ACTA UNIVERSITATIS UPSALIENSIS
Uppsala Dissertations from
the Faculty of Science and Technology
61

Pritha Mahata

Model Checking Parameterized Timed Systems

Dissertation for the Degree of Doctor of Philosophy in Computer Science. Presented at Uppsala University, on 23rd March, 2005.

ABSTRACT

Mahata, P. 2005: Model Checking Parameterized Timed Systems. Acta Universitatis Upsaliensis. *Uppsala Dissertations from the Faculty of Science and Technology 61.* 180 pp. Uppsala, Sweden. ISBN 91-554-6158-1

In recent years, there has been much advancement in the area of verification of infinite-state systems. A system can have an infinite state-space due to unbounded data structures such as counters, clocks, stacks, queues, etc. It may also be infinite-state due to parameterization, i.e., the possibility of having an arbitrary number of components in the system. For parameterized systems, we are interested in checking correctness of all the instances in one verification step.

In this thesis, we consider systems which contain both sources of infiniteness, namely: (a) real-valued clocks and (b) parameterization. More precisely, we consider two models : (a) the *timed Petri net* (TPN) model which is an extension of the classical Petri net model; and (b) the *timed network* (TN) model in which an arbitrary number of timed automata run in parallel.

We consider verification of *safety* properties for timed Petri nets using forward analysis. Since forward analysis is necessarily incomplete, we provide a semi-algorithm augmented with an acceleration technique in order to make it terminate more often on practical examples. Then we consider a number of problems which are generalisations of the corresponding ones for *timed automata* and *Petri nets*. For instance, we consider *zenoness* where we check the existence of an infinite computation with a finite duration. We also consider two variants of the *boundedness* problem: *syntactic boundedness* in which both live and dead tokens are considered; *semantic boundedness* where only *live tokens* are considered. We show that the former problem is decidable, while the latter is not. Finally, we show undecidability of *LTL model checking* both for dense and discrete timed Petri nets.

Next we consider *timed networks*. We show undecidability of safety properties in case each component is equipped with two or more clocks. This result contrasts previous decidability result for the case where each component has a single clock. Also, we show that the problem is decidable when clocks range over the discrete time domain. This decidability result holds when the processes have any finite number of clocks. Furthermore, we outline the border between decidability and undecidability of safety for TNs by considering several syntactic and semantic variants.

Pritha Mahata, Department of Information Technology, Uppsala University, P. O. Box 337, SE 751 05 Uppsala, Sweden.

ISSN 1104-2516
ISBN 91-554-6158-1.

Printed in Elanders Gotab, Stockholm, Sweden
Distributor: Uppsala University Library, Box 510, SE-751 20 Uppsala, Sweden

To my family

Acknowledgments

I will always remain indebted to my supervisor Prof. Parosh Aziz Abdulla for his excellent guidance through the long journey of my doctoral studies. He has brought me to the area of formal verification and has tried his best to teach me how to be a good researcher. His demand for perfection has really been educational. His insight always led me to the right path in the course of my research. He has always encouraged me to go beyond my limits. I am grateful to him for his numerous valuable suggestions and comments during the writing of this thesis.

I am thankful to Dr. Richard Mayr for sharing his valuable experience with me during his visits to Uppsala. It has been a privilege to work with him.

I wish to thank Dr. Aletta Nylén for helping me to get started in the area of verification of timed systems and for many long discussions.

I enjoyed working with Johann Deneux. I was lucky to have such a co-author with an excellent eye for details. I am much in debt to him for his careful reading and comments towards a large part of this thesis.

I would also like to thank Dr. Julien d'Orso and Rezine Ahmed for some interesting discussions.

Special thanks to Mayank Saksena, Rezine Ahmed, Noomene Ben Henda, Lisa Kaati and Prof. Bengt Jonsson for reviewing and commenting on the parts of this thesis.

I am grateful to our prefect Dr. Ivan Christoff for maintaining a lively, research environment in DoCS.

I am thankful to Ms. Anne-Marie Nilsson and Ms. Marianne Ahrne for their continuous help in all administrative issues. The life would have been really difficult without their kind and efficient co-operation.

My parents and my elder sister always inspired me to stay in the academics. My sister steered me towards science. Without their love, support and guidance, my life would have been impossible to live.

My husband, Kaushik has been a role model to me as a researcher. I would not have experienced the world of research if I were not with him. He has not only been an infinite source of support, he has also guided me during the tougher times in the past four years. This thesis would have been impossible without his love and encouragement.

This research was partially supported by the European project *ADVANCE* and the *Swedish Research Council*.

Contents

Verifiering av parametriserade tidssystem

Datorsystem har blivit en del av vårt dagliga liv. Datorsystem ingår i allt från TV-apparater och mobiltelefoner till kärnkraftverk och rymdraketer. Ett litet fel i en datorkomponent kan få förödande effekter, i säkerhetskritiska datorsystem som tex ett flygkontrollsystem kan ett fel i en datorkomponent utgöra ett direkt hot mot människoliv.

Händelser som när raketen Ariane exploderade strax efter starten, problemen med omställningen till år 2000 samt tillfällen då medicinsk utrustning har satts ur funktion visar hur viktigt det är att försäkra sig om att datorsystem fungerar och inte innehåller fel som kan leda till allvarliga skador eller dyrbara driftstopp. Det är därför viktigt att, både ur säkerhetsmässig och ekonomisk synvinkel att datorsystemen är korrekta och analyseras grundligt för att de ska bli så pålitliga som möjligt.

Testning är idag den dominerande metoden för att validera och verifiera att datorsystemen fungerar korrekt, men utvecklingen inom designen av systemkomponenter har gjort det omöjligt att verifiera att ett datorsystem verkligen fungerar korrekt genom att bara använda sig av testning. Ett datorsystem kan nämligen bete sig som förväntat i hundra testkörningar för att nästa gång, utan någon som helst synbar anledning, bete sig helt annorlunda. En annan nackdel med testning är att eftersom det bara kan användas på ett redan existerande system eller på en prototyp av systemet leder detta till att många fel upptäcks sent, eftersom datorsystemet redan utvecklats. Detta kan i många fall leda till stora kostnader i form av felsökning och om-design av systemet.

För att undvika dessa problem kan man använda sig av formell verifiering. Formell verifiering består i huvudsak av två delar. Den första delen utgörs av att man skapar en matematisk modell över hur datorsystemet fungerar och den andra delen består i att man analyserar modellen för att verifiera att systemet faktiskt uppfyller de krav man ställer på det.

Genom användningen av formell verifiering kan flera systemegenskaper som inte kan verifieras med hjälp av testning säkerställas, tex säkerhetsegenskaper. Dessutom undviker att man att fel introduceras under den tidiga utvecklingsfasen, och om fel ändå uppkommer, så ser man till att felen upptäcks så tidigt som möjligt.

Utvecklingen av datorsystem går mot allt mer komplicerade system vilket också medför att sannolikheten för att fel ska kunna inträffa ökar. När systemen blir komplicerade ökar storleken på den matematiska modellen av systemet och det blir svårare att analysera modellen. Analysen av ett komplicerat system måste utföras automatiskt med hjälp av en dator eftersom ett datorsystem kan innehålla miljontals hårdvarukomponenter och tusentals rader programkod.

Avhandlingen fokuserar i första hand på att utveckla nya metoder som effektivt kan hantera verifiering av komplicerade datorsystem, tex realtidssystem, kommunikationsprotokoll och mobila system.

Under senare år har det skett en stor utveckling inom området för verifiering av system med en oändlig tillståndsrymd. Ett system kan ha en oändlig tillståndsrymd om det innehåller obegränsade datastrukturer som t.ex. räknare, klockor, stackar och köer. Ett system kan också ha en oändlig tillståndsrymd om det är parametriserat, d.v.s. det finns möjlighet att ha ett godtyckligt antal komponenter i systemet. För parametriserade system, är vi intresserade av att undersöka om alla instanser av systemet är korrekta i ett verifieringssteg.

Den här avhandlingen beskriver system som är oändliga dels för att de innehåller obegränsade datastrukturer som klockor med reella värden, och dels för att de är parametriserade. Mer precist så kommer två olika modeller att behandlas: (a) tids-Petrinät (timed Petri net, TPN) som är en utvidgning av den klassiska Petrinätmodellen, och (b) tidsnätverk (timed network, TN) i vilken ett godtyckligt antal tidsautomater körs parallellt.

Vi behandlar verifiering av säkerhetsegenskaper för tids-Petrinät genom användning av framåt-analys. Eftersom framåt-analys inte nödvändigtvis är fullständig, så tillhandahåller vi en semi-algoritm utvidgad med accelereringstekniker för att terminering ska ske oftare på praktiska exempel. Vi kommer också att behandla ett antal problem som är generaliseringar av de motsvarande problemen för tidsautomater och Petrinät. Till exempel, så behandlar vi *zenoness*-problemet, där vi undersöker huruvida det finns en oändlig beräkning som tar ändligt lång tid. Vi behandlar även två varianter av det s.k. *boundedness*-problemet: *syntaktisk boundedness* där både levande och döda tecken (tokens) betraktas; och *semantisk boundedness* där endast levande tecken betraktas. Vi visar att det förra problemet är avgörbart, och att det senare ej är det. Slutligen, så visar vi oavgörbarhet för *LTL model checking* både för kontinuerliga och diskreta tids-Petrinät.

Därefter behandlar vi *tidsnätverk*. Vi visar oavgörbarhet för säkerhetsegenskaper då varje komponent har två eller fler klockor. Detta resultat står i kontrast till det tidigare avgörbarhetsresultatet för fallet då varje komponent har en enda klocka. Vidare visar vi att problemet är avgörbart då klockorna antar diskreta värden. Detta avgörbarhetsresultat gäller då processerna har ett ändligt antal klockor. Slutligen, så ger vi en översikt över gränsen för det oavgörbara och avgörbara för säkerhet hos TN, genom att betrakta flera syntaktiska och semantiska variationer.

Chapter 1

Introduction

It is evident now in the software industry that the cost and the time required for testing and debugging a software is much higher than that for designing and implementation. When one uses a simulation of a complex system or tests it, one is never sure whether the system is correct or there *still* exist some subtle *bugs*. It is not even possible to estimate the number of such bugs still hidden in the implementation. As the complexity of the system increases, say by introducing new features (photo transfer, video transfer, etc) into a mobile phone, it can turn out to be completely impossible to "test" whether all software and hardware associated with the phone are working correctly.

Even though one may be tolerant if the phone has some spurious misbehaviours, no one will probably like to have eavesdropping on the phone, which can happen due to errors in telephone switching networks. Failure is also unacceptable in electronic commerce, highway and air traffic-control systems, medical instruments, etc. Nowadays, there are frequent news of failure caused by a software or a hardware system. For instance, the Ariane 5 rocket exploded in June 4, 1996 due to an unhandled exception caused by a conversion of a 64-bit floating-point number to a 16-bit signed integer. So, clearly the need for reliable hardware and software systems is critical.

A very attractive and increasingly promising alternative to simulation and testing is the approach of *formal verification*. While simulation and testing explore *some* of the possible behaviours of the system, leaving open a possibility of unexplored wrong behaviours, formal verification conducts an *exhaustive exploration* of all possible behaviours. Thus, a correctness certificate from a formal verification method is necessary for all (complex) software and hardware systems. Actually it is claimed that this approach already yielded a dramatic effect on the SLAM project [25] of Microsoft, which plans to incorporate formal verification in the Windows driver development kit.

1.1 Model Checking

Several approaches to formal verification have been proposed over the years. One of them is *Model Checking* [47] which was initially introduced as a technique for verifying finite, concurrent systems such as sequential circuit designs and communication protocols. Model checking allows verification of a desired

behavioural property of a given system (model) through exhaustive enumeration of all the (actual or symbolic) states reachable by the system and the behaviours that traverse through them. This approach has two advantages over other approaches.

- It is fully automatic and does not require user supervision or expertise in mathematical disciplines (which is a requirement for using *theorem proving*).
- It produces counterexamples in case the design contains an error. This counterexample can then be used to pinpoint the source of the error.

Applying model checking to a system consists of several tasks. The first task is to convert the design of a system to a suitable formal model M for which model checking algorithms exist. Often, this may require some abstraction due to limitation of memory (abstraction is achieved by avoiding some unimportant details). *Models* are usually specified as *transition systems* which consist of two parts, a set of states and a reflexive, transitive, binary relation (called, *transition relation*) on the set of states. Secondly, one needs to know which property (often called *specification*) the design is expected to satisfy. Then one may use a *model checker* which checks: "does the model M satisfy the property ?" If the model checker says "yes", we are happy with our model. Otherwise, the model checker tool supplies a counter-example (often called *error trace*, i.e., a trace which does not satisfy the specification), which can then be used to modify the model/system.

1.2 Model Checking Finite-State Systems

A state is a snapshot of a system at a moment in time. It is everything that we need to know about the system at that moment in order to determine the future for all forthcoming input sequences. More precisely, it consists of the values of all variables and control locations. A system is called *finite-state* if the number of possible states of the system is finite. However, this number may be exceedingly high for practical systems. The model checking of finite-state systems started with the works of [45, 119] in the early eighties by Clarke, Emerson, Sistla, Queille, and Sifakis. During model-checking, large finite-state systems may suffer from an explosion of its state-space. The battle against state-space explosion was started in 1987 by McMillan, with the use of *reduced ordered binary decision diagrams* (ROBDDs, introduced by Bryant [38]) for model checking [105]. ROBDDs are used to characterize a finite set of states, by canonically representing a set of evaluations of a boolean formula. This allowed description of transition systems by ROBDDs and thus efficient model checking of large finite-state systems [39, 40].

In parallel, another technique came up for combatting state-explosion: *partial order reduction*. This technique is mainly used in model checking concurrent, asynchronous software systems when most of the activities by different processes are performed *independently* (two events are considered independent

if executing them in any order results in the same global state). The partial order reduction technique makes it possible to decrease the number of interleavings of such independent events to be considered during model checking. They use the fact that when a specification cannot distinguish between two interleaving sequences that differ only in the order of concurrently executed events, then it is sufficient to analyse only one of them. The works of [112, 92, 131, 75, 114, 106, 65] investigate different methodologies for partial order reduction.

There are still other techniques for fighting state-space explosion in model checking finite-state systems (complex circuits and protocols) : e.g., *compositional reasoning, symmetry, etc.* In *compositional reasoning* [48, 78, 79, 98, 116], the specification of the system can often be decomposed into properties that describe the behaviour of small parts of the system. An obvious strategy is to check local properties for each small part of the system, show that conjunction of each of these properties satisfy the global specification and then infer that complete system satisfies this global specification as well. *Symmetry* is also used to fight state-space explosion. Finite-state system often consists of replicated components. This fact is then used to reduce the model of the system, e.g., in [46, 59].

1.3 Model Checking Infinite-State Systems

In general, software systems often have an infinite state-space rather than a finite one. A system may have an infinite state space since it operates on unbounded data structures, e.g. timed automata [17], hybrid automata [82], data-independent systems [90, 134], relational automata [42], counter machines [87, 4], pushdown processes [41], lossy channel systems [7], completely specified protocols [66] etc. A system may also have an infinite state-space since it has an unbounded control part, e.g. Petri nets [61, 88], and parameterized systems [71, 8, 93], in which the topology of the system is parameterized by the number of processes inside the system.

The theory of computability [86] already provides a limitation on what can be computed by an algorithm. It shows that there cannot be an algorithm which can take any arbitrary algorithm P as input and decide whether P terminates. This immediately limits the systems which can be verified automatically.

Initially model checking was advocated only for finite state systems. Since the last decade, a major challenge has been to extend the applicability of model checking to the context of infinite-state systems. Roughly, there are two methods for model checking infinite-state systems.

- *Abstraction.* One way to verify infinite-state systems is to come up with an *abstraction function* which maps a set of concrete states to a finite set of abstract states. The goal is to have a finite *abstract image* of the concrete system and apply finite-state model checking techniques on the abstract image [97, 44]. However, inventing the abstraction function is usually very hard for an arbitrary system.

- *Extending Symbolic Methods.* Another way is to find a finite, symbolic representation of an infinite set of states of the system under consideration. Thus, a *symbolic state* represents a (possibly) infinite set of actual states of a system. In this thesis, we will be interested in verifying safety properties. For instance, we ask questions such as: "Given an infinite-state system, a set I of initial states and a set B of bad states, is there a path from some initial state in I to a bad state in B ?" The symbolic algorithms to solve such problems usually do one of the following.

  Forward reachability analysis: start with a set of symbolic states representing I and then compute all its *post-images*, i.e., the symbolic states reached by one or more transitions from the current symbolic state. Finally we check whether B contains some state in one of the post-images.

  Backward reachability analysis: start with a set of symbolic states representing B and then again compute the *pre-images*, i.e., the set of symbolic states from which the current symbolic state is reached in one or more steps. Finally we check whether I contains a state in any of the pre-images.

  Unfortunately, symbolic algorithms for infinite-state systems do not always terminate.

  In some cases, e.g., for systems modelled by timed automata [19], the symbolic algorithm is guaranteed to terminate. Timed automata are extended finite state machines operating on real valued-clocks. Therefore, they have an infinite state-space. However, it is possible to have a finite partitioning of the infinite state space. Each partition is called a *region*. A *region* defines the local state, integral part of the clocks and the ordering on the fractional parts of the clock values. States in the same region behave in the same manner, i.e, the effect of a transition on the states in a region results in states belonging to the same region. In [19], it was shown that the states in a region are equivalent with respect to a finite bisimulation relation. Several tools [135, 28] employ this technique for verifying real-time systems. In some other cases, e.g., for lossy channel systems [7] and (timed) Petri nets [13], it is not possible to have an equivalence relation on the state-space as a finite bisimulation relation, however it may be possible to define a preorder which is a simulation relation [5], such that the larger states simulate smaller states. Furthermore, transition systems in such cases are monotonic with respect to the preorder, i.e., transition from a smaller state can be monotonically imitated from a larger state. To prove termination for these systems, one has to show that there is no infinite sequence of symbolic states such that one does not ever encounter a larger state, i.e, the ordering on the set of states is a *well-quasi-ordering*. The symbolic algorithms for lossy channel systems and (timed) Petri nets both employ a *backward reachability* approach. The algorithms start from a symbolic set of unwanted or 'bad' states and compute pre-images iteratively. The algorithm then checks whether any initial symbolic state is reached in the process.

However, employing a *forward reachability* approach may yield non-termination in some cases. Even though one may find an acceleration scheme for some systems (e.g., Petri nets [91]), there are systems (e.g., lossy channel systems [104], timed Petri nets, etc) for which forward analysis may not terminate. Nevertheless, the work in [2] and our work in [1] show that it is worthwhile to design semi-algorithms using acceleration schemes which let many examples terminate more often. In fact, *forward reachability analysis* is practically very attractive. The set of forward reachable states contains much more information about system behaviour than backward reachable states. This is due to the fact that forward closure characterizes the set of states which arises during the execution of the system, in contrast to backward closure which only describes the states from which the system may fail. This implies for instance that forward analysis can often be used for constructing a symbolic graph which is a finite-state abstraction of the system, and which is in simulation or bisimulation of the original system (see e.g. [29, 97]). Furthermore, it is known that in general the complexity of a backward reachability algorithm is very high [125]. In many cases, thus forward analysis is more efficient than the backward one [68].

1.4 Model Checking Parameterized Systems

Currently, a main challenge in model checking is to extend its applicability to *parameterized systems*. The description of such a system is parameterized by the number of components, and the challenge is to check correctness of all instances in one verification step. In general, a symbolic algorithm for a network of an arbitrary number of components, which tries to find the set of all reachable symbolic states, may not terminate [22, 128].

Most existing methods (e.g.[71, 52, 63, 57]) for model checking of parameterized systems consider the case where each individual component is modelled as a finite-state process. Such families of finite-state systems arise frequently in the design of both hardware and software systems. Typically circuit and protocol designs are parameterized, that is they define an infinite family of systems. For instance, a bus protocol (e.g., cache coherence protocol, broadcast protocol) can accommodate an arbitrary number of processors and a mutual exclusion protocol can be given for an arbitrary number of processes. The authors in [71] construct a *Petri net* for a parameterized system of finite-state processes and then use the reachability algorithm for Petri nets to model check a given parameterized system. Petri nets are good for modelling concurrent systems. A *Petri net* is a directed bipartite graph with two kinds of nodes: *places* and *transitions*. Places may hold *tokens* and transitions may consume some tokens from a place and produce some other tokens in other places. A state of Petri net is called *marking*, which assigns a number of tokens to each place. Since each place can hold an unbounded number of tokens, Petri nets are also infinite-state systems. In the construction of [71], the number of tokens in a place p of Petri net determine how many processes are there in state p in

the parameterized system.

Cache coherence protocols and broadcast protocols are analysed in [52] and [63] respectively, by effectively constructing Petri nets with transfer arcs and for which they give the aforementioned backward reachability algorithm.

In [71, 52, 63], the set topology of parameterized systems is considered. Works of [58, 43] consider systems with ring topology and arbitrary graph topology respectively. Both use token-passing as the method of communication, reduce the system consisting of an arbitrary number of processes to a system of constant size and finally use finite-state model checking on the reduced system.

Regular model checking is also used to verify parameterized systems with linear [3, 93, 33, 11, 12] or tree-like topology [10, 36]. In regular model checking, states are represented by words (trees) over a finite alphabet and the transitions are represented by a regular length-preserving relation on words (trees) and one is interested in computing the transitive closure. The above works necessarily produced semi-algorithms, since the transitive closure of a regular relation is not necessarily regular [86].

There are in fact many more approaches like network invariants [23, 117], predicate abstraction [96], or symmetry [46, 59] for verifying parameterized systems.

1.5 Model Checking Parameterized Timed Systems

Parameterized timed systems are families of systems in which each component is a simple timed automaton rather than a finite-state machine. Timed automata have an infinite state-space due to the real-valued clocks in the model. In *parameterized timed systems*, thus the infiniteness of state-spaces results from two sources : (a) parameterization and (b) real-valued local clocks.

For analysing parameterized timed systems, we consider two models : (a) the *timed Petri net* (TPN) model which is an extension of the classical Petri net model; and (b) the *timed network* (TN) model in which an arbitrary number of timed automata run in parallel.

A number of timed extensions of Petri nets [122, 107, 30, 121, 85, 50, 126, 72, 14], have been proposed in order to capture the timing aspects of the concurrent systems (see [37] for a survey). We use the TPN model of Abdulla and Nylén [14], in which each token is equipped with a real-valued clock and each arc between a place and a transition is equipped with a time interval. TPNs allow arbitrarily many tokens in each place and therefore can model parameterized timed systems. We consider TPNs for analysing several properties of concurrent, real-time systems in this thesis.

Timed networks model was introduced by Abdulla and Jonsson in [9], to analyse parameterized timed systems. A *timed network* represents a family of systems, each consisting of a finite-state *controller*, together with a finite number of arbitrarily many *timed processes* (timed automata). A timed process operates on a finite number of real-valued clocks. This means that a timed network operates on an unbounded number of clocks, and therefore its behaviour cannot

be captured by that of a timed automaton [19]. In [9], the authors use backward analysis of timed networks to verify mutual exclusion for a parameterized version of Fischer's protocol [124]. This protocol achieves mutual exclusion by defining timing constraints on an arbitrary set of processes each with one clock.

1.6 Contribution

The contribution of this thesis is divided into two parts. In Part I of this thesis, we consider timed Petri net model for analysing parameterized systems with one clock per process. We consider timed network model for analysing multi-clock parameterized systems in Part II of this thesis.

1.6.1 Part I: Single-clock timed systems

We consider verification of *safety* properties for timed Petri nets using forward analysis. Since forward analysis is necessarily incomplete, we provide a semi-algorithm augmented with an acceleration technique in order to make it terminate more often on practical examples. Then we consider a number of problems which are generalisations of the corresponding ones for *timed automata* and *Petri nets*. For instance, we consider *zenoness* where we check the existence of an infinite computation with a finite duration. We also consider two variants of the *boundedness* problem: *syntactic boundedness* in which both live and dead tokens are considered; *semantic boundedness* where only *live tokens* are considered. We show that the former problem is decidable, while the latter is not. Finally, we show undecidability of *LTL model checking* both for dense and discrete timed Petri nets.

In the following, we give an outline of each chapter in Part I.

Chapter 2

In this chapter, we give some preliminary definitions.

Chapter 3

In this chapter, we introduce (timed) Petri nets and give some examples which are modelled as timed Petri nets.

Chapter 4

In Chapter 4, we define the *coverability problem* and show how safety can be reduced to coverability problem for (timed) Petri nets. We also give an overview of forward and backward reachability analysis for Petri nets. Finally, we recall the backward analysis for timed Petri nets.

Chapter 5

In Chapter 5, we show how to verify safety properties for TPN models using *forward analysis*. Forward analysis is practically very appealing. The set of forward reachable states contains much more information about system behaviour than backward reachable states. We provide an abstraction of the set of reachable markings by taking its *downward closure* and introduce a symbolic representation for downward closed sets, which we call *region generators*. Each region generator denotes the union of an infinite number of *regions* [18], and thus characterizes a state of TPN with arbitrary number of processes in their local states and their respective clock values. We show that region generators allow the basic operations in forward analysis, i.e, checking membership, entailment, and computing the post-images with respect to a single transition. Since forward analysis is incomplete, we also give an acceleration scheme to make the analysis terminate more often. If the algorithm terminates, then we compute an abstract graph which is a simulation of the original timed Petri net. This allowed us to analyse parameterized versions of Fischer's protocol, Lynch and Shavit's protocol [100] and a producer/consumer protocol [109].

A preliminary version of this chapter is published as follows.

Forward Reachability Analysis of Timed Petri Nets. Parosh Abdulla, Johann Deneux, Pritha Mahata and Aletta Nylén. In Proc. FORMATS-FTRTFT '04. Joint Conference on Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault-Tolerant Systems, Grenoble, France, September 22-24, 2004. Volume 3253 of Lecture Notes in Computer Science. Pages 343-362. Full version is available as Technical Report 2003-056. Department of Information Technology, Uppsala University, Sweden.

Chapter 6

In this chapter, we investigate the progress property for a timed Petri net model. A fundamental progress property for timed systems is that it should be possible for time to *diverge* [129]. This requirement is justified by the fact that timed processes cannot be infinitely fast. Computations violating this property are called *zeno* and markings from which zeno computations start are called *zeno-markings*. The problem which asks whether a marking is a zeno-marking is called *zenoness-problem*. The zenoness problem was left open by Escrig, et.al [51] both for dense-timed Petri nets and for discrete-timed Petri nets (where behaviour is interpreted over the discrete time domain).

In this chapter, we show how to compute the set of all zeno-markings for a timed Petri net with both dense-time and discrete-time semantics.

Chapter 7

In this chapter, we consider several verification problems for timed Petri nets.

(i) *Token liveness:* whether a token is *alive* in a marking, i.e., whether there is a computation from the marking which eventually consumes the token. We show decidability of this problem by reducing it to the *coverability problem*.

(ii) *Boundedness:* whether the size of reachable markings is finite. We consider two versions of the problem; namely *semantic boundedness* where only live tokens are taken into consideration in the markings, and *syntactic boundedness* where also dead tokens are considered. We show undecidability of semantic boundedness, while we prove that syntactic boundedness is decidable.

(iii) *Repeated Reachability:* Finally, we consider *repeated reachability* problem, in which one asks the question: whether starting from a marking, there is a computation which visits a place infinitely often. We show undecidability of this problem both for dense and discrete-timed Petri nets. This implies that LTL model checking is also undecidable for both of these variants of timed Petri nets.

A preliminary version of Chapter 6 and a part of Chapter 7 appears in:

Zenoness, Syntactic Boundedness and Token-Liveness of Dense-Timed Petri Nets. Parosh Abdulla, Pritha Mahata and Richard Mayr. In Proc. FSTTCS '04. Foundations of Software Technology and Theoretical Computer Science, Chennai, India, December 16-18, 2004. Volume 3328 of Lecture Notes in Computer Science. Pages 59-71. Also available as Technical Report 2004-034. Department of Information Technology, Uppsala University, Sweden.

The other part of Chapter 7 will appear in:

Model Checking LTL for Timed Petri Nets. Parosh Abdulla, Pritha Mahata and Aletta Nylén. In Proc. ARTES '05.

1.6.2 Part II: Multi-clock timed systems

In Part II of this thesis, we extend the timed network model of [9] for multi-clock timed systems, where each component of the system can be equipped with a finite number of real-valued clocks (instead of just one clock). This extension is natural, because timed network modelling of real-time systems is natural.

The authors in [9] showed decidability of the *controller state reachability problem* for TNs: given a state of the controller, is there a computation from an initial configuration leading to that state? This problem is relevant since it can be shown, using standard techniques, that checking large classes of safety properties can be reduced to controller state reachability. The decidability result in [9] is given subject to the restriction that each timed process has a single clock. The paper [9] leaves open the case of *multi-clock* TNs, i.e., TNs where each timed process may have several clocks.

In the literature, there are many applications where a number of timed automata [19] run in parallel and where each of the timed automata has more than one clock. For instance, the Phillips audio control protocol with bus collision [27] has two clocks per sender of audio signals. Also, the system described in [101] consists of an arbitrary number of nodes, each of which is connected to a set of LANs. Each node maintains timers to keep track of sending and receiving of messages from other nodes connected to the same set of LANs. In a similar way to Fischer's protocol, it is clearly relevant to ask whether we can verify correctness of the protocol in [27] regardless of the number of senders, or the protocol in [101] regardless of the number of nodes.

The question is then whether the decidability result of [9] can be extended to multi-clock systems. In fact, we answer this question negatively. Also, we show that the problem is decidable when clocks range over the discrete time domain. This decidability result holds when the processes have any finite number of clocks. Furthermore, we outline the border between decidability and undecidability of safety for TNs by considering several syntactic and semantic variants.

In the following, we give an outline of each chapter in Part II.

Chapter 8

In this chapter, we show that it is sufficient to allow two clocks per process in order to get undecidability of the controller-state reachability problem for multi-clock TNs. The undecidability result is shown through a reduction from the classical reachability problem for 2-counter machines.

Chapter 9

In this chapter, we show the decidability of the controller state reachability problem when timed networks are interpreted over the discrete time domain. The algorithm adapts the backward reachability algorithm for discrete timed networks. We also show that the complexity of this algorithm is non-primitive recursive.

A preliminary version of Chapter 8 and Chapter 9 appeared in:

Multi-Clock Timed Networks. Parosh Abdulla, Johann Deneux and Pritha Mahata. In Proc. LICS '04. In 19th Annual IEEE Symposium on Logic in Computer Science, Turku, Finland, July 13-17, 2004. IEEE Computer Society Press. Pages 345-354. Also available as Technical Report 2004-012. Department of Information Technology, Uppsala University, Sweden.

Chapter 10

In this chapter, we restrict the expressivity of timed networks syntactically (by syntactic removal of equality testing) and study two syntactic subclasses of timed networks : *closed timed networks* and *open timed networks*. In the former, only non-strict clock constraints (of the form $x \leq 3$ or $x \geq 2$) are allowed, while in the latter one, only strict clock constraints (of the form $x < 3$ or $x > 2$) are allowed. It turns out that controller state reachability problem is decidable for closed timed networks, while it remains undecidable for open timed networks. The latter result strengthens the undecidability result of Chapter 8. But the undecidability proof is little complicated for open timed networks.

In this chapter, we also consider *robust timed networks* by introducing "timing fuzziness" and semantic removal of equality testing. The problem remains undecidable even for robust semantics of TNs.

A preliminary version of this chapter appeared in:

Closed, Open and Robust Timed Networks. Parosh Abdulla, Johann Deneux and Pritha Mahata. In Proc. Infinity '04. In 6th Workshop on Verification of Infinite-State Systems, London, U.K, September 4, 2004. Also available as Technical Report 2004-033, Department of Information Technology, Uppsala University, Sweden.

Chapter 11

In this chapter, we conclude and give directions for future research.

1.7 Other Works

- *Regular Tree Model Checking*. Parosh Abdulla, Bengt Jonsson, Pritha Mahata and Julien d'Orso. In Proc. Conference on Computer-Aided Verification (CAV '02), Copenhagen, Denmark, July 27-31, 2002.
- *Downward Closed Language Generators*. Parosh Abdulla, Pritha Mahata and Aletta Nylén. In 14th Nordic Workshop on Programming Theory (NWPT '02), Tallinn, Estonia, November 20-22, 2002.

