prove that every derivation of $G$ faithfully simulates a run of $M$. The initial configuration of the components of $G$ is the same as the initial configuration of $M$. It is enough to prove by induction on $p \geq 0$ that for all $w, w' \in \Sigma^*$, for all $s, s' \in S$, for all $\gamma_1 \ldots \gamma_n, \gamma'_1 \ldots \gamma'_n \in \Gamma^*$, and for all $\theta_1 \ldots \theta_m, \theta'_1 \ldots \theta'_m \in \Theta^*$, if

$$\langle s, ww'; \gamma_1, \ldots, \gamma_n; \theta_1, \ldots, \theta_m \rangle \Rightarrow^p_M \langle s', w'; \gamma'_1, \ldots; \gamma'_n, \theta'_1, \ldots, \theta'_m \rangle$$

then:

$$(h(s))_1(\gamma_1)_2 \ldots (\gamma_n)_{n+1}(\theta_1)_{n+2} \ldots (\theta_m)_{n+m+2} \Rightarrow^p_G$$
$$w(h(s'))_1(\gamma'_1)_2 \ldots (\gamma'_n)_{n+1}(\theta'_1)_{n+2} \ldots (\theta'_m)_{n+m+2}.$$

The base step $p = 0$ is obvious. If $p > 0$ then there are two cases:

1. If $s \neq s_f$ then, after the first step, for some $g_1, \ldots, g_n \in \Gamma^*$, $t_1, \ldots, t_m \in \Theta^*$, $s_1 \in S$ and for some $a \in \Sigma \cup \{\epsilon\}$ and $w_1 \in \Sigma^*$ such that $w = aw_1$, the configuration of $M$ is $\langle s_1, w_1 w'; g_1\gamma_1, \ldots, g_n\gamma_n; t_1\theta_1, \ldots, t_m\theta_m\rangle$. Hence, the transition

   $$(s, a, g_1, \ldots, g_n, t_1, \ldots t_m, s_1) \in \delta.$$

   By clause (1) of the definition of $P$,

   $$s \to a(h(s_1))_1 (g_1)_2 \ldots (g_n)_{n+1} (t_1)_{n+2} \ldots (t_m)_{n+m+2}$$

   is in $P$:

   $$(s)_1(\gamma_1)_2 \ldots (\gamma_n)_{n+1}(\theta_1)_{n+2} \ldots (\theta_m)_{n+m+2} \Rightarrow_G$$
   $$a(h(s_1))_1(g_1\gamma_1)_2 \ldots (g_n\gamma_n)_{n+1}(t_1\theta_1)_{n+2} \ldots (t_m\theta_m)_{n+m+2}.$$

2. If $s = s_f$ then assume that there is $\gamma_i = \gamma\gamma''$ for $\gamma \in \Gamma, \gamma'' \in \Gamma^*$, $1 \leq i \leq n$ and such that $\gamma_j = \epsilon$ for every $1 \leq j < i$. Hence, after the first step (a pop-queue transition), for some $s_1 \in S$, $a \in \Sigma \cup \{\epsilon\}$ and $w_1 \in \Sigma^*$ such that $w = aw_1$, the configuration of $M$ is $\langle s_1, w_1 w'; \epsilon, \ldots, \epsilon\gamma'', \gamma_{i+1}, \ldots, \gamma_n; \theta_1, \ldots, \theta_m\rangle$. By clause (2) of the definition of $P$, $\gamma(h(s_1)_1$ is in $P$. Hence:

   $$(\gamma\gamma'')_{i+1} \ldots (\gamma_n)_{n+1}(\theta_1)_{n+2} \ldots (\theta_m)_{n+m+2} \Rightarrow_G$$
   $$(h(s_1)_1(\gamma'')_{i+1} \ldots (\gamma_n)_{n+1}(\theta_1)_{n+2} \ldots (\theta_m)_{n+m+2}.$$

   The case where all $\gamma_i = \epsilon$, (i.e., the first step of $M$ is a pop-stack-transition) is completely analogous, using clause (3) of the definition of $P$.

If $w \in \Sigma^*$ is accepted by $M$ then:

$$\langle s_0, w; \gamma_0, \epsilon, \ldots, \epsilon; \epsilon, \ldots, \epsilon\rangle \Rightarrow_M^* \langle s_f, \epsilon; \epsilon, \ldots, \epsilon; \epsilon, \ldots, \epsilon\rangle$$

Hence,

$$A \Rightarrow_G (h(s_0))_1(\gamma_0)_1\rangle \Rightarrow_G^* w(h(s_f))_1$$

and, since $h(s_f) = \epsilon$, $w$ is generated by $G$.

To complete the proof that $L(G) = L(M)$ we have also to show that $L(G) \subseteq L(M)$, but the proof, which is completely analogous, is omitted. ∎

### 6.8.3 Proof of Theorem 17

Theorem 17. Given a static multi-queue discrete timed automaton $\mathcal{A}$. The language $\rightsquigarrow^{\mathcal{A}}$ is semilinear.

*Proof.* We will construct a multi-queue-stack machine $M$ to accept the language $\rightsquigarrow^{\mathcal{A}}$. From Theorem 16, $\rightsquigarrow^{\mathcal{A}}$ is semilinear.

$M$ uses its own queues to simulate the queues in $\mathcal{A}$, and uses its own stacks to store the clock values in $\mathcal{A}$. Given a pair of string encodings of configurations $\alpha$ and $\beta$ (separated by a delimiter "#" not in the queue alphabet of $\mathcal{A}$) of $\mathcal{A}$ on $M$'s one-way input tape, $M$ first copy $\alpha$ into its queues (to record the queue contents in $\alpha$) and into stacks (to record the clock values in $\alpha$). Thus, $M'$ input head stops at the beginning of $\beta$. $M$ starts simulating $\mathcal{A}$ from configuration $\alpha$. A push to a queue in $\mathcal{A}$ is faithfully simulated by the same push to the queue in $M$. Clock progress of a clock $x_i$ in $\mathcal{A}$ is simulated by pushing a symbol $\tau_i$ (the reason that we use different symbols $\tau_i$ for each clock will be clear later.) onto the $i$-th stack in $M$. A clock reset of a clock $x_i$ in $\mathcal{A}$ is simulated by pushing a symbol $\theta_i$ ($\theta_i$ are also different.) onto the $i$-th stack in $M$. Thus, the clock value of $x_i$ is exactly the number of symbol $\tau_i$'s from the top of the $i$-th stack down to the first "bottom" symbol $\theta_i$ in $M$ (assuming at the very beginning of the simulation when all the stacks are empty, a symbol $\theta_i$ was pushed.). When $\mathcal{A}$ pops the first nonempty queue (say the $j$-th queue), $M$ also pops the $j$-th queue of its own. Since this can happen only at the final state of $\mathcal{A}$, the final state of $M$ corresponds to the final state of $\mathcal{A}$.

So far, $M$ faithfully simulates $\mathcal{A}$. At some moment, $M$ nondeterministically guesses that configuration $\beta$ has been reached. Then $M$ checks this is the case by popping its own queues and stacks to check that the queue contents and the stack words (taken the part before the first bottom symbol $\theta_i$) are consistent with the queue contents and the clock values on the rest of the input tape. But there is a problem here. Since, in $M$, popping a queue (or a stack) happens only at the final state of $M$. The popping depends only on the top symbol of the queue (or the stack) and the symbol under the input head. Popping a queue can be proceeded by comparing the top with the input symbol–when the two are not the same, $M$ aborts (without accepting the input). But when popping a stack (say the $i$-th stack) in order to check the clock value of $x_i$ in $\beta$, $M$ must make sure that after popping the part before the first bottom symbol $\theta_i$ on the stack (at this moment the clock value of $x_i$ in $\beta$ has been compared.), popping the rest of the stack wouldn't be misunderstood as a continuation of the comparison. This can be solved by using a padding technique. That is, in the string encoding of $\beta$, the substring representing the clock value $x_i$ (we also assume that this substring was encoded using a different symbol $\phi_i$ for each $i$. The reason will be made clear in moments.) is padded by a unary string (built from a new symbol $\varphi_i$). The length $t$ of this unary string is a guess of the number of symbols on the rest of the $i$-th stack (after the first bottom symbol $\theta_i$). Thus, $M$ can pop the rest of the $i$-th stack while reading the padding word. What if the guess was wrong? If $t$ is too large or too small, $M$ may continue to pop the $(i+1)$-th stack or the input head moves to the substring for clock $x_{i+1}$. But recall we used different sets of stack symbols $\theta_i$ and $\tau_i$ for different $i$, and different sets of symbols $\phi_i$ and $\varphi_i$ to encode the clock value $x_i$ and its padding word, $M$ can always detect a wrong guess.