Part I

Single-Clock Timed Systems

Chapter 2

Preliminaries

We use $\mathbb{N}, \mathbf{Z}, \mathbb{R}^{\geq 0}, \mathbb{R}^{>0}$ to denote the sets of natural numbers, integers, non-negative and positive real numbers respectively. We also use a set *Intrv* of intervals of natural numbers. An open interval is written as (k_1, k_2) where $k_1 \in \mathbb{N}$ and $k_2 \in \mathbb{N} \cup \{\infty\}$. Intervals can also be closed in one or both directions, e.g. $[k_1, k_2)$ is closed to the left and open to the right.

2.1 Multisets

For a set A, we use $A^{\circledast}$ to denote the set of finite multisets over A. We view a multiset over A as a mapping from A to $\mathbb{N}$. Sometimes, we write multisets as lists, so $b = [2.4\ ,\ 5.1\ ,\ 5.1\ ,\ 2.4\ ,\ 2.4]$ represents a multiset b over $\mathbb{R}^{\geq 0}$ where $b(2.4) = 3$, $b(5.1) = 2$ and $b(x) = 0$ for $x \neq 2.4, 5.1$. We may also write b as $\left[2.4^3\ ,\ 5.1^2\right]$. For multisets b_1 and b_2 over $\mathbb{N}$, we write $b_1 \leq^m b_2$ ($b_1 \geq^m b_2$) if $b_1(a) \leq b_2(a)$ ($b_1(a) \geq b_2(a)$) for each $a \in A$. We define the *addition* $b_1 + b_2$ of multisets b_1, b_2 to be the multiset b where $b(a) = b_1(a) + b_2(a)$, and (assuming $b_1 \leq^m b_2$) we define the *subtraction* $b_2 - b_1$ to be the multiset b where $b(a) = b_2(a) - b_1(a)$, for each $a \in A$. We say that $b_1 <^m b_2$ if $b_1 \leq^m b_2$ and $b_1 \neq b_2$. We use ϵ to denote the empty multiset .

2.2 Words

For a set A, we use A^* to denote the set of finite words over A. Recall that $A^{\circledast}$ denotes the set of finite multisets over A. In the above, we used ϵ to denote the empty multiset. Abusing notations, we use ϵ also for the empty word.

For a word $w \in L$, we use $|w|$ to denote the length of w, and $w(i)$ to denote the i^{th} element of w where $1 \leq i \leq |w|$. We use $w_1 \bullet w_2$ to denote the concatenation of the words w_1 and w_2.

Assume that $\preceq$ is an ordering on a set A. We define the ordering $\leq^w$ on the set of words over A such that $w_1 \leq^w w_2$ if there is a strictly monotonic injection $h : \{1, \ldots, |w_1|\} \rightarrow \{1, \ldots, |w_2|\}$ where $w_1(i) \preceq w_2(h(i))$ for $i : 1 \leq i \leq |w_1|$.

2.3 Vectors

For a natural number k, we let A^k denote the set of vectors of size k over A. We use ω to denote an arbitrarily large number. For a natural number k, we also let $\mathbb{N}_\omega^k$ denote the set of vectors of size k over $\mathbb{N} \cup \{\omega\}$. For any $z \in \mathbb{N}$, we assume that $z + \omega = \omega + z = \omega + \omega = \omega$.

As usual, we write members of A^k (vectors) as tuples $(a_1, \ldots, a_k)$ where $a_1, \ldots, a_k \in A$. In later chapters, if the size of the vectors is known from the context, instead of $(a_1, \ldots, a_k)$, we write $\vec{a}$. We also use $\vec{a}(i)$ to denote the i-th component a_i in $\vec{a}$.

Assuming that $\preceq$ is the ordering on A, we define the ordering $\leq^k$ on A^k such that $(a_1, \ldots, a_k) \leq^k (a'_1, \ldots, a'_k)$ iff $a_i \preceq a'_i$ for $i : 1 \leq i \leq k$.

Also, we allow elements of vectors from different alphabets. For instance, $A \times B \times C$ denotes the set of vectors of size 3 and members of $A \times B \times C$ are tuples (a, b, c) where $a \in A, b \in B, c \in C$.

2.4 Well-Quasi-Ordering

A *quasi-ordering* $\preceq$ on a set A, is a reflexive, transitive binary relation on the set A. A quasi-ordering is said to be a *well-quasi ordering* if for each infinite sequence $a_0, a_1, a_2, \ldots$ with $a_0, a_1, a_2, \ldots \in A$, there are always i, j with $i < j$ such that $a_i \preceq a_j$. In such case, A is said to be a *(well-)quasi-ordered* set. For instance, a finite set A with equality as the ordering on A, is trivially well-quasi-ordered. Furthermore, from [54], we have the following.

Lemma 2.1. The ordering $\leq^m$ on the set $A^{\circledast}$ of all finite multisets over a well-quasi-ordered set A is a well-quasi ordering.

Given $\preceq$ as a well-quasi-ordering on A and the fact that well-quasi-ordering is preserved on the set of words (also, vectors) built over A ([84]), we get the following.

Lemma 2.2. Given a set A and an well-quasi-ordering $\preceq$ on the set A, the ordering $\leq^w$ on the set A^* of all finite words over A is a well-quasi-ordering.

Notice that the underlying set A need not be finite. In fact, to define regions in Chapter 4, we use the underlying set A to be the set of multisets $B^{\circledast}$ for a finite set B. In that case $\preceq$ is the ordering $\leq^m$ defined on multisets.

In a similar manner, it follows that the ordering on the set of vectors is also a well-quasi-ordering if the underlying sets are well-quasi-ordered.

2.5 Upward and Downward Closedness

Given a quasi-ordered set A with $\preceq$ as the ordering and a set $B \subseteq A$, B is said to be *upward closed* if $a_1 \in B, a_2 \in A$ and $a_1 \preceq a_2$ implies $a_2 \in B$. A set M is said to be *canonical* if for any $a, b \in M$ with $a \neq b$, $a \not\preceq b$. We say that $M \subseteq A$ is

a *minor set* of A, denoted A_{min}, if for all $a \in A$, there exists a $b \in M$ such that $b \preceq a$ and M is canonical. It is shown in [5] that if $\preceq$ is a well-quasi-ordering, then for each set A, there is a finite minor set A_{min}. Given a set $B \subseteq A$, we define the *upward closure* $B\uparrow$ to be the set $\{a \in A \mid \exists a' \in B : a' \preceq a\}$. A *downward closed* set B and the *downward closure* $B\downarrow$ are defined in a similar manner. Often, we use $a\uparrow$, $a\downarrow$, a instead of $\{a\}\uparrow$, $\{a\}\downarrow$, $\{a\}$ respectively.

2.6 Transition Systems

The operational semantics of a formal model is usually given as a *transition system*. A *transition system* is denoted by a pair $(S, \longrightarrow)$ where S is a (possibly infinite) set of states of the modelled system and $\longrightarrow \subseteq S \times S$ is a transition relation on the set of states. We use $\xrightarrow{*}$ to denote the reflexive, transitive closure of $\longrightarrow$. Given two states $s_1, s_2 \in S$, if $s_1 \xrightarrow{*} s_2$, then we say that s_2 is *reachable* from s_1. Also, given a set S_1 of states, we let $Reach(S_1) = \left\{s' \mid \exists s \in S_1 . s \xrightarrow{*} s'\right\}$. We also use two functions called *Pre* and *Post* for computing *pre-images* and *post-images* of a set S_1 of states as follows. We define $Pre(S_1) = \{s' \mid \exists s \in S_1 . s' \longrightarrow s\}$ and $Post(S_1) = \{s' \mid \exists s \in S_1 . s \longrightarrow s'\}$. We use Pre^* and $Post^*$ to denote the transitive closure of *Pre* and *Post* respectively.

We say that a transition system is *monotonic* if the transition relation $\longrightarrow$ is monotonic with respect to $\preceq$, i.e., given states $s_1, s_2, s_3 \in s$, $s_1 \longrightarrow s_2$ and $s_1 \preceq s_3$, there is a state $s_4 \in S$ such that $s_3 \longrightarrow s_4$ and $s_2 \preceq s_4$.

In the following chapters, we give the operational semantics of different models as transition systems. In fact, all the models considered in this thesis induce monotonic transition systems.

Chapter 3

(Timed) Petri Nets

Petri nets are one of the most widely used graphical models for analysis and verification of concurrent systems. They were first introduced by C. A. Petri [115]. Petri nets offer two advantages in the analysis of concurrent systems: ease of modelling and formal analysis. In the last four decades, much research has been done to develop algorithms for model checking of Petri nets. In this chapter, we define Petri nets and show how they can be extended in order to also capture timing aspects of concurrent systems.

3.1 Petri Nets

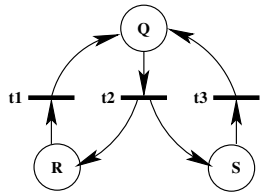

Figure 3.1: A small Petri net.

A *Petri net* N is a directed bipartite graph consisting of two kinds of nodes, namely, *places* (represented by circles) and *transitions* (represented by bars). If there is an arc from a place p to a transition t, then p is called an *input place* of t. Similarly, if there is an arc from a transition t to a place p, then p is called an *output place* of t. Formally, a *Petri Net (PN)* is a tuple $N = (P, T, F)$ where P is a finite set of places, T is a finite set of transitions and $F \subseteq P \times T \cup T \times P$ is the flow relation.

Furthermore, *tokens* reside in the places of a Petri net. A state (called a *marking*) of a Petri net is given by the number of tokens in each place. Formally, a marking is a multiset over the set of places of N. The state of N can be changed by *firing* transitions. Firing of a transition t corresponds to removing tokens from the input places and adding tokens to the output places of t.

A Petri net N induces a transition system $(\mathsf{M}, \longrightarrow)$ where M is the set of

markings of N and $\longrightarrow = \bigcup_{t \in T} \longrightarrow_t$, such that $\longrightarrow_t$ is given as follows.

Given a transition t, the set I of its input places and the set O of its output places, we say that t is enabled from a marking M if $I \leq^m M$. In such a case, $M \longrightarrow_t M'$ where $M' = (M - I) + O$. Given $\mathbf{T} = t_1 \dots t_n$ as a sequence of transitions (where $t_1, \dots, t_n \in T$), we say that $M_1 \longrightarrow_{\mathbf{T}} M_2$ iff there are markings $M'_1, \dots, M'_{n-1}$ such that $M_1 \longrightarrow_{t_1} M'_1 \longrightarrow_{t_2} \dots \longrightarrow_{t_n} M_2$.

Example 3.1. Figure 3.1 shows an example of a Petri net where the set of places is $\{Q, R, S\}$ and the set of transitions is $\{t_1, t_2, t_3\}$. A marking of the given net is given by $M_0 = [Q, R]$. Also, $M_0 \longrightarrow_{t_2} M_1$ where $M_1 = [R, R, S]$. □

3.1.1 Parameterized Systems

One application of Petri nets, which is of particular interest in this thesis, is the modelling of *parameterized systems*. A *parameterized system* consists of an arbitrary number of identical components of (finitely many) different types, which are typically finite-state machines. Many resource allocation algorithms, network protocols, mutual exclusion algorithms, etc can be handled in this model. Following [71], we show how to construct a Petri net for modelling such parameterized systems through an example. Suppose, we are given two

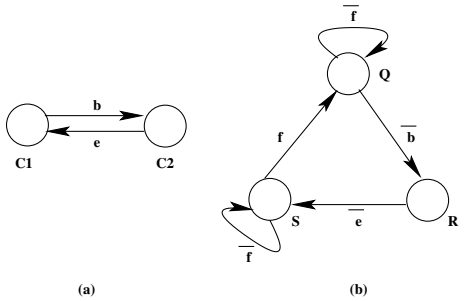

Figure 3.2: A system consisting of processes of two different types.

types of processes, a controller process and a template for user processes, described in Figure 3.2.(a) and (b) respectively, as communicating finite state machines. The controller can synchronize on the action labels b, e with some user process and then both the controller and the user process can change their states simultaneously. While the controller remains in the same state, two user processes can also synchronize on the action label f. The controller and the user processes can also perform transitions independently (not shown in this example).

The set of places in the corresponding Petri net is given by $P = \{C_0, C_1, C_2\} \cup \{Q, R, S\}$ where we add a new place C_0 in the Petri net for the initialization purpose (explained later). The number of tokens in each place corresponds

to the number of processes in the corresponding state. We ensure that there is always a token in either C_0 or C_1 or C_2. The number of tokens in places Q, R, S determine how many processes are in those states. In Figure 3.3, we give a Petri net N which simulates the communication between the controller and an arbitrary number of user processes. In Figure 3.4, we give a marking $[C_1, R, R, Q]$ of N which represents a state of the parameterized system with the controller in state C_1, two user processes in state R and one user process in state Q respectively.

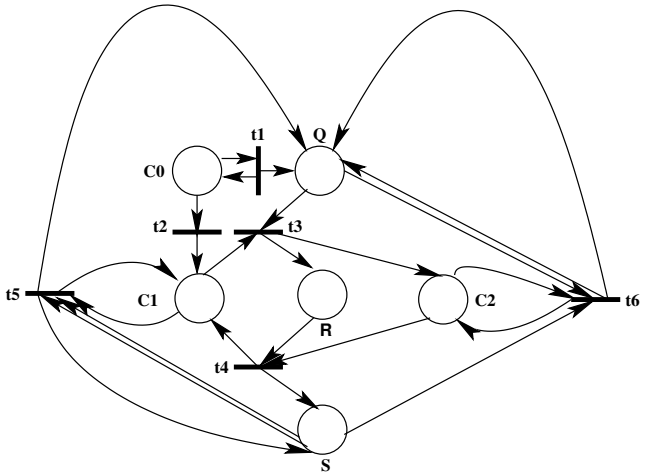

Figure 3.3: Petri net.

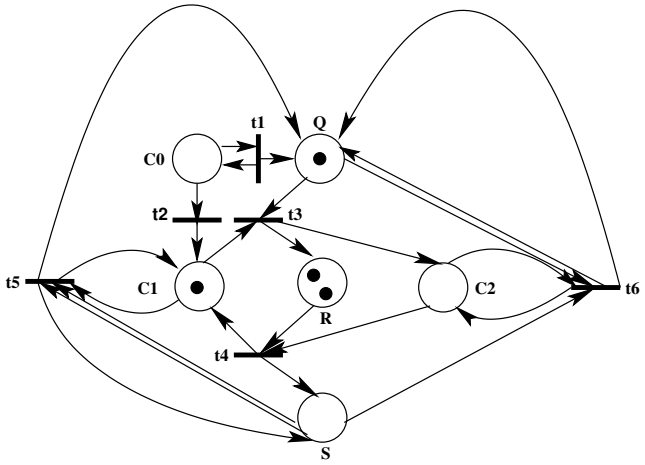

Figure 3.4: A state of the parameterized system.

There are two phases in the Petri net modelling a parameterized system (see

Figure 3.3). Firstly, we have *initialization* phase in which we simulate the participation of any arbitrary number of user processes. Secondly, we simulate the communication between different processes.

- An arbitrary number of tokens in place Q is created by firing the transition t_1 successively from the initial marking $[C_0]$. This simulates arbitrary number of processes starting at the initial state Q of the user processes. Then by firing t_2, we move the token from C_0 to C_1 and finish the initialization. Then the usual communication between different processes takes place.

- A controller process and a user process can synchronize on the action labels b and $\overline{b}$ (see Figure 3.2). For instance in Figure 3.2, the controller process in state C_1 may communicate with a process in state Q and the two processes may change their states to C_2 and R respectively. This is simulated by firing the transition t_3. Firing of the transition t_4 corresponds to the synchronization on the action label e and $\overline{e}$ and can be explained in a similar manner. Finally, the communication between user processes with action label f and $\overline{f}$ is simulated by firing the transitions t_5 and t_6. In these transitions, a token moves from place S to place Q while another token is present in place S or place Q. Notice that in the last two transitions, the token for the controller stays in the same place (C_1 or C_2).

3.2 Timed Petri Nets

In order to capture the timing aspects of the concurrent systems, we consider the timed Petri net model of [14], in which each token is equipped with a real-valued clock. The firing conditions of a transition include the usual ones for Petri nets. Additionally, each arc between a place and a transition is labelled with a sub-interval of natural numbers. When firing a transition, tokens which are removed (added) from (to) places should have ages in the intervals of corresponding arcs. Notice that this model assumes a lazy (non-urgent) behaviour of timed Petri net model.

Timed Petri Nets *Timed Petri Net (TPN)* is a tuple $N = (P, T, In, Out)$ where P is a finite set of *places*, T is a finite set of *transitions* and In, Out are partial functions from $T \times P$ to $Intrv$.

If $In(t, p)$ ($Out(t, p)$) is defined, we say that p is an *input (output) place* of t.

Unlike the black tokens of untimed Petri nets, the tokens in TPNs are equipped with real-valued ages. We use the pair (p, x) to denote a token of TPN in place p with age x. Abusing notations, we write $p(x)$ instead of (p, x). We shall also use a similar notation[1] for arcs of TPNs, i.e., we write $p(\mathcal{I})$ instead of $(p, \mathcal{I})$ where $\mathcal{I} \in Intrv$.

[1] Later, we use a similar notation, too. We shall write $p(n)$ instead of (p, n) where $n \in \mathbb{N}$.

For each transition $t \in T$, there is a set of input arcs $\mathcal{A}_{in}(t) = \{p(\mathcal{I}) \mid In(t,p) = \mathcal{I}\}$ and a set of output arcs $\mathcal{A}_{out}(t) = \{p(\mathcal{I}) \mid Out(t,p) = \mathcal{I}\}$. We let *max* be the maximal finite constant that appears in the arcs of the TPN.

Markings A *marking* M (state) of N is a multiset over $P \times \mathbb{R}^{\geq 0}$. The marking M defines the numbers and ages of tokens in each place in the net. That is, $M(p(x))$ defines the number of tokens with age x in place p. For example, if $M = [p_1(2.5)\ ,\ p_1(1.3)\ ,\ p_2(4.7)\ ,\ p_2(4.7)]$, then, in the marking M, there are two tokens with ages 2.5 and 1.3 in place p_1, and two tokens each with age 4.7 in place p_2. Abusing notation again, we define, for each place p, a multiset $M(p)$ over $\mathbb{R}^{\geq 0}$, where $M(p)(x) = M(p(x))$. Notice that untimed Petri nets are a special case in our model where all intervals are of the form $[0, \infty)$.

For a marking M of the form $[p_1(x_1)\ ,\ \ldots\ ,\ p_n(x_n)]$ and $\delta \in \mathbb{R}^{\geq 0}$, we use $M^{+\delta}$ to denote the marking $[p_1(x_1 + \delta)\ ,\ \ldots\ ,\ p_n(x_n + \delta)]$.

Transitions There are two types of transition relations: *timed* and *discrete* transitions. A *timed transition* $\longrightarrow_{T=\delta}$ increases the age of each token in a marking by the same real number δ. Formally, for $\delta \in \mathbb{R}^{\geq 0}$, $M_1 \longrightarrow_{T=\delta} M_2$ if $M_2 = M_1^{+\delta}$. We use $M_1 \longrightarrow_{Time} M_2$ to denote that $M_1 \longrightarrow_{T=\delta} M_2$ for some $\delta \in \mathbb{R}^{\geq 0}$.

We define the set of *discrete transitions* $\longrightarrow_{Disc}$ as $\bigcup_{t \in T} \longrightarrow_t$, where $\longrightarrow_t$ represents the effect of firing the transition t. More precisely, $M_1 \longrightarrow_t M_2$ if the set of input arcs $\mathcal{A}_{in}(t)$ is of the form $\{p_1(\mathcal{I}_1), \ldots, p_k(\mathcal{I}_k)\}$, the set of output arcs $\mathcal{A}_{out}(t)$ is of the form $\{q_1(\mathcal{J}_1), \ldots, q_\ell(\mathcal{J}_\ell)\}$, and there are multisets $b_1 = [p_1(x_1)\ , \ldots\ , p_k(x_k)]$ and $b_2 = [q_1(y_1)\ , \ldots\ , q_\ell(y_\ell)]$ such that the following holds:
- $b_1 \leq^m M_1$
- $x_i \in \mathcal{I}_i$, for $i : 1 \leq i \leq k$.
- $y_i \in \mathcal{J}_i$, for $i : 1 \leq i \leq \ell$.
- $M_2 = (M_1 - b_1) + b_2$.

Intuitively, a transition t may be fired only if for each input arc to the transition, there is a token with the "right" age in the corresponding input place. These tokens will be removed when the transition is fired. The newly produced tokens have ages belonging to the relevant intervals.

We define $\longrightarrow = \longrightarrow_{Time} \cup \longrightarrow_{Disc}$. Here we recall the definitions of reachability for (timed) Petri nets. We say that M_2 is *reachable* from M_1 if $M_1 \xrightarrow{*} M_2$. We define $Reach(M)$ to be the set $\left\{M' \mid M \xrightarrow{*} M'\right\}$.

A *M-computation* π from a marking M is $M_0 \longrightarrow_{Time} M_1' \longrightarrow_{Disc} M_1 \longrightarrow_{Time} M_2' \longrightarrow_{Disc} \ldots$ of markings and transitions such that $M_0 = M$. Abusing notations, we also write an M_0-computations as $M_0 \longrightarrow M_1' \longrightarrow M_1 \ldots M_n' \longrightarrow M_n$. Furthermore, sometimes we write $M_1 \longrightarrow_{Disc} M_2$ instead of $M_1 \longrightarrow_{T=0} M_2' \longrightarrow_{Disc} M_2$.

Example 3.2. Figure 3.5 shows an example of a TPN where $P = \{Q, R, S\}$

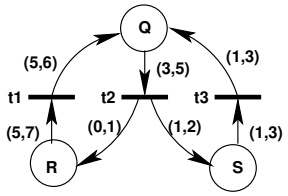

Figure 3.5: A small timed Petri net.

and $T = \{t_1, t_2, t_3\}$. For instance, $In(t_2, Q) = (3, 5)$, $Out(t_2, R) = (0, 1)$ and $Out(t_2, S) = (1, 2)$. A marking of the given net is $M_0 = [Q(2.0),\ R(4.3),\ R(3.5)]$. A timed transition from M_0 is given by $M_0 \longrightarrow_{T=1.5} M_1$ where $M_1 = [Q(3.5),\ R(5.8),\ R(5.0)]$. An example of a discrete transition is given by $M_1 \longrightarrow_{t_2} M_2$ where $M_2 = [R(0.2),\ S(1.6),\ R(5.8),\ R(5.0)]$. □

Remarks:

- One may wonder why instead of always producing a new token of age 0 after a discrete transition, we associate an interval with output arcs ! This is to allow resetting of clocks even to non-zero values (off course, we also allow the clocks to be reset to zero value).
- We assume a lazy (non-urgent) behaviour of TPNs. This means that we may choose to "let time pass" instead of firing enabled transitions, even if that disables a transition by making some of the needed tokens "too old".
- TPNs cannot be modelled within the context of real-time automata [18], as the latter operate on a finite number of clocks.

3.2.1 Parameterized Timed Systems

In fact TPNs are infinite in two directions: they have unbounded number of tokens, and each token has a real-valued clock. This implies that TPNs can, among other things, model parameterized timed systems (systems consisting of an arbitrary number of single-clock, identical timed processes) [14]. Given the descriptions of a finite number of timed processes, one can construct a TPN in a manner similar to that in Section 3.1.1. Additionally, conditions over clocks of timed processes are translated into intervals on the arcs of the TPN.

3.2.2 Related TPN Models

A number of other timed extensions of Petri nets [122, 107, 30, 121, 85, 50, 126, 72], have been proposed in order to capture the timing aspects of the concurrent systems (see [37, 133] for a survey).

In [107, 30], an interval $[t_{min}, t_{max}]$ is associated with each transition of a Petri net, where t_{min} and t_{max} represent the minimum and maximum time

delay between the enabling of the transition and its firing time respectively. The firing is *urgent*, i.e., if the transition is enabled at time t, it has to be fired before time $t + t_{max}$. In [121, 85], the authors propose to associate a firing duration to each transition. A transition starts firing as soon as it is enabled and stops exactly after a time duration. In [50, 126], a waiting time is associated with each place. When a token arrives at a place, it has to wait until the waiting time associated with the place elapses and the firing should take place immediately. In [72], each token is equipped with a real-valued clock representing the "age" of the token and each transition is associated with a *time condition*. In this timed Petri net model, the age of the token gives the time elapsed since it was produced and the time condition is a function from the timestamp of the tokens in the input places to a set of time values. For instance, if there are two input places p_1, p_2 for a transition t, the time condition may be $[time(p_1) + t_{min}, time(p_2) + t_{max}]$. This model assumes that there are never two tokens in a place which can contribute to a transition. Also, unlike the previous models, the transitions are not urgent in this model.

Our TPN model is more general than all the previous models and the behaviours of the systems modelled by all other models are subsumed by the behaviour of our model. Also, our model assumes a lazy (non-urgent) behaviour of timed Petri met model as in [72]. However, the assumption of [72] for singly enabled token in input places of a transition is not required in this model.

3.2.3 Examples

In this section, we show a few examples which we verify in Chapter 5.

Fischer's Protocol

First we describe a parameterized version of Fischer's protocol. The purpose of the protocol is to guarantee mutual exclusion in a concurrent system consisting of an arbitrary number of processes. The example was suggested by Schneider et al. [124].

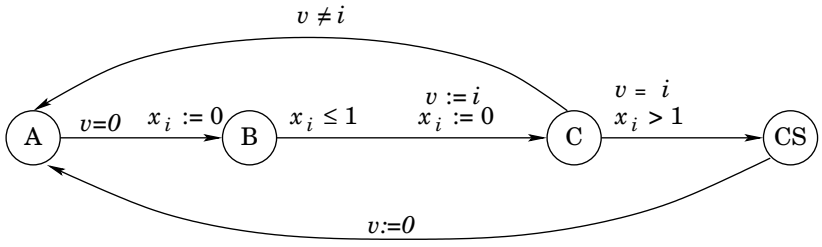

Figure 3.6: Fischer's Protocol for Mutual Exclusion

The protocol consists of an arbitrary number of processes, each running the

code graphically described in Figure 3.6. Each process i has a local clock x_i, and a control state, which assumes values in the set $\{A, B, C, CS\}$ where A is the initial state and CS is the critical section. The processes read from and write to a shared variable v, whose value is either 0 or the index (a positive integer) of one of the processes.

All processes start in state A. If the value of the shared variable is 0, a process wishing to enter the critical section can proceed to state B and reset its local clock. From state B, the process can proceed to state C within one time unit or get stuck in B forever. When making the transition from B to C, the process resets its local clock and sets the value of the shared variable to its own index. The process now has to wait in state C for more than one time unit, a period of time which is strictly greater than the one used in the timeout of state B. If the value of the shared variable is still the index of the process, the process may enter the critical section, otherwise it may return to state A and start over again. When exiting the critical section, the process resets the shared variable to 0.

The paper [14] gives a model of the protocol in our TPN formalism. The processes running the protocol are modelled by tokens in the places A, B, C, CS, $A!$, $B!$, $C!$ and $CS!$. The places marked with ! represent that the value of the shared variable is the index of the process modelled by the token in that place. We use a place *udf* to represent that the value of the shared variable is 0. A token in place *udf* means that the variable $v = 0$ and an absence of a token in *udf* means that some process i has its id assigned to the variable.

A straightforward translation of the description in Figure 3.6 yields the timed Petri net model in Figure 3.7. q is used to denote an arbitrary process state among A, B, C, CS. We illustrate translation of two transitions in Figure 3.6 by the transitions in the TPN model of Figure 3.7. Translations of other transitions can be explained in a similar manner.

In Figure 3.6, a process in state A changes its state to B if the variable value is undefined. Furthermore, it resets its clock. This is translated to the transition *initiate* in the TPN model. The transition *initiate* is fired if there is a token in place A and a token in place *udf* (denoting that the variable is undefined). Firing of the transition removes the token from the place A, adds a token with age 0 to place B (corresponds to resetting the clock and changing state to B in Figure 3.6) and leaves the variable undefined by returning a token in place *udf*.

Secondly, in Figure 3.6, a process in state B changes its state to C if its clock value is less than 1. Then it also assigns its own process id i to the variable v and resets its clock. This transition is translated to three transitions *choose1*, *choose2* and *choose3* in the TPN model. There are 3 cases.

$v = 0$. Consider a token in *udf* (denoting $v = 0$) and a token in B with age less than 1 (modelling a process in state B). Then firing transition *choose1* consumes these two tokens and puts a token of age 0 in $C!$. This means that a process in C modelled by the token in $C!$ has its id assigned to the shared variable v and has reset it clock.

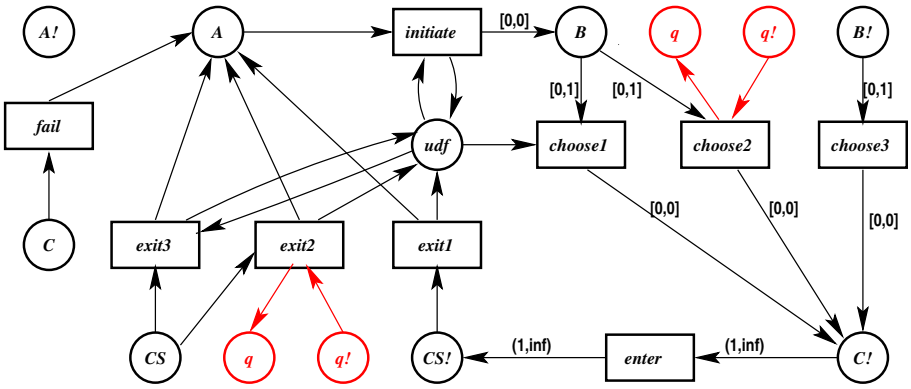

Figure 3.7: TPN model of Fischer's Protocol for Mutual Exclusion

$v = j$ ***where*** $j \neq i$**.** If there is a token in place $q!$ (i.e, some other process has its id j assigned to the shared variable) and there is a token in place B (modelling a process in state B) with clock value less than 1, we fire *choose2* and change the state of the process in $q!$ to q by removing a token from place $q!$ and adding a token to q. Also, the token from place B is moved to $C!$ and the new age of the token is 0.

$v = i$**.** If there is a token in place $B!$ (modelling a process which already has its id assigned to the shared variable), we fire the transition *choose3*, remove the token from $B!$ and add a token to $C!$ with age 0.

The critical section is modelled by the places CS and $CS!$, so mutual exclusion is violated when the total number of tokens in those places is at least two.

Lynch and Shavit's Mutual Exclusion Protocol

Lynch and Shavit [100] modified Fischer's protocol in such a way that mutual exclusion property becomes time-independent. This protocol uses an integer variable v_1 (same as v in Fischer) and an extra boolean variable v_2, shared between processes. The code for each process in Lynch and Shavit's protocol is shown in Figure 3.8 and the corresponding TPN model is shown in Figure 3.9.

In Figure 3.9, the processes running this protocol are modelled by tokens in the places A', B', C', A, B, C, CS, W, X, $A'!$, $B'!$, $C'!$ $A!$, $B!$, $C!$, $CS!$, $W!$ and $X!$. The places marked with ! represent that the value of the shared variable v_1 is the index of the process modelled by the token in that place. We use a place *udf* to represent that the value of the shared variable v_1 is undefined (0). We use two places $false$ and $true$ to represent the variable v_2. A token in place *udf* means that the variable $v_1 = 0$ and an absence of a token in *udf* means that some process i has its id assigned to the variable. A token in place $false$ and no token in place $true$ mean that the shared variable v_2 has value $false$. Shared variable v_2 with value $true$ is represented in a similar manner. Also, we

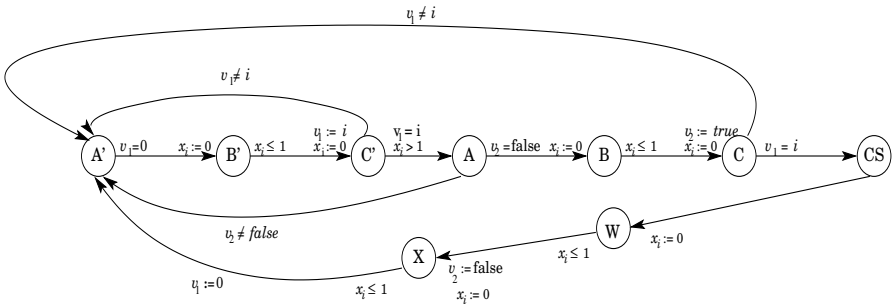

Figure 3.8: One process running Lynch and Shavit's mutual exclusion protocol

consider $q \in \{A', B', C', CS, A, B, C, W, X\}$ and $false' = false$. Notice that in this protocol, it is not compulsory to have a delay before entering the critical section CS!.

A straightforward translation of the description in Figure 3.8 yields the timed Petri net model in Figure 3.9.

Producer/Consumer System

In a traditional producer/consumer system, the producer produces items and stores it into a buffer, whereas the consumer consumes the items from the buffer. Figure 3.10 shows a timed Petri net model of the producer/consumer system [109]. A token in the place *producer_ready* means that the producer can produce *items*; firing transition *produce* creates new *items* in place *store*. The consumer consumes items of age 1 by firing *consume* if the place *consumer_ready* has a token; firing *consume* also puts back a token in place *tmp*. A transition *get_ready* is used to move the token from *tmp* back to the place *consumer_ready* if there are still items of age 0 in *store*. To make this possible, old items (of age greater than 1) in *store* are recycled by the producer using the transition *recycle*. Firing *recycle* removes an old item from the *store* if *producer_ready* has a token and puts back a fresh item (a token of age 0) back to *store*. The transition *switch* moves the control from the producer to the consumer by consuming a token from *producer_ready* and adding a token to *consumer_ready*. The transition *switch* also adds a token of age 0 to *store*, and thus lets the consumer to consume at least once. Also, the transition *done* switches the control back from the consumer to the producer by doing the reverse. These two transitions make sure that the items are not simultaneously accessed by the producer and the consumer.

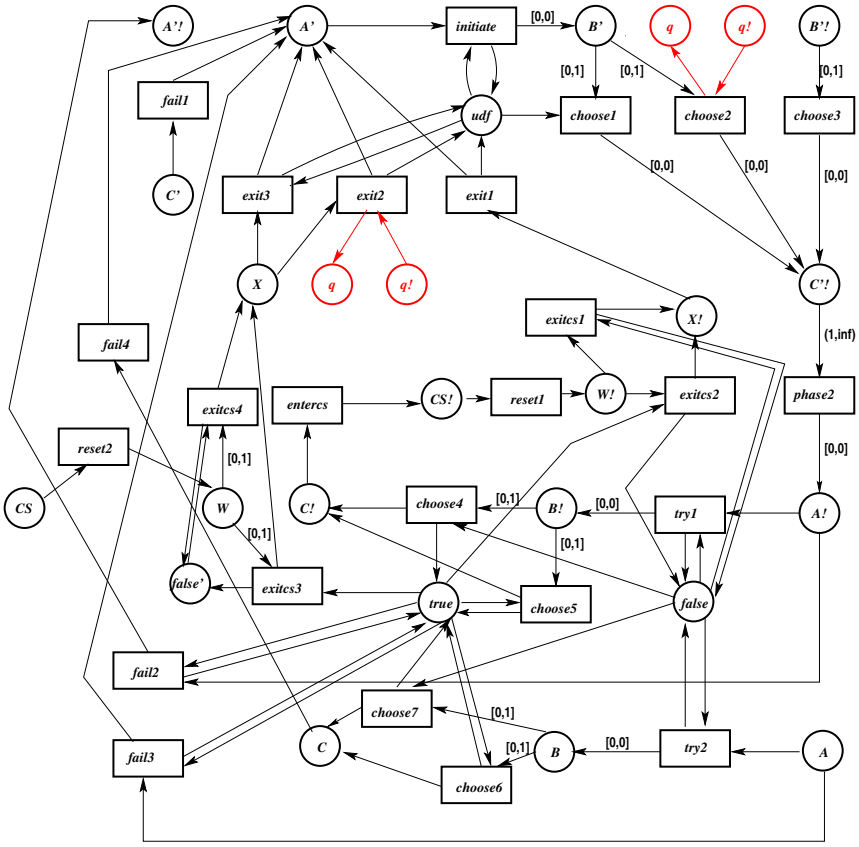

Figure 3.9: TPN model for the parameterized version of Lynch and Shavit's protocol

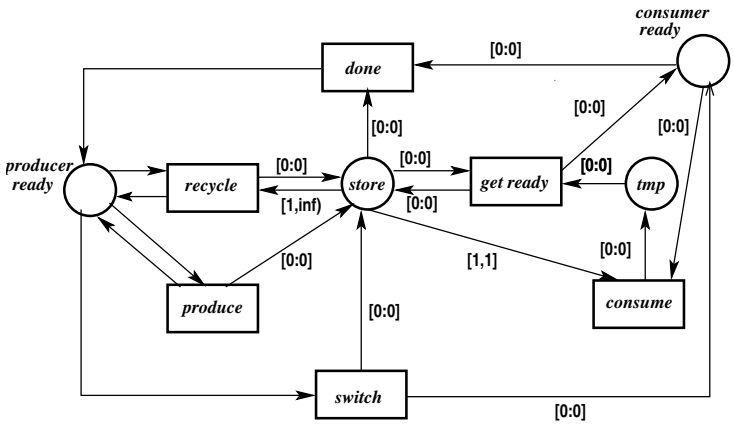

Figure 3.10: TPN model for Producer/Consumer System

Chapter 4

Coverability

In this chapter, we introduce a central problem in verification, called the *coverability* problem for (timed) Petri nets and then recall different existing methods for solving it.

4.1 Coverability Problem

The *coverability problem* for Petri nets (as well as for timed Petri nets) is defined as: to check whether an upward closed set of *final markings* is reachable from a set of initial markings.

COVERABILITY

Instance: A set of initial markings M_{init} and a finite set M_{fin} of final markings.

Question: $Reach(\mathsf{M}_{init}) \cap (\mathsf{M}_{fin} \uparrow) = \emptyset$?

Notice that the set $\mathsf{M}_{fin} \uparrow$ is upward closed with respect to the ordering $\leq^m$ (see Chapter 2). We use the set $\mathsf{M}_{fin} \uparrow$ to represent a set of "bad markings" which we do not want to occur during the execution of the system. Safety is then equivalent to non-reachability of $\mathsf{M}_{fin} \uparrow$.

Here, we recall that one can use the standard reduction of safety to reachability as in [132, 75]. Given a transition system $(S, \longrightarrow)$ and the specification of the system given as a regular set L of traces (represented by a finite state machine), the following method is used to decide whether system satisfies the specification (i.e., system is safe). First we compute the complement $\overline{L}$ ($\overline{L}$ represents the unwanted behaviours of the system). Then we compose the transition system $(S, \longrightarrow)$ with the finite-state machine describing $\overline{L}$. This composition is computable in case of Petri nets. Finally we check whether $S \times F$ is reachable from the initial state of the composed transition system, where F is the set of final states in the automaton with language $\overline{L}$. Notice that the set $S \times F$ is trivially upward-closed.

To solve coverability problem for Petri nets, one may either compute the set of *forward reachable markings*, i.e., all the markings reachable from the initial markings as shown in [91]; or compute *backward reachable markings*, i.e., all the markings from which a final marking is reachable, following the general algorithm in [5].

4.2 Forward Analysis of Petri Nets

The Karp-Miller algorithm [91] is the classical method used for checking coverability in untimed Petri nets. The algorithm in [91] constructs the *coverability graph* starting from an initial marking. At each node of the coverability graph, one produces successor nodes (*post-images*) which are reachable from the current node by performing a single transition. A path in the coverability graph is pruned in two cases: (a) if the current node already appears in the existing part of the coverability graph (in this case, current node is removed from the graph and an edge is put from the parent of the current node to the already existing copy of the current node), (b) or if there is no successor of the current node. The algorithm also detects paths in the coverability graph which lead from a marking M_1 to a marking M_2 such that $M_1 <^m M_2$ (since, the ordering on markings of a Petri net is simply the ordering $\leq^m$ on multisets over natural numbers). In such a case, it makes an over-approximation of the set of reachable markings by putting ω (interpreted as "unboundedly many tokens") in each place p with $M_1(p) < M_2(p)$. Notice that the resulting markings are from $\mathbb{N}_\omega^k$ where k is the number of places and we call such markings as ω-markings. This over-approximation preserves safety properties. The termination of this algorithm is guaranteed, since the ordering $\leq^m$ is a well-quasi-ordering (Chapter 2).

4.3 Backward Analysis

To check reachability of an upward closed set, we may also perform a reachability analysis backwards using the symbolic method according to [120, 45, 127, 5]. This method is a standard fix-point iteration method to check whether a set of states F is reachable. A high-level description of the method is that to check whether a state in F is reachable, it computes the set of all states from which a state in F is reachable. In this method, at each iteration $j \geq 1$, one computes the set of states from which a state in F is reachable by j or less steps. More precisely, j-th approximation is obtained from the $(j-1)$-th approximation by adding the pre-image of the $(j-1)$-th approximation to it. If this procedure converges, then one checks for the intersection of the result with the set of initial states. The algorithm is applicable for reachability analysis of a system, if we can solve the following three problems for the system.

- Finding a suitable representation of infinite sets of states.
- Finding a method for computing pre-images.
- Proving that the iteration always converges.

To represent sets of states, we use *constraints* . Each constraint c characterizes a potentially infinite upward-closed set $[\![\mathsf{c}]\!]^\uparrow$ of states. Suppose the ordering on the states is given by $\preceq$. Then, if a state $s \in [\![\mathsf{c}]\!]^\uparrow$ and $s \preceq s'$, then $s' \in [\![\mathsf{c}]\!]^\uparrow$. A set $\mathbb{C}$ of constraints characterizes the union $\bigcup_{\mathsf{c}\in\mathbb{C}} [\![\mathsf{c}]\!]^\uparrow$, denoted by $[\![\mathbb{C}]\!]^\uparrow$. In

general, an infinite number of constraints can appear in the analysis of an infinite-state system. To handle this, we introduce an *entailment* relation $\sqsubseteq$ on the set of constraints. We define $\sqsubseteq$ on $\mathbb{C}$ such that $\mathsf{c_1} \sqsubseteq \mathsf{c_2}$ iff $[\![\mathsf{c_2}]\!]^{\uparrow} \subseteq [\![\mathsf{c_1}]\!]^{\uparrow}$. We extend $\sqsubseteq$ on sets of constraints $\mathbb{C}$ such that with $\mathbb{C}_1, \mathbb{C}_2 \subseteq \mathbb{C}$, $\mathbb{C}_1 \sqsubseteq \mathbb{C}_2$ if for all $\mathsf{c_2} \in \mathbb{C}_2$, there is a $\mathsf{c_1} \in \mathbb{C}_1$ such that $\mathsf{c_1} \sqsubseteq \mathsf{c_2}$. The key step in the analysis is to decide $\sqsubseteq$ and prove that $\sqsubseteq$ is a well-quasi-ordering on $\mathbb{C}$. The termination of the algorithm is guaranteed if the entailment relation $\sqsubseteq$ on constraints is a well-quasi-ordering.

To apply the above algorithm to any system, we need the following.

- Define constraints representing sets of states of the system.
- Given a set of constraints $\mathbb{C}_1$, show that its pre-image $Pre(\mathbb{C}_1)$ characterizing the set $\{s \mid \exists s' \in [\![\mathbb{C}_1]\!]^{\uparrow}, s \longrightarrow s'\}$ is computable as a finite set of constraints.
- Define an ordering $\sqsubseteq$ on $\mathbb{C}$ such that $\sqsubseteq$ is computable and prove that $\sqsubseteq$ is a well-quasi-ordering.
- Given a marking M and a set $\mathbb{C}$ of constraints, whether $M \in [\![\mathbb{C}]\!]^{\uparrow}$.

4.3.1 Backward Analysis for Petri Nets

To perform a backward analysis for Petri nets, we do the following.

- We define a *constraint* of a Petri net, syntactically the same as a marking. However, a constraint c represents an upward closed set $\{\mathsf{c}\}\uparrow$ of markings where $\{\mathsf{c}\}\uparrow$ is upward-closed with respect to $\leq^m$, i.e., $[\![\mathsf{c}]\!]^{\uparrow} = \{\mathsf{c}\}\uparrow$.
- The pre-image $Pre(\mathbb{C})$ for a finite set $\mathbb{C}$ of constraints of Petri nets is effectively computable [68].
- The entailment $\sqsubseteq$ between two constraints $\mathsf{c_1}, \mathsf{c_2}$, denoted by $\mathsf{c_1} \sqsubseteq \mathsf{c_2}$ is computed as $\mathsf{c_1} \leq^{\mathsf{m}} \mathsf{c_2}$. From Lemma 2.1, we have that the ordering $\leq^m$ on the set $P^{\circledast}$ of multisets over a finite set P is a well-quasi-ordering. Notice that $P^{\circledast}$ denotes the set of markings of a Petri net with P as its set of places.

The general algorithm now allows us to start from a set of constraints $\mathbb{C}_0 = \mathsf{M}_{fin}$ (i.e., $[\![\mathbb{C}_0]\!]^{\uparrow} = \mathsf{M}_{fin}\uparrow$). Then we define the sequence $\mathbb{C}_0, \mathbb{C}_1, \mathbb{C}_2, \ldots$ of sets of constraints such that $\mathbb{C}_{i+1} = \mathbb{C}_i \cup Pre(\mathbb{C}_i)$. Intuitively $\mathbb{C}_i$ denotes the set of markings from which a state belonging to $\mathbb{C}_0$ is reachable in i or less steps. Abusing notations, we define $Pre^*(\mathbb{C}_0) = \bigcup_{i \geq 0} \mathbb{C}_i$. Then $\mathbb{C}_0$ is reachable from M_{init} if there is a marking in M_{init} which is also in $Pre^*(\mathbb{C}_0)$. Notice that for each i, $[\![\mathbb{C}_i]\!]^{\uparrow} \subseteq [\![\mathbb{C}_{i+1}]\!]^{\uparrow}$. Since $\leq^m$ is a well-quasi-ordering, there is a j such that $[\![\mathbb{C}_j]\!]^{\uparrow} = [\![\mathbb{C}_{j+1}]\!]^{\uparrow}$. Otherwise, there will be an infinite sequence of constraints, $\mathsf{c_0} \not\sqsubseteq \mathsf{c_1} \not\sqsubseteq \mathsf{c_2} \ldots$ such that $\mathsf{c_i} \in \mathbb{C_i}$ for all $i \geq 0$ and this violates the assumption that $\leq^m$ ($\sqsubseteq$) is a well-quasi-ordering. It is easy to see $\mathbb{C}_l = \mathbb{C}_j$ for all $l \geq j$, meaning that $Pre^*(\mathbb{C}_0) = \mathbb{C}_j$.

- Finally we check membership of each marking $M \in \mathsf{M}_{init}$ in the set of markings given by $[\![Pre^*(\mathbb{C}_0)]\!]^{\uparrow}$. This is done by checking whether there is a constraint $\mathsf{c} \in Pre^*(\mathbb{C}_0)$ such that $\mathsf{c} \leq^{\mathsf{m}} M$.

4.4 Coverability for TPNs

As we mentioned in the previous two sections, the coverability problem for Petri nets can be solved both using forward and backward analysis. However, for timed Petri nets these two analyses exhibit surprisingly different behaviours. For TPNs, the set of backward reachable states is computable. In [14], the backward reachability algorithm given above for Petri nets is adapted to the case for TPNs. However the set of forward reachable states is in general not computable for TPNs. Therefore any procedure for performing forward reachability analysis on TPNs is necessarily incomplete. In the following, we show that we can perform the required operations for the backward analysis of TPNs. In the next chapter, we give a semi-algorithm for the forward analysis of TPNs.

4.4.1 Backward Analysis for TPNs

In the previous section, we saw how to define constraints for Petri nets and that the ordering on those constraints is a well-quasi-ordering. For Petri nets, a constraint is syntactically the same as a marking of Petri net, which is in turn simply a multiset. Thus, the ordering on the constraints of Petri nets is a well-quasi-ordering.

For TPNs, we cannot consider a constraint as a marking. This is due to the fact that marking of a TPN is a mapping from $P \times \mathbb{R}^{\geq 0}$ to $\mathbb{N}$. Thus, the underlying alphabet, i.e, $P \times \mathbb{R}^{\geq 0}$ is no longer a finite alphabet and the equality ordering on $P \times \mathbb{R}^{\geq 0}$ is trivially not a well-quasi-ordering.

To perform backward analysis for TPNs, first we define *constraints* to represent a possibly infinite set of markings of TPNs. We use constraints for TPNs which generalizes the notion of *regions* used to verify properties of (non-parameterized) timed automata [18]. In the following, we define *regions* for TPNs formally.

Regions

Regions were first designed for timed automata (which operate on a finite number of clocks) [18], and are therefore not sufficiently powerful to capture the behaviour of TPNs. A *region* defines the integral parts of clock values up to max (the exact age of a token is irrelevant if it is greater than max), and also the ordering of the fractional parts among clock values. For TPNs, we need to use a variant which also defines the place in which each token (clock) resides. Intuitively, each region for TPNs is a sequence of multisets of tokens. The idea is that elements belonging to the same multiset correspond to clocks with equal fractional parts while the ordering among multisets in a word corresponds to increasing fractional parts of the clock values.

Formally, following Godskesen [77] we represent a region by a triple (b_0, w, b_{max}) where

- $b_0 \in (P \times \{0, \ldots, max\})^{\circledast}$. b_0 is a multiset of pairs. A pair of the form $p(n)$ represents a token with age exactly n in place p.
- $w \in \left((P \times \{0, \ldots, max-1\})^{\circledast}\right)^{*}$. w is a word over the set $(P \times \{0, \ldots, max-1\})^{\circledast}$, i.e., w is a word where each element in the word is a multiset over $P \times \{0, \ldots, max-1\}$. The pair $p(n)$ represents a token in place p with age x such that $x \in (n, n+1)$. Pairs in the same multiset represent tokens whose ages have equal fractional parts. The order of the multisets in w corresponds to the order of the fractional parts.
- $b_{max} \in P^{\circledast}$. b_{max} is a multiset over P representing tokens with ages strictly greater than max. Since the actual ages of these tokens are irrelevant, the information about their ages is omitted in the representation.

Each region characterizes an infinite set of markings as follows. Assume a marking $M = [p_1(x_1), \ldots, p_n(x_n)]$ and a region $\mathcal{R} = (b_0, b_1 \bullet b_2 \bullet \cdots \bullet b_m, b_{m+1})$. Let b_j be of the form $\left[q_{j1}(y_{j1}), \ldots, q_{j\ell_j}(y_{j\ell_j})\right]$ for $j : 0 \leq j \leq m$ and b_{m+1} be of the form $\left[q_{m+1\,1}, \ldots, q_{m+1\,l_{m+1}}\right]$. We say that M *satisfies* $\mathcal{R}$, written $M \models \mathcal{R}$, if there is a bijection h from the set $\{1, \ldots, n\}$ to the set of pairs $\{(j,k) \mid (0 \leq j \leq m+1) \wedge (1 \leq k \leq \ell_j)\}$ such that the following conditions are satisfied:

- $p_i = q_{h(i)}$. Each token should be in the same place as that required by the corresponding element in $\mathcal{R}$.
- $h(i) = (j,k)$ and $j = m+1$ if $x_i > max$. Tokens older than max should correspond to elements in multiset b_{m+1}. The actual ages of these tokens are not relevant.
- $h(i) = (j,k)$ and $j \leq m$ if $x_i \leq max$ and $\lfloor x_i \rfloor = y_{j\,k}$. The integral part of the age of tokens should agree with the natural number specified by the corresponding elements in w.
- $h(i) = (0,k)$ for some k if $fract(x_i) = 0$ and $x_i \leq max$. Tokens with zero fractional parts correspond to elements in multiset b_0.
- $h(i_1) = (j_1, k_1)$ and $h(i_2) = (j_2, k_2)$ for $j_1 \leq j_2 \leq m$ if $x_{i_1}, x_{i_2} < max$ and $fract(x_{i_1}) \leq fract(x_{i_2})$. Tokens with equal fractional parts correspond to elements in the same multiset (unless they belong to b_{m+1}). The ordering among multisets inside $\mathcal{R}$ reflects the ordering among fractional parts in clock values.

Given a marking M, it is straightforward to find the unique region $\mathcal{R}_M$ such that $M \models \mathcal{R}_M$.

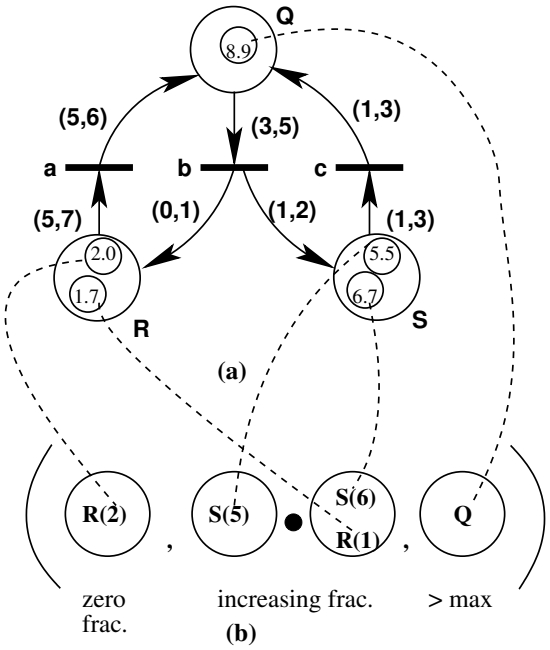

Figure 4.1: Marking M in (a) satisfies region $\mathcal{R}$ in (b).

We let $[\![\mathcal{R}]\!]^{=} = \{M \mid M \models \mathcal{R}\}$. For a set $\mathbf{R}$ of regions, $[\![\mathbf{R}]\!]^{=} = \bigcup_{\mathcal{R}\in\mathbf{R}} [\![\mathcal{R}]\!]^{=}$. The region construction defines an equivalence relation $\equiv$ on the set of markings such that $M_1 \equiv M_2$ if, for each region $\mathcal{R}$, it is the case that $M_1 \in [\![\mathcal{R}]\!]^{=}$ iff $M_2 \in [\![\mathcal{R}]\!]^{=}$. Following [18], it can be easily shown that $\equiv$ is a congruence on the set of markings, i.e, if $M_1 \longrightarrow M_2$ and $M_1 \equiv M_3$ then there is an M_4 such that $M_2 \equiv M_4$ and $M_3 \longrightarrow M_4$.

Example 4.1. Consider the TPN N in Figure 3.5 with $max = 7$. Figure 4.1(a) shows a marking $M = [R(2.0)\,, S(5.5),\ R(1.7),\ S(6.7),\ Q(8.9)]$. Figure 4.1(b) shows the unique region $\mathcal{R} = ([R(2)]\,,\ [S(5)]\ \bullet\ [R(1),\ S(6)]\,,\ [Q])$. such that $M \models \mathcal{R}$. In Figure 4.1(b), each circle corresponds to a multiset of tokens of N with same fractional parts. Dotted lines show how the tokens of M in TPN corresponds to elements in the region $\mathcal{R}$. □

Ordering

We define an ordering on markings which extends the equivalence relation $\equiv$ on markings induced by the definition of regions. We define an ordering $\preceq$ on the set of markings of TPNs such that $M_1 \preceq M_2$ if there is an M_2' with $M_1 \equiv M_2'$ and $M_2' \leq^m M_2$. In other words, $M_1 \preceq M_2$ if we can delete a number of tokens from M_2 and as a result obtain a new marking which is equivalent to M_1. We let $M_1 \prec M_2$ denote that $M_1 \preceq M_2$ and $M_1 \not\equiv M_2$.

Next, we consider the following lemma which states that $\longrightarrow$ is *monotonic* with respect to the ordering $\preceq$.

Lemma 4.2. If $M_1 \longrightarrow M_2$ and $M_1 \preceq M_3$ then there is an M_4 such that $M_2 \preceq M_4$ and $M_3 \longrightarrow M_4$.

Proof. Suppose that $M_1 \preceq M_3$. By definition of $\preceq$ there is an M_3' with $M_1 \equiv M_3'$ and $M_3' \leq^m M_3$. From the definition of $\leq^m$ we know that there is an M_3'' such that $M_3 = M_3' + M_3''$. Since $\equiv$ is a congruence, there is M_4' such that $M_2 \equiv M_4'$ and $M_3' \longrightarrow M_4'$. We define $M_4 = M_4' + M_4''$ where $M_4'' = M_3''$ if $M_3' \longrightarrow_t M_4'$ for some discrete transition t, and $M_4'' = (M_3'')^{+\delta}$ if $M_3' \longrightarrow_{T=\delta} M_4'$ for some $\delta \in \mathbb{R}^{\geq 0}$. □

Example 4.3. Consider the TPN N in Figure 3.5 where $max = 7$. Recall the marking M and the region $\mathcal{R}$ in Figure 4.1. Also, consider marking $M' = [R(2.0)\,, S(5.7),\ R(1.8),\ S(6.8),\ Q(9.9)]$. Notice that M' satisfies the region $\mathcal{R}$ and $M \equiv M'$. Let $M'' = M' + [R(1.2),\ Q(9.2)]$. Since $M' \leq^m M''$ and $M \equiv M'$, we have $M \preceq M''$ (see Figure 4.2). □

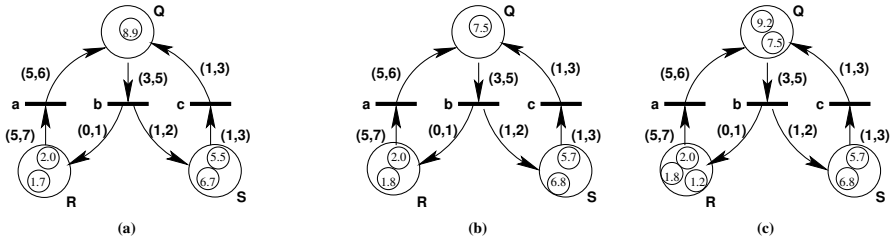

Figure 4.2: (a) M. (b) M'. (c) M''.

Finally, we define an ordering $\sqsubseteq$ on regions such that if $\mathcal{R}_1 = \left(b_0^1, w^1, b_{max}^1\right)$ and $\mathcal{R}_2 = \left(b_0^2, w^2, b_{max}^2\right)$ then $\mathcal{R}_1 \sqsubseteq \mathcal{R}_2$ iff $b_0^1 \leq^m b_0^2$, $w^1 \leq^w w^2$, and $b_{max}^1 \leq^m b_{max}^2$. This means that the entailment on the regions of TPNs is decidable.

From Lemma 2.1, Lemma 2.2 and the fact that $P \times \{0, \ldots, max\}$ is a finite alphabet and that the well-quasi-ordering is preserved when vectors are built over well-quasi-ordered sets [84], it follows that

Lemma 4.4. The ordering $\sqsubseteq$ on the set of regions is a well-quasi-ordering.

Backward Analysis

For backward analysis of TPNs, we use regions as our constraints. Given a constraint (region) $\mathcal{R}$ of TPN, notice that $[\![\mathcal{R}]\!]^{\uparrow} = [\![\mathcal{R}\uparrow]\!]^{=}$. This means that a marking $M \in [\![\mathcal{R}]\!]^{\uparrow}$ iff there is a region $\mathcal{R}'$ such that $\mathcal{R} \sqsubseteq \mathcal{R}'$ and $M \models \mathcal{R}'$. Notice that we already showed how to compute $[\![\mathbf{R}]\!]^{\uparrow}$ for a set $\mathbf{R}$ of constraints in the above.

From [9], we have that $Pre(\mathbf{R})$ for a set $\mathbf{R}$ of constraints (regions) is effectively constructible.

We already showed that the entailment on the constraints (regions) of TPNs is decidable. Notice that $\mathcal{R}_1 \sqsubseteq \mathcal{R}_2$ means that $[\![\mathcal{R}_2]\!]^{\uparrow} \subseteq [\![\mathcal{R}_1]\!]^{\uparrow}$. From Lemma 4.4, we have that $\sqsubseteq$ is a well-quasi-ordering.

Finally we check membership of each marking[1] $M \in \mathsf{M}_{init}$ in the set of markings given by $[\![Pre^*(\mathbf{R}_0)]\!]^{\uparrow}$. This is done by checking whether there is a region $\mathcal{R} \in Pre^*(\mathbf{R}_0)$ and a marking $M' \preceq M$ such that $M' \models \mathcal{R}$. Thus, all the problems to perform the backward analysis for TPNs are solved and we get the following.

Lemma 4.5. Given a set $\mathbf{R}$ of regions, the set $Pre^*(\mathbf{R})$ is effectively constructible as a finite set of regions [14].

Notice that the infiniteness in TPNs due to real-valued ages of tokens are handled by regions. However, the infiniteness due to unbounded number of tokens in a place is handled by the notion of constraints characterizing upward closed sets of regions.

[1] We assume that such a marking contains tokens with only integer ages.

Chapter 5

Forward Analysis of TPNs

In this chapter, we give a semi-algorithm for performing forward reachability analysis for TPNs. We provide an abstraction of the set of reachable markings by taking its *downward closure*. The abstraction is exact with respect to coverability (consequently with respect to safety properties), i.e, a given TPN satisfies any safety property exactly when the downward closure of the set of reachable states satisfies the same property. Moreover, the downward closure has usually a simpler structure than the exact set of reachable states.

The set of reachable markings (and its downward closure) is in general infinite. So, we introduce a symbolic representation for downward closed sets, which we call *region generators*. Each region generator denotes the union of an infinite number of *regions* (Chapter 4). Thus region generators handle the infiniteness due to unbounded number of tokens. *Region generators* are similar to *regular expressions* in the context of TPNs.

First we give an overview of the forward analysis and then show that region generators allow the basic operations in forward analysis, i.e, checking membership, entailment, and computing the post-images with respect to a single transition. Since forward analysis is incomplete, we also give an acceleration scheme to make the analysis terminate more often. The scheme computes, in one step, the effect of an arbitrary number of firings of a single discrete transition interleaved with timed transitions.

We have implemented the forward reachability procedure and used the tool to compute the reachability set for a parameterized version of Fischer's protocol, Lynch and Shavit's mutual exclusion protocol and a simple producer/consumer protocol. Also, we used the tool for generating finite state abstractions of these protocols.

Remark: To do forward analysis for TPNs, one might first think of extending the Karp-Miller algorithm[91]. However it turns out that it is not obvious how to extend this algorithm for TPNs. In the case of TPNs, if $M_1 \prec M_2$ (in fact even if $M_1 <^m M_2$), one cannot over-approximate M_2 by introducing ω. For instance, consider the Petri net in Figure 5.1. If we start from marking $M_0 = [p(0)]$, we can let time pass by one unit, reach $M_1 = [p(1)]$, then fire transition t and reach a marking $M_2 = [p(0), q(0)]$. However, it is not the case that unboundedly many tokens with age $q(0)$ are generated, even though $M_0 <^m M_2$. Termination is therefore not guaranteed.

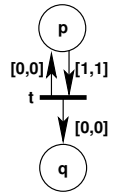

Figure 5.1: A timed Petri net for which Karp-Miller algorithm cannot be applied.

5.1 Coverability

In the last chapter, we showed that TPNs induce a monotonic transition system (Lemma 4.2). It follows that analyzing coverability will not be affected by taking the downward closure (with respect to the ordering $\preceq$ on the markings) of the set of reachable markings.

Lemma 5.1. For a set of markings M_{init} and an upward closed set M of markings, we have $\mathit{Reach}(\mathsf{M}_{init}) \cap \mathsf{M} = \emptyset$ iff $(\mathit{Reach}(\mathsf{M}_{init}))\downarrow \cap \mathsf{M} = \emptyset$.