$M$ continues the comparisons and accepts the input when all the comparisons are successful. From Theorem 16, the language accepted by $L(M)$ is semilinear. But $L(M)$ is different from the language $\leadsto^{\mathcal{A}}$, but not too different. By applying a homomorphism mapping all the padding words to null strings, and mapping $\phi_i$ to a normal symbol (such as '1') encoding the clock value $x_i$, $L(M)$ becomes exactly the binary reachability $\leadsto^{\mathcal{A}}$. Since semilinear languages are closed under homomorphisms, $\leadsto^{\mathcal{A}}$ is semilinear. ∎

### 6.8.4 Proof of Theorem 19

Theorem 19. The truth value of $\exists F$ with respect to a multi-queue discrete timed automaton $\mathcal{A}$ is decidable for any MQDTA-formula $F$.

*Proof.* Let $L(F)$ be the language of the string encodings of the tuples of all the configurations that satisfy a MQDTA-formula $F$. Thus, $L(F) = \bigcup_i \left(L(\alpha^i \leadsto^{\mathcal{A}} \beta^i) \cap L(f_i)\right)$. We will show that $L(F)$ is a semilinear language. Since all the proofs are constructive, the semilinear set of $L(F)$ can be effectively constructed from $F$ and $\mathcal{A}$. Thus, testing whether $\exists F = false$, which is equivalent to testing the emptiness of $L(F)$, is decidable [GS64]. Since semilinearity is closed under union, without loss of generality we show that $L_1 \cap L_2$ is a semilinear language, where $L_1 = L(\alpha \leadsto^{\mathcal{A}} \beta)$ and $L_2 = L(f)$. From Theorem 18, $L_1$ is a semilinear language. Notice that $L_2$ is locally commutative, since the contents of the queues are only used by count variables. $P(L_2)$, the result of applying the Parikh transform $P$, is a semilinear language. Thus, from Lemma 5, part (1), $L_2$ is a semilinear language. Hence, $L_1 \cap L_2$ is a semilinear language from Lemma 5, part (2). ∎

### 6.8.5 Proof of Theorem 21

Theorem 21. Given a reversal-bounded $k$-multi-counter $m$-multi-pushdown machine $M$, $L(M)$ is an effective semilinear language. Thus, the emptiness problem for reversal-bounded $k$-multi-counter $m$-multi-pushdown machines is decidable.

*Proof.* It suffices to show that languages accepted by reversal-bounded MCPMs are effective semilinear languages. The decidability result follows from the well-known result that the emptiness problem is decidable for semilinear sets [GS64].

Without loss of generality, we assume each counter in $M$ makes at most 1-reversal (by introducing new counters an $r$-reversal-bounded counter can make at most 1 reversal [BB74]) and notice that counters become zero when $M$ accepts. We will use the technique in [I78] to construct an MCPM $M'$ without counters to simulate $M$. For each counter $x_i$ with $1 \leq i \leq k$, we introduce two new symbols $+_i$ and $-_i$ to indicate increasing and decreasing $x_i$ by 1. Define a homomorphism

$$g : (\Sigma \cup \{+_i, -_i : 1 \leq i \leq k\})^* \rightarrow \Sigma^*$$

with $g(+_i) = g(-_i) = \epsilon$ for $1 \leq i \leq k$ and $g(a) = a$ for $a \in \Sigma$. Since stack operations in $M$ may be exactly simulated by $M'$ on its own $m$ stacks, $M'$, working on a string $y \in (\Sigma \cup \{+_i, -_i : 1 \leq i \leq k\})^*$ faithfully simulates $M$ working on the string $x = g(y) \in L(M)$, except when $M$ updates its counters. In the latter case, suppose $M$ increases (decreases) the counter $x_i$ by 1: then $M'$ will check that the symbol under the input head is $+_i$ ( $-_i$). In order to simulate $M$ comparing $x_i$ against zero, $M'$ has a Boolean variable $s_i$ in its finite control to indicate whether $x_i$ is positive. Since $x_i$ is at most 1 reversal, $s_i$ is set to 1 if an increment has been made. However, when $x_i$ is decremented, $s_i$ is nondeterministically chosen either 1 or 0 and $M'$ will make sure that once $s_i$ is chosen 0 no occurrences of $+_i$ and $-_i$ can appear on the input tape afterwards. During the simulation, $M'$ also uses its finite control to make sure that any occurrence of $-_i$ in $y$ must appear to the right of all occurrences of $+_i$ on the input tape. It can be seen that $x \in L(M)$ iff there exists $y \in L(M')$ with $g(y) = x$ such that for all $1 \leq i \leq k$, the number of $+_i$ and the number of $-_i$ in $y$ are the same (counters go back to 0). By Theorem 20, $L(M')$ is an effective semilinear language. Define $L'$ to be the set of all strings in $(\Sigma \cup \{+_i, -_i : 1 \leq i \leq k\})^*$, with the number of $+_i$ and the number of $-_i$ being the same for each $1 \leq i \leq k$. Clearly, $L'$ is semilinear and commutative. Therefore, $L(M') \cap L'$ is an effective semilinear language [GS64]. The result follows by noticing that $g$ is a homomorphism and $L(M) = g\,(L(M') \cap L')$. (Without using $L'$, the result can be established using a technique in the proof of Theorem 2.1 in [I78].) ∎

### 6.8.6 Proof of Corollary 1

Corollary 1. For every reversal-bounded $k$-multi-counter $m$-multi-pushdown machine $M$ and for every regular language $R$, a reversal-bounded multi-counter multi-pushdown machine $\hat{M}$ can be constructed such that $L(\hat{M}) = L(M) \cap R$.

*Proof.* In the proof of Theorem 21, $g$ is a homomorphism. Thus, $g^{-1}(R)$ is regular. It is known [CBCC96] that MCPMs without counters are effectively closed under intersection with a regular language. Thus, we can construct an MCPM without counters, $M''$, accepting $L(M') \cap g^{-1}(R)$. The theorem follows from the same proof by noticing that $L(M) = g(L(M'') \cap L')$ and that, by the choice of $g$ in the proof of Theorem 21 $g(g^{-1}(R)) = R$. ∎

### 6.8.7 Proof of Corollary 2

Corollary 2. Let $\mathcal{L}$ be the class of languages accepted by reversal-bounded multi-counter multi-pushdown machines (MCPMs). Then

1. $\mathcal{L}$ is the smallest class of languages such that: a) it contains the languages accepted by MCPMs without counters; b) it is closed under homomorphism and intersection with semilinear commutative languages.

2. $\mathcal{L}$ is closed under permutation.

*Proof.* To prove (1), we first show part (b) (while part (a) is obvious): given an MCPM $M'$, a homomorphism $h$ and a semilinear commutative language $L$, $h(L(M'))$ and $L(M') \cap L$

are in $\mathcal{L}$. For $h(L(M'))$, an MCPM $M''$ is constructed such that in every move every $a \in \Sigma$ is replaced by $h(a)$. Clearly, $M''$ accepts $h(L(M'))$. For $L(M') \cap L$, notice that $L$, being semilinear and commutative, can be accepted by a deterministic reversal-bounded multi-counter machine (i.e., a reversal-bounded MCPM without pushdowns): this can be deduced from the fact [I78] that a tuple language (like the Parikh set $p(L)$ of $L$) is semilinear iff it can be accepted by a deterministic reversal-bounded multi-counter machine. Thus, a reversal-bounded MCPM $M''$ can be easily constructed by "intersecting" $M'$ with the deterministic reversal-bounded multi-counter machine. To prove that $\mathcal{L}$ is also the smallest class for which (a) and (b) hold, let $M$ be a reversal-bounded MCPM. Then, from the proof of Theorem 21, $L(M) = g(L(M') \cap L')$, for some MCPM without counters $M'$, some homomorphism $g$ and some semilinear commutative language $L'$.

To show (2), let $M$ be any reversal-bounded MCPM and let $L \subseteq \Sigma^*$ be the set of the permutations of the words of $L(M)$, where $\Sigma = \{a_1, \ldots, a_n\}$, for some $n \geq 1$. A reversal-bounded MCPM $M'$ to accept $L$ is constructed as below. With a word (that is supposedly a permutation of a word in $L(M)$) given on the input tape of $M'$, $M'$ first reads through the input word while using new counters $c_i$ to store the number of occurrences of each $a_i$, $1 \leq i \leq n$. Then $M'$ starts to simulate $M$ by guessing each input symbol for $M$ while $M$ reads the input tape. When an $a_i$ is guessed, $M'$ decrements the counter $c_i$ by 1. $M'$ accepts iff $M$ accepts and all the counters $c_i$ $(1 \leq i \leq n)$ are 0. ∎

### 6.8.8 Proof of Theorem 22

Theorem 22. Given a reversal-bounded multi-counter multi-pushdown machine (MCPM) $M$ without input, $\leadsto^M$ can be accepted by a reversal-bounded MCPM with an input tape.

*Proof.* We will construct a reversal-bounded MCPM $M'$ to accept $\leadsto^M$. With a string encoding

$$[\alpha]^r\$[\beta]$$

on its input tape, $M'$ first copies the counter values (causing at most one reversal for each counter in $M'$) and the stack contents to its own $k$ counters and $m$ stacks. Note that stack words in $[\alpha]^r$ are reversed, thus, currently the top of each stack $S_i$ in $M'$ is exactly the leftmost symbol in $\alpha_{S_i}$. At this time, $M'$'s input head is at the delimiter \$. Then $M'$

starts to simulate $M$ from the state $\alpha_{\mathbf{q}}$, using its own $k$ counters and $m$ stacks. Since counters in $M$ are reversal-bounded, so are the $k$ counters in $M'$. At some moment, $M'$ (a nondeterministic machine), guesses that $M$ has reached the configuration $\beta$. $M'$ then checks whether the current state is the same as the state $\beta_{\mathbf{q}}$ encoded in $[\beta]$ on the input tape. Then it checks whether each of its $k$ counters has the same value as the unary string encoded in $[\beta]$ (while reading the input tape and decreasing the counter: this will cause at most one reversal for each counter), and also checks whether the content of each of its $m$ pushdown stacks is the same as the stack word encoded in $[\beta]$ (while reading the input tape and popping the stack). Whenever an inconsistency is detected, $M'$ aborts (without accepting the input). Clearly, $M'$ accepts $\rightsquigarrow^M$ and $M'$ is reversal-bounded. ∎

### 6.8.9 Proof of Theorem 23

Theorem 23. Given a multi-reversal-bounded multi-piecewise-monotonic multi-blind counter machine (MMMCM) $M$ specified as above, $L(M)$ is an effective semilinear language. Thus, the emptiness problem for MMMCMs is decidable.

*Proof.* We define an MCPM $M'$ with $k$ reversal-bounded counters and $m+1$ pushdown stacks to simulate $M$. Given a word on the input tape, $M'$ starts to simulate the computations of $M$ with that word as input. $M'$ first pushes $Z_0$ to each of the $m+1$ stacks. Each counter $y_j$ ($0 \leq j \leq m$) corresponds to the $j$-th pushdown stack $S_j$ in $M'$. Each $S_j$ uses alphabet $\Gamma = \{+1, -1, Z_0\}$ where the symbol '+1' is used as the unary encoding of non-negative counter values, and '-1' is used as the unary encoding of negative counter values, $Z_0$ is the initial stack symbol. Counter operations on reversal-bounded counters $x_i$ ($1 \leq i \leq k$) in $M$ are faithfully simulated by $M'$ on its own reversal-bounded counters $\hat{x}_i$.

Operations on $y_j$ are simulated on the stack $S_j$. For the unrestricted counter $y_0$, $M'$ can use $S_0$ (the first stack) to simulate directly an increase or a decrease by a push or a pop of the stack. A test of $y_0$ is simulated by testing the top symbol of $S_0$ against $Z_0$. For every other counter $y_j$, with $j \neq 0$, increasing and decreasing is simulated by pushing '+1' or '-1' onto $S_j$. A reset of $y_j$ is simulated by simply pushing $Z_0$ onto $S_j$. Thus, $Z_0$ is considered the current "bottom" of the stack. An append transition from $y_{j_1}$ to $y_{j_2}$ is simulated by popping $S_{j_1}$ while pushing the popped symbol onto $S_{j_2}$ until $S_{j_1}$ hits the current bottom

$Z_0$; but before doing this, we will make sure all stacks before $S_{j_1}$ are emptied. Recall that according to the semantics, the append transition is firable, if all counters $y_n$, which are not blind, before $y_{j_1}$ are 0. That is, all stacks before $S_{j_1}$ have top symbol $Z_0$. Thus, we first need to empty every $S_n$ by repeatedly popping (proceeding in order from $S_0$ to $S_{j_1-1}$). After the append transition, $M'$ pushes $Z_0$ on every stack before $S_{j_1}$. Test of $y_j$ ($j \neq 0$) against 0 is a little more complicated. However, for a blind counter, this test is not allowed. Thus, we only need to discuss tests against 0 for piecewise-monotonic counters $y_j$. By definition, $y_i$ does not reverse before it goes back to 0 (by either reset or append transitions). Thus, in $S_j$, the stack word between two consecutive $Z_0$ or above the last (from bottom to top) $Z_0$ is unary; it is either all '+1's, all '-1's or $\epsilon$. Thus, without popping $S_j$, $M'$ remembers in its finite control whether a '+1' or '-1' was pushed onto $S_j$ after the last time $Z_0$ was pushed on. Then $M'$ uses this information later to test $y_j$.

State transitions and input head movements are faithfully simulated by $M'$. Clearly, $M$ accepts an input word iff $M'$ accepts it. Thus, $L(M') = L(M)$. From Theorem 21, the theorem follows. ∎

### 6.8.10 Proof of Theorem 24

Theorem 24. The binary reachability $\leadsto^M$ for a multi-reversal-bounded multi-piecewise-monotonic multi-blind counter machine $M$ without an input tape is an effective semilinear language; i.e., it is definable as a Presburger formula which can be effectively computed.

*Proof.* We will construct a reversal-bounded MCPM $M'$ to accept the language $\leadsto^M$. The proof is similar to the proof of Theorem 22. That is, with $[\alpha]\$[\beta]$ (now they are integer tuples) on the input tape, $M'$, as in the proof of Theorem 23, uses its own reversal-bounded counters and $m+1$ pushdown stacks to simulate $M$'s reversal-bounded counters, the unrestricted counter, and the piecewise-monotonic and blind counters. $M'$ first push $Z_0$ to each of the $m+1$ pushdown stacks, and sets counter values and stack contents by reading $[\alpha]$. At this time, $M'$ starts to simulate $M$ starting from configuration $\alpha$, as shown in the proof of Theorem 23. At some time, $M'$ guesses that $M$ has reached $[\beta]$. Then, $M'$ starts to compare its reversal-bounded counter values directly against those (represented as a unary string in $[\beta]$) on the input tape while reading the input tape. Comparing the $i$-th stack $S_i$ with $\beta_{y_i}$ in $[\beta]$ can be done by popping $S_i$ up to the first $Z_0$ while reading the input tape. Doing this is fine with the unrestricted counter $y_0$ and all the piecewise-monotonic counters, since, as we pointed out in the proof of Theorem 23, before the first $Z_0$ the stack word is unary. However, if $y_i$ is a blind counter, the stack word before the first $Z_0$ on $S_i$ is not unary; i.e., it is a string of '-1' and '+1' indicating the increments applied to $y_i$ after a reset. For each blind counter, $M'$ introduces two new reversal-bounded counters $\hat{y}_i^+$ and $\hat{y}_i^-$ with value 0. During the process of popping $S_i$ up to the first $Z_0$, $M'$ increases $\hat{y}_i^+$ by 1 if the current top symbol is '+1' and increases $\hat{y}_i^-$ by 1 if the current top symbol is '-1'. Then, $M'$ calculates $\hat{y}_i^+ := \hat{y}_i^+ - \hat{y}_i^-$ directly. This will cause one counter reversal for each of them. Afterwards, $M'$ uses $\hat{y}_i^+$, which is the expected value of the blind counter $y_i$, to proceed with the comparison with $\beta_{y_i}$ in $[\beta]$ on the input tape. Certainly, since the words after the first $Z_0$ on each $S_i$ is useless, $M'$ empties each $S_i$ before it continues to compare $S_{i+1}$. $M'$ aborts without accepting the input if any comparison fails. Clearly, $M'$ is a reversal-bounded MCPM and accepts $\leadsto^M$. Thus, from Theorem 22, $\leadsto^M$ is an effective semilinear language. Noticing that $\leadsto^M$ is a tuple language (i.e., it consists of integer tuples), $\leadsto^M$ is effectively definable as a Presburger formula, since the semilinear set of $\leadsto^M$ can be computed. ∎