Proof. It is obvious that $\mathit{Reach}(\mathsf{M}_{init}) \cap \mathsf{M} = \emptyset$ if $(\mathit{Reach}(\mathsf{M}_{init}))\downarrow \cap \mathsf{M} = \emptyset$. We show the other direction. Suppose that there is a marking $M \in (\mathit{Reach}(\mathsf{M}_{init}))\downarrow \cap \mathsf{M}$. This means that there is a marking $M' \in (\mathit{Reach}(\mathsf{M}_{init}))$ such that $M \preceq M'$ and $M \in \mathsf{M}$, i.e., $M' \in \mathsf{M}$, since M is upward closed. This implies that $(\mathit{Reach}(\mathsf{M}_{init})) \cap \mathsf{M} \neq \emptyset$. Contradiction. □

Since $\mathsf{M}_{fin}\uparrow$ (in the definition of the coverability problem) is upward closed by definition, it follows from Lemma 5.1 that taking downward closure of $\mathit{Reach}(M_{init})$ gives an exact abstraction with respect to coverability.

5.2 Region Generators

In this section we introduce *region generators* which we define in a hierarchical manner. First, we introduce *multiset* and *word language generators* and then describe how a region generator characterizes a potentially infinite set (language) of regions.

5.2.1 Multiset Language Generators (Mlgs)

We define *multiset language generators (mlgs)*, each of which characterizes a language which consists of multisets over a finite alphabet.

Let A be a finite alphabet. A *multiset language* (over A) is a subset of $A^{\circledast}$. We will consider multiset languages which are downward closed with respect to the relation $\leq^m$ on multisets (Chapter 2). If L is downward closed then $b_1 \in L$ and $b_2 \leq^m b_1$ implies $b_2 \in L$.

We define *(downward closed) multiset language generators* (or *mlgs* for short) over the finite alphabet A. Each mlg ϕ over A defines a multiset language over A, denoted $L(\phi)$, which is downward closed. The set of mlgs over A and their languages are defined as follows :

- An *expression* over A is of one of the following two forms:
  - an *atomic expression* a where $a \in A$. $L(a) = \{[a]\ ,\ \epsilon\}$.
  - a *star expression* of the form $S^{\circledast}$ where $S \subseteq A$. $L(S^{\circledast}) = \{[a_1\ ,\ \ldots\ ,\ a_m] \mid m \geq 0\ \wedge\ a_1, \ldots, a_m \in S\}$.
- An *mlg* ϕ is a (possibly empty) sequence $e_1 + \cdots + e_\ell$ of expressions. $L(\phi) = \{b_1 + \cdots + b_\ell \mid b_1 \in L(e_1), \cdots, b_\ell \in L(e_\ell)\}$. We denote an empty mlg by ϵ and assume that $L(\epsilon) = \{\epsilon\}$.

We also consider sets of mlgs which we interpret as unions. If $\Phi = \{\phi_1, \cdots, \phi_m\}$ is a set of mlgs, then $L(\Phi) = L(\phi_1) \cup \cdots \cup L(\phi_m)$. We assume $L(\emptyset) = \emptyset$.

Theorem 5.2. For each downward closed multiset language L over an alphabet A there is a set Φ of mlgs over A such that $L = L(\Phi)$.

Proof. See Appendix. □

Sometimes we identify mlgs with the languages they represent, so given two mlgs ϕ_1, ϕ_2, we write $\phi_1 \subseteq \phi_2$ (rather than $L(\phi_1) \subseteq L(\phi_2)$), and given a multiset b and a mlg ϕ, we write $b \in \phi$ (rather than $b \in L(\phi)$), etc.

Normal Form An mlg ϕ is said to be in *normal form* if it is of the form $e + e_1 + \cdots + e_k$ where e is a star expression and $e_1, \ldots, e_k$ are atomic expressions and for each $i : 1 \leq i \leq k$, $e_i \not\sqsubseteq e$. A set of mlgs $\{\phi_1, \cdots, \phi_m\}$ is said to be *normal* if ϕ_i is in *normal form* for each $i : 1 \leq i \leq m$, and $\phi_i \not\sqsubseteq \phi_j$ for each $i, j : 1 \leq i \neq j \leq m$.

For each mlg ϕ, there is a unique (up to commutativity of the operators) normal mlg ϕ' such that $L(\phi') = L(\phi)$. We can derive ϕ' from ϕ by performing the following operations.

- Delete each atomic expression a from ϕ in case there is a star expression of the form $S^{\circledast}$ in ϕ such that $a \in S$. The language of the mlg is preserved since $L(a + S^{\circledast}) = L(S^{\circledast})$.
- Merge all star expressions using the property that $L({S_1}^{\circledast} + {S_2}^{\circledast}) = L((S_1 \cup S_2)^{\circledast})$.

A set of mlgs $\Phi = \{\phi_1, \cdots, \phi_m\}$ is said to be *normal* if each mlg ϕ_i is normal and $\phi_i \not\sqsubseteq \phi_j$ for $1 \leq i \neq j \leq m$. We can transform each set of mlgs into normal form by transforming each member of Φ into normal from as described above, and by eliminating redundant members of Φ using the entailment algorithm described below.

From now on, (sets of) mlgs will always be assumed to be in a normal form.

Entailment In the following, we give an algorithm for computing entailment $\subseteq$ for (sets of) mlgs.

The relation $\subseteq$ is the least partial order on expressions satisfying

$$\begin{array}{ll} a \subseteq S^{\circledast} & \text{if } a \in S \\ {S_1}^{\circledast} \subseteq {S_2}^{\circledast} & \text{if } S_1 \subseteq S_2 \end{array}$$

Given the algorithm for entailment of expressions, we can compute the entailment of mlgs as follows:

Consider the base cases. $\epsilon \subseteq \phi_2$ and $\phi_1 \not\subseteq \epsilon$ if $\phi_1 \neq \epsilon$. Given two non-empty mlgs $\phi_1 = e_1 + \phi_1'$ and $\phi_2 = e_2 + \phi_2'$, we have $\phi_1 \subseteq \phi_2$ iff one of the following holds.

1. $e_1 = a$, $e_1 \not\subseteq e_2$ and $\phi_1 \subseteq \phi_2'$.
2. $e_1 = e_2 = a$ and $\phi_1' \subseteq \phi_2'$.
3. $e_2 = S^{\circledast}$, $e_1 \subseteq e_2$ and $\phi_1' \subseteq \phi_2$.

In a normal mlg, we assume that the atomic expressions in an mlg are sorted. This means that the entailment algorithm will have linear complexity.

Next, we consider entailment for sets of mlgs. We use the following lemma.

Lemma 5.3. For mlgs $\phi, \phi_1, \ldots, \phi_m$, if $\phi \subseteq \{\phi_1, \cdots, \phi_m\}$, then $\phi \subseteq \phi_i$ for some $i \in \{1, \ldots, m\}$.

Proof. See Appendix. □

From Lemma 5.3 and the algorithm for entailment of mlgs, it follows that

Theorem 5.4. Entailment among mlgs can be checked in linear time and entailment of sets of mlgs can be checked in quadratic time.

Example 5.5. Consider a finite alphabet $A = \{a, b, c\}$ and the set of multisets $A^{\circledast}$ over A. Given mlg $\phi_1 = \{a, b\}^{\circledast} + c$ (i.e., the multiset language over A containing at most one c), examples of multisets in $L(\phi_1)$ are $\left[a^2, b\right], [b, c], \left[a^3, b^2, c\right]$. Consider $\phi_2 = b + c$, i.e., the multiset language containing at most one b and one c. $L(\phi_2) = \{\epsilon, [c], [b], [b, c]\}$. Notice that ϕ_1, ϕ_2 are in normal form and $\phi_2 \subseteq \phi_1$. Furthermore $L(\phi_1)$ and $L(\phi_2)$ are both downward closed. Figure 5.2 graphically describes ϕ_1 and ϕ_2. Sets are drawn as ellipses and mlgs are shown as circles. □

5.2.2 Word Language Generators (Wlgs)

In this section, we consider languages where each word is a sequence of multisets over a finite alphabet A, i.e., each word is a member of $(A^{\circledast})^*$ (recall that for a set A, we use A^* to denote the set of finite words over A (Chapter 2). The language is then a subset of $(A^{\circledast})^*$. Notice that the underlying alphabet, namely $A^{\circledast}$ is infinite.

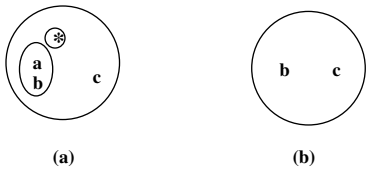

Figure 5.2: Mlgs (a) ϕ_1. (b) ϕ_2.

For a word $w \in L$, observe that $w(i)$ is a multiset over A.

Recall that $\leq^m$ and $\leq^w$ are the orderings on the set of multisets and the set of words respectively (Chapter 2). This means that $w_1 \leq^w w_2$ if there is a strictly monotonic injection $h : \{1, \ldots, |w_1|\} \rightarrow \{1, \ldots, |w_2|\}$ where $w_1(i) \leq^m w_2(h(i))$ for $i : 1 \leq i \leq |w_1|$.

We shall consider languages which are downward closed with respect to $\leq^w$. In a similar manner to mlgs, we define downward closed *word language generators (wlgs)* and word languages as follows.

- A *word expression* over A is of one of the following two forms:
  - a *word atomic expression* is an mlg ϕ over A. $L(\phi) = \{b \mid b \in \phi\} \cup \{\epsilon\}$.
  - a *word star expression* of the form $\{\phi_1, \cdots, \phi_k\}^*$, where $\phi_1, \ldots, \phi_k$ are mlgs over A.
    $L(\{\phi_1, \cdots, \phi_k\}^*) =$
    $\{b_1 \bullet \cdots \bullet b_m \mid (m \geq 0)$ and $b_1, \ldots, b_m \in L(\phi_1) \cup \cdots L(\phi_k)\}$.
- A *word language generator (wlg)* ψ over A is a (possibly empty) concatenation $e_1 \bullet \cdots \bullet e_\ell$ of word expressions $e_1, \ldots, e_\ell$. $L(\psi) = \{w_1 \bullet \cdots \bullet w_\ell \mid w_1 \in L(e_1) \wedge \cdots \wedge w_\ell \in L(e_\ell)\}$.

Notice that the concatenation operator is associative, but not commutative (as is the operator + for multisets). Again, we denote the empty wlg by ϵ. For a set $\Psi = \{\psi_1, \cdots, \psi_m\}$ of wlgs, we define $L(\Psi) = L(\psi_1) \cup \cdots \cup L(\psi_m)$. We assume $L(\epsilon) = \{\epsilon\}$ and $L(\emptyset) = \emptyset$. We also identify wlgs with word languages as we did with mlgs and multiset languages.

Theorem 5.6. For each downward closed word language L, there is a set Ψ of wlgs such that $L = L(\Psi)$.

Proof. See Appendix. □

Normal Form A word atomic expression e of the form ϕ is said to be in normal form if ϕ is a normal mlg. A word star expression $\{\phi_1, \ldots, \phi_k\}^*$ is said be in normal form if the set of mlgs $\{\phi_1, \ldots, \phi_k\}$ is in normal form.

A wlg $\psi = e_1 \bullet \cdots \bullet e_\ell$ is said to be *normal* if

- $e_1, \ldots, e_\ell$ are normal,
- $e_i \bullet e_{i+1} \not\subseteq e_i$ for each $i : 1 \leq i < \ell$, and
- $e_i \bullet e_{i+1} \not\subseteq e_{i+1}$, for each $i : 1 \leq i < \ell$.

A set of wlgs $\{\psi_1, \cdots, \psi_m\}$ is said to be *normal* if $\psi_1, \ldots, \psi_m$ are normal and $\psi_i \not\subseteq \psi_j$ for each $i, j : 1 \leq i \neq j \leq m$.

For each wlg ψ, there is a unique *normal* wlg ψ' such that $L(\psi) = L(\psi')$. We can derive ψ' from ψ using normalisation, checking entailment for mlgs and the entailment algorithm for wlgs described below. Normal form for sets of wlgs can be defined in a similar manner to mlgs. We can transform a set of wlgs $\Psi = \{\psi_1, \cdots, \psi_m\}$ into normal form using the normalization procedure above, and by eliminating redundant wlgs using the entailment algorithm below.

From now on, (sets of) wlgs will always be reduced to a normal form.

Entailment Now, we extend the algorithm for checking entailment of mlgs to check entailment of wlgs of the form $\psi \subseteq \psi'$.

First, we extend $\subseteq$ such that

- $\phi \subseteq \{\phi_1, \ldots, \phi_k\}^*$ if $\phi \subseteq \{\phi_1, \cdots, \phi_k\}$.
- $\{\phi_1, \cdots, \phi_k\}^* \subseteq \{\phi'_1, \cdots, \phi'_{k'}\}^*$ if $\{\phi_1, \cdots, \phi_k\} \subseteq \{\phi'_1, \cdots, \phi'_{k'}\}$.

The above entailment of word expressions can be computed using the entailment algorithm for multisets.

Entailment of wlgs is very similar to the entailment of mlgs. But, concatenation is not commutative. Therefore, given two non-empty wlgs $\psi_1 = e_1 \bullet \psi'_1$ and $\psi_2 = e_2 \bullet \psi'_2$, we have $\psi_1 \subseteq \psi_2$ iff one of the following holds.

- $e_1 \not\subseteq e_2$ and $\psi_1 \subseteq \psi'_2$.
- $e_1 \subseteq e_2$, e_2 is an atomic expression and $\psi'_1 \subseteq \psi'_2$
- $e_1 \subseteq e_2$, e_2 is a star expression, $\psi'_1 \subseteq \psi_2$.

For sets of wlgs, we use a lemma similar to Lemma 5.3.

Lemma 5.7. For wlgs $\psi, \psi_1, \ldots, \psi_m$, if $\psi \subseteq \{\psi_1, \cdots, \psi_m\}$, then $\psi \subseteq \psi_i$ for some $i \in \{1, \ldots, m\}$.

Proof. See Appendix. □

From Theorem 5.4, Lemma 5.7 and the above algorithm for computing entailment of wlgs, we conclude that

Theorem 5.8. Entailment of wlgs can be computed in quadratic time and entailment of a set of wlgs can be computed in cubic time.

Example 5.9. Consider the same alphabet A and mlgs $\phi_1 = \{a, b\}^{\circledast} + c$ and $\phi_2 = b + c$. Consider a wlg $\psi_1 = \{\phi_2\}^* \bullet \phi_1$. Example of a word in $L(\psi_1)$ is $[b, c] \bullet [b] \bullet \left[a^3\right]$. Consider a wlg $\psi_2 = \{\phi_1\}^* \bullet \phi_2$. Example of a word in $L(\psi_2)$ is $\left[a^2, b^3, c\right] \bullet \left[a^3, c\right] \bullet [b, c]$. Notice that $\psi_1 \subseteq \psi_2$, but and $\psi_2 \not\subseteq \psi_1$. Figure 5.3 graphically describes ψ_1 and ψ_2. □

Figure 5.3: Wlgs (a) ψ_1. (b) ψ_2.

5.2.3 Region Generators

A *region generator* θ is a triple $(\phi_0, \psi, \phi_{max})$ where ϕ_0 is an mlg over $P \times \{0, \ldots, max\}$, ψ is a wlg over $P \times \{0, \ldots, max - 1\}$, and ϕ_{max} is an mlg over P. The language $L(\theta)$ contains exactly each region of the form (b_0, w, b_{max}) where $b_0 \in \phi_0$, $w \in \psi$, and $b_{max} \in \phi_{max}$. We observe that if $\theta_1 = \left(\phi_0^1, \psi^1, \phi_{max}^1\right)$ and $\theta_2 = \left(\phi_0^2, \psi^2, \phi_{max}^2\right)$ then $\theta_1 \subseteq \theta_2$ iff $\phi_0^1 \subseteq \phi_0^2$, $\psi^1 \subseteq \psi^2$, and $\phi_{max}^1 \subseteq \phi_{max}^2$. In other words entailment between region generators can be computed by checking entailment between the individual elements.

For a region generator θ, we define $[\![\theta]\!]^{\downarrow}$ to be $\cup_{\mathcal{R} \in L(\theta)} [\![\mathcal{R}]\!]^{=}$. In other words, a region generator θ:

- defines a language $L(\theta)$ of regions; and
- denotes a set of markings, namely all markings which belong to the denotation $[\![\mathcal{R}]\!]^{=}$ for some region $\mathcal{R} \in L(\theta)$.

A finite set $\Theta = \{\theta_1, \ldots, \theta_m\}$ of region generators denotes the union of its elements, i.e, $[\![\Theta]\!]^{\downarrow} = \bigcup_{1 \leq i \leq m} [\![\theta_i]\!]^{\downarrow}$.

Given a marking M and a region generator θ, it is straightforward to check whether $M \in [\![\theta]\!]^{\downarrow}$ from the definition of $[\![\mathcal{R}]\!]^{=}$ and $[\![\theta]\!]^{\downarrow}$.

Here, we recall that in Section 5.1, we showed how to decide the entailment $\sqsubseteq$ on regions. However for forward analysis, $\mathcal{R}_1 \sqsubseteq \mathcal{R}_2$ iff for each $M_1 \in [\![\mathcal{R}_1]\!]^{\downarrow}$ and $M_2 \in [\![\mathcal{R}_2]\!]^{\downarrow}$, $M_1 \preceq M_2$. By Theorem 5.2 and Theorem 5.6 it follows that for each set $\mathbf{R}$ of regions which is downward closed with respect to $\sqsubseteq$, there is a finite set of region generators Θ such that $L(\Theta) = \mathbf{R}$.

From this we get the following.

Theorem 5.10. For each set M of markings which is downward closed with respect to $\preceq$ there is a finite set of region generators Θ such that $\mathsf{M} = [\![\Theta]\!]^{\downarrow}$.

From this, it follows that $\theta_1 \subseteq \theta_2$ iff for each $M_1 \in [\![\theta_1]\!]^{\downarrow}$ and $M_2 \in [\![\theta_2]\!]^{\downarrow}$, $M_1 \preceq M_2$.

Example 5.11. Consider again the TPN in Figure 3.5 with $max = 7$. Examples of mlgs over $\{Q, R, S\} \times \{0, \ldots, 7\}$ are $R(2)$, $S(2)$, $\{S(6), R(1)\}^{\circledast} + S(5)$, etc. $S(2) \bullet \left\{\{S(6), R(1)\}^{\circledast} + S(5)\right\}^{*}$ is an example of a wlg over $\{Q, R, S\} \times \{0, \ldots, 6\}$ and $\{Q\}^{\circledast}$ is an mlg over $\{Q, R, S\}$. Finally, an example of region generator is given by $\theta = \left(R(2),\ S(2) \bullet \left\{\{S(6), R(1)\}^{\circledast} + S(5)\right\}^{*},\ \{Q\}^{\circledast}\right)$. Figure 5.4(a) shows the region generator graphically and Figure 5.4(b) shows an example of a region in the language of the region generator in Figure 5.4(a). □

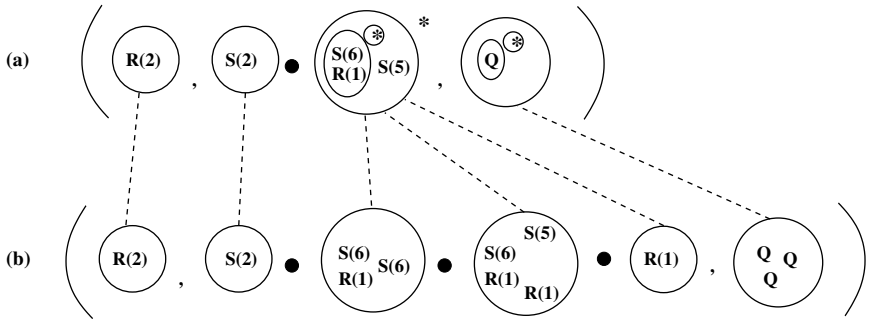

Figure 5.4: (a) Region Generator θ. (b) A region $\mathcal{R} \in L(\theta)$.

5.3 Forward Analysis

We present a version of the standard symbolic forward reachability algorithm which uses region generators as a symbolic representation. The algorithm inputs a set of region generators Θ_{init} characterizing the set M_{init} of initial markings, and a set M_{fin} of final markings and tries to answer whether $[\![\Theta_{init}]\!]^{\downarrow} \cap \mathsf{M}_{fin} \uparrow = \emptyset$. The algorithm computes the sequence $\Theta_0, \Theta_1, \ldots$ of sets of region generators such that $\Theta_{i+1} = \Theta_i \cup succ(\Theta_i)$ with $\Theta_0 = \Theta_{init}$. If $[\![\Theta_i]\!]^{\downarrow} \cap \mathsf{M}_{fin} \uparrow \neq \emptyset$ (amounts to checking membership of elements of M_{fin} in $[\![\Theta]\!]^{\downarrow}$), or if $\Theta_{i+1} = \Theta_i$, then the procedure is terminated. We define $succ(\Theta)$ to be $Post_{Time}(\Theta) \cup \bigcup_{t \in T}(Post_t(\Theta) \cup Step_t(\Theta))$. $Post_{Time}$ and $Post_t$, defined in Section 5.4, compute the effect of timed and discrete transitions respectively. $Step_t$, defined in Section 5.5, implements acceleration. Also, whenever there are two region generators θ_1, θ_2 in a set of region generators such that $\theta_1 \subseteq \theta_2$, we remove θ_1 from the set.

Even if we know by Theorem 5.10 that there is finite set Θ of region generators such that $Reach([\![\Theta_{init}]\!]^{\downarrow}) = [\![\Theta]\!]^{\downarrow}$, the following holds due to undecidability of structural termination for TPNs.

Theorem 5.12. Given a region generator θ_{init} we cannot in general compute a set Θ of region generators such that $Reach([\![\theta_{init}]\!]^{\downarrow}) = [\![\Theta]\!]^{\downarrow}$.

The aim of acceleration is to make the forward analysis procedure terminate more often.

5.4 Post-Image of a Region Generator

In this section, we consider the post-image of a region generator θ with respect to timed and discrete transitions respectively.

5.4.1 Timed Post-image

To give the intuition of computing the post-image of a region generator with respect to timed transitions, first we show how to perform this operation on regions.

$\mathbf{Post}_{Time}$ for regions

We define $Post_{Time}$ such that it corresponds to letting time pass.

We compute the post-image of a region $\mathcal{R}$ with respect to time as a finite set of regions such that $[\![Post_{Time}(\mathcal{R})]\!]^{=} = \{M' \mid \exists M \in [\![\mathcal{R}]\!]^{=}.\ M \longrightarrow_{Time} M'\}$. For a set $\mathbf{R}$ of regions $Post_{Time}(\mathbf{R}) = \bigcup_{\mathcal{R}\in\mathbf{R}} Post_{Time}(\mathcal{R})$.

First, we define a function *Rotate* such that given an input region $\mathcal{R}$, $Rotate(\mathcal{R})$ returns a region as described in the following. Later we use *Rotate* to define $Post_{Time}$.

Consider a marking M and a region $\mathcal{R} = (b_0, w, b_{max})$ such that $M \models \mathcal{R}$. Three cases are possible:

1. If $b_0 = \epsilon$, i.e., there are no tokens in M with ages whose fractional parts are equal to zero. Let w be of the form $w_1 \bullet b_1$. The behaviour of the TPN from M due to passage of time is decided by a certain subinterval of $\mathbb{R}^{\geq 0}$ which we denote by $stable(M)$. This interval is defined by $[0 : 1-x)$ where x is the highest fractional part among the tokens whose ages are less than max. Those tokens correspond to b_1 in the definition of $\mathcal{R}$. We call $stable(M)$ the *stable period* of M.

   Suppose that time passes by an amount $\delta \in stable(M)$. If $M \longrightarrow_{T=\delta} M_1$ then $M_1 \models \mathcal{R}$, i.e., $M_1 \equiv M$. In other words, if the elapsed time is in the stable period of M then all markings reached through performing timed transitions are equivalent to M. The reason is that, although the fractional parts have increased (by the same amount), the relative ordering of the fractional parts, and the integral parts of the ages are not affected. This case does not yield a new marking by letting time pass.

   Next consider $\delta = 1-x$. As soon as we leave the stable period, the tokens which originally had the highest fractional parts (those corresponding to

b_1) will now change: their integral parts will increase by one while fractional parts will become equal to zero. Therefore, we reach a new marking M_2, where $M_2 \models \mathit{Rotate}(\mathcal{R})$ and $\mathit{Rotate}(\mathcal{R})$ is of the form $\left(b_1^{+1}, w_1, b_{max}\right)$. Here, b_1^{+1} is the result of replacing each pair $p(n)$ in b_1 by $p(n+1)$.

2. If $b_0 \neq \epsilon$, i.e., there are some tokens whose ages do not exceed max and whose fractional parts are equal to zero. We divide the tokens in b_0 into two multisets: *young tokens* whose ages are strictly less than max, and *old tokens* whose ages are equal to max. The stable period $\mathit{stable}(M)$ here is the point interval $[0:0]$. Suppose that we let time pass by an amount $\delta : 0 < \delta < 1 - x$, where x is the highest fractional part of the tokens whose ages are less than max. Then the fractional parts for the tokens in b_0 will become positive. The young tokens will still have values not exceeding max, while the old tokens will now have values strictly greater than max. This means that if $M \longrightarrow_{T=\delta} M_1$ then $M_1 \models \mathit{Rotate}(\mathcal{R})$ where $\mathit{Rotate}(\mathcal{R})$ is of the form $(\epsilon, \mathit{young} \bullet w, b_{max} + \mathit{old})$. Here, *young* and *old* are sub-multisets of b_0 such that $\mathit{young}(p(n)) = b_0(p(n))$ if $n < max$, and $\mathit{old}(p) = b_0(p(max))$, where $p(n) \in b_0$. Since the fractional parts of the tokens in *young* are smaller than all other tokens, we put *young* first in the second component of the region. Also, the ages of the tokens in *old* are now strictly greater than max, so they are added to the third component of the region.

3. If $b_0 = \epsilon, w = \epsilon$, all tokens have age greater than max. Now, if we let time pass by any amount $\delta \geq 0$ and $M \longrightarrow_{T=\delta} M_1$, then $M_1 \models \mathcal{R}$. When all tokens reach age of max, aging of tokens becomes irrelevant. This case yields only a marking which is equivalent to M with respect to $\equiv$.

Notice that in cases 1 and 2, the stable period is the largest interval during which the marking does not change the region it belongs to. Markings in case 3 never change their regions and are therefore considered to be "stable forever" with respect to timed transitions. Also, we observe that each of first two cases above correspond to "rotating" the multisets in b_0 and w, sometimes also moving them to b_{max}.

We define Rotate^* to be the reflexive transitive closure of Rotate. It computes the set of all regions which we can generate by letting time pass by any amount.

In $\mathit{Rotate}^*(\mathcal{R})$, we apply Rotate to each new region generated, except when a region is of the form (ϵ, ϵ, b). It is straightforward to verify that such a region will be eventually generated (by increasing the age of the tokens, all tokens will eventually become old). This gives us the following lemma.

Lemma 5.13. Rotate^* is effectively constructible and $\mathit{Post}_{Time} = \mathit{Rotate}^*$.

Example 5.14. For the TPN in Figure 3.5, $max = 7$, consider the region $\mathcal{R}$ in Figure 5.5(a). $\mathit{Post}_{Time}(\mathcal{R})$ computes a number of regions. We show first

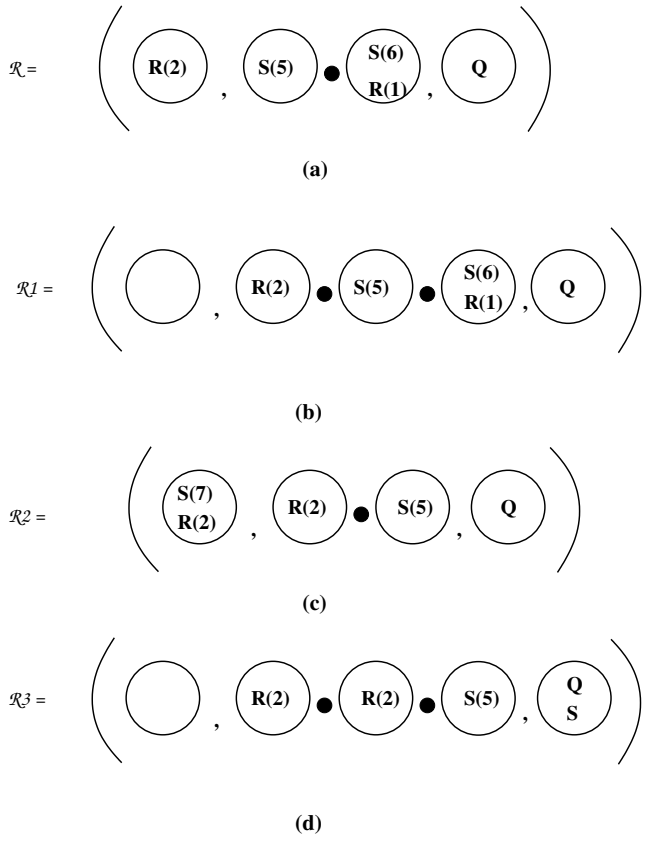

Figure 5.5: A few regions in $Post_{Time}(\mathcal{R})$

three of them in Figure 5.5(b), (c) and (d). Consider the following markings

$$M = [R(2.0), S(5.5), R(1.7), S(6.7), Q(8.9)]$$
$$M_1 = [R(2.1), S(5.6), R(1.8), S(6.8), Q(9.0)]$$
$$M_2 = [R(2.3), S(5.8), R(2.0), S(7.0), Q(9.2)]$$
$$M_3 = [R(2.4), S(5.9), R(2.1), S(7.1), Q(9.3)]$$

and Now, $M \longrightarrow_{T=0.1} M_1 \longrightarrow_{T=0.2} M_2 \longrightarrow_{T=0.1} M_3$ and $M \models \mathcal{R}$, $M_1 \models \mathcal{R}_1$, $M_2 \models \mathcal{R}_2$ and $M_3 \models \mathcal{R}_3$. □

Post$_{Time}$ for region generators

For an input region generator θ, we shall characterize the set of all markings which can be reached from a marking in $[\![\theta]\!]^{\downarrow}$ through the passage of the time. We shall compute $Post_{Time}(\theta)$ as a finite set of region generators such that $[\![Post_{Time}(\theta)]\!]^{\downarrow} = \{M' \mid \exists M \in [\![\theta]\!]^{\downarrow}.\ M \longrightarrow_{Time} M'\}$.

First, we introduce some notations. Let ϕ be an mlg of the form

$\{a_1, \ldots, a_k\}^{\circledast} + a_{k+1} + \cdots + a_{k+\ell}$. Notice that, by the normal form defined in Section 5.2, we can always write ϕ in this form. We define $\sharp\phi$ to be the pair (b, b') where $b = [a_1, \ldots, a_k]$ and $b' = [a_{k+1}, \ldots, a_{k+\ell}]$.

Let ϕ be an mlg over $P \times \{0, \ldots, max\}$ with $\sharp\phi = (b, b')$. We define $young(\phi)$ and $old(\phi)$ to be mlgs over $P \times \{0, \ldots, max - 1\}$ and P respectively such that the following holds: let $\sharp young(\phi) = (b_1, b'_1)$ and $\sharp old(\phi) = (b_2, b'_2)$ such that

- $b(p(n)) = b_1(p(n))$ and $b'(p(n)) = b'_1(p(n))$ if $n < max$.
- $b(p(max)) = b_2(p)$ and $b'(p(max)) = b'_2(p)$.

In other words, from ϕ, we obtain an mlg given by $young(\phi)$ which characterizes tokens younger than max and an mlg $old(\phi)$ which characterizes tokens older than max.

Let ϕ be an mlg over $P \times \{0, \ldots, max - 1\}$ of the form $\{p_1(n_1), \ldots p_k(n_k)\}^{\circledast} + p_{k+1}(n_{k+1}) + \cdots + p_{k+\ell}(n_{k+\ell})$. We use ϕ^{+1} to denote the mlg $\{p_1(n_1 + 1), \cdots, p_k(n_k + 1)\}^{\circledast} + p_{k+1}(n_{k+1} + 1) + \cdots + p_{k+\ell}(n_{k+\ell} + 1)$. That is, we replace each occurrence of a pair $p(n)$ in the representation of ϕ by $p(n+1)$.

For a word atomic expression (an mlg) $\phi = \{a_1, \ldots, a_k\}^{\circledast} + a_{k+1} + \cdots + a_{k+\ell}$, we define $alph(\phi)$ as the set of symbols given by $\{a_1, \ldots, a_{k+\ell}\}$. For a word star expression $e = \{\phi_1, \ldots, \phi_k\}^*$, $alph(e) = \bigcup_i alph(\phi_i)$ for $i : 1 \leq i \leq k$.

We are now ready to define the function $Post_{Time}(\theta_{in})$ for some input region generator θ_{in}. We start from θ_{in} and perform an iteration, maintaining two sets V and W of region generators. Region generators in V are already analyzed and those in W are yet to be analyzed. We pick (also remove) a region generator θ from W, add it to V (if it is not already included in V). We update W and V with new region generators according to the rules described below. We continue until W is empty. At this point we take $Post_{Time}(\theta_{in}) = V$. Depending on the form of θ, we update W and V according to one of the following cases.

- If θ is of the form $(\phi_0, \psi, \phi_{max})$, where $\phi_0 \neq \epsilon$. We add a region generator $(\epsilon, young(\phi_0) \bullet \psi, \phi_{max} + old(\phi_0))$ to W. This step corresponds to one rotation according to case 2 in the computation of *Rotate*.
- If θ is of the form $(\epsilon, \psi \bullet \phi, \phi_{max})$. Here the last element in the second component of the region generator is an atomic expression (an mlg). We add the region generator $\left(\phi^{+1}, \psi, \phi_{max}\right)$ to W. This step corresponds to one rotation according to case 1 for computation of *Rotate*.
- If θ is of the form $\left(\epsilon, \psi \bullet \{\phi_1, \ldots, \phi_k\}^*, \phi_{max}\right)$. Here, the last expression in the second component of the region generator is a star expression. This case is similar to the previous one. However, the tokens corresponding to $\{\phi_1, \ldots, \phi_k\}^*$ now form an unbounded sequence with strictly increasing fractional parts. We add

$$\left(\phi_i^{+1}, \left\{young(\phi_1^{+1}), \ldots, young(\phi_k^{+1})\right\}^* \bullet \psi \bullet \{\phi_1, \ldots, \phi_k\}^*, \phi_{max} + Old^{\circledast}\right)$$

to V, and

$$\left(\phi_i^{+1}, \left\{young(\phi_1^{+1}), \ldots, young(\phi_k^{+1})\right\}^* \bullet \psi, \phi_{max} + Old^{\circledast}\right)$$

to W, for $i : 1 \leq i \leq k$. Here, Old is the union of the sets of symbols occurring in the set of mlgs $\left\{old(\phi_1^{+1}), \ldots, old(\phi_k^{+1})\right\}$, i.e $Old = alph(old(\phi_1^{+1})) \cup \cdots \cup alph(old(\phi_k^{+1}))$. This step corresponds to performing a sequence of rotations of the forms of case 1 and case 2 together for *Rotate*.

Notice that we add one of the newly generated region generators directly to V (and its "successor" to W). This is done in order to avoid an infinite loop where the same region generator is generated all the time.

- If θ is of the form $(\epsilon, \epsilon, \phi_{max})$, i.e., all tokens have ages which are strictly greater than max, then we do not add any element to W.

The termination of this algorithm is guaranteed due to the fact that after a finite number of steps, we will eventually reach a point where we analyze region generators which will only characterize tokens with ages greater than max (i.e. will be of the form $(\epsilon, \epsilon, \phi_{max})$).

Example 5.15. For the TPN in Figure 3.5, where $max = 7$. Consider an input region generator $\theta = (Q(7) + R(6), \{S(6) + Q(5)\}^*, \epsilon)$ in Figure 5.6(a). We show a few region generators computed by $Post_{Time}(\theta)$ starting from θ in Figure 5.6(b) to Figure 5.6(g). Notice the rotation of first and second part of the region generators and sometimes moving to third part of the region generator, corresponding to the 'rotation' described in $Post_{Time}$ for regions. Also, notice that the region generator in (c) and (f) correspond to several 'rotations' of regions in their languages. Furthermore, notice that we need to apply normalisation after computing $Post_{Time}$, since the region generators in Figures 5.6(f) and 5.6(g) are not in normal form. Furthermore, notice that the region generator in Figure 5.6(g) is included in the region generator in Figure 5.6(f) and the former one will be removed during the forward analysis from the working set of the region generators.

□

5.4.2 Discrete Post-image

To give the intuition of computing the post-image of a region generator with respect to a discrete transition, first we show how to perform this operation on regions.

Post$_t$ for regions

We define $Post_{Disc}$ such that it corresponds to firing discrete transitions. $Post_{Disc}$ is computed as the union of $Post_t$ for all transitions t in the TPN where $Post_t$ characterizes the effect of running t once.

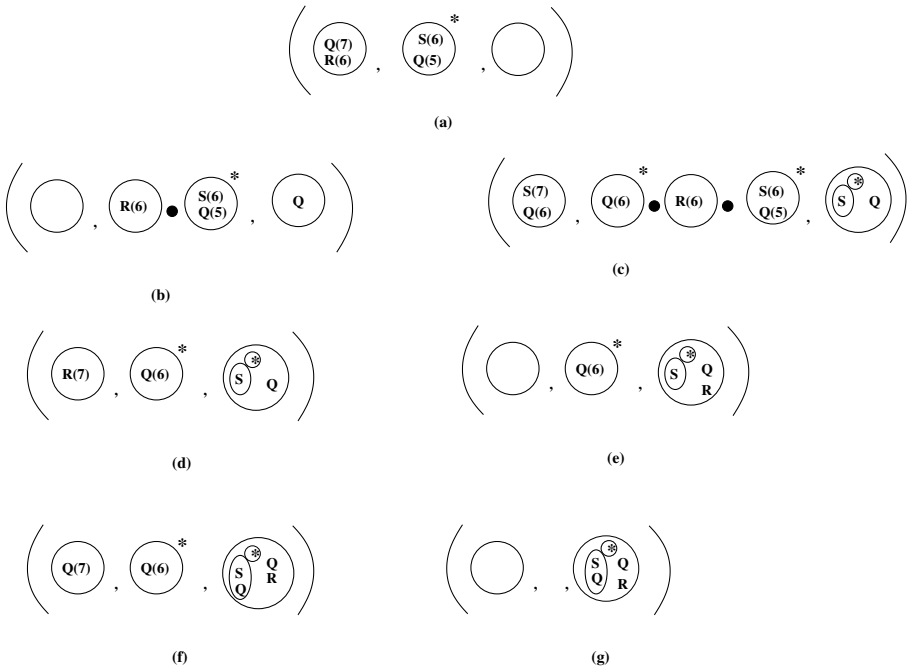

Figure 5.6: Given θ in (a), (b),...,(g) shows the region generators computed by $Post_{Time}(\theta)$.

For an input region $\mathcal{R}$, we shall characterize the set of all markings which can be reached from a marking in $[\![\mathcal{R}]\!]^{=}$ through execution of transition t. We shall compute $Post_t(\mathcal{R})$ as a finite set of regions such that $[\![Post_t(\mathcal{R})]\!]^{=} = \{M' \mid \exists M \in [\![\mathcal{R}]\!]^{=}.\ M \longrightarrow_t M'\}$. For a set $\mathbf{R}$ of regions $Post_t(\mathbf{R}) = \bigcup_{\mathcal{R}\in\mathbf{R}} Post_t(\mathcal{R})$.

Let $\mathcal{R} = (b_0, w, b_{max})$. To give an algorithm for $Post_t$, we need to define an *addition* and a *subtraction* operation for regions.

An addition (subtraction) corresponds to adding (removing) a token in a certain age interval. Let $\mathcal{I}$ be an interval of the form $[w, z)$ and let $\mathcal{R}$ be a region of the form $\mathcal{R} = (b_0, b_1 \bullet \ldots \bullet b_m, b_{m+1})$. We define the *addition* $\mathcal{R} \oplus p(\mathcal{I})$ as a set of regions (the addition of other types of intervals can be defined in a similar manner).

We define $\mathcal{R} \oplus p(\mathcal{I})$ to be the union of the following four sets:

1. A set containing $(b_0 + [p(n)], b_1 \bullet \ldots \bullet b_m, b_{m+1})$, for each $n : w \leq n < z$. This corresponds to adding a token with zero fractional part.

2. A set containing regions $(b_0, b_1 \bullet \ldots \bullet b'_i \bullet \ldots \bullet b_m, b_{m+1})$, where $1 \leq i \leq m$ and $b'_i = b_i + [p(n)]$, for each $n : w \leq n < z$ with $n < max$. An element added according to this case corresponds to adding a token with a fractional part equal to that of some other token.

3. A set containing regions $(b_0, b_1 \bullet \ldots \bullet b_i \bullet [p(n)] \bullet \ldots \bullet b_m, b_{m+1})$, where n satisfies the same conditions as in 2. In this case, the fractional part differs from all other tokens.

4. A singleton set containing $(b_0, b_1 \bullet \ldots \bullet b_m, b_{m+1} + [p])$ if $z = \infty$.

Given $\mathcal{R}$ and $\mathcal{I}$ of the above forms, we define the subtraction $\mathcal{R} \ominus p(\mathcal{I})$ as the union of the following sets:

1. A singleton set containing $(b_0 - [p(n)], b_1 \bullet \ldots \bullet b_m, b_{m+1})$, for each $n : w \leq n < z$. This corresponds to subtracting a token with zero fractional part.

2. A set containing regions $(b_0, b_1 \bullet \ldots b_i' \ldots \bullet b_m, b_{m+1})$ where $b_i' = b_i - [p(n)]$, where $1 \leq i \leq m$ and $w \leq n < z$, $n < max$. This corresponds to subtracting a token with non-zero fractional part.

3. A singleton set containing $(b_0, b_1 \bullet \ldots \bullet b_m, b_{m+1} - [p])$ in case $z = \infty$. This corresponds to removing a token with age greater than max.

4. If all the above sets are empty, then $\mathcal{R} \ominus p(\mathcal{I})$ is undefined.

We extend $\oplus, \ominus$ to sets of regions in the obvious manner.

We also extend $\oplus, \ominus$ for a set $\mathcal{A}$ of pairs of the form $p(\mathcal{I})$ as follows. $\mathcal{R} \oplus \mathcal{A} = \bigcup_{p(\mathcal{I}) \in \mathcal{A}} \mathcal{R} \oplus p(\mathcal{I})$.

Let $\mathcal{A}_{in}(t)$ be the set of input arcs given by $\{p(\mathcal{I}) \mid In(t,p) = \mathcal{I}\}$ and the set of output arcs $\mathcal{A}_{out}(t)$ be given by $\{p(\mathcal{I}) \mid Out(t,p) = \mathcal{I}\}$.

We define

$$Post_t(\mathcal{R}) = (\mathcal{R} \ominus \mathcal{A}_{in}(t)) \oplus \mathcal{A}_{out}(t)$$

.

Example 5.16. For the TPN in Figure 3.5, consider a marking $M = [Q(3.5)]$ and a region $\mathcal{R} = (\epsilon, Q(3), \epsilon)$. $M \models \mathcal{R}$. Consider the transition t_2. We show the regions computed by $Post_t$ in Figure 5.7. Since, $\mathcal{R} \ominus [Q(3.5)] = (\epsilon, \epsilon, \epsilon)$, we show the result of $((\epsilon, \epsilon, \epsilon) \oplus R((0,1))) \oplus S((1,2))$ by $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ which together denote all possible markings that can be created by firing t_2 from M. □

We let $Post = Post_{Time} \cup Post_{Disc}$. From the definition of $Post_{Time}, Post_t$, we get the following.

Lemma 5.17. Given a region $\mathcal{R}$, $Post(\mathcal{R})$ is effectively constructible.

Post$_t$ for region generators

For an input region generator θ, we compute (the downward closure of) the set of all markings which can be reached from a marking in $[\![\theta]\!]^{\downarrow}$ by firing a

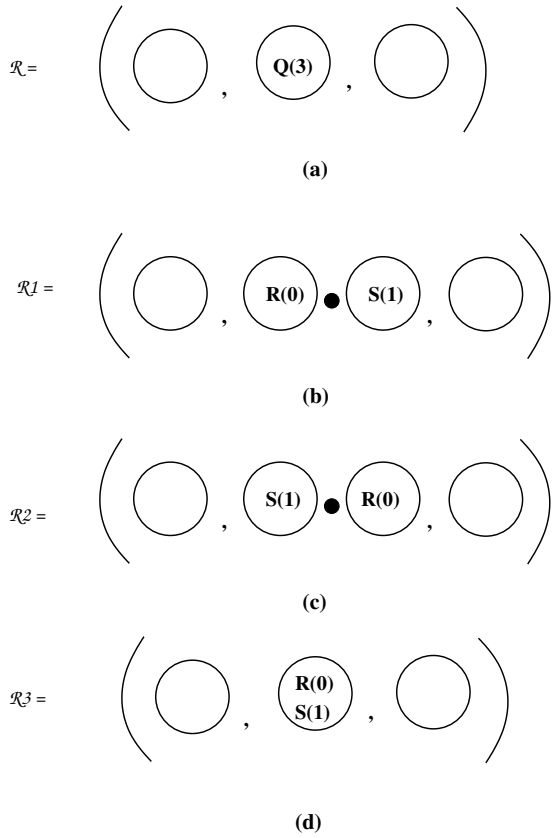

Figure 5.7: Regions in $Post_{t_2}(\mathcal{R})$

discrete transition t, i.e we compute $Post_t(\theta)$ as a finite set of region generators s.t $[\![Post(\theta)]\!]^{\downarrow} = \left\{M' \mid \exists M \in [\![\theta]\!]^{\downarrow}.M \longrightarrow_t M'\right\}\downarrow$.

Notice that from a downward closed set of markings, when we execute a timed transition, the set of markings reached is always downward-closed. But this is not the case for discrete transitions. Therefore, we consider the downward closure of the set of reachable markings in the following algorithm.

To give an algorithm for $Post_t$, we use an *addition* and a *subtraction* operation for region generators. An addition (subtraction) corresponds to adding (removing) a token in a certain age interval. These operations have hierarchical definitions reflecting the hierarchical structure of region generators.

We start by defining addition and subtraction for mlgs, defined over a finite set $P \times \{0, \ldots, max\}$.

Given a *normal* mlg $\phi = S^{\circledast} + a_1 + \cdots + a_\ell$ and a pair $p(n)$ where p is a place and n denotes the integral part of the age of a token in p, we define the *addition* $\phi \oplus p(n)$ to be the mlg $\phi + p(n)$.

The subtraction $\phi \ominus p(n)$ is defined by the following three cases.

- If $p(n) \in S$, then $\phi \ominus p(n) = \phi$. Intuitively, the mlg ϕ describes markings with an unbounded number of tokens each with an integral part equal to n, and each residing in place p. Therefore, after removing one such a token, we will still be left with an unbounded number of them.
- If $p(n) \notin S$ and $a_i = p(n)$ for some $i : 1 \leq i \leq \ell$ then $\phi \ominus p(n) = S^{\circledast} + a_1 + \cdots + a_{i-1} + a_{i+1} + \cdots + a_\ell$.
- Otherwise, the operation is undefined.

Addition and subtraction from mlgs over P is similar where instead of $p(n)$, we simply add (subtract) p.

Now, we extend the operations to wlgs defined over mlgs of the above form.

The addition $\psi \oplus p(n)$ is a wlg ψ consisting of the following three sets of wlgs.

1. For each ψ_1, ψ_2, and ϕ with $\psi = \psi_1 \bullet \phi \bullet \psi_2$, we have
   $\psi_1 \bullet (\phi \oplus p(n)) \bullet \psi_2 \in (\psi \oplus p(n))$.
2. For each ψ_1, ψ_2 and $\psi = \psi_1 \bullet \{\phi_1, \cdots, \phi_k\}^* \bullet \psi_2$, we have for $i : 1 \leq i \leq k$,
   $\psi_1 \bullet \{\phi_1, \cdots, \phi_k\}^* \bullet (\phi_i \oplus p(n)) \bullet \{\phi_1, \cdots, \phi_k\}^* \bullet \psi_2 \in (\psi \oplus p(n))$.
3. For each ψ_1 and ψ_2 with $\psi = \psi_1 \bullet \psi_2$, we have
   $\psi_1 \bullet p(n) \bullet \psi_2 \in (\psi \oplus p(n))$.

Intuitively, elements added according to the first two cases correspond to adding a token with a fractional part equal to that of some other token. In the third case the fractional part differs from all other tokens.

We define the subtraction $\psi \ominus p(n)$, where ψ is a wlg, to be a set of wlgs, according to the following two cases.

- If there is a star expression $e = \{\phi_1, \cdots, \phi_k\}^*$ containing the token we want to remove, i.e., if ψ is of the form $\psi_1 \bullet e \bullet \psi_2$, and if any of the operations $\phi_i \ominus p(n)$ is defined for $i : 1 \leq i \leq k$, then $\psi \ominus p(n) = \{\psi\}$.
- Otherwise, the set $\psi \ominus p(n)$ contains wlgs of the form $\psi_1 \bullet \phi' \bullet \psi_2$ such that ψ is of the form $\psi_1 \bullet \phi \bullet \psi_2$ and $\phi' \in (\phi \ominus p(n))$.

Now we describe how to use the addition and subtraction operations for computing $Post_t$. Addition and subtraction of pairs of the form $p(n)$ can be easily extended to pairs of the form $p(\mathtt{N})$ where $\mathtt{N} \subseteq \{0, \ldots, max\}$, e.g $\psi \ominus p(\mathtt{N}) = \{\psi \ominus p(n) \mid n \in \mathtt{N}\}$.

We recall that, in a TPN, the effect of firing a transition is to remove tokens from the input places and add tokens to the output places. Furthermore, the tokens which are added or removed should have ages in the corresponding intervals. The effect of of firing transitions from the set of markings characterized by

a region generator $\theta = (\phi_0, \psi, \phi_{max})$ can therefore be defined by the following operations.

First, we assume an interval $\mathcal{I}$ of the form (x, y). The subtraction $\theta \ominus p(\mathcal{I})$ is given by the union of the following sets of region generators.

- $(\phi_0 \ominus p(\mathtt{N}), \psi, \phi_{max})$ where each $n \in \mathtt{N}$ is a natural number in the interval $\mathcal{I}$. Intuitively, if the age of the token that is removed has a zero fractional part, then $\mathtt{N}$ contains the valid choices of integral part.
- $(\phi_0, \psi', \phi_{max})$ such that $\psi' \in \psi \ominus p(\mathtt{N})$, where $\mathtt{N} = \{n \mid n \in \mathbb{N} \wedge x \leq n < y\}$ i.e., each n is a valid choice of integral part for the age of the token if it has a non-zero fractional part.
- $(\phi_0, \psi, \phi_{max} \ominus p)$ if $\mathcal{I}$ is of the form (x, ∞), i.e., the age of the token may be greater than max.

Addition is defined in a similar manner. The addition and subtraction operations will be similar if the interval is closed to the left. However, if the interval is closed to the right, the last rule is undefined.

We extend definition of subtraction and addition for subtracting a set of tuples $p(\mathcal{I})$ in the obvious manner. For a set of region generators Θ, we define $\Theta \oplus p(\mathcal{I}) = \bigcup_{\theta \in \Theta} (\theta \oplus p(\mathcal{I}))$. Subtraction for a set of region generators is defined in a similar manner.

Let $\mathcal{A}_{in}(t)$ be the set of input arcs given by $\{p(\mathcal{I}) \mid In(t, p) = \mathcal{I}\}$ and the set of output arcs $\mathcal{A}_{out}(t)$ be given by $\{p(\mathcal{I}) \mid Out(t, p) = \mathcal{I}\}$.

We define,

$$Post_t(\theta) = (\theta \ominus \mathcal{A}_{in}(t)) \oplus \mathcal{A}_{out}(t)$$

Example 5.18. For the TPN in Figure 3.5, consider an input region generator $\theta = (\epsilon, \{R(6)\}^*, \epsilon)$ shown in Figure 5.8(a) and a transition t_1 of the TPN. We show the region generators computed by the above algorithm in Figure 5.8(b), (c) and (d). Notice that we have $\{R(6)\}^* \ominus R(6) = \{R(6)\}^*$. We show the result of $\theta \oplus Q((5, 6))$ in Figure 5.8. Notice that we normalised the resulting set of region generators. □

5.5 Acceleration

In this section, we explain how to accelerate the firing of a single transition interleaved with timed transitions from a region generator. We give a criterion which characterizes when acceleration can be applied. If the criterion is satisfied by an input region generator θ_{in} with respect to a transition t, then we compute a finite set $Accel_t(\theta_{in})$ of region generators such that $[\![Accel_t(\theta_{in})]\!]^{\downarrow} = \left\{M' \mid \exists M \in [\![\theta_{in}]\!]^{\downarrow}.\ M(\longrightarrow_{Time} \cup \longrightarrow_t)^* M'\right\}\downarrow$. We shall not compute the set $Accel_t(\theta_{in})$ in a single step. Instead, we will present a procedure $Step_t$ with the following property: for each region generator θ_{in}, there is an $n \geq 0$ such that $Accel_t(\theta_{in}) = \bigcup_{0 \leq i \leq n} (Post_{Time} \circ Step_t)^i(\theta_{in})$. In other

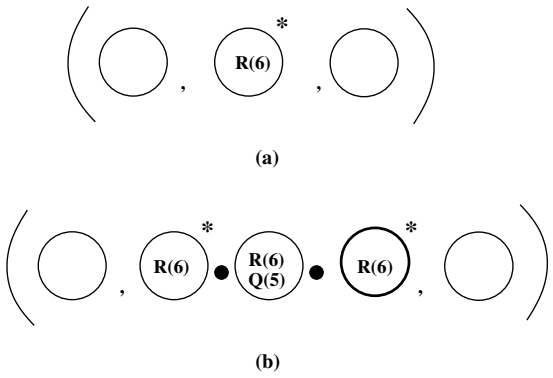

Figure 5.8: Given θ in (a), (b) shows the region generator computed by $Post_{t_1}(\theta)$.

words, the set $Accel_t(\theta_{in})$ will be fully generated through a finite number of applications of $Post_{Time}$ followed by $Step_t$. Since the reachability algorithm of Section 5.3 computes both $Post_{Time}$ and $Step_t$ during each iteration, we are guaranteed that all region generators in $Accel_t(\theta_{in})$ will eventually be produced.

To define $Step_t$ we need some preliminary definitions.

Given a symbol $a \in A$ and an mlg ϕ over A of the form $S^{\circledast} + a_1 + \cdots + a_\ell$, we say that a is a $\circledast - symbol$ in ϕ if $a \in S$. Intuitively, a is a $\circledast - symbol$ in an mlg ϕ if it can occur arbitrarily many times in the multisets in ϕ.
Given a wlg $\psi = e_1 \bullet \cdots \bullet e_l$ over A, we say that a symbol $a \in A$ is a

- $\circledast - symbol$ in ψ if there is an $i : 1 \leq i \leq l$ such that a is a $\circledast - symbol$ for some mlg ϕ occurring in wlg ψ.
- $* - symbol$ in ψ if there is an $i : 1 \leq i \leq l$ such that $a \in alph(e_i)$ and e_i is a word star expression.

Intuitively, a is a $* - symbol$ in ψ if it can occur an arbitrary number of times in arbitrarily many consecutive multisets in a word given by the wlg ψ.

In this section, we show how to perform acceleration when intervals are open, i.e of the form (x, y). It is straightforward to extend the algorithms to closed intervals (see [6] for details).

To compute the effect of acceleration, we define an operation $\uplus$.
Accelerated addition $\uplus$ corresponds to repeatedly adding an arbitrary number of tokens of the form $p(n)$ (with all possible fractional parts) to a region generator θ.

First we define the operation $\uplus$ for mlgs. Given an mlg ϕ and a pair $p(n)$, the accelerated addition $\phi \uplus p(n)$ is given by an mlg $\phi + \{p(n)\}^{\circledast}$.

Given a wlg ψ, $\psi \uplus p(n)$ can be inductively defined as follows.

- If $\psi = \epsilon$, then $\psi \uplus p(n) = \left\{\{p(n)\}^{\circledast}\right\}^*$.

- If $\psi = \phi \bullet \psi'$, then
  $\psi \uplus p(n) = \left\{\{p(n)\}^{\circledast}\right\}^* \bullet (\phi \uplus p(n)) \bullet (\psi' \uplus p(n))$
- If $\psi = \{\phi_1, \cdots, \phi_n\}^* \bullet \psi'$, then
  $\psi \uplus p(n) = \{\phi_1 \uplus p(n), \cdots, \phi_n \uplus p(n)\}^* \bullet (\psi' \uplus p(n))$

Accelerated addition can be extended to sets of pairs of the form $\{p(n_1), \ldots, p(n_k)\}$. Given a wlg ψ, we define $\psi \uplus \{p(n_1), \ldots, p(n_k)\} = \psi \uplus p(n_1) \uplus \cdots \uplus p(n_k)$.

Given a region generator $\theta = (\phi_0, \psi, \phi_{max})$ and a pair $p(\mathcal{I})$ where $\mathcal{I} = (x, y)$, we define
$\theta \uplus p(\mathcal{I}) = \left(\phi_0 + {S_1}^{\circledast},\ \psi \uplus S_2,\ \phi_{max} + \{p_{max}\}^{\circledast}\right)$ where

- $S_1 = \{p(n) \mid n \in \mathbb{N} \wedge x < n < y\}$.
- $S_2 = \{p(n) \mid n \in \mathbb{N} \wedge x \leq n < y\}$.
- $p_{max} = p$ if $y = \infty$, $p_{max} = \epsilon$ otherwise.

For a set of pairs, $\mathcal{A} = \{p_1(\mathcal{I}_1), \cdots, p_k(\mathcal{I}_k)\}$, we define $\theta \uplus \mathcal{A} = \theta \uplus p_1(\mathcal{I}_1) \uplus \cdots \uplus p_k(\mathcal{I}_k)$.

Acceleration Criterion: For a discrete transition t, to check whether we can fire t arbitrarily many times interleaved with timed transitions, first we categorize the input places of t with respect to a region generator $\theta = (\phi_0, \psi, \phi_{max})$ and the transition t.

Type 1 place An input place p of t is said to be of *Type 1* if one of the following holds. Given $In(t, p) = (x, y)$,

- there is an integer n such that $x < n < y$ and $p(n)$ is a $\circledast - symbol$ in ϕ_0.
- there is an integer n such that $x \leq n < y$ and $p(n)$ is a $\circledast - symbol$ or a $* - symbol$ in ψ.
- p is a $\circledast - symbol$ in ϕ_{max} and $y = \infty$.

Intuitively, unbounded number of tokens with the "right age" are available in an input place p of Type 1.

Type 2 place An input place p of t is of *Type 2* if it is not of Type 1, but it is an output place and both the following holds.

1. Given $In(t, p) = \mathcal{I}$, $\theta \ominus p(\mathcal{I}) \neq \emptyset$. Intuitively, for a Type 2 place, there is initially at least one token of the "right age" for firing t.
2. $In(t, p) \cap Out(t, p)$ is a non-empty interval. Intuitively, a token generated as output in any firing may be re-used as an input for the next firing.

We accelerate if each input place of t is a Type 1 place or a Type 2 place.

Acceleration: Let $\mathcal{A}_{in}(t), \mathcal{A}_{out}(t)$ be the set of input and output arcs as defined in Section 5.4. Now, given a region generator θ, we describe acceleration in steps.

- First we subtract input tokens from all input places. Then we add tokens to Type 2 places (places which always re-use an output token as an input for next firing). Formally we compute a set of region generators $\Theta = (\theta \ominus \mathcal{A}_{in}(t)) \oplus T_2$ where $T_2 = \{p(\mathcal{I}) \mid p \text{ is of Type } 2 \wedge p(\mathcal{I}) \in \mathcal{A}_{out}(t)\}$ is the set of output arcs from Type 2 places.

- Next, we accelerate addition for each region generator in Θ and add tokens of all possible ages in the output places which are not of Type 2 (Type 2 places re-use input tokens, therefore do not accumulate tokens), i.e, we compute

$$Step_t(\theta) = \bigcup_{\theta' \in \Theta} \theta' \uplus (\mathcal{A}_{out}(t) \setminus T_2)$$

Example 5.19. Consider the TPN in Figure 3.5 and the region generator θ in Figure 5.9(a). Figure 5.9(b) illustrates the region generator computed by the acceleration algorithm from θ with respect to transition t_1. Notice that all region generators generated by $Post_t$ is entailed by the region generator in Figure 5.9(b). □

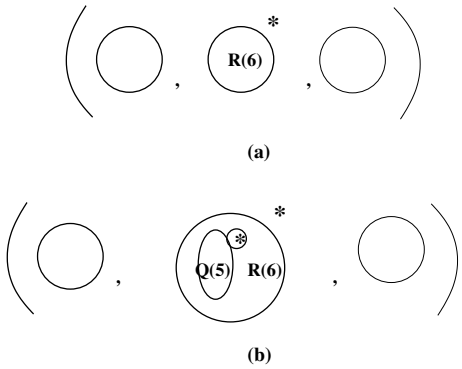

Figure 5.9: Given θ in (a), (b) shows the region generator computed by $Step_{t_1}(\theta)$.

5.6 Experimental Results

We have implemented a prototype based on our algorithm and used it to verify two protocols described in Chapter 3.

Parameterized Model of Fischer's protocol ([14])

In order to prove the mutual exclusion property, we specify markings with two tokens in $CS, CS!$ as the bad markings. We use $\left(\{A(0), A(1)\}^{\circledast} + udf(0), \left\{\{A(0)\}^{\circledast}\right\}^{*}, \{A\}^{\circledast}\right)$ as the initial region generator θ_{init}. θ_{init} characterizes arbitrarily many processes in A having any clock value (age) and one token in udf with age 0. Furthermore, to prove that mutual exclusion is guaranteed, we checked the membership of the bad markings (characterizing an upward closed set of bad states) in the computed set of region generators.

Parameterized Model of Lynch and Shavit's protocol ([100])

The specification of the bad markings is the same as the case of Fischer's protocol. We use $\left(\{A'(0), A'(1)\}^{\circledast} + udf(0) + false(0), \left\{\{A'(0)\}^{\circledast}\right\}^{*}, \{A'\}^{\circledast}\right)$ as the initial region generator θ_{init}. θ_{init} characterizes arbitrarily many processes in A' having any clock value (age), one token in udf with age 0 and one token in $false$ with age 0 denoting that v_2 is false initially.

Producer-Consumer System([109])

We consider the producer/consumer system mentioned in [109][1]. We use $(producer\_ready(0), \epsilon, \epsilon)$ as the initial region generator θ_{init} which characterizes a single token in place $producer\_ready$ with age 0.

Results

Our program computes the reachability set for all the protocols. The procedure fails to terminate without the use of acceleration in each of the above cases. It took 1.16MB memory and 2.12s to analyse Fischer's protocol, 187MB memory and 34 mins to analyse Lynch and Shavit's protocol and 1.02MB memory and 1.25s to analyse producer/consumer system on a 1 GHz processor with 256 MB RAM.

5.6.1 Abstract Graph

Using forward analysis of a TPN, our tool also generates a graph $\mathcal{G}$ which is a finite-state abstraction of the TPN. Each state in $\mathcal{G}$ corresponds to a region generator in the reachability set. Edges of $\mathcal{G}$ are created as follows. Consider two region generators θ_1, θ_2 in the reachability set. If there is a region generator $\theta_2' \in Post_t(\theta_1)$ such that $\theta_2' \subseteq \theta_2$, then we add an edge $\theta_1 \xrightarrow{t} \theta_2$ to $\mathcal{G}$. Similarly, if there is a region generator $\theta_2' \in Post_{Time}(\theta_1)$ such that $\theta_2' \subseteq \theta_2$, then we add an edge $\theta_1 \xrightarrow{\tau} \theta_2$. Notice that each region generator in the post-image should

[1] [109] considers a TPN model with local time in each place.

be included in some region generator in the computed set. It is straightforward to show that the abstract graph simulates the corresponding TPN model.

The graph obtained by the above analysis contains 10 states and 59 edges in the case of Fischer's protocol; 82 states and 770 edges in the case of Lynch and Shavit's protocol; and 11 states and 49 edges in the case of producer/consumer system. Furthermore, we use *The Concurrency Workbench* [49] to minimize the abstract graphs modulo weak bisimilarity. Figure 5.10, Figure 5.11 and Figure 5.12 shows the minimized finite state labelled transition systems for the above protocols.