### 6.8.11 Proof of Theorem 25

Theorem 25. Let $S$ be a set of configurations of a multi-reversal-bounded multi-piecewise-monotonic multi-blind counter machine $M$ without an input tape that is definable by a Presburger formula. Then both $Post^*_M(S)$ and $Pre^*_M(S)$ are effectively definable by Presburger formulas.

*Proof.* By definition, the postimage $Post^*_M(S)$ of $S$ with respect to $M$ is defined as

$$\{\beta : \alpha \leadsto^M \beta, \alpha \in S\}.$$

Notice that from Theorem 24, $\leadsto^M$ is definable by a Presburger formula. $Post^*_M(S)$ may be rewritten as

$$\{\beta : \exists \alpha \in S(\alpha \leadsto^M \beta)\}$$

This is definable by a Presburger formula, since $S$ is definable by a Presburger formula and Presburger formulas are closed under quantification. The case for $Pre^*_M(S)$ is similar.

In fact, given $M$ and the Presburger formula for $S$, the Presburger formulas for $Post^*_M(S)$ and $Pre^*_M(S)$, are also effectively computable, since $\rightsquigarrow^M$ is effectively semilinear, and it is well known that quantification elimination for a Presburger formula is effective. ∎

# Chapter 7

# Conclusions

In this dissertation, we studied verification problems (with an emphasis on safety analysis) for real-time infinite state systems, from both an applicational and a theoretical view point. In applications, we anchored the research to an existing high level specification language for real-time system called ASTRAL. In theory, we proposed two new models for infinite state systems and investigated the decidability of a number of safety properties.

Due to the fact that ASTRAL is a very expressive language, the research started by defining a subset of the ASTRAL language such that an automatic or semi-automatic verification procedure exists for the subset. This subset is defined in Chapter 2 as Small-ASTRAL. Small-ASTRAL preserves the most important timing features of ASTRAL and can be used to specify real-time systems containing parameterized constants. Even though automatically verifying Small-ASTRAL specifications is not possible, we are able to build a symbolic bounded-testing tool in Chapter 3 to debug Small-ASTRAL specifications. The tool is implemented to carry out image calculations on a bounded-depth execution tree of a specification. Since the tool is intended to be used as a specification debugger instead of a verifier, we have proposed a number of approximation techniques, such as partial image, random walk and dynamic environment generation, to speed up the debugging procedure. An extensive set of experiments showed that the approximation techniques proposed are effective in debugging a specification in a much shorter time than without using them.

In addition, we defined a subset of Small-ASTRAL such that an automatic verification procedure exists for the subset. The subset is called Mini-ASTRAL. As a special case, Chapter 4 shows that a history-independent Mini-ASTRAL specification can be related to a timed automaton [AD94]. We show that history-independent Mini-ASTRAL has a decidable safety analysis problem, by showing a stronger result: the binary reachability of a (discrete time) timed automaton is Presburger. The proof technique used, which separates controls (i.e., enabling conditions of a transition) from clock progressions, is more important than the results. It has already been used in other extensions of timed automata [DIBKS00b, PD00a, PD00b].

Based on a new construction for eliminating history operators in Mini-ASTRAL, we are able to show, in Chapter 5, that the entire Mini-ASTRAL language has a decidable safety analysis problem. That is, Mini-ASTRAL can be automatically verified.

In the theory portion of this research, we have investigated two new models for infinite state systems. The first model is a timed automaton coupled with a multi-queue automaton (inspired by the GCG model). The resulting machine can be effectively used for modeling queue-related systems, such as real-time process schedulers, while retaining the decidability of a class of Presburger formulas over the binary reachability set, with control-state variables, clock value variables and counter variables. The second model is a multi-reversal-bounded multi-blind multi-piecewise-monotonic counter machine (MMMCM). This model is capable of providing an alternative theoretical tool to analyze various timed models. We have shown that the binary reachability of MMMCMs is effectively semilinear.

There are many issues in this research that are left as future research. Since the model-checker proposed in Chapter 3 is primarily intended for use as a specification debugger and fixed point computation is out of the question, additional approximation techniques that already exist in the testing area can also be investigated. The approximation techniques proposed in Chapter 3 may also be useful in model checkers using different specification languages, as long as only safety properties are considered. For debugging a general temporal property formulated in a temporal logic, we still do not know how well these approximation approaches will work. This is another area for further research.

The price paid for modularized model-checking of real-time systems is the introduction of history-dependency in the form of past formulas. The translation in Chapter 5 from a past formula to a finite state machine suffers from nonelementary blow-up in size, which limits its usefulness in practice. Studies on special formats for past formulas leading to a short translation have practical importance. Such a short translation would result in more efficient modularized model-checking procedures for real-time systems. Therefore, this is another area that merits further research.

A short fall in this research is that we have not looked into automatic verification of general temporal logic formulas in addition to safety properties. Recently, we have made some progress in this direction by considering a restricted stack-queue model and a linear temporal logic [PD00c]. The issue, however, is far from being completed. The MMMCM model presented in Chapter 6 can be regarded as a base for studying other real-time system models. We have used the binary reachability result for MMMCM models to study verification problems of (discrete) Rectangle Automata [PD00b].