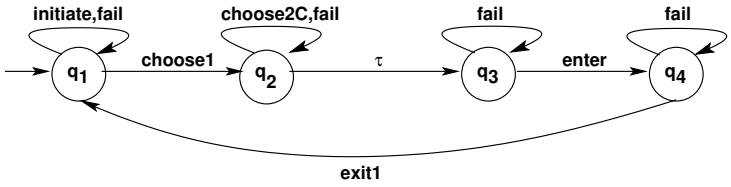

Figure 5.10: Minimized abstract graph for Fischer's protocol.

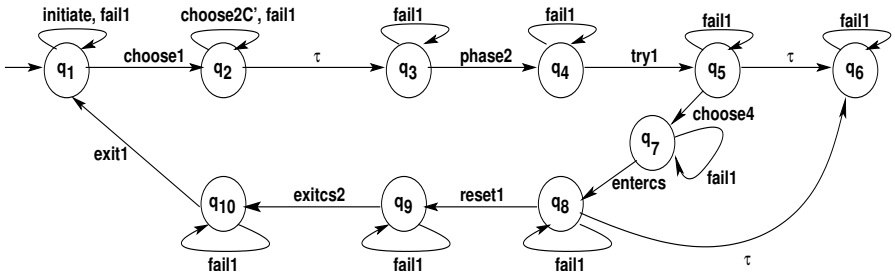

Figure 5.11: Minimized abstract graph for Lynch and Shavit's protocol.

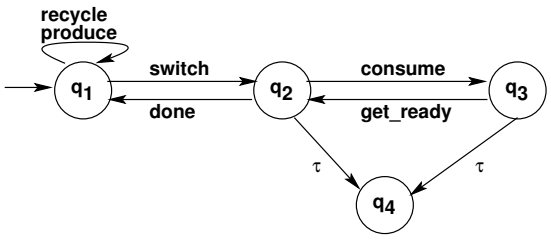

Figure 5.12: Minimized abstract graph for producer/consumer protocol.

5.7 Related Work

The paper [2] considers *simple regular expressions (SRE)* as representations for downward closed languages over a *finite* alphabet. SREs are used for performing forward reachability analysis of lossy channel systems. SREs are not sufficiently powerful in the context of TPNs, since they are defined on a finite alphabet, while in the case of region generators the underlying alphabet is infinite (the set of multisets over a finite alphabet).

Both [53] and [68] consider (untimed) Petri nets and give symbolic representations for upward closed sets and downward closed sets of markings, respectively. The works in [67, 31, 32] give symbolic representation for FIFO automata. These representations are designed for weaker models (Petri nets and FIFO automata) and cannot model the behaviour of TPNs.

Timed Petri nets are considered in [14]. The symbolic representation in this paper characterizes upward closed sets of markings, and can be used for backward analysis, but not for forward analysis.

5.8 Appendix - Proofs of Lemmas

5.8.1 Proof of Theorem 5.2

First, we show some auxiliary lemmas.

Lemma A.1 For a finite alphabet A and a multiset $\kappa \in A^{\circledast}$, there is a set of mlgs Φ such that a multiset $\varrho \in L(\Phi)$ iff $\kappa \not\leq^m \varrho$.

Proof. Let κ be of the form $\left[a_1^{l_1}, \ldots, a_m^{l_m}\right]$. Let e_1 be a star expression $\{b_1, \cdots, b_k\}^{\circledast}$ where $b_1, \ldots, b_k \in A \setminus \{a_1, \ldots, a_m\}$. Notice that $A = \{a_1, \ldots, a_m\}$ implies that $L(e_1) = \{\epsilon\}$. We define Φ as a set of mlgs ϕ_i of the form $e_1 + a_1^{l_1} + \cdots + a_i^{l_i - 1} + \cdots + a_m^{l_m}$ where $i : 1 \leq i \leq m$.

First we show that $\varrho \in L(\Phi) \implies \kappa \not\leq^m \varrho$ by contraposition. Assume $\kappa \leq^m \varrho$. We prove that $\varrho \notin L(\phi_i)$ for each $i : 1 \leq i \leq m$. From the definition of $\leq^m$, we know that ϱ is of the form $\kappa + \vartheta$ where $\vartheta \in A^{\circledast}$. From definition of ϕ_i, $a_i^{l_i} \notin L(\phi_i)$ for each $i : 1 \leq i \leq m$. Therefore, $\kappa + \vartheta \notin L(\phi_i)$ for each i. Therefore, $\varrho \notin L(\Phi)$.

Next, we prove that $\kappa \not\leq^m \varrho \implies \varrho \in L(\phi)$. Let κ' be the largest proper sub-multiset of κ which satisfies $\kappa' \leq^m \varrho$. This means that ϱ is of the form $\kappa' + \vartheta$ where ϑ is a multiset over $A \setminus \{a_1, \ldots, a_m\}$. Thus $\vartheta \in L(e_1)$. Since κ' is a proper submultiset of κ, $\kappa' \in L(a_1^{l_1} + \cdots + a_i^{l_i - 1} + \cdots + a_m^{l_m})$ where $i : 1 \leq i \leq m$. Thus, $\varrho \in L(\Phi)$. □

Lemma A.2 For set of mlgs Φ_1, Φ_2, there is a set of mlgs $\Phi_1 \cap \Phi_2$ such that $L(\Phi_1 \cap \Phi_2) = L(\Phi_1) \cap L(\Phi_2)$.

Proof. First we consider the intersection of mlgs ϕ_1, ϕ_2.
In case, ϕ_1, ϕ_2 are atomic expressions, we have

- if both are atomic expressions, then either of the following holds.
  1. if $\phi_1 = \phi_2 = a$, then $\phi_1 \cap \phi_2 = a$.
  2. $\phi_1 \cap \phi_2 = \epsilon$, otherwise.
- If one of them is a star expression $\{a_1, \cdots, a_l\}^{\circledast}$ and the other one is a, then $\phi_1 \cap \phi_2 = a$ if $a \in \{a_1, \ldots, a_m\}$ for $a, a_1, \ldots, a_m \in A$. Otherwise, $\phi_1 \cap \phi_2 = \epsilon$.
- If both of them are star expressions, i.e, $\phi_1 = \{a_1, \cdots, a_m\}^{\circledast}$ and $\phi_2 = \{b_1, \cdots, b_n\}^{\circledast}$, then $\phi_1 \cap \phi_2 = \{c_1, \cdots, c_k\}^{\circledast}$ where $\{c_1, \ldots, c_k\} = \{a_1, \ldots, a_m\} \cap \{b_1, \ldots, b_n\}$.

If ϕ_1 and ϕ_2 are mlgs, then if either of them is empty, their intersection is also empty. Suppose that we are given two non-empty mlgs $\phi_1 = e_{11} + \cdots + e_{1k}$ and $\phi_2 = e_{21} + \cdots + e_{2m}$. Define ϕ_{1i} to be the result of deleting the expression e_{1i} from ϕ_1. Define ϕ_{2j} in a similar manner. Then, $\phi_1 \cap \phi_2$ is the union of all sets of mlgs ϕ_{ij}, for $i : 1 \leq i \leq k$ and $j : 1 \leq j \leq m$, computed according to one of the following four cases.

1. if e_{1i} and e_{2j} are atomic expressions.
   $$\phi_{ij} = (e_{1i} \cap e_{2j}) + (\phi_{1i} \cap \phi_{2j}).$$
2. if e_{1i} is an atomic expression and e_{2j} is a star expression.
   $$\phi_{ij} = (e_{1i} \cap e_{2j}) + (\phi_{1i} \cap \phi_2).$$
3. if e_{1i} is a star expression and e_{2j} is an atomic expression.
   $$\phi_{ij} = (e_{1i} \cap e_{2j}) + (\phi_1 \cap \phi_{2j}).$$
4. if e_{1i} and e_{2j} are star expressions.
   $$\phi_{ij} = \{(e_{1i} \cap e_{2j}) + (\phi_{1i} \cap \phi_2)\,,\, (e_{1i} \cap e_{2j}) + (\phi_1 \cap \phi_{2j})\}$$

Intuitively, due to commutativity of multiset addition, we intersect all pairs of expressions in two mlgs and repeat the intersection with the rest of the two mlgs. Notice that, if one of e_{1i}, e_{2j} is a star expression, (say, e_{2j}), then we consider whole of ϕ_2 as the "rest" of the mlg. Also notice that we assume that $+$ can be distributed over sets of mlgs.
Now, if $\Phi_1 = \{\phi_1, \cdots, \phi_m\}$ and $\Phi_2 = \{\phi'_1, \cdots, \phi'_n\}$, then $\Phi_1 \cap \Phi_2 = \{\phi_{11}, \cdots, \phi_{n_1 n_2}\}$ where $\phi_{ij} = \phi_i \cap \phi'_j$ for each $i : 1 \leq i \leq m$ and $j : 1 \leq j \leq n$. □

Main proof of Theorem 5.2: Finally, we assume a downward closed language L. If $L = \emptyset$, then $L = L(\Phi)$ where $\Phi = \emptyset$. Otherwise, complement of L is upward closed and can be characterized by a finite set of multisets $\{M_1, \ldots, M_n\}$ over A by Dickson's Lemma [54]. Thus, a multiset $\varrho \in L$, if and only if $M_i \not\leq^m \varrho$ for any $i : 1 \leq i \leq n$. $M_i \in A^{\circledast}$ for each $i : 1 \leq i \leq n$ and by Lemma A.1 and Lemma A.2, it follows that there are sets of mlgs, $\Phi_1, \ldots, \Phi_n$ such that $L = L(\Phi_1) \cap \cdots \cap L(\Phi_n)$.

5.8.2 Proof of Lemma 5.3

We prove the lemma by contraposition. Assume $\phi \not\subseteq \phi'$ for any $\phi' \in \{\phi'_1, \ldots, \phi'_n\}$.

We show that there is a multiset $M \in L(\phi)$, but $M \notin L(\phi')$ for any ϕ' such that $\phi' \in \{\phi'_1, \ldots, \phi'_n\}$. Thus $M \notin L(\{\phi'_1, \cdots, \phi'_n\})$. Therefore, $\phi \not\subseteq \{\phi'_1, \cdots, \phi'_n\}$.

Let ϕ' be of the form $e'_1 + \cdots + e'_k$.

Induction hypothesis **(IH):** For a mlg $\phi = e_1 + \cdots + e_m$ with $m \geq 1$, we have $e_1 + \cdots + e_m \not\subseteq \phi' \implies M_1 + \cdots + M_m \notin L(\phi')$ where multisets $M_i \in L(e_i)$ for $i : 1 \leq i \leq m$ and $M = M_1 + \ldots + M_m$.

Base case ($m = 1$):

First, we prove the claim where ϕ is an atomic expression a. In that case, we define M to be a multiset containing a singleton element a. $a \notin L(e'_i)$ for any $i : 1 \leq i \leq k$. Hence, $a \notin L(\phi')$.

Second, we prove the claim where ϕ is a star expression $\{a_1, \ldots, a_l\}^{\circledast}$. In this case, we define M such that $M(a) = k + 1$ for all $a \in \{a_1, \cdots, a_l\}$, i.e $M = \left[a_1^{k+1}, \ldots, a_l^{k+1}\right]$ and $l > 0$. We use induction on k to show that $M \notin L(\phi')$. The base case ($k = 0$) is trivial. For the induction step, we assume $k > 0$. For each $i : 1 \leq i \leq k$, assuming $\phi' = e'_i + \phi''_i$, we show the claim. There are two cases.

- e'_i is atomic. By the induction hypothesis, we have that $\left[a_1^k, \ldots, a_l^k\right] \notin L(\phi''_i)$. Since e'_i is atomic (contains a singleton), $\left[a_1^{k+1}, \ldots, a_l^{k+1}\right] \notin L(\phi')$.
- e'_i is star expression. We know that $\{a_1, \ldots, a_l\}^{\circledast} \not\subseteq e'_i$ (otherwise, $\phi \subseteq \phi'$ and that is contradiction). Since e'_i is a star expression and $\{a_1, \ldots, a_l\}^{\circledast} \not\subseteq e'_i$, there must be a symbol $a \in \{a_1, \ldots, a_l\}$ such that $a \notin L(e'_i)$. This implies that $[a_1, \ldots, a_l] \notin L(e'_i)$. By the induction hypothesis, we have that $\left[a_1^k, \ldots, a_l^k\right] \notin L(\phi''_i)$. This implies that $\left[a_1^{k+1}, \ldots, a_l^{k+1}\right] \notin L(\phi')$.

Inductive Step ($m > 1$): Let ϕ be of the form $e_1 + \cdots + e_m$. We define $M = M_1 + \cdots + M_m$ where M_i is derived from e_i in the same manner to derivation of M from expressions in the special case above. We show that M satisfies the claim. We use induction on m. If $e_1 \not\subseteq \phi'$, then this case reduces to the case above. Otherwise, we know that $m > 1$ and $e_1 \subseteq \phi''$ such that $\phi' = \phi'' + \phi'''$ where ϕ'' is a minimum mlg (i.e, a mlg consisting of the least number of expressions) which satisfies $e_1 \subseteq \phi''$. Assume that ϕ'' is of the form

$e'_i + \phi'_i$ where $i : 1 \leq i \leq n$ where n is the number of expressions in ϕ''. For each i, we have two possible cases.

- If e'_i is atomic. Since ϕ'' is a minimum mlg which satisfies $e_1 \subseteq \phi''$, $M_1 \notin L(\phi'_i)$. Furthermore, we know that $e_2 + \cdots + e_m \not\subseteq \phi'''$ (otherwise, $\phi \subseteq \phi'$, contradiction). By induction hypothesis, it follows that $M_2 + \cdots + M_m \notin L(\phi''')$. Since e'_i is atomic, we infer that $M_1 + \cdots + M_m \notin L(\phi')$.

- If e'_i is a star expression. Since ϕ'' is a minimum mlg which satisfies $e_1 \subseteq \phi''$, $M_1 \notin L(\phi'_i)$. Furthermore, since e'_i is a star expression, we know that $e_2 + \cdots + e_m \not\subseteq e'_i + \phi'''$ (otherwise, $\phi \subseteq \phi'$, contradiction). By induction hypothesis, it follows that $M_2 + \cdots + M_m \notin L(e'_i + \phi''')$. We infer that $M_1 + \cdots + M_m \notin L(\phi')$.

5.8.3 Proof of Theorem 5.6

First, we show some auxiliary lemmas.

Lemma A.3 For an infinite alphabet $A^{\circledast}$ and a non-empty word $\mu \in (A^{\circledast})^*$, there is a wlg ψ such that a word $\nu \in L(\psi)$ iff $\mu \not\leq^w \nu$.

Proof. Let μ be of the form $M_1 \bullet M_2 \bullet \ldots \bullet M_m$ where $M_1, \ldots, M_m$ are multisets over a finite alphabet A. Let e_i be a star expression Φ_i^* where Φ_i is obtained from multiset M_i for each $i : 1 \leq i \leq m$ as shown in Lemma A.1, satisfying that a multiset $M \in L(\Phi_i)$ if $M_i \not\leq^m M$ for $i : 1 \leq i \leq m$.

On the other hand, for each multiset M_i, it is easy to construct a smallest mlg ϕ_i such that $M_i \in L(\phi_i)$ for each $i : 1 \leq i \leq m$.

We define wlg ψ by $e_1 \bullet \phi_1 \bullet \cdots \bullet e_{m-1} \bullet \phi_{m-1} \bullet e_m$.

First we show that $\nu \in L(\psi) \implies \mu \not\leq^w \nu$ by contraposition. Assume $\mu \leq^w \nu$. We prove that $\nu \notin L(\psi)$. From the definition of $\leq^w$, we know that ν is of the form $\nu_1 \bullet M_1 \bullet \nu_2 \bullet \cdots M_m \bullet \nu_{m+1}$ where $\nu_i \in (A^{\circledast})^*$. From definition of e_i, we know that $M_i \notin L(e_i)$ and hence, $\nu_i \bullet M_i \notin L(e_i)$ for each $i : 1 \leq i \leq m$. This implies that $\nu_1 \bullet M_1 \bullet \nu_2 \bullet \cdots \bullet \nu_m \bullet M_m \notin L(e_1 \bullet \phi_1 \bullet \cdots \bullet \phi_{m-1} \bullet e_m) = L(\psi)$, i.e $\nu \notin L(\psi)$.

Next, we prove that $\mu \not\leq^w \nu \implies \nu \in L(\psi)$. Let l be the largest natural number such that $M_1 \bullet \ldots \bullet M_l \leq^w \nu$. Obviously, $0 \leq l < m$. This means that ν is of the form $\nu_0 \bullet M_1 \bullet \nu_1 \bullet M_2 \cdots \bullet \nu_{l-1} \bullet M_l \bullet \nu_l$, where ν_i is a word over $A^{\circledast} \setminus (M_{i+1} \uparrow)$ for $i : 0 \leq i < l$, where $M_{i+1} \uparrow$ denotes the upward closure of multiset M_{i+1}. Furthermore, we know that M_{l+1} does not occur in ν_l (otherwise, we will have $M_1 \bullet \cdots \bullet M_{l+1} \leq^w \nu$ violating the maximality of l). This implies that $\nu_i \in L(e_{i+1})$ for each $i : 0 \leq i \leq l$. From this and the fact that $M_i \in L(\phi_i)$, we have $\nu \in L(\psi)$. □

Lemma A.4 For wlgs ψ_1, ψ_2, there is a set of wlgs $\psi_1 \cap \psi_2$ such that $L(\psi_1 \cap \psi_2) = L(\psi_1) \cap L(\psi_2)$.

Proof. In case ψ_1 and ψ_2 are atomic expressions (mlgs), $\psi_1 \cap \psi_2$ is same as intersection of two mlgs. In case, one of them is a star expression, i.e, $\psi_1 = \{\phi_1, \ldots, \phi_k\}^*$ and $\psi_2 = \phi_2$, then $\psi_1 \cap \psi_2 = \{\phi_1 \cap \phi_2, \ldots, \phi_k \cap \phi_2\}$. If both of them are star expressions, i.e, $\psi_1 = \{\phi_1, \ldots, \phi_k\}^*$ and $\psi_2 = \{\phi'_1, \ldots, \phi'_m\}^*$, then $\psi_1 \cap \psi_2 = \{\phi_1 \cap \phi'_1, \ldots, \phi_k \cap \phi'_m\}^*$. Let $\psi_1 = e_1 \bullet \psi'_1$ and $\psi_2 = e_2 \bullet \psi'_2$ be non-empty wlgs. We have four cases depending on the form of e_1 and e_2.

1. e_1 and e_2 are atomic expressions,
   $\Psi = \{(e_1 \cap e_2) \bullet (\psi'_1 \cap \psi'_2)\ ,\ (\psi_1 \cap \psi'_2)\ ,\ (\psi'_1 \cap \psi_2)\}$
2. e_1 is an atomic expression and e_2 is a star expression.
   $\Psi = \{(e_1 \cap e_2) \bullet (\psi'_1 \cap \psi_2)\ ,\ (\psi_1 \cap \psi'_2)\}$
3. e_1 is a star expression and e_2 is an atomic expression.
   $\Psi = \{(e_1 \cap e_2) \bullet (\psi_1 \cap \psi'_2)\ ,\ (\psi'_1 \cap \psi_2)\}$
4. e_1 and e_2 are star expressions.
   $\Psi = \{(e_1 \cap e_2) \bullet (\psi_1 \cap \psi'_2)\ ,\ (e_1 \cap e_2) \bullet (\psi'_1 \cap \psi_2)\}$

Notice that we assume the operator $\bullet$ can be distributed over sets of wlgs. □

Main Proof of Theorem 5.6: Consider a downward closed language L of words over multisets. If $L = \emptyset$, then $L = L(\Psi)$ where $\Psi = \emptyset$. Otherwise, complement of L is upward closed and can be characterized by a finite set of words over multisets given by $\{w_1, \ldots, w_n\}$ (by Higman's theorem[84]). Thus, a word $\nu \in L$, if and only if $w_i \not\leq^w \nu$ for any $i : 1 \leq i \leq n$. $w_i \in (A^{\circledast})^*$ for each $i : 1 \leq i \leq n$ and by Lemma A.3 and Lemma A.4, it follows that there are wlgs, $\psi_1, \ldots, \psi_n$ such that $L = L(\psi_1) \cap \cdots \cap L(\psi_n)$.

5.8.4 Proof of Lemma 5.7

Assume $\psi \not\subseteq \psi'$ for any $\psi' \in \{\psi'_1, \ldots, \psi'_n\}$. We show that there is a word $w \in L(\psi)$ such that $w \notin L(\psi')$ for any $\psi' \in \{\psi'_1, \ldots, \psi'_n\}$ which implies that $w \not\subseteq L(\{\psi'_1, \cdots, \psi'_n\})$. This proves that $\psi \not\subseteq \{\psi'_1, \cdots, \psi'_n\}$.

Let k be the number of expressions in wlg $\psi' \in \{\psi'_1, \cdots, \psi'_n\}$, i.e, $\psi' = e'_1 \bullet \ldots \bullet e'_k$.

Induction hypothesis (**IH**): For a wlg $\psi = e_1 \bullet \cdots \bullet e_m$ with $m \geq 1$, we have $e_1 \bullet \cdots \bullet e_m \not\subseteq \psi' \implies w_1 \bullet \cdots \bullet w_m \notin L(\psi')$ where words $w_i \in L(e_i)$ for $i : 1 \leq i \leq m$ and $w = w_1 \bullet \cdots \bullet w_m$.

Base case ($m = 1$):

First, we prove the claim where ψ is an atomic expression, i.e a mlg ϕ. In that case, we follow the proof steps in the general case of Lemma 5.3 and define w to be a word containing a single multiset M derived from ϕ for some natural number k_m where k_m is the length of longest mlg among all mlgs in $e'_1, \cdots, e'_k$. Given, $\psi \not\subseteq \psi'$ and ψ is atomic, $M \notin L(e'_i)$ for any $i : 1 \leq i \leq k$. Hence, $w \notin L(\psi')$.

Second, we prove the claim where ψ is a star expression $e = \{\phi_1, \cdots, \phi_l\}^*$ with ϕ_i is a mlg for $i : 1 \leq i \leq l$. In this case, we define w such that $w = (M_1 \bullet \ldots \bullet M_l)^{k+1}$ and M_i is derived from ϕ_i as before. We use induction on k (length of ψ') to show that $w \notin L(\psi')$. The base case ($k = 0$) is trivial. For the induction step, we assume $k > 0$. There are two cases.

- e'_k is atomic. By the induction hypothesis, we have that $(M_1 \bullet \ldots \bullet M_l)^k \notin L(e'_1 \bullet \cdots \bullet e'_{k-1})$. Since e'_k is atomic, $(M_1 \bullet \ldots \bullet M_l)^{k+1} \notin L(\psi')$.
- e'_k is star expression. We know that $e \nsubseteq e'_k$ (otherwise, $\psi \subseteq \psi'$ and that is contradiction). Since e'_k is a star expression and $e \nsubseteq e'_k$, there must be a mlg ϕ_i in e such that $i : 1 \leq i \leq l$ and $M_i \notin L(e'_k)$. This implies that $M_1 \bullet \ldots \bullet M_l \notin L(e'_k)$. By the induction hypothesis, we have that $(M_1 \bullet \ldots \bullet M_l)^k \notin L(e'_1 \bullet \cdots \bullet e'_{k-1})$. This implies that $(M_1 \bullet \ldots \bullet M_l)^{k+1} \notin L(\psi')$.

Inductive Step ($m > 1$): Let ψ be of the form $e_1 \bullet \cdots \bullet e_m$. We define $w = w_1 \bullet \cdots \bullet w_m$ where w_i is derived from e_i in the same manner to derivation of w from expressions in the special case above. We show that w satisfies the claim. We use induction on m. If $e_1 \nsubseteq \psi'$, then this case reduces to the case above. Otherwise, we know that $m > 1$. Let k_1 be the minimum natural number such that $e_1 \subseteq e'_1 \bullet \cdots \bullet e'_{k_1}$. Now, we have two possible cases.

- If e'_{k_1} is atomic. Since k_1 is the minimum natural number satisfying $e_1 \subseteq e'_1 \bullet \cdots \bullet e'_{k_1}$, $w_1 \notin L(e'_1 \bullet \cdots \bullet e'_{k_1-1})$. Furthermore, we know that $e_2 \bullet \cdots \bullet e_m \nsubseteq e'_{k_1+1} \bullet \cdots \bullet e'_k$. (otherwise, $\psi \subseteq \psi'$, contradiction). By induction hypothesis, it follows that $w_2 \bullet \cdots \bullet w_m \notin L(e'_{k_1+1} \bullet \cdots \bullet e'_k)$. Since e'_{k_1} is atomic, we infer that $w_1 \bullet \cdots \bullet w_m \notin L(\psi')$.
- If e'_{k_1} is a star expression. Since k_1 is the minimum natural number satisfying $e_1 \subseteq e'_1 \bullet \cdots \bullet e'_{k_1}$, $w_1 \notin L(e'_1 \bullet \cdots \bullet e'_{k_1-1})$. Furthermore, since e'_{k_1} is a star expression, we know that $e_2 \bullet \cdots \bullet e_m \nsubseteq e'_{k_1} \bullet \cdots \bullet e'_k$ (otherwise, $\psi \subseteq \psi'$, contradiction). By induction hypothesis, it follows that $w_2 \bullet \cdots \bullet w_m \notin L(e'_{k_1} \bullet \cdots \bullet e'_k)$. We infer that $w_1 \bullet \cdots \bullet w_m \notin L(\psi')$.

Chapter 6

Zenoness

Recently, several verification problems have been studied for TPNs (see e.g. [123, 51, 14, 15]). These problems are both extensions of classical problems previously studied for standard (untimed) Petri nets, and problems which are related to the timed behavior of TPNs.

A fundamental progress property for timed systems is that it should be possible for time to *diverge* [129]. This requirement is justified by the fact that timed processes cannot be infinitely fast. Computations violating this property are called *zeno*. Given a TPN and a marking M, we check whether M is a *zeno-marking*, i.e., whether there is an infinite computation from M with a finite duration. The zenoness problem is solved in [16] for timed automata using the region graph construction. Since region graphs only deal with a finite number of clocks, the algorithm of [16] cannot be extended to check zenoness for TPNs. In this chapter, we solve the zenoness problem for TPNs.

To do this, we consider a subclass of *transfer net* [69] which we call *simultaneous-disjoint transfer net (SD-TN)* . This class is an extension of standard (untimed) Petri nets, in which we also have *transfer* transitions which may move all tokens in one place to another with the restriction that (a) all such transfers take place simultaneously and (b) the sources and targets of all transfers are disjoint.

Given a TPN N, we perform the following three steps:

- Derive a corresponding SD-TN N'.
- Characterize the set of markings in N' from which there are infinite computations[1].
- Re-interpret the set computed above as a characterization of the set of zeno-markings in N.

In fact, the above procedure solves a more general problem than that of checking whether a given marking is zeno; namely it gives a characterization of the set of all zeno-markings.

The zenoness problem was left open in [51] both for dense TPNs (the model we consider so far in this thesis) and for discrete TPNs (where behavior is

[1] Such a characterization is not computable for general transfer nets [104].

interpreted over the discrete time domain). The construction given in this chapter considers the dense case. Later we show how to modify the construction to deal with the discrete case.

Notations

First we give some preliminary definitions.

For a M-computation $\pi = M \longrightarrow_{T=x_1} M'_1 \longrightarrow_{Disc} M_1 \dots M_{n-1} \longrightarrow_{T=x_n} M'_n \longrightarrow_{Disc} M_n$, we use $\pi(i)$ to denote $M \longrightarrow_{T=x_1} M'_1 \longrightarrow_{Disc} M_1 \dots M_{i-1} \longrightarrow_{T=x_i} M'_i \longrightarrow_{Disc} M_i$. The *delay* $\Delta(\pi)$ of the computation is defined as follows.

- $\Delta(\pi(0)) = 0$.
- $\Delta(\pi(i)) = \Delta(\pi(i-1)) + x_i$ for $i : 1 \leq i \leq n$.

Intuitively, the delay of a computation is the total amount of time passed in all timed transitions of the computation.

Zenoness

A zeno-computation of a timed Petri net is an infinite computation that has a finite delay.

ZENONESS-PROBLEM

Instance: A timed Petri net N, and a marking M of N.

Question: Is there an infinite M-computation π and a finite number m such that $\Delta(\pi) < m$?

A marking is called a *zeno-marking* of N if the answer to the above problem is 'yes'.

We consider a timed Petri net N. We let *ZENO* denote the set of all zeno-markings of N.

The decidability of the zenoness-problem for timed Petri nets (i.e., the problem if $M \in ZENO$ for a given marking M, or, more generally, constructing *ZENO*) was mentioned in [51] by Escrig, et.al as as an open problem for both discrete and dense-timed Petri nets. In this chapter, we show that for any TPN, a characterization of the set *ZENO* can be effectively computed. We also show that this implies the computability of *ZENO* also for discrete-timed Petri nets.

The following outline explains the main steps of our proof.

Step 1. We translate the original timed Petri net N into an untimed *simultaneous-disjoint-transfer net* N'. Simultaneous-disjoint-transfer nets are a subclass of *transfer nets* where all transfers happen at the same time and do not affect each other (i.e., sources and targets of a transfer are all disjoint). The computations of N' represent, in a symbolic way, the computations of N that can be performed in time less than $1 - \delta$ for some predefined $0 < \delta < 1$.

Step 2. We consider the set *INF* of markings of N', from which an infinite computation is possible. *INF* is upward-closed and can therefore be characterized by a finite set INF_{min} of minimal elements. While INF_{min} is not computable for general transfer nets [56, 104], it is computable for simultaneous-disjoint-transfer nets, as shown in Lemma 6.33.

Step 3. We re-interpret the set *INF* (resp. INF_{min}) of N' markings in the context of the timed Petri net N and construct from it a characterization of the set *ZENO*.

For the ease of explanation, we first show Step 1 and Step 3. Then, we show how to perform Step 2.

6.1 Step 1 : Translating TPNs to Simultaneous-Disjoint-Transfer Nets

First we define *simultaneous-disjoint-transfer nets*.

Definition 6.1. A simultaneous-disjoint-transfer net (short SD-TN) N is described by a tuple $(P, T, Input, Output, Trans)$ where

- P is a set of places,
- T is a set of ordinary transitions,
- $Input, Output : T \longrightarrow 2^P$ are functions that describe the input and output places of every transition, respectively (as in ordinary Petri nets), and
- *Trans* describes the simultaneous and disjoint transfer transition. We have $Trans = (I, O, ST)$ where $I \subseteq P$, $O \subseteq P$, and $ST \subseteq P \times P$. $Trans$ consists of two parts: (a) I and O describe the input and output places of the Petri net transition part; (b) the pairs in ST describe the source and target places of the transfer part. Furthermore, the following restrictions on *Trans* must be satisfied:
  - If $(sr, tg), (sr', tg') \in ST$ then sr, sr', tg, tg' are all different and $\{sr, tg\} \cap (I \cup O) = \emptyset$.

Let $M : P \longrightarrow \mathbb{N}$ be a marking of N. We use $\leq^m$ as the ordering on the set of markings (Chapter 2). The firing of normal transitions $t \in T$ is defined just as for ordinary Petri nets. The transfer transition *Trans* is enabled at M iff $\forall p \in I.\, M(p) \geq^m 1$. Firing *Trans* yields the new marking M' where

$$
\begin{array}{ll}
M'(p) = M(p) & \text{if } p \in I \cap O \\
M'(p) = M(p) - 1 & \text{if } p \in I - O \\
M'(p) = M(p) + 1 & \text{if } p \in O - I \\
M'(p) = 0 & \text{if } \exists p'.\, (p, p') \in ST \\
M'(p) = M(p) + M(p') & \text{if } (p', p) \in ST \\
M'(p) = M(p) & \text{otherwise}
\end{array}
$$

The restrictions above ensure that these cases are disjoint.

We use $M \longrightarrow M'$ to denote that M' is reached from M either by executing an ordinary Petri net transition $t \in T'$ or a transfer transition $Trans$.

In the following, sometimes we use *transfer* transition to mean simultaneous-disjoint transfer transitions.

6.1.1 Construction of SD-TN from a TPN

For a given TPN $N = (P, T, In, Out)$ we construct a SD-TN $N' = (P', T', Input, Output, Trans)$. The intuition is that N' simulates symbolically all computations of N which can happen in time $< 1 - \delta$ for some predefined $\delta > 0$. First we show how to construct the places of SD-TN. Then we show how to simulate a discrete transition of N by a set of transitions of N'. Finally, we show how to simulate timed transitions of N by simultaneous-disjoint-transfers and a set of normal discrete transitions as in ordinary PNs.

As in Chapter 3, we let max be the maximal finite constant that appears in the arcs of the TPN. We define a finite set of symbols $Sym := \{k \mid k \in \mathbb{N}, 0 \leq k \leq max\} \cup \{k+ \mid k \in \mathbb{N}, 0 \leq k \leq max\} \cup \{k- \mid k \in \mathbb{N}, 1 \leq k \leq max\}$ and a total order on Sym by $k < k+ < (k+1)- < (k+1)$ for every k.

Constructing places of SD-TN

We let $P' = \{p(sym) \mid p \in P, sym \in Sym\}$, i.e., for every place $p \in P$ of N we have a set containing places of the form $p(sym)$ such that $sym \in Sym$.

A token in place $p(k)$ encodes a token of age exactly k on place p. A token in $p(k+)$ encodes a token in place p of an age x which satisfies $k < x \leq k + \delta$ for some a-priori defined $0 < \delta < 1$. This means that the age of this token cannot reach $k+1$ in any computation taking time $< 1 - \delta$. A token in $p(k-)$ encodes a token in p whose age x satisfies $k - 1 + \delta < x < k$ and which may or may not reach age k during a computation taking time $1 - \delta$. For instance, given $\delta = 0.6$, a TPN token $p(1.5)$ is encoded as $p(1+)$ while another TPN token $p(2.7)$ is encoded as $p(3-)$. The SD-TN tokens $p(k), p(k+)$ and $p(k-)$ are called *symbolic encodings* of the corresponding TPN token $p(a)$.

In particular, the age of a $p(k-)$ token could be chosen arbitrarily close to k, such that its age could reach (or even exceed) k in computations taking an arbitrarily small time.

Translating Discrete Transitions

First we define a function $enc : Intrv \longrightarrow 2^{Sym}$ as follows.

$$\begin{array}{lcl} enc([x,y]) & := & \{sym \in Sym \mid x \leq sym \leq y\} \\ enc((x,y]) & := & \{sym \in Sym \mid x < sym \leq y\} \\ enc([x,y)) & := & \{sym \in Sym \mid x \leq sym < y\} \\ enc((x,y)) & := & \{sym \in Sym \mid x < sym < y\} \end{array}$$

This means that given an interval $[1,2]$, $enc([1,2]) = \{1, 1+, 2-, 2\}$. We say that $enc(\mathcal{I})$ is an *encoding* of interval $\mathcal{I}$.

For every transition $t \in T$ in the TPN N we have a set $T'(t)$ of new transitions in N'. The intuition is that the transitions in $T'(t)$ encode all possibilities of the age intervals of input and output tokens.

Example 6.2. Consider the TPN in Figure 6.1(1) The only (discrete) transition t has an input arc from place p labeled $[0,1]$ and two output arcs both labelled $[0,0]$ to places p and q respectively. Translation of this transition in SD-TN would yield 4 different transitions in $T'(t)$ with input arcs from places $p(0), p(0+), p(1-)$ and $p(1)$, respectively (as shown in Figure 6.1(2)(a), 6.1(b), 6.1(c) and 6.1(d)). □

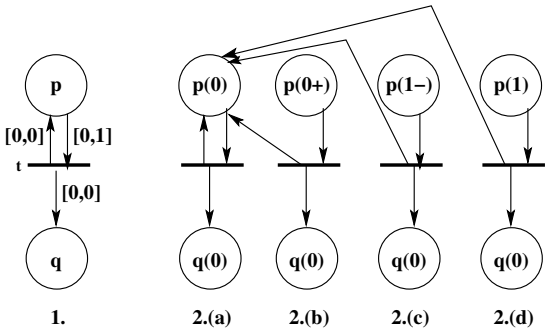

Figure 6.1: Simulating (1) t in TPN by (2) a set $T'(t)$ consisting of 4 transitions in 2.(a), 2.(b), 2.(c) and 2.(d).

Example 6.3. Consider the TPN in Figure 6.2(1) The only (discrete) transition t has input arc from place p as in Figure 6.1(1), but the output arc to place q is labelled by $[0,1]$. This will yield 16 different transitions in $T'(t)$ (shown in Figure 6.2(2)), since $enc([0,1]) = \{0, 0+, 1-, 1\}$.

□

Each transition t of TPN N yields a set $T'(t)$ of transitions in the corresponding SD-TN N'. Each transition in the set $T'(t)$ is of the form $t'(A,B)$ where A and B are the set of input and output places of $t'(A,B)$ respectively, i.e., $Input(t'(A,B)) = A$ and $Output(t'(A,B)) = B$. In the following, for each transition t in TPN, we compute a set $\mathcal{P}_{in}(t)$ ($\mathcal{P}_{out}(t)$) which contains the set of input (output) places for each transition in $T'(t)$.

For every $t \in T$, consider the set of input arcs $\mathcal{A}_{in}(t) = \{p_1(\mathcal{I}_1), \ldots, p_m(\mathcal{I}_m)\}$ and the set of output arcs $\mathcal{A}_{out}(t) = \{p'_1(\mathcal{J}_1), \ldots, p'_\ell(\mathcal{J}_\ell)\}$. Now, we define $\mathcal{P}_{in}(t) \subseteq 2^{P'}$ where each element in $\mathcal{P}_{in}(t)$ is a set A of places and is given by

$$A = \{p_1(sym_1), \ldots, p_m(sym_m)\}$$

where $sym_i \in enc(\mathcal{I}_i)$ for $i : 1 \leq i \leq m$. Intuitively, each set A in $\mathcal{P}_{in}(t)$ corresponds to a unique combination of encodings of input tokens of t in N.

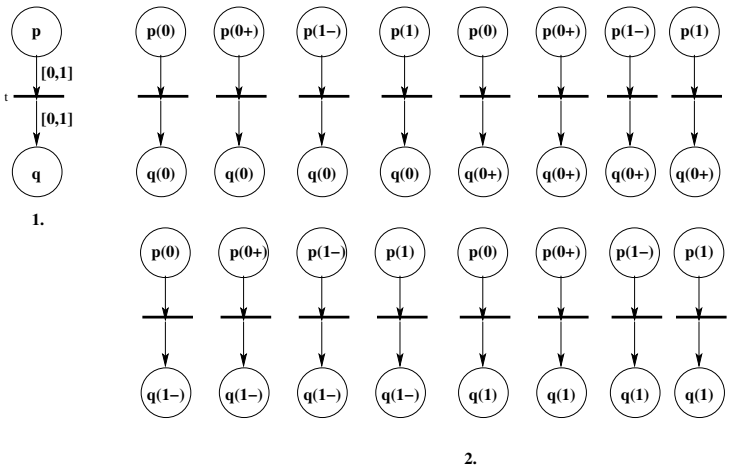

Figure 6.2: Simulating (1) t in TPN by (2) a set $T'(t)$ consisting of 16 transitions.

For every $t \in T$ we define $\mathcal{P}_{out}(t) \subseteq 2^{P'}$ in a similar manner. We define $\mathcal{P}_{out}(t)$ where each element in $\mathcal{P}_{out}(t)$ is a set B of places and is given by

$$B = \{p'_1(sym'_1), \ldots, p'_\ell(sym'_\ell)\}$$

where $sym'_i \in enc(\mathcal{J}_i)$ for $i : 1 \leq i \leq \ell$. Similarly, each set B in $\mathcal{P}_{out}(t)$ corresponds to a unique combination of encodings of output tokens of t in N.

We define $T'(t) := \{t'(A, B) \mid A \in \mathcal{P}_{in}(t), B \in \mathcal{P}_{out}(t)\}$ and finally $T' := \bigcup_{t \in T} T'(t)$.

Example 6.4. Consider the example in Figure 6.1. Here, $In(t, p) = [0, 1]$ and $Out(t, p) = [0, 0]$; $enc([0, 1]) = \{0, 0+, 1-, 1\}$ and $enc([0, 0]) = \{0\}$. Then $\mathcal{P}_{in}(t) = \{\{p(0)\}, \{p(0+)\}, \{p(1-)\}, \{p(1)\}\}$ and $\mathcal{P}_{out}(t) = \{q(0)\}$. The four transitions in Figure 6.1.2 given by $t'(\{p(0)\}, \{q(0)\})$, $t'(\{p(0+)\}, \{q(0)\})$, $t'(\{p(1-)\}, \{q(0)\})$ and $t'(\{p(1)\}, \{q(0)\})$ respectively. $T'(t)$ consists of the above four transitions. □

Translating Timed Transitions

So far, the transitions in T' only encode the discrete transitions of N. The transfer arc will be used to encode the passing of time. However, since we need to keep discrete transitions and time-passing separate, we must first modify the net to obtain alternating discrete phases and time-passing phases.

First we add two extra places p_{disc} and p_{time} to P' which act as control-states for the different phases. Then we modify all transitions $t \in T'$ by adding p_{disc} to $Input(t)$ and $Output(t)$. Thus normal transitions can fire iff p_{disc} is marked.

The transitions which encode the passing of time include both the

simultaneous-disjoint transfer transition and several normal transitions as follows.

- After an arbitrarily small amount of time < 1 passes, all tokens of age k have an age $> k$. This is encoded by the simultaneous-disjoint transfer arc, which moves all tokens from places $p(k)$ to places $p(k+)$. Formally, $\mathit{Trans} := (I, O, ST)$ where $I := \{p_{disc}\}$, $O := \{p_{time}\}$, and $ST := \{(p(k), p(k+)) \mid 0 \leq k \leq max\}$. Note that *Trans* starts the time-passing phase and switches the control-state from p_{disc} to p_{time}.
- Next we add two new sets of transitions to T', which encode what happens to tokens of age $k-$ when (a small amount of) time passes. Their age might either stay below k, reach k or exceed k. Notice that we do not need to do anything in the first case.
  - For every $k \in \{1, \ldots, max\}$ we have a transition with input places p_{time} and $p(k-)$ and output places p_{time} and $p(k)$. This encodes the second scenario.
  - Furthermore, for every $k \in \{1, \ldots, max\}$ we have a transition with input places p_{time} and $p(k-)$ and output places p_{time} and $p(k+)$. This encodes the third scenario.
- Finally, we add an extra transition t_{switch} with input place p_{time} and output place p_{disc}, which switches the net back to normal discrete mode.

Note that after a time-passing phase the only tokens on places $p(k)$ are those which came from $p(k-)$, because all tokens on $p(k)$ were first transferred to $p(k+)$ by the transfer transition.

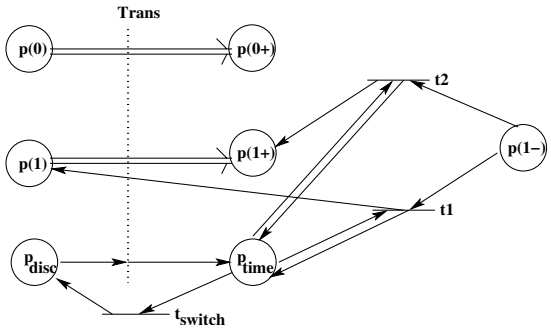

Figure 6.3: Simulating timed transition in TPN for $< 1 - \delta$ time.

Example 6.5. In Figure 6.3, we simulate the timed transitions of a TPN with a single place p and $max = 1$. The transfer transition is shown in dotted line, the transfer arcs are shown as double-arrow from the source of the transfer to the target of the transfer (namely, from $p(0)$ to $p(0+)$ and from $p(1)$ to $p(1+)$). The Petri net part of transfer (input from p_{disc} and output to p_{time}) is shown

as ordinary arcs. t_1 and t_2 move a token from $p(1-)$ to $p(1)$ and to $p(1+)$ respectively, if there is a token in p_{time}. Finally, t_{switch} moves the token from p_{time} to p_{disc} and ends the time-passing phase. □

6.2 Step 3: Constructing ZENO

In this section, we show how to compute the set ZENO.

Definition 6.6. Let N be a TPN and N' the corresponding SD-TN, defined as in Section 6.1.1. We say that a marking M' of N' is a *standard marking* if $M'(p_{disc}) = 1$ and $M'(p_{time}) = 0$. We denote by *INF* the set of all markings of N' from which infinite computations start. Since *INF* is upward-closed with respect to $\leq^m$ and $\leq^m$ is a well-quasi-ordering, *INF* can be characterized by its finitely many minimal elements (Chapter 4). Let INF_{min} be such a set of minimal elements (markings). Let INF' and INF'_{min} be the restriction to standard markings of *INF* and INF_{min}, respectively.

The following definitions establish the connection between the markings of the timed Petri net N and the markings of the SD-TN N'.

Definition 6.7. For every δ with $0 < \delta < 1$ we define a function $int_\delta : (P \times \mathbb{R}^{\geq 0})^{\circledast} \longrightarrow (P' \longrightarrow \mathbb{N})$ that maps a marking M of N to its corresponding marking M' in N'. $M' := int_\delta(M)$ is defined as follows. For any k we have

$$\begin{array}{lcl} M'(p(k)) & := & M(p(k)) \\ M'(p(k+)) & := & \sum_{k<x\leq k+\delta} M(p(x)) \\ M'(p((k+1)-)) & := & \sum_{k+\delta<x<k+1} M(p(x)) \end{array} \qquad \begin{array}{lcl} M'(p_{disc}) & := & 1 \\ M'(p_{time}) & := & 0 \end{array}$$

Note that $M' = int_\delta(M)$ is a standard marking according to Def. 6.6.

For instance, for $\delta = 0.8$ and a marking $M = [p(1), p(0.5), p(0.95), p(1.9)]$, we have $int_\delta(M) = [p(1), p(0+), p(1-), p(2-), p_{disc}]$.

The intuition is as follows. In an infinite computation π starting at M with $\Delta(\pi) < 1 - \delta$, no token $p(x)$ with $k < x \leq k + \delta$ can reach age $k + 1$ by aging. This is reflected in N' by the fact that $p(k+)$ tokens are not affected during the time-passing phase. On the other hand, tokens $p(x)$ with $k + \delta < x < k + 1$ can reach an age $\geq k + 1$ by aging. This is reflected in N' by the fact that $p((k + 1)-)$ tokens can become $p(k + 1)$ or $p((k + 1)+)$ tokens during the time-passing phase.

Lemma 6.8. Consider a TPN N with marking M_0, the corresponding SD-TN N' constructed as above, and $0 < \delta < 1$. If there exists an infinite M_0-computation π such that $\Delta(\pi) < 1 - \delta$ then there exists an infinite $int_\delta(M_0)$-computation π' in N', i.e., $int_\delta(M_0) \in INF'$.

Proof. We show that for every infinite M-computation $\pi = M_0 \longrightarrow_{T=x_1} M'_1 \longrightarrow_t M_1 \ldots\ldots$ with $\Delta(\pi) < 1 - \delta$ there is a corresponding infinite computation in N' of the form $int_\delta(M_0) \stackrel{*}{\longrightarrow} int_{\delta_1}(M'_1) \longrightarrow int_{\delta_1}(M_1) \stackrel{*}{\longrightarrow}$

$int_{\delta_2}(M'_2) \longrightarrow int_{\delta_2}(M_2) \ldots$ with $\delta_0 = \delta$ and for all i, $1 > \delta_{i+1} \geq \delta_i$. Let π_i be the infinite suffix of π starting at M_i. The values of δ_i will be defined such that $\Delta(\pi_i) < 1 - \delta_i$. (The condition $\delta_{i+1} \geq \delta_i$ is required, because $\Delta(\pi_{i+1}) \leq \Delta(\pi_i)$.)

For every discrete transition step $M'_i \longrightarrow_t M_i$ there exists a transition step in N' of the form $int_{\delta_i}(M'_i) \longrightarrow_{dt} int_{\delta_i}(M_i)$, where $dt \in T'(t)$ by the construction in Section 6.1.1 and Def. 6.7.

For every timed transition step $M_i \longrightarrow_{T=x_{i+1}} M'_{i+1}$ (with $x_{i+1} < 1 - \delta_i$) we have $\delta_{i+1} = \delta_i + x_{i+1}$. By the construction in Section 6.1.1 and Def. 6.7 there is a sequence of transitions in N' (the encoding of the time-passing phase) of the form $int_{\delta_i}(M_i) \stackrel{*}{\longrightarrow} int_{\delta_i + x_{i+1}}(M'_{i+1})$. Note in particular that if some token $p(x)$ with $k + \delta_i < x < k + 1$ reaches an age equal to (or greater than) $k + 1$ in the transition from M_i to M'_{i+1} then its encoding $p((k+1)-)$ can be transformed into a token $p(k+1)$ or $p((k+1)+)$ in the time-passing phase of N'. Furthermore, all tokens in M_i with fractional part 0 are transformed into tokens with a positive fractional part in M'_{i+1}. In N' this is encoded by the fact that all $p(k)$ tokens become $p(k+)$ tokens in the time-passing phase. □

The reverse implication of Lemma 6.8 does not generally hold. The fact that $int_\delta(M) \in INF'$ for some marking M of a TPN N does not imply that there is an infinite M-computation in the corresponding TPN. The infinite $int_\delta(M)$-computation in N' depends on the fact that the $p(k-)$ tokens do (or don't) become $p(k)$ or $p(k+)$ tokens at the right step in the computation. For example, in an infinite computation taking time 0.5, two different tokens $p(0.8)$ and $p(0.9)$ are both interpreted as $p(1-)$ in N'. However, $p(0.8)$ cannot become $p(1)$ by aging unless $p(0.9)$ becomes $p(1.1)$, while their symbolic encodings $p(1-)$ can become $p(1)$ or $p(1+)$ in any order.

To establish a reverse correspondence between markings of N' and markings of N we need the following definitions.

Definition 6.9. Consider a TPN $N = (P, T, In, Out)$. Let N' be the corresponding SD-TN with places $P' = \{p(sym) \mid p \in P, sym \in Sym\} \cup \{p_{disc}, p_{time}\}$ and a standard marking $M' : P' \longrightarrow \mathbb{N}$. We let M'^-, M'^+ to be sub-markings of M' such that

- $M'^-(p(k-)) = M'(p(k-))$ for each place of the form $p(k-)$ in P'; $M'^-(p(k+)) = 0$ and $M'^-(p(k)) = 0$ for each place of the form $p(k+)$ and $p(k)$ in P' respectively.
- $M'^+(p(k+)) = M'(p(k+))$ for each place of the form $p(k+)$ in P'. But $M'^+(p(k-)) = 0$ and $M'^+(p(k)) = 0$ for each place of the form $p(k-)$ and $p(k)$ in P' respectively.

We define $perm(M'^-)$ as the set of all words $w_- = b_1 \bullet \ldots \bullet b_n \in \left((P \times \{0, \ldots, max - 1\})^{\circledast}\right)^*$ such that for all p and $k < max$ we have that $M'^-(p((k+1)-)) = b_1(p(k)) + \ldots + b_n(p(k))$. Similarly, let $perm(M'^+)$ be the set of all words $w_+ = b_1 \bullet \ldots \bullet b_n \in \left((P \times \{0, \ldots, max - 1\})^{\circledast}\right)^*$ such that for all p and $k < max$, we have $M'^+(p((k)+)) = b_1(p(k)) + \ldots + b_n(p(k))$.

Intuitively, $perm(M'^{-})$ describes all possible permutations of the fractional parts of tokens in a TPN marking M which are symbolically encoded as $p(k-)$ tokens in the corresponding SD-TN standard marking M'. $perm(M'^{+})$ can be explained in a similar manner.

Example 6.10. Let $max = 1$.
Consider $M' = [p_{disc}, p(1), q(1+), p(0+), q(1-), q(1-)]$. Then $perm(M'^{-}) = \{[q(0)] \bullet [q(0)]\ ,\ [q(0), q(0)]\}$ and $perm(M'^{+}) = \{[p(0)]\}$. Notice that $q(1+)$ does not belong to $perm(M'^{+})$ (since $max = 1$). □

Every symbolic marking M' of the SD-TN defines a set of TPN markings, depending on which permutation of the $p(k-)$ tokens and $p(k+)$ tokens is chosen.

Definition 6.11. Let N' be a SD-TN. Every standard marking $M' : P' \longrightarrow \mathbb{N}$ induces a set of regions $Reg(M', w_+, w_-)$ of the form $(b_0, w_+ \bullet w_-, b_{max})$, where $b_0(p(k)) = M'(p(k))$ for all p and all $k \leq max$, $w_+ \in perm(M'^{+})$, $w_- \in perm(M'^{-})$ and $b_{max}(p) = M'(p(max+))$ for all p.

Example 6.12. Again consider $M' = [p_{disc}, p(1), q(1+), p(0+), q(1-), q(1-)]$ and the sets $perm(M'^{+}), perm(M'^{-})$ in Example 6.10. $Reg(M', w_+, w_-)$ consists of 2 regions shown in Figure 6.4. □

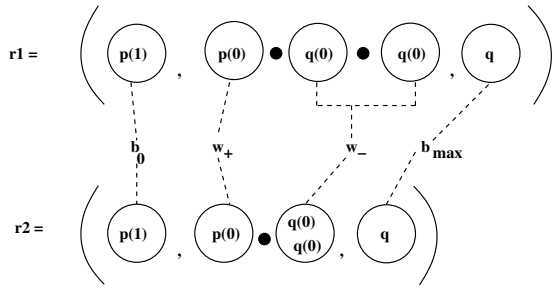

Figure 6.4: $Reg(M', w_+, w_-) = \{\mathcal{R}_1, \mathcal{R}_2\}$

Next we show how an infinite computation of SD-TN corresponds to a zeno computation in the TPN. (Recall from Chapter 4 that $[\![\mathbf{R}]\!]^{\uparrow} = [\![\mathbf{R}\uparrow]\!]^{=}$ for a set $\mathbf{R}$ of regions.)

Lemma 6.13. Let N be a TPN with corresponding SD-TN N' and $M' \in INF'$.

$$\exists w_- \in perm(M'^{-}).\, \forall w_+ \in perm(M'^{+}).\, [\![Reg(M', w_+, w_-)]\!]^{\uparrow} \subseteq ZENO$$

Proof. Since $M' \in INF'$ there is an infinite M'-computation $\pi' = M' \longrightarrow M'_1 \longrightarrow M'_2 \longrightarrow \ldots$. This computation contains a (possibly infinite) number of time-passing phases (where the control-token shifts to the place p_{time}) $tpp_1, tpp_2, \ldots$. Now consider the *original* $p(k-)$ tokens in M' which become $p(k)$ tokens or $p(k+)$ tokens in the i-th time-passing phase tpp_i. Other tokens

which were *newly created* during the computation π' are not considered. Let α_i be the multiset of $p(k-)$ tokens in M' which become $p(k+)$ tokens in tpp_i and β_i the multiset of $p(k-)$ tokens in M' which become $p(k)$ tokens in tpp_i. (Note that this does not happen by the simultaneous discrete transfer transition, but by normal transitions in the time-passing phase.) We have $\alpha_i, \beta_i \leq^m M'^-$, but not necessarily $\Sigma_{i\in\mathbb{N}}\alpha_i + \beta_i = M'^-$, because $p(k-)$ tokens can also be used by normal transitions in the discrete phase or never become $p(k)$ or $p(k+)$ tokens at all. Let $\gamma := M'^- - \Sigma_{i\in\mathbb{N}}\alpha_i + \beta_i$. Since M'^- is finite, there exists a smallest number m such that $\alpha_i + \beta_i = \emptyset$ for all $i > m$. It follows that there exists a minimal number n such that π' has an infinite suffix π'' starting at M'_n, such that in π'' no *original* $p(k-)$ token of M' becomes a $p(k)$ or $p(k+)$ token.

We define $w_2 \in perm(M'^-)$ by $w_2 := \gamma\beta_m\alpha_m \ldots \beta_1\alpha_1$.

We need to prove that $\forall w_1 \in perm(M'^+).[\![Reg(M', w_1, w_2)\uparrow]\!]^= \subseteq ZENO$. For this it suffices to show that $[\![Reg(M', w_1, w_2)]\!]^= \subseteq ZENO$, because $ZENO$ is upward-closed. Now let $w_1 \in perm(M'^+)$ and $M \in [\![Reg(M', w_1, w_2)]\!]^=$. We need to show that $M \in ZENO$, i.e., that there exists an infinite time-bounded M-computation π.

Since $M \in [\![Reg(M', w_1, w_2)]\!]^=$ there exists a δ with $0 < \delta < 1$ and $int_\delta(M) = M'$. We construct an infinite M-computation π with $\Delta(\pi) < 1 - \delta$. The computation π has the form $M \longrightarrow M_{j_1} \longrightarrow M_{j_2} \longrightarrow \ldots$ where the sequence $\{j_i\}_{i\in\mathbb{N}}$ is a subsequence of $1, 2, \ldots$ and $int_{\delta_{j_i}}(M_{j_i}) = M'_{j_i}$ and $\delta_{j_i} = \delta + \Delta(M \longrightarrow M_{j_1} \longrightarrow M_{j_2} \longrightarrow \ldots \longrightarrow M_{j_i})$.

For every simulation of a discrete transition of N in π' (i.e., not in the time-passing phase) of the form $M'_i \longrightarrow M'_{i+1}$ where $int_{\delta_i}(M_i) = M'_i$ there is a corresponding discrete transition in π of the form $M_i \longrightarrow M_{i+1}$ where $\delta_{i+1} = \delta_i$ and $int_{\delta_{i+1}}(M_{i+1}) = M'_{i+1}$. This follows directly from Def. 6.1.

Now we consider the i'-th time-passing phase for $1 \leq i' \leq m$. For every sequence of transitions $M'_i \stackrel{*}{\longrightarrow} M'_l$ in π' representing the i'-th time-passing phase there is a corresponding single time-transition in π of the form $M_i \longrightarrow_{T=\varepsilon_{i'}} M_l$, where $int_{\delta_i}(M_i) = M'_i$, $\delta_l = \delta_i + \varepsilon_{i'}$ and $int_{\delta_l}(M_l) = M'_l$. The delay $\varepsilon_{i'}$ is chosen as $\varepsilon_{i'} := 1 - f_{i'}$ where $f_{i'}$ is the fractional part of the age of those tokens in M_i which are mapped to $\beta_{i'}$ by int_{δ_i}. This ensures that in this timed transition the right tokens (of those originally present in M) reach (those mapped to $\beta_{i'}$) or exceed (those mapped to $\alpha_{i'}$) the next higher integer age. For the other tokens of M_i, which were newly created during π we can arbitrarily choose the values of their fractional parts, i.e., for every combination of these values there is a possible computation which implements it. Thus one can assume that these fractional parts are conveniently chosen such that they do (or don't) reach (or exceed) the next higher integer age, just as required by the condition $int_{\delta_l}(M_l) = M'_l$. Since $int_\delta(M) = M'$, only those tokens in M with a fractional part $> \delta$ were mapped to $p(k-)$ tokens in M' and only those tokens can reach (or exceed) age k in π'. Thus we can choose the $\varepsilon_{i'}$ in such a way that $\sum_{i'=1}^m \varepsilon_{i'} < 1 - \delta$. Thus we get $\lambda := (1 - \delta) - \sum_{i'=1}^m \varepsilon_{i'} > 0$. (The quantity λ will be used to determine the $\varepsilon_{i'}$ for $i' > m$.)

Now we consider the i'-th time-passing phase for $i' > m$. These are the time-passing phases in the infinite suffix π'' mentioned above. For them, it works

like the case above, except that the delays $\varepsilon_{i'}$ do no longer depend on the initial marking M, because $\alpha_{i'} + \beta_{i'} = \emptyset$ for $i' > m$. As shown above, none of the original tokens of M are involved in these i'-th time-passing phases for $i' > m$. The only tokens involved in this (reaching or exceeding the next higher integer age in this phase) are tokens newly generated in π (which have an age greater than δ and are mapped to $p(k-)$). The fractional parts of their ages can be chosen conveniently such that they reach or exceed the next higher integer age exactly as required for the correspondence with the computation π'. In particular, their ages can be chosen arbitrarily close to the next higher integer age such that the required delay $\varepsilon_{i'}$ can be made arbitrarily small. We choose $\varepsilon_{i'} := (\lambda/2) * 2^{-i'}$ for $i' > m$.

So we obtain $\Delta(\pi) = \sum_{i' \in \mathbb{N}} \varepsilon_{i'} = \sum_{1 \leq i' \leq m} \varepsilon_{i'} + \sum_{i' > m} \varepsilon_{i'} \leq \sum_{1 \leq i' \leq m} \varepsilon_{i'} + \lambda/2 < \sum_{1 \leq i' \leq m} \varepsilon_{i'} + \lambda = 1 - \delta$ and thus $\Delta(\pi) < 1 - \delta$ as required and therefore $M \in \mathit{ZENO}$. □

In the following, we give the algorithm to compute the set $ZENO$. The algorithm, in fact, computes a set of regions Z given by Definition 6.14 and we prove in Lemma 6.15 and Lemma 6.16 that $[\![Z]\!]^{\uparrow} = ZENO$.

Definition 6.14. Let N be a TPN with corresponding SD-TN N'.

$$Z := \bigcup_{M' \in INF'_{min}} \; \bigcup_{w_+ \in perm(M'^+)} \; \bigcap_{w_- \in perm(M'^-)} Pre^*(Reg(M', w_+, w_-))$$

6.2.1 Proof of Correctness

Now we show that *ZENO* is equal to the set $[\![Z]\!]^{\uparrow}$ and that it is effectively constructible.

Lemma 6.15. $[\![Z]\!]^{\uparrow} \subseteq \mathit{ZENO}$.

Proof. Let $M \in [\![Z]\!]^{\uparrow}$. Then there is an $M' \in INF'_{min}$ and an $w_+ \in perm(M'^+)$ such that $M \in \bigcap_{w_- \in perm(M'^-)} Pre^*(Reg(M', w_+, w_-))$. We choose $w_- \in perm(M'^-)$ according to Lemma 6.13 and so obtain $M \in Pre^*(Reg(M', w_+, w_-))$ and $[\![Reg(M', w_+, w_-)]\!]^{\uparrow} \subseteq \mathit{ZENO}$. Thus $M \in \mathit{ZENO}$, since $Pre^*(\mathit{ZENO}) = \mathit{ZENO}$. □

Lemma 6.16. $\mathit{ZENO} \subseteq [\![Z]\!]^{\uparrow}$.

Proof. Sine *ZENO* is upward-closed, it is enough to show that $ZENO_{min} \subseteq [\![Z]\!]^{\uparrow}$. Let $M \in ZENO_{min}$. By the definition of *zeno-marking*, there exists an infinite M-computation π and a finite number m such that $\Delta(\pi) < m$. Thus there exists a marking M_1 such that $M \xrightarrow{*} M_1$ and an infinite M_1-computation π_1 with $\Delta(\pi_1) < 1/2$. Since M_1 contains finitely many tokens and π_1 is infinite, there exists an infinite suffix of π_1 such that none of the original tokens of M_1 is used in this suffix. Thus there exist markings M_2, M_3 and M_4 and a finite computation π_2 such that

- $M_1 \xrightarrow{\pi_2} M_2 = M_3 + M_4$
- All tokens in M_3 were created during π_2.
- There is an infinite M_3-computation π_3 with $\Delta(\pi_2\pi_3) < 1/2$ (and thus $\Delta(\pi_3) < 1/2$).

Let $M_3' := int_{1/2}(M_3)$. Then we have $M_3' \in INF'$ by Lemma 6.8. From Definition 6.11, we have that there exist permutations $w_+ \in perm(M_3'^+)$ and $w_- \in perm(M_3'^-)$ such that $M_3 \in [\![Reg(M_3', w_+, w_-)]\!]^\uparrow$.