# Appendix A

# ASTRAL Specification of a Railroad Crossing

```
/* This specification was written by P. Kolano, and it has been
verified using the PVS theorem prover. */

SPECIFICATION Railroad_Crossing
  GLOBAL SPECIFICATION Railroad_Crossing
    PROCESSES
              the_gate: Gate,
              the_sensors: array [ 1..n_tracks ]  of Sensor
    TYPE
              pos_integer: TYPEDEF i: integer ( i > 0 ) ,
              pos_real: TYPEDEF i: integer ( i > 0 ) ,
              gate_position:  ( raised, raising, lowered, lowering ) ,
              sensor_id: TYPEDEF i: id ( IDTYPE  ( i )  = Sensor )
    CONSTANT
              n_tracks: pos_integer,
              min_speed, max_speed: pos_real,
              dist_R_to_I, dist_I_to_out: pos_real,
              response_time, wait_time: pos_real,
              RImax: pos_real,
              RIImin: pos_real,
              RIImax: pos_real
    AXIOM
      /*since the Omega library can not handle division,
        we therefore have to introduce RImax to represent
```

```
      dist_R_to_I/max_speed, RIImin to represent
      (dist_R_to_I + dist_I_to_out)/min_speed, and
      RIImax to represent (dist_R_to_I + dist_I_to_out)/max_speed*/

            max_speed >= min_speed
          & response_time < RImax
          & max_speed * RImax <= dist_R_to_I
          & max_speed * RImax + max_speed > dist_R_to_I
          & min_speed * RIImin <= dist_R_to_I + dist_I_to_out
          & min_speed * RIImin + min_speed >dist_R_to_I+dist_I_to_out
          & max_speed * RIImax <= dist_R_to_I+ dist_I_to_out
          & max_speed * RIImax + max_speed >dist_R_to_I+dist_I_to_out
  SCHEDULE

    /* gate will be down before fastest train reaches crossing */
             ( EXISTS s: sensor_id
                        ( s.train_in_R
                        & EXISTS t2: time
                                   ( t2 <= NOW
                                   & s.Call ( enter_R, t2 )
                                   & now - t2 >= RImax )  )
          ->   the_gate.position = lowered )
    /* gate will be up after slowest train exits
       crossing and a reasonable wait time has elapsed */

          &  ( FORALL s: sensor_id
                        ( ~s.train_in_R
                        &  ( EXISTS t: time
                                      ( s.Call ( enter_R, t ) )
                        ->   EXISTS t1: time
                                  ( t1 <= NOW
                                  & s.Call ( enter_R, t1 )
                                  & now - t1 >= RIImin + wait_time ) ) )
          ->   the_gate.position = raised )
END Railroad_Crossing

PROCESS SPECIFICATION Sensor
  LEVEL Top_Level
    IMPORT
              pos_real, max_speed, min_speed, dist_R_to_I,
              dist_I_to_out, response_time, RImax, RIImax,
              RIImin
    EXPORT
              train_in_R, enter_R
    CONSTANT
              enter_dur, exit_dur: pos_real
    VARIABLE
```

```
          train_in_R: boolean
AXIOM
          response_time >= enter_dur
        & RIImin >= response_time + exit_dur
ENVIRONMENT

  /* only one train will be in the region at the
     same time on the same track */
          Call ( enter_R, now )
        & EXISTS t: time
                   ( t >= 0
                   & t <= now
                   & Call [ 2 ]  ( enter_R, t )  )
     -> Call(enter_R) - Call [ 2 ] ( enter_R ) > RIImin
INITIAL
          ~train_in_R
INVARIANT

  /* once a sensor reports a train, it will keep
     reporting a train at least as long as it takes the
     fastest train to exit the region */
          Change ( train_in_R, now )
        & ~train_in_R
     ->   0 <= now -  ( RIImax - response_time )
        & FORALL t: time
                   ( now -  ( RIImax - response_time ) <= t
                   & t < now
                ->   past ( train_in_R, t )  )
SCHEDULE

  /* train will be sensed within enter_dur of call */
           ( now >= response_time
           & Call ( enter_R, now - response_time )
        ->   train_in_R )
  /* sensor will be reset when the slowest
     train is beyond the crossing */

        &  ( now >= RIImin
           & Call ( enter_R, now - RIImin )
        ->   ~train_in_R )
TRANSITION enter_R
  ENTRY           [ TIME : enter_dur ]
           ~train_in_R
  EXIT
           train_in_R = TRUE
TRANSITION exit_I
  ENTRY           [ TIME : exit_dur ]
```

```
              train_in_R
            & now - Start ( enter_R )  >= RIImin - exit_dur
      EXIT
              train_in_R = FALSE
  END Top_Level
END Sensor

PROCESS SPECIFICATION Gate
  LEVEL Top_Level
    IMPORT
              pos_real, gate_position, max_speed, dist_R_to_I,
              dist_I_to_out, wait_time, response_time,
              sensor_id, the_sensors.train_in_R,
              the_sensors.enter_R, RImax, RIImax, RIImin
    EXPORT
              position
    CONSTANT
              lower_dur, raise_dur, up_dur, down_dur: pos_real,
              raise_time, lower_time: pos_real
    VARIABLE
              position: gate_position
    AXIOM
              wait_time >= raise_dur + raise_time + up_dur
            & RImax >= response_time + lower_dur + lower_time
                                      + down_dur + raise_dur
            & RImax >= response_time + lower_dur + lower_time
                                      + down_dur + up_dur
    INITIAL
              position = raised
    SCHEDULE

      /* gate will be down before fastest train reaches crossing */
               ( EXISTS s: sensor_id
                   ( s.train_in_R
                    & now - Change(s.train_in_R )
                          >= RImax - response_time)
            ->   position = lowered )
      /* gate will be up after slowest train exits
         crossing and enough time has elapsed for gate to be
         raised */

            &  ( FORALL s: sensor_id
                   ( FORALL t: time
                          ( now - wait_time <= t
                            & t <= now
                         ->   ~past ( s.train_in_R, t )  )  )
            ->   position = raised )
```

```
      IMPORTED VARIABLE

        /* once a sensor reports a train, it will keep
           reporting a train at least as long as it takes the
           fastest train to exit the region */
                FORALL s: sensor_id
                         ( Change ( s.train_in_R, now )
                         & ~s.train_in_R
                      ->   0 <= now -  ( RIImax - response_time )
                         & FORALL t: time
                                 ( now - ( RIImax - response_time ) <= t
                                 & t < now
                              -> past ( s.train_in_R, t ) ) )
      TRANSITION lower
        ENTRY           [ TIME : lower_dur ]
                  ~ ( position = lowering
                   |  position = lowered )
                & EXISTS s: sensor_id
                           ( s.train_in_R )
        EXIT
                  position = lowering
      TRANSITION down
        ENTRY           [ TIME : down_dur ]
                  position = lowering
                & now - End ( lower )  >= lower_time
        EXIT
                  position = lowered
      TRANSITION raise
        ENTRY           [ TIME : raise_dur ]
                  ~ ( position = raising
                   |  position = raised )
                & FORALL s: sensor_id
                           ( ~s.train_in_R )
        EXIT
                  position = raising
      TRANSITION up
        ENTRY           [ TIME : up_dur ]
                  position = raising
                & now - End ( raise )  >= raise_time
        EXIT
                  position = raised

    END Top_Level
  END Gate
END Railroad_Crossing
```

# Appendix B

# Small-ASTRAL grammar

```
//GLOBAL SPECIFICATION
    global_spec:
        GLOBAL SPECIFICATION IDENTIFIER
        PROCESSES processes_decl_list
        type_clause
        axiom_clause
        constant_clause
        define_clause
        environment_clause
        invariant_clause
        schedule_clause
        END IDENTIFIER

//PROCESS SPECIFICATIONS

    process_spec_list:
        process_spec
        | process_spec_list process_spec

    process_spec:
        import_clause
        export_clause
        environment_clause
        impvar_clause
        type_clause
        axiom_clause
        variable_clause
        constant_clause
        define_clause
```

```
        initial_clause
        invariant_clause
        schedule_clause
        trans_decl_list

//make each part clear
    processes_decl_list:
        processes_decl
        | processes_decl_list COMMA processes_decl

    processes_decl:
        id_list COLON IDENTIFIER
        | id_list COLON ARRAY OPENSQUARE id_integer CLOSESQUARE
              OF IDENTIFIER
        | id_list COLON ARRAY OPENSQUARE id_integer DOT DOT
              id_integer CLOSESQUARE OF IDENTIFIER

     import_clause:
        /* empty */
        | IMPORT id_dot_list

     export_clause:
        /* empty */
        | EXPORT id_list

     environment_clause:
        /* empty */
        | ENVIRONMENT small_astral_wff

     impvar_clause:
          /* empty */
        | IMPORTED VARIABLE small_astral_wff


     axiom_clause:
        /* empty */
        | AXIOM small_astral_wff


     invariant_clause:
        /* empty */
        | INVARIANT small_astral_wff

     schedule_clause:
        /* empty */
        | SCHEDULE small_astral_wff
```

```
constant_clause:
   /* empty */
   | CONSTANT const_var_decl_list

variable_clause:
   /* empty */
   | VARIABLE const_var_decl_list

type_clause:
     /* empty */
   | type_decl_list


define_clause:
     /* empty */
   | IDENTIFIER COLON any_type EQEQ small_astral_wff
   | IDENTIFIER OPENROUND id_type_list CLOSEROUND COLON
         any_type EQEQ small_astral_wff

trans_decl_list:
   trans_decl
   | trans_decl_list trans_decl

//TRANSITION
     trans_decl:
        TRANSITION trheading
        ENTRY OPENSQUARE TIME COLON duration CLOSESQUARE small_astral_wff
        EXIT small_astral_wff
        opt_except_list

     trheading:
          IDENTIFIER
        | IDENTIFIER OPENROUND id_type_list CLOSEROUND

     duration:
        IDENTIFIER
        | INTEGER_CONST

     opt_except_list:
         /* empty */
        | except_decl_list

     except_decl_list:
        except_decl
        | except_decl_list except_decl
```

```
    except_decl:
       EXCEPT OPENSQUARE TIME COLON duration CLOSESQUARE small_astral_wff
       EXIT small_astral_wff

//CONSTANT and VARIABLE declarations

    const_var_decl_list:
         const_var_decl
       | const_var_decl_list COMMA const_var_decl

    const_var_decl:
         const_var_list COLON any_type

    const_var_list:
         const_var
       | const_var_list COMMA const_var

    const_var:
         IDENTIFIER
       | IDENTIFIER OPENROUND any_type_list CLOSEROUND

    any_type_list:
         any_type
       | any_type_list COMMA any_type

//TYPE DECLARATION
   type_decl:
         IDENTIFIER colon_is OPENROUND id_list CLOSEROUND
       | IDENTIFIER colon_is STRUCTURE OF OPENROUND
             id_type_list CLOSEROUND
       | IDENTIFIER colon_is TYPEDEF IDENTIFIER colon_is any_type
             OPENROUND small_astral_wff CLOSEROUND.

    colon_is:
         COLON
       | IS

    id_dot_list:
         IDENTIFIER
       | IDENTIFIER optional_n DOT IDENTIFIER
       | id_dot_list COMMA IDENTIFIER
       | id_dot_list COMMA IDENTIFIER optional_n DOT IDENTIFIER

    optional_n:
         /* empty */
       | OPENSQUARE INTEGER_CONST CLOSESQUARE
```

```
     id_list:
          IDENTIFIER
        | id_list COMMA IDENTIFIER

     id_type_list:
          id_list COLON any_type
        | id_type_list COMMA id_list COLON any_type

     any_type:
          INTEGER
        | BOOLEAN
        | TIME
        | ID
        | IDENTIFIER

//WFF
    small_astral_wff:
          small_astral_wff IFF small_astral_wff
        | small_astral_wff NIFF small_astral_wff
        | small_astral_wff IMPLIES small_astral_wff
        | small_astral_wff NIMPLIES small_astral_wff
        | small_astral_wff OR small_astral_wff
        | small_astral_wff NOR small_astral_wff
        | small_astral_wff AND small_astral_wff
        | small_astral_wff NAND small_astral_wff
        | NOT small_astral_wff
        | small_astral_wff EQ small_astral_wff
        | small_astral_wff NEQ small_astral_wff
        | small_astral_wff LT small_astral_wff
        | small_astral_wff NLT small_astral_wff
        | small_astral_wff LTE small_astral_wff
        | small_astral_wff NLTE small_astral_wff
        | small_astral_wff GT small_astral_wff
        | small_astral_wff NGT small_astral_wff
        | small_astral_wff GTE small_astral_wff
        | small_astral_wff NGTE small_astral_wff
        | small_astral_wff PLUS small_astral_wff
        | small_astral_wff MINUS small_astral_wff
        | small_astral_wff TIMES INTEGER_CONST
        | small_astral_wff DIVIDE INTEGER_CONST
        | small_astral_wff MOD INTEGER_CONST
        | MINUS small_astral_wff
        | IDTYPE OPENROUND IDENTIFIER CLOSEROUND
        | sec sec_times OPENROUND id_combo, IDENTIFIER CLOSEROUND
        | sec sec_times OPENROUND id_combo CLOSEROUND
        | NOCHANGE OPENROUND id_combo CLOSEROUND
```

```
   | PAST OPENROUND id_combo, IDENTIFIER CLOSEROUND
   | FORALL IDENTIFIER : any_type OPENROUND
            small_astral_wff CLOSEROUND
   | EXISTS IDENTIFIER : any_type OPENROUND
            small_astral_wff CLOSEROUND
   | becomes
   | id_combo DOT small_astral_wff

becomes:
     IDENTIFIER OPENROUND number_id_list
                CLOSEROUND BECOMES small_astral_wff
   | IDENTIFIER comp_spec BECOMES small_astral_wff
   | IDENTIFIER OPENROUND number_id_list
                CLOSEROUND comp_spec BECOMES small_astral_wff


id_combo:
     IDENTIFIER
   | IDENTIFIER PRIME
   | IDENTIFIER OPENROUND number_id_list CLOSEROUND
   | IDENTIFIER PRIME OPENROUND number_id_list CLOSEROUND
   | IDENTIFIER comp_spec
   | IDENTIFIER PRIME comp_spec
   | IDENTIFIER OPENROUND number_id_list CLOSEROUND comp_spec
   | IDENTIFIER PRIME OPENROUND number_id_list CLOSEROUND comp_spec

number_id_list:
     number_id
   | number_id_list number_id

number_id:
     BOOLEAN
   | ID
   | IDENTIFIER

sec:
     START
   | END
   | CALL
   | CHANGE

sec_times:
     /* empty */
   | OPENSUARE 1 CLOSESQUARE
   | OPENSUARE 2 CLOSESQUARE

comp_spec:
```

```
  OPENSQUARE IDENTIFIER CLOSESQUARE
| OPENSQUARE IDENTIFIER CLOSESQUARE comp_spec
```

# Bibliography


[A99] R. Alur. "Timed automata," *Proceedings of the International Conference on Computer Aided Verification,* LNCS 1633, pages 8-22, 1999.

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. "Model-checking in dense real time," *Information and Computation,* 104(1): 2-34, 1993.

[AD94] R. Alur and D. Dill. "Automata for modeling real-time systems," *Theoretical Computer Science,* 102(2): 183-236, 1994.

[AH94] R. Alur and T. A. Henzinger. "A really temporal logic," *Journal of the ACM,* 41(1): 181-204, 1994.

[AHV93] R. Alur, T. A. Henzinger, and M. Y. Vardi. "Parametric real-time reasoning," *Proceedings of the ACM Symposium on Theory of Computing,* pages 592-601, 1993.

[B60] J. R. Büchi . "On a decision method in a restricted second order arithmetic," *Proceedings of the International Congress on Logic, Methodology and Philosophy,* pages 1-11, 1960.

[B86] R.E. Bryant. "Graph-Based algorithms for Boolean function manipulation," *IEEE Transactions on Computers,* 35(8): 677-691, 1986.

[B92] R.E. Bryant. "Symbolic Boolean manipulation with ordered binary decision diagrams," *ACM Computing Surveys,* 24(3): 293-318, 1992.

[BB74] B. Baker and R. Book. "Reversal-bounded multipushdown machines," *Journal of Computer and System Sciences,* 8(3): 315-332, 1974.

[BBKT95] K. Brink, L. Bun, J. van Katwijk and W. J. Toetenel. "Hybrid specification of control systems," *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems,* 1995

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. "Reachability analysis of pushdown automata: application to model-checking," *Proceedings of the International Conference on Concurrency Theory,* LNCS 1243, pages 135-150, 1997.

[BER95] A. Bouajjani, R. Echahed, and R. Robbana, *"On the Automatic Verification of Systems with Continuous Variables and Unbounded Discrete Data Structures,"*, *Hybrid System II*, LNCS 999, pages 64-85, 1995

[BF99] B. Berard and L. Fribourg, *"Automated verification of a parametric real-time program: the ABR conformance protocol,"*, *Proceedings of the International Conference on Computer Aided Verification,* LNCS 1633, pages 96-107, 1999

[BLPWY99] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise and W. Yi. "Efficient timed reachability analysis using clock difference diagrams," *Proceedings of the International Conference on Computer Aided Verification,* LNCS 1633, pages 341-353, 1999.

[Bjorner et al. 96] N. S. Bjorner, A. Browne, E. Chang, M. Colon, A. Kapur, Z. Manna, H. Sipma, and T. Uribe. "STeP: deductive-algorithmic verification of reactive and real-time systems," *Proceedings of the International Conference on Computer Aided Verification,* LNCS 1102, pages 415-418, 1996.

[BCCC92] L. Breveglieri, A. Cherubini, and S. Crespi Reghizzi. "Real-Time scheduling by queue automata," *Proceedings of the International School and Symposium Formal Techniques in Real Time and Fault Tolerant Systems,* LNCS 571, pages 131-148, 1992.

[BCCC99] L. Breveglieri, A. Cherubini and S. Crespi Reghizzi. "Modelling operating systems schedulers with Multi-Stack-Queue Grammars," *Proceedings of the International Symposium on Fundamentals of Computation Theory,* LNCS 1684, pages 161-172, 1999.

[BCF91] G. Buonanno, A. Coen-Porisini and W. Fornaciari. "Hardware specification using the assertion language ASTRAL," *Proceedings of the Advanced Research Workshop on Correct Hardware Design Methodologies,* Torino, Italy, June 1991.

[BCM92] J. R. Burch, E. M. Clarke, and K. L. McMillan. "Symbolic model-checking: $10^{20}$ states and beyond," *Information and Computation*, 98(2): 142-170, 1992.

[BGL98] T. Bultan, R. Gerber, and C. League. "Verifying systems with integer constraints and Boolean predicates: a composite approach," *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis,* pages 113-123, 1998.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. "Symbolic model-checking of infinite state systems using Presburger arithmetic," *Proceedings of the International Conference on Computer Aided Verification,* pages 400-411, 1997.

[BGP99] T. Bultan, R. Gerber, and W. Pugh. "Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations and experimental results," *ACM Transactions on Programming Languages and Systems*, 21(4): 747-789, 1999.

[BS97] D. Beauquier and A. Slissenko. "The railroad crossing problem: towards semantics of timed algorithms and their model-checking in high-level languages," *Proceedings of the 7th International Joint Conference on Theory and Practice of Software Development,* LNCS 1214, pages 202-212, 1997.

[BMSU97] N. S. Bjorner, Z. Manna, H. B. Sipma, and T. E. Uribe. "Deductive verification of real-time systems using STeP," *Proceedings of the International AMAST Workshop on Real-time Systems,* LNCS 1231, pages 22-43, 1997.

[BW94] B. Boigelot and P. Wolper, *"Symbolic verification with periodic sets," Proceedings of the International Conference on Computer Aided Verification,* LNCS 818, pages 55-67, 1994

[C76] L. A. Clarke. "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering,* 2(3): 215-222, 1976.

[CBCC96] A. Cherubini, L. Breveglieri, C. Citrini, and S. Crespi Reghizzi. "Multi-push-down languages and grammars," *International Journal of Foundations of Computer Science,* 7(3): 253-291, 1996.

[CC99] P. Cousot and R. Cousot. "Refining model checking by abstract interpretation," *Automated Software Engineering,* 6(3): 69-95, 1999.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic verification of finite state concurrent systems using temporal logic specifications," *Proceedings of the 10th ACM Symposium on Principles of Programming Languages,* pages 117-126, 1983.

[CF97] G. Cécé and A. Finkel. "Programs with quasi-stable channels are effectively recognizable," *Proceedings of the International Conference on Computer Aided Verification,* LNCS 1254, pages 304-315, 1997.

[CGHJLMN93] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. "Verification of the Futurebus+ cache coherence protocol," *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and their Applications*, April 1993.

[CGJ98] C. Colby, P. Godefroid and L. J. Jagadeesan. "Automatically closing open reactive programs," *SIGPLAN Notices,* 33(5): 345-357, 1998.

[CGK97] A. Coen-Porisini, C. Ghezzi and R. A. Kemmerer. "Specification of real-time systems using ASTRAL," *IEEE Transactions on Software Engineering,* 23(9): 572-598, 1997.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. "Model-checking and abstraction," *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 342-354, 1992.

[CJ98] H. Comon and Y. Jurski. "Multiple counters automata, safety analysis and Presburger arithmetic," *Proceedings of the International Conference on Computer Aided Verification,* LNCS 1427, pages 268-279, 1998.

[CJ99] H. Comon and Y. Jurski. "Timed automata and the theory of real numbers," *Proceedings of the International Conference on Concurrency Theory,* pages 242-257, 1999.

[CKM94] A. Coen-Porisini, R. A. Kemmerer and D. Mandrioli. "A formal framework for ASTRAL intralevel proof obligations," *IEEE Transactions on Software Engineering*, 20(8): 548-561, 1994.

[CPK94] A. Coen-Porisini, P. San Pietro and R. A. Kemmerer. "Formal semantics definition for ASTRAL," Technical Report TRCS94-15, Department of Computer Science, UC Santa Barbara, 1994.

[CS00] A. Cherubini and P. San Pietro. "Tree adjoining languages and multi-pushdown languages," To appear in *Theory of Computation Systems*.

[CS96] A. Cherubini and P. San Pietro. "A polynomial-time parsing algorithm for k-depth languages," *Journal of Computer and System Sciences,* 52(1): 61-79, 1996.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper and M. Yannakakis. "Memory-efficient algorithms for the verification of temporal properties," *Formal Methods in System Design*, 1(2): 275-288, 1992.

[CW96] Edmund Clarke and Jeannette Wing. "Formal methods: state of the art and future directions," *ACM Computing Surveys,* 28(4): 626-643, 1996.

[D96] D. L. Dill. "The Mur$\varphi$ verification system," *Proceedings of the International Conference on Computer Aided Verification,* LNCS 1102, pages 390-393, 1996.

[DGG97] D. Dams, R. Gerth and O. Grumberg. "Abstract interpretation of reactive systems," *ACM Transactions on Programming Languages and Systems* , 19(2): 253-291, 1997.

[D97] Z. Dang. Master Thesis, 1997, UC Santa Barbara.

[DK94] J. Douglas and R. A. Kemmerer. "Aslantest: a symbolic execution tool for testing Aslan formal specifications," *Proceedings of the International Symposium on Software Testing and Analysis,* pages 15-27, 1994.

[DK97] Z. Dang and R. Kemmerer, "Using the ASTRAL model-checker for cryptographic protocol analysis," *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols,* Rutgers University, 1997.

[DK99a] Z. Dang and R. A. Kemmerer. "Using the ASTRAL model-checker to analyze Mobile IP," *Proceedings of the International Conference on Software Engineering,* pages 132-141, 1999.

[DK99b] Z. Dang and R. A. Kemmerer. "A symbolic model-checker for testing ASTRAL real-time specifications," *Proceedings of the International Conference on Real-time Computing Systems and Applications,* pages 174-181, 1999.

[DK00a] Z. Dang and R. A. Kemmerer. "Dynamic environment generations for an ASTRAL process," Technical Report TRCS00-03, Computer Science Department, University of California at Santa Barbara, 2000.

[DK00b] Z. Dang and R. A. Kemmerer. "Three approximation techniques for ASTRAL symbolic model checking of infinite state real-time systems," *Proceedings of the International Conference on Software Engineering,* pages 345-354, 2000.

[DIBKS00a] Z. Dang, O. H. Ibarra, T. Bultan, R. A. Kemmerer, and J. Su. "Decidable approximations on discrete clock machines with parameterized durations," Manuscript, 2000.

[DIBKS00b] Z. Dang, O. H. Ibarra, T. Bultan, R. A. Kemmerer, and J. Su. "Binary reachability analysis of discrete pushdown timed automata," *Proceedings of the International Conference on Computer Aided Verification,* pages 69-84, 2000.

[DIBKS00c] Z. Dang, O. H. Ibarra, T. Bultan, R. A. Kemmerer, and J. Su. "Past machines," Manuscript, 2000.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. "Hints on test data selection: Help for the practicing programmer," *IEEE Computer,* 11(4): 34-41, 1978.

[DN81] J. W. Duran and S. Ntafos. "A report on random testing," *Proceedings of the International Conference on Software Engineering,* pages 179-183, 1981.

[DW80] J. W. Duran and J. J. Wiorkowski. "Quantifying software validity by sampling," *IEEE Transactions on Reliability,* 29(3): 141-144, 1980.

[DW95] David L. Dill and Howard Wong-Toi. "Verification of real-time systems by successive over and under approximation," LNCS 939, pages 409-422, 1995.

[E90] E.A. Emerson. "Temporal and modal logic," *Handbook of Theoretical Computer Science*, pages 997-1072, 1990.

[Esp97] J. Esparza. "Decidability of model-checking for infinite state concurrent systems," *Acta Informatica*, 34(2): 85-107, 1997.

[EH86] E.A. Emerson and J.Y.Halpern. "'Sometime' and 'not never' revisited: on branching versus linear time temporal logic," *Journal of the ACM,* 33(1): 151-178, 1986.

[FS00] A. Finkel and G. Sutre. "Decidability of reachability problems for classes of two counters automata," *Proceedings of the 17th International Symposium on Theoretical Aspects of Computer Science*, 2000.

[FWW97] A. Finkel, B. Willems and P. Wolper. "A direct symbolic approach to model-checking pushdown systems," *Proceedings of the INFINITY Workshop*, 1997.

[G75] S.A. Greibach. "A note on the recognition of one-counter languages," *RAIRO*, 9(2): 5-12, 1975.

[G78] S.A. Greibach. "Remarks on blind and partially blind one-way multicounter machines," *Theoretical Computer Science,* 7(2): 311-324, 1978.

[G98] B. Grahlmann. "The state of PEP," *Proceedings of the International AMAST Workshop,* pages 522-526, 1998.

[GF91] A. Gabrielian and M. K. Franklin. "Multilevel specification of real time systems," *Communications of the ACM*, 34(1): 50-60, 1991.

[GL92] R. Gerber and I. Lee. "A layered approach to automating the verification of real-time systems," *IEEE Transactions on Software Engineering,* 18(1992): 768-784.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. "Simple on-the-fly automatic verification of linear temporal logic," *Proceedings of the 15th International Workshop on Protocol Specification, Testing, and Verification*, Warsaw, June 1995.

[GS64] S. Ginsburg and E. Spanier. "Bounded Algol-like languages," *Transactions of American Mathematical Society,* 113(2): 333-368, 1964.

[GW94] P. Godefroid and P. Wolper. "A partial approach to model-checking," *Information and Computation*, 110(2): 305-326, 1994.

[HHW97] T. A. Henzinger and P. Ho, H. Wong-Toi. "HYTECH: a model checker for hybrid systems," *Proceedings of the International Conference on Computer Aided Verification,* pages 460-463, 1997.

[HKPV95] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. "What is decidable about hybrid automata," *Proceedings of the ACM Symposium on Theory of Computing,* pages 373-382, 1995.

[HL94] C. Heitmeyer and N. Lynch. "The generalized railroad crossing: a case study in formal verification of real-time systems," *Proceedings of the 15th Real-time Systems Symposium*, pages 120-131, 1994.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. "Symbolic model-checking for real-time systems," *Information and Computation,* 111(2): 193-244, 1994.

[H69] C. A. R. Hoare. "An axiomatic basis for computer programming," *Communications of the ACM,* 12(10): 576-580, 1969.

[H91] G. J. Holzmann. "Design and validation of computer protocols," Prentice Hall, 1991.

[H97] G. Holzmann. "The model checker SPIN," *IEEE Transactions on Software Engineering,* 23(2): 279-295, 1997.

[H82] W. E. Howden. "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering,* 8(7): 371-379, 1982.

[I78] O. H. Ibarra. "Reversal-bounded multicounter machines and their decision problems," *Journal of the ACM,* 25(1): 116-133, 1978.

[IJTW95] O. H. Ibarra, T. Jiang, N. Tran, and H. Wang. "New decidability results concerning two-way counter machines," *SIAM Journal of Computing*, 24(1): 123-137, 1995.

[ISDBK00] O. H. Ibarra, J. Su, Z. Dang, T. Bultan, and R. A. Kemmerer. "Counter machines: decidable properties and applications to verification problems," To appear in *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science*, 2000.

[JW96] D. Jackson and J. Wing. "Lightweight formal methods," *IEEE Computer*, 29(1): 16-30, 1996.

[JM86] F. Jahanian and A. K. Mok. "Safety analysis of timing properties in real-time systems," *IEEE Transactions on Software Engineering,* 12(9): 890-904, 1986.

[KE85] R. A. Kemmerer and S. T. Eckman. "UNISEX: a UNIX-based Symbolic EXecutor for Pascal," *Software - Practice and Experience,* 15(5): 439-458, 1985.

[K76] J. C. King. "Symbolic execution and program testing," *Communications of the ACM*, 19(2): 385-394, 1976.

[K99a] P. Z. Kolano. "Proof assistance for real-time systems using an interactive theorem prover," *Proceedings of the International AMAST Workshop on Real-Time and Probabilistic Systems,* LNCS 1601, pages 315-333, 1999.

[K99b] P. Z. Kolano. PhD. Dissertation, Department of Computer Science, University of California at Santa Barbara, 1999.

[KDK99] P. Z. Kolano, Z. Dang and R. A. Kemmerer. "The design and analysis of real-time systems using the ASTRAL software development environment," *Annals of Software Engineering*, 7(2): 177-210, 1999.

[KK67] G. Kreisel and J. L. Krevine. "Elements of mathematical logic," North-Holland, 1967.

[LP85] O. Lichtenstein and A. Pnueli. "Checking that finite state concurrent programs satisfy their linear specification," *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages,* pages 97-107, 1985.

[M70] Y. Matijasevic. "Enumerable sets are Diophantine," *Soviet Math. Dokl*, 11(2): 354-357, 1970.

[M76] P. M. Merlin. "A methodology for the design and implementation of communication protocols," *IEEE Transactions on Communications,* 24(6): 1036-1043, 1976.

[M92] K. L. McMillan. "Symbolic model-checking - an approach to the state explosion problem," PhD thesis, Department of Computer Science, Carnegie Mellon University, 1992.

[M98] R. Mayr. "Decidability and complexity of model-checking problems for infinite state systems," Ph.D. Thesis, Inst. für Informatik, Techn. Universität München, 1998.

[MT89] F. Moller and C. Tofts, *"A temporal calculus of communicating systems," Proceedings of the International Conference on Concurrency Theory,* LNCS 458, pp. 401-415, 1989

[ND96] C. Norris Ip and D. L. Dill. "Better verification through symmetry," *Formal Methods in System Design*, 9(1): 41-75, 1996.

[O78] D. C. Oppen. "A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic," *Journal of Computer and System Sciences*, 16(2): 323-332, 1978.

[P66] R. Parikh. "On context-free languages," *Journal of the ACM*, 13(4): 570-581, 1966.

[P77] A. Pnueli. "The temporal logic of programs," *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science,* pages 46-57, 1977.

[P92] W. Pugh. "The Omega test: a fast and practical integer programming algorithm for dependence analysis," *Communications of the ACM*, 8(1): 102-104, 1992.

[PD00a] P. San Pietro and Z. Dang. "Automatic verification of multi-queue discrete timed automata," Manuscript, 2000.

[PD00b] P. San Pietro and Z. Dang. "Binary reachability analysis of discrete timed rectangle automata," Manuscript, 2000.

[PD00c] P. San Pietro and Z. Dang. "Model-checking discrete timed multi-stack multi-queue machines," Manuscript, 2000.

[PD00d] P. San Pietro and Z. Dang. "A new class of counter machines," Manuscript, 2000.

[R69] M. O. Rabin. "Decidability of second order theories and automata on infinite trees," *Transactions of American Mathematical Society*, 141(1): 1-35, 1969.

[S93] A. Udaya Shankar. "An introduction to assertion reasoning for concurrent systems," *ACM Computing Surveys*, 25(2): 225-262, 1993.

[SC85] A.P. Sistla and E. M. Clarke. "The complexity of propositional linear temporal logic," *Journal of the ACM*, 12(6): 733-774, 1985.

[VW88] M. Y. Vardi and P. Wolper. "Reasoning about infinite computation paths," IBM Research Reports RJ6209, 1988.

[Wa97] I. Walukiewicz. "Pushdown processes: games and model-checking," *Proceedings of the International Conference on Computer Aided Verification,* pages 63-74, 1996.

[W00] F. Wang. “Efficient data structure for fully symbolic verification of real-time software systems,” *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems,* LNCS 1785, pages 157-171, 2000.

[WC81] L. J. White and E. I. Cohen. “A domain strategy for computer program testing,” *IEEE Transactions on Software Engineering,* 6(2): 247-257, 1980.

[W86] P. Wolper. “Expressing interesting properties of programs in propositional temporal logic,” *Proceedings of the 13th ACM Symposium on Principles of Programming Languages,* pages 184-193, 1986.

[W83] P. Wolper. “Temporal logic can be more expressive,” *Information and Control,* 56(1): 72-99, 1983.

[WB98] P. Wolper and B. Boigelot. “Verifying systems with infinite but regular state spaces,” *Proceedings of the International Conference on Computer Aided Verification,* pages 88-97, 1998.

[Y98] S. Yovine. “Model-checking timed automata,” LNCS 1494, pages 114-152, 1998.