Since INF' is upward-closed, there exists a marking $M_3'' \in INF'_{min}$ such that $M_3'' \leq^m M_3'$ (consequently, $M_3''^+ \leq^m M_3'^+$, $M_3''^- \leq^m M_3'^-$ and $perm(M_3''^+) \subseteq perm(M_3'^+)$ and $perm(M_3''^-) \subseteq perm(M_3'^-)$). This means that there also exist permutations $w_+' \in perm(M_3''^+)$ with $w_+' \leq^w w_+$ and $w_-' \in perm(M_3''^-)$ with $w_-' \leq^w w_-$ and $[\![Reg(M_3'', w_+', w_-')]\!]^\uparrow \supseteq [\![Reg(M_3', w_+, w_-)]\!]^\uparrow$. It follows that $M_3 \in [\![Reg(M_3'', w_+', w_-')]\!]^\uparrow$.

Now consider all those tokens in M_3 which are mapped to $p(k-)$ tokens in M_3', i.e. those with a fractional part of their age which is $\geq^m 1/2$. These tokens were all created during π_2 and none of them had an integer age during π_2, because $\Delta(\pi_2) < 1/2$. Thus, the fractional parts of their ages are totally independent and any permutation is possible, i.e., for any permutation there is a computation which implements it. Therefore, for every $w_- \in perm(M_3'^-)$ there is a marking $M_3^{w_-}$ in N such that

- $M_1 \xrightarrow{*} M_3^{w_-} + M_4$
- $M_3^{w_-} \in [\![Reg(M_3', w_+, w_-)]\!]^\uparrow$.

Since $M_3'' \leq^m M_3'$ we get that for every $w_-' \in perm(M_3''^-)$ there is a marking $M_3^{w_-'}$ in N s.t

- $M_1 \xrightarrow{*} M_3^{w_-'} + M_4$
- $M_3^{w_-'} \in [\![Reg(M_3'', w_+', w_-')]\!]^\uparrow$.

It follows that

$$M \in [\![\bigcap_{w_-' \in perm(M_3''^-)} Pre^*(Reg(M_3'', w_+', w_-'))]\!]^\uparrow$$

with $M_3'' \in INF'_{min}$ and $w_+' \in perm(M_3''^+)$, and thus $M \in [\![Z]\!]^\uparrow$. □

Notice that union and intersection of (sets of) regions are easily computable as finite sets of regions.

6.3 Step 2: Computing INF'_{min}

Computability of the set *ZENO* (in the last section) requires that the minimal elements of any upward closed set is effectively constructible. In this section, we show for any SD-TN, how to construct the minimal elements INF_{min} for INF. Then INF'_{min} is obtained by just restricting INF_{min} to standard markings.

For constructing INF_{min}, we use a result by Valk and Jantzen [130]. Our algorithm depends on the concepts of semi-linear languages, Presburger Arithmetic and Parikh's Theorem, described in the following. Recall that we use $(v_1, \ldots, v_n)$ or $\vec{v}$ interchangeably to denote a vector of size n (Chapter 2).

6.3.1 Semilinear Sets

First we define *linear* sets.

Definition 6.17. A set $L \subseteq \mathbb{N}^n$ is said to be *linear*, written as $L = L(\vec{v_0}; \vec{u_1}, \ldots, \vec{u_r})$ where $\vec{v_0}, \vec{u_1}, \ldots, \vec{u_r} \in \mathbb{N}^n$. We let L denote the set of all elements $\vec{v}$ in $\mathbb{N}^n$ of the form $\vec{v_0} + \Sigma_1^m k_i \vec{v_i}$ with $\vec{v_1}, \ldots, \vec{v_m} \in \{\vec{u_1}, \ldots, \vec{u_r}\}$ and $k_1, \ldots, k_m \in \mathbb{N}$.

Example 6.18. $L((0,0);(0,2),(2,0)) = \{(0,0) + k_1(0,2) + k_2(2,0) \mid k_1, k_2 \in \mathbb{N}\}$ is linear. □

Definition 6.19. A subset of $\mathbb{N}^n$ is said to be *semilinear* if it is a finite union of linear sets.

Theorem 6.20. [73] Semilinear sets are closed under union, intersection, complementation and first order quantification.

Next we define the Parikh mapping φ. Given a finite alphabet $\Sigma = \{a_1, \ldots, a_n\}$, φ is a function from Σ^* to $\mathbb{N}^n$, defined by $\varphi(w) = (\#_{a_1}(w), \ldots, \#_{a_n}(w))$, where $\#_{a_i}(w)$ is the number of occurrences of a_i in w. Thus $\varphi(\epsilon) = 0^n$ and $\varphi(w_1 \bullet \ldots \bullet w_m) = \Sigma_1^m \varphi(w_i)$. Finally, given a language $L \subseteq \Sigma^*$, $\varphi(L) = \{\varphi(w) \mid w \in L\}$. If $\varphi(L)$ is semilinear for a language L, then L is called a semilinear language.

Theorem 6.21. (Parikh's Theorem) [113] $\varphi(L)$ is effectively semilinear for each context-free language L.

As a special case, Theorem 6.21 holds for regular languages, since every regular language is a context-free language [113].

Example 6.22. Let $\Sigma = \{a_1, a_2, a_3\}$. Then $\varphi(a_1a_2a_1a_3a_2a_3a_3) = (2,2,3) \in L((2,0,1);(0,1,1))$. Also, $\varphi(ab^*ca) = \{(2,0,1) + n * (0,1,0) \mid n \in \mathbb{N}\}$. □

6.3.2 Presburger Arithmetic

Presburger arithmetic is the first-order theory of the integers with addition and the ordering relation over $\mathbf{Z}$, also denoted as $(\mathbf{Z}, \leq, +)$. Formally, Presburger

arithmetic is the first-order theory over *atomic formulas* of the form

$$\Sigma_{1\leq i\leq n}\ a_i x_i \quad \sim c$$

where a_i, c are integer constants, x_i-s are variables ranging over integers and $\sim \in \{=, \neq, <, \leq, >, \geq\}$. This means that a *Presburger formula* ρ is either an *atomic formula*, or it is constructed from the Presburger formulas ρ_1, ρ_2 recursively as follows:

$$\rho := \neg\rho_1 \ \mid\ \rho_1 \wedge \rho_2 \ \mid\ \rho_1 \vee \rho_2 \ \mid\ \exists x_i.\rho_1(x_1, \ldots, x_n)$$

where $\rho_1(x_1, \ldots, x_n)$ is a Presburger formula over free variables $x_1, \ldots, x_n$ and $1 \leq i \leq n$.

Theorem 6.23. (Presburger) [26] Presburger arithmetic is decidable.

As a shorthand notation, we work with $\mathbf{Z}_\omega = \mathbf{Z} \cup \{\omega\}$ instead of the usual $\mathbf{Z}$. This is not a problem, since Presburger-arithmetic on $\mathbf{Z}_\omega$ can easily be reduced to Presburger-arithmetic on $\mathbf{Z}$ as follows. For every variable x one adds an extra variable x' which is used in such a way that the original state $x = k < \omega$ is represented by $(x, x') = (k, 0)$ and the original state $x = \omega$ is represented by $(x, x') = (0, 1)$. It is easy to encode the usual properties like $\omega + k = \omega - k = \omega + \omega = \omega$.

Theorem 6.24. [74] A subset of $\mathbb{N}^n$ is semilinear iff it is definable in Presburger Arithmetic.

6.3.3 Result from Valk and Jantzen

We recall a result from [130].

Theorem 6.25. Given an upward-closed set $V \subseteq \mathbb{N}^k$, the minor [2] set V_{min} of V is effectively computable iff for any vector $\vec{u} \in \mathbb{N}^k_\omega$ the predicate $\vec{u}\downarrow \cap V \neq \emptyset$ is decidable.

Proof. Assume that the minimal elements of V, denoted by V_{min} can be computed. Then $V = V_{min} + \mathbb{N}^k$ gives a semilinear representation of V. Since $\vec{u}\downarrow$ is also a semilinear set, a representation of which can be found effectively, the predicate $\vec{u}\downarrow \cap V \neq \emptyset$ is decidable.

On the other hand, assume that the predicate is decidable for any vector $\vec{u} \in \mathbb{N}^k_\omega$. The following method then effectively constructs V_{min}. First start with a singleton set of vectors $W_0 := \{(\omega, \ldots, \omega)\}$ with k ω-s. We let W_i to denote the set of vectors we need to consider in the i-th iteration and $\mathcal{V}_i$ to denote the set of minimal elements found for V_{min} in the i-th iteration. Initially $\mathcal{V}_0 := \emptyset$. We let $pred_V(\vec{u})$ denote $\vec{u}\downarrow \cap V \neq \emptyset$. We repeat the following.

Stage 1: In this stage, we perform the following two loops sequentially.

[2] Recall from Chapter 2 that V_{min} denotes the *minor set* of V.

Loop 1 We choose some vector $\vec{u}$ from W_i and compute $pred_V(\vec{u})$. If the value is false, then we remove u from W_i. We get out of this loop if $pred_V(\vec{u})$ is true or $W_i = \emptyset$.

After exiting from the above loop if $W_i = \emptyset$, then $V_{min} = \mathcal{V}_i$ and we stop the algorithm. Otherwise, $pred_V(\vec{u})$ is true; $\vec{u}\downarrow$ contains at least one element of V_{min} and one such element will be found in the next loop.

Loop 2 We repeat the following until all co-ordinates of $\vec{u}$ are considered. Choose some co-ordinate $\vec{u}(i)$ of $\vec{u}$ which has not yet been considered and replace $\vec{u}(i)$ in $\vec{u}$ by the smallest natural number such that $pred_V(\vec{u})$ for this new vector is still true.

The above computed new vector will then be an element of V_{min}. So, we update $\mathcal{V}_{i+1} = \mathcal{V}_i \cup \{\vec{u}\}$.

Stage 2: Let the new found vector be $\vec{u} = (z_1, \ldots, z_k)$. In this stage, we try to find other vectors in V_{min}. We let $W_i' = \{(z_1', \ldots, z_k') \in \mathbb{N}_\omega^k \mid \exists j : 1 \leq j \leq k : z_j' = z_j - 1 \text{ and } z_m' = \omega \text{ for all } m \neq j\}$. We update $W_{i+1} := min(W_i, W_i')$ where $min(W, W') = \left\{min(\vec{u}, \vec{u'}) \mid \vec{u} \in W, \vec{u'} \in W'\right\}$ and min of two vectors are evaluated component-wise. Then we increment the iterator by $i := i + 1$ and go back to Loop 1. □

6.3.4 Computing INF_{min} for a Petri net

We use the result of Valk and Jantzen to compute INF_{min} for a Petri net. To apply this algorithm, we require the computability of the predicate $M\downarrow \cap\ INF \neq \emptyset$ ($pred_{INF}(\vec{M})$) for any ω-marking M. The decidability of this predicate is shown in the following lemma (also shown in [34]).

Lemma 6.26. [34] Given a Petri net N with k places and an ω-marking (introduced in Chapter 4) $M_0 \in \mathbb{N}_\omega^k$, it is decidable if $M_0\downarrow \cap$ INF $\neq \emptyset$.

Proof. We show that if $M_0\downarrow \cap\, INF \neq \emptyset$ then this condition will be detected by the following construction. Furthermore, we prove that the construction does not yield any false positives.

Construction:

Let $\mathcal{C} = (G, \longrightarrow)$ with $G \subseteq \mathbb{N}_\omega^k$ be the coverability graph of N from initial marking M_0, which is computable [91]. Now we give an outline of the algorithm to check whether $M_0\downarrow \cap\, INF \neq \emptyset$. For each ω-marking M in the coverability graph, we compute the automaton $\mathbb{A}_M$, compute the Parikh image $L(\mathbb{A}_M)$ and from it, find a Presburger formula ρ_M characterizing loops with positive effects. We say that $M_0\downarrow \cap$ INF$\neq \emptyset$ is true if and only if there is a solution to such a Presburger formula ρ_M.

First, for every ω-marking M in the coverability graph, we compute a finite-state automaton $\mathbb{A}_M$ on finite sequences such that its labelled transition graph is the largest strongly connected subgraph containing M, and its initial state

and only final state is M. Let l be the number of edges in $\mathbb{A}_M$. We label every arc in $\mathbb{A}_M$ with a unique symbol Λ_i for $i : 1 \leq i \leq l$. To every Λ_i, we assign an *effect-vector* $\vec{\zeta_i} \in \mathbb{Z}^k$ that describes the effect of the transition that was fired in the step from one node to the other.

The aim is to find a cyclic path in $\mathbb{A}_M$ from a marking M back to M where the sum of all the effect-vectors of all traversed arcs is $\geq \vec{0}$. Here, the effect-vector of an arc that is traversed j times is counted j times. Such a cyclic path with positive overall effect corresponds to a possible infinite computation of the system N. Given an automaton $\mathbb{A}_M$ with M as its initial and the only final state, every word in $L(\mathbb{A}_M)$ corresponds to a cyclic path from M to M. For any word w, let $|w|_{\Lambda_i}$ be the number of occurrences of Λ_i in w. The question now is if there is a word $w \in L(\mathbb{A}_M)$ such that

$$\Sigma_{1 \leq i \leq l} |w|_{\Lambda_i} \vec{\zeta_i} \geq \vec{0}.$$

Such words characterize loops starting and ending in the same marking of a net. We show how to answer the above question in the following.

- First we compute the Parikh image of $L(\mathbb{A}_M)$, i.e., the set $\{(|w|_{\Lambda_1}, \ldots, |w|_{\Lambda_l}) \mid w \in L(\mathbb{A}_M)\}$. This set is effectively semilinear by Parikh's Theorem.
- By Theorem 6.24, we compute a Presburger formula $\rho(x_1, \ldots, x_l)$ from the semilinear set computed above. The variables $x_1, \ldots, x_l$ count the number of times each edge Λ_i appears in a word $w \in L(\mathbb{A}_M)$.
- Finally, to decide if $\Sigma_{1 \leq i \leq l} |w|_{\Lambda_i} \vec{\zeta_i} \geq \vec{0}$, we check the satisfiability of $\rho_{\mathbb{A}} = \rho(x_1, \ldots, x_l) \wedge \Sigma_{1 \leq i \leq l} x_i \vec{\zeta_i} \geq \vec{0}$, which is again a Presburger formula. By Theorem 6.23, we can decide whether this formula is satisfiable.

We check this condition for every $\mathbb{A}_M$ and we say that $M_0 \downarrow \cap \text{INF} \neq \emptyset$ is true if and only if the condition for some automaton $\mathbb{A}_M$ holds.

Correctness: Now we show the correctness of the above construction. If $M_0 \downarrow \cap \mathit{INF} \neq \emptyset$ then there exists a marking $M \in \mathbb{N}^k$ with $M \leq^m M_0$ and $M \in \mathit{INF}$. Thus there exists an infinite M-computation π. By Dickson's lemma, there are markings M', M'' and a sequence of transitions Seq such that $M \xrightarrow{*} M' \longrightarrow_{\mathit{Seq}} M''$ and $M' \leq^m M''$. Thus the total effect of Seq is non-negative.

Now, from [91], we know that there is a ω-marking $M_{\mathcal{C}}$ in the coverability graph such that $M'' \leq^m M_{\mathcal{C}}$. Due to monotonicity of the transition relation, there is a path labelled with transitions in Seq and which leads us from $M_{\mathcal{C}}$ to a ω-marking larger than $M_{\mathcal{C}}$. Repeating this process from the larger node will finally lead us to a node which is largest of all ω-markings larger than $M_{\mathcal{C}}$. We will reach such a node $M_{\mathcal{C}}^{max}$, since the graph is finite. This means that we can fire transitions in Seq from $M_{\mathcal{C}}^{max}$ and we get back to $M_{\mathcal{C}}^{max}$ itself (since there are no ω-marking larger than $M_{\mathcal{C}}^{max}$ in $\mathcal{C}$ and by monotonicity Seq leads to a larger or equal node in $\mathcal{C}$). So, $M_{\mathcal{C}} \longrightarrow_{\mathit{Seq}} M_{\mathcal{C}}$, i.e., there are ω-markings $M_1, \ldots, M_n$ such that $M_{\mathcal{C}} \longrightarrow M_1 \longrightarrow \ldots \longrightarrow M_n = M_{\mathcal{C}}$ with effect-vectors

$\vec{\zeta_1}, \ldots, \vec{\zeta_n}$ such that $\Sigma_{1 \leq i \leq n} \quad \vec{\zeta_i} \geq \vec{0}$. This is the condition checked in our construction.

On the other hand, suppose that there is a word $w \in L(\mathbb{A}_{M_\mathcal{C}})$ for some ω-marking $M_\mathcal{C}$ in the coverability graph such that $\Sigma_{1 \leq i \leq l} |w|_{\Lambda_i} \vec{\zeta_i} \geq \vec{0}$. This means that there is a ω-marking $M_\mathcal{C}$ from which there is a path (through a sequence Seq of transitions) back to itself with non-negative effect. From [91], we know that there are markings M' reachable from M_0 which agree with $M_\mathcal{C}$ in its finite coordinates, and can be made arbitrarily large in the coordinates equal to ω. We can choose one such marking M' such that it contains enough tokens in its ω-coordinates to be able to perform one iteration of Seq. Now, Seq has a non-negative effect. This means that one can repeatedly execute Seq starting from M'. The reachability of such an M' from M_0 and a non-negative loop from M' corresponds to an infinite M_0-computation. This means that $M_0 \downarrow \cap\, INF \neq \emptyset$.

□

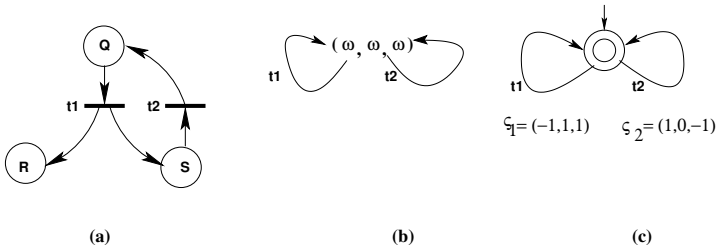

Figure 6.5: (a). A small Petri net, (b). Coverability graph for this net from (ω, ω, ω). (c) Automaton $A_{(\omega,\omega,\omega)}$.

Example 6.27. Consider the Petri net in Figure 6.5(a) and the coverability graph (Figure 6.5(b)) of the above Petri net from a ω-marking $M = (\omega, \omega, \omega)$ [3] where $M(Q) = M(R) = M(S) = \omega$. We show that $M \downarrow \cap\, INF \neq \emptyset$. The automaton produced for the single node in the coverability graph is shown in Figure 6.5(c). Notice that $\Lambda_1 = t_1$ and $\Lambda_2 = t_2$. Also, the effect-vectors $\vec{\zeta_1}$ and $\vec{\zeta_2}$ show the effect of firing t_1 and t_2 respectively. Notice that $L(A_{(\omega,\omega,\omega)}) = \{w \mid w \in \{t_1, t_2\}^*\}$. This means that $\varphi(L(A_{(\omega,\omega,\omega)})) = L((0,0); (1,0), (0,1))$. Finally, we compute a Presburger formula $\rho(x_1, x_2)$ for the above linear set and from it, construct the formula $\rho(x_1, x_2) \wedge x_1\vec{\zeta_1} + x_2\vec{\zeta_2} \geq \vec{0}$. One of the solutions of this formula is given by $x_1 = x_2 = k$ for any natural number k. This means $M \downarrow \cap\, INF \neq \emptyset$. □

Example 6.28. In the above, we show an example for computing $pred_{INF}(M)$ for an ω-marking M. Now we show how to compute INF_{min} for the same

[3] Markings of a Petri net are written as multisets over places and vectors over the set of natural numbers interchangeably.

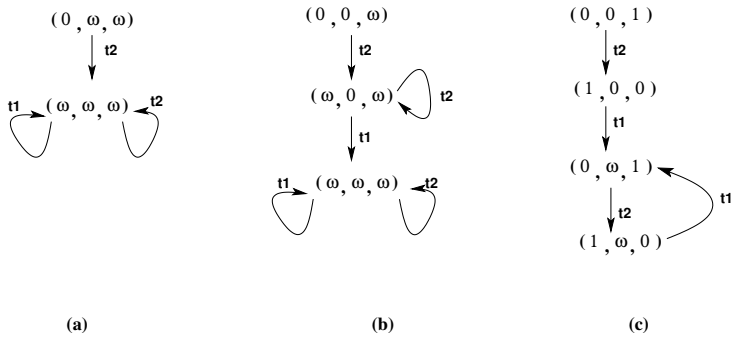

Figure 6.6: (a). Coverability graph from $(0, \omega, \omega)$ (b). Coverability graph from $(0, 0, \omega)$. (c) Coverability graph from $(0, 0, 1)$.

Petri net using Valk and Jantzen's algorithm. We start with a single marking (ω, ω, ω). Immediately, we get out of Loop 1, since $pred_{INF}((\omega, \omega, \omega))$ is true (as shown in Example 6.27). In Loop 2, one finds a minimal element in INF_{min}. This is done by first reducing the first co-ordinate for Q in (ω, ω, ω) to 0. In Figure 6.6(a), we show the coverability graph from $(0, \omega, \omega)$. $pred_{INF}((0, \omega, \omega))$ is true, since we reach a node (ω, ω, ω) in the coverability graph from $(0, \omega, \omega)$ and $pred_{INF}((\omega, \omega, \omega))$ is already shown to be true in the previous example. Then we replace the ω in place R to 0 and compute the coverability graph for $(0, 0, \omega)$ in Figure 6.6(b). $pred_{INF}((0, 0, \omega))$ is true again by the same reasoning. Notice that $pred_{INF}((0, 0, 0))$ is false. So, finally we show the coverability graph from marking $(0, 0, 1)$ in Figure 6.6(c) and $pred_{INF}((0, 0, 1))$ is true. Thus $(0, 0, 1)$ is included in INF_{min}.

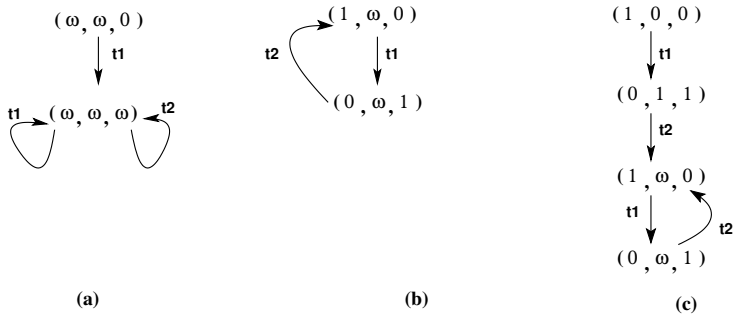

Figure 6.7: (a). Coverability graph from $(\omega, \omega, 0)$ (b). Coverability graph from $(1, \omega, 0)$. (c) Coverability graph from $(1, 0, 0)$.

In Stage 2, we have $W'_0 = \{(\omega, \omega, 0)\}$ and $W_1 = min((\omega, \omega, \omega), (\omega, \omega, 0)) = \{(\omega, \omega, 0)\}$.

Now we go to Loop 1 again. From Figure 6.7(a), it is evident that $pred_{INF}((\omega, \omega, 0))$ is true. Now, we again perform Loop 2. We find that

$pred_{INF}((0,\omega,0))$ is false, but $pred_{INF}((1,\omega,0))$ is true (the coverability graph from $(1,\omega,0)$ is shown in Figure 6.7(b)). Finally we show the coverability from $(1,0,0)$ in Figure 6.7(c) and it follows that $pred_{INF}((1,0,0))$ is true. Thus $(1,0,0)$ is another member of INF_{min}.

In Stage 2, we have $W_1' = (0,\omega,\omega)$ and $W_2 = min((0,\omega,\omega),(\omega,\omega,0)) = (0,\omega,0)$. Now $pred_{INF}((0,\omega,0))$ is false and $W_2 = \emptyset$ and we are finished. Thus $INF_{min} = \{(0,0,1),(1,0,0)\}$. □

6.3.5 Computing INF_{min} for SD-TNs

To compute INF_{min} for SD-TNs, we will again use Valk and Jantzen's Theorem 6.25. This algorithm requires decision of the predicate $M_0 \downarrow \cap \text{INF} \neq \emptyset$ for any given $M_0 \in \mathbb{N}_\omega^k$ for an SD-TN. First we construct a coverability graph for a given SD-TN.

Coverability graph for SD-TN

For any SD-TN N with initial marking M_0, the coverability graph can be effectively constructed. We use ω-markings from $\mathbb{N}_\omega^k$ (where k is the number of places). One proceeds from M_0 similarly as in the Karp-Miller construction [91], except for the transfer arc. The detection of loop is done slightly differently in the two cases (with and without transfer arc).

1. *Loop without transfer arc:* If one encounters the case $M_1 \longrightarrow_{Seq} M_2$ with

   - $M_1 <^m M_2$,
   - *Seq* is a sequence of transitions of N such that the transfer arc was not used in *Seq*,

   then we replace M_2 by $M_2 + \omega(M_2 - M_1)$ as in the case of Petri nets. Notice that $\omega M = M'$ such that $M'(p) = \omega$ for all place p with $M(p) > 0$. Obviously, *Seq* can be repeated arbitrarily often to yield an arbitrarily high numbers of tokens on the places where M_2 is strictly larger than M_1.

2. *Loop containing transfer arc:* Let M_1 and M_2 be two markings *reached just after transfers*, i.e., $\longrightarrow_{Trans} M_1 \longrightarrow_{Seq} M_1' \longrightarrow_{Trans} M_2$ (where *Seq* may contain other transfers). If $M_1 <^m M_2$ then we replace M_2 by $M_2 + \omega(M_2 - M_1)$. The sequence of transitions $\longrightarrow_{Seq}\longrightarrow_{Trans}$ can be repeated arbitrarily often to yield arbitrarily high numbers of tokens on the places where M_2 is strictly bigger than M_1. This is possible, because in SD-TN the set of places which are sources of transfers and the set of places which are targets of transfers are disjoint by Def. 6.1. Thus the transfers in $\longrightarrow_{Seq}\longrightarrow_{Trans}$ do not negatively affect those places p where $M_1(p) <^m M_2(p)$. This point does not carry over to general transfer nets. In particular, all transfer-target places, once marked by ω in this construction, will stay ω in the future. Furthermore, all transfer source places are empty after the transfer, since all transfers are simultaneous.

3. If one reaches an ω-marking encountered before, then one creates a loop.

It is easy to show that the so-generated coverability graph is finite. Suppose that there is an infinite sequence $M_0, M_1, \ldots$ of nodes in the coverability graph. Now, there are two cases.

- In this infinite sequence, there is only a finite number of occurrence of transfer transition $Trans$. Suppose M_r was the last marking produced by transfer transition. Consider the sequence $M_{r+1}, M_{r+2}, \ldots$. This sequence is still infinite. By Dickson's lemma, in any such infinite sequence of markings of SD-TN, there are always two markings such that $i < j$ and $M_i \leq^m M_j$, where $i, j \geq r + 1$.
- There is an infinite number of markings produced by the transfer transition $Trans$, which appear in the sequence $M_0, M_1, \ldots$. We take the subsequence $M'_0, M'_1, \ldots$ of $M_0, M_1, \ldots$ such that each marking M'_i for $i \geq 0$ is a marking produced by the transfer transition. Since there are infinitely many transfer transitions in sequence $M_0, M_1, \ldots$, the sequence $M'_0, M'_1, \ldots$ is also infinite. Now, like the previous case, we will always find two markings in the sequence $M'_0, M'_1, \ldots$ by Dickson's lemma.

This ensures the finiteness of the coverability graph for any SD-TN.

Note that the above construction implies that place-boundedness is decidable for simultaneous-disjoint transfer nets, while it is undecidable for general transfer nets [56, 104].

Remark: Notice that if a place p is a source of a transfer transition, then $M_1(p) <^m M_2(p)$ does not in general imply that p will contain arbitrarily large number of tokens. This is due to the fact that the loop may contain a transfer transition which will remove all tokens from p.

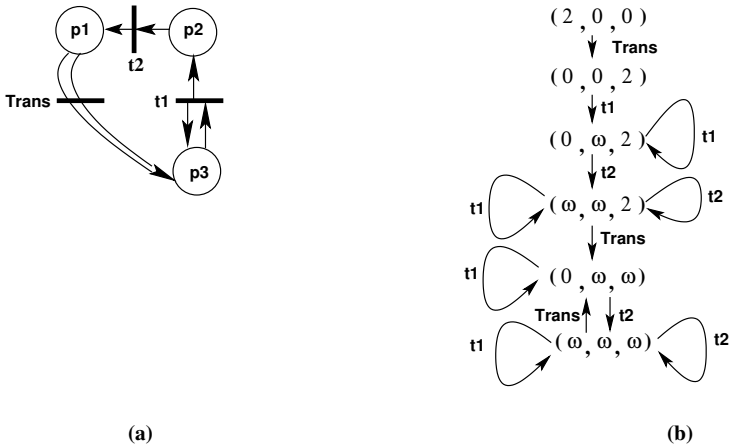

Figure 6.8: (a) A small SD-TN. (b) Coverability graph $\mathcal{C}$ for this net.

Example 6.29. Consider a small SD-TN shown in Figure 6.8(a). In Figure 6.8(b), we show the coverability graph $\mathcal{C}$ from a marking $M = (2, 0, 0)$ of SD-TN where $M(p_1) = 2, M(p_2) = 0$ and $M(p_3) = 0$. We omit the transfer arcs in the coverability graph if the source place of transfer does not contain a token. Notice that $Trans = ((p_1, p_3), \emptyset, \emptyset)$ and $(0, 0, 2)$ and $(0, \omega, \omega)$ are the only ω-AT-markings in $\mathcal{C}$. □

Computing $pred_{INF}$ for SD-TNs

In Lemma 6.30, we use the following definitions.

By Def. 6.1 of SD-TN, the source places and target places of transfers are disjoint and thus after a simultaneous transfer all source places are empty. We call a marking an 'after transfer marking' (AT-marking) if it is reached just after firing *Trans*. We represent markings as vectors in $\mathbb{N}^k$ of the form (transfer source places, other places). So AT-markings have the form $(\overrightarrow{0}, \overrightarrow{v})$ with $\overrightarrow{0} \in \mathbb{N}^{k'}$ and $\overrightarrow{v} \in \mathbb{N}^{k''}$ with $k = k' + k''$ where k' is the number of transfer source places. The corresponding markings in the coverability graph $\mathcal{C}$ are called ω-AT-markings and have the form $(\overrightarrow{0}, \overrightarrow{v})$ with $\overrightarrow{v} \in \mathbb{N}_\omega^{k''}$.

Lemma 6.30. Given an SD-TN N with k places and an ω-marking $M_0 \in \mathbb{N}_\omega^k$, it is decidable if $M_0{\downarrow} \cap INF \neq \emptyset$.

Proof. First we give an algorithm to detect the non-emptiness of the set $M_0{\downarrow} \cap INF$. Let $\mathcal{C} = (G, \longrightarrow)$ with $G \subseteq \mathbb{N}_\omega^k$ be the coverability graph of N from initial marking M_0. An infinite computation π from a marking M in $M_0{\downarrow}$ is detected as follows. There are two cases. Either there are finitely many or infinitely many transfers in such an infinite computation.

- In the first case, the transfer transition $Trans$ is used only finitely often and π has an infinite suffix π' which starts at some marking M' and only normal Petri net transitions are used in π'. Since $M \xrightarrow{*} M'$, there is a node $M_{\mathcal{C}}$ in $\mathcal{C}$ such that $M' \leq^m M_{\mathcal{C}}$. To find out whether there is a positive effect of such cycles consisting of ordinary Petri net transitions, we let N' be the ordinary Petri net obtained from N by removing the transfer transition *Trans*. So π' is an infinite M'-computation of N'. Let $INF_{N'} \subseteq \mathbb{N}^k$ be the (upward-closed) set of markings from which infinite computations of N' start. So we have $M_{\mathcal{C}}{\downarrow} \cap INF_{N'} \neq \emptyset$. In fact, we consider each ω-marking $M_{\mathcal{C}} \in G$ and detect the presence of an infinite computation with just ordinary Petri net transitions if the condition (Cond1)

  $$\exists M_{\mathcal{C}} \in G. M_{\mathcal{C}}{\downarrow} \cap INF_{N'} \neq \emptyset$$

  holds. This is a problem about ordinary Petri nets and has been shown to be decidable (Lemma 6.26).

- In the second case, the transfer transition *Trans* is used infinitely often in π. Recall that in Lemma 6.26, we construct automata from the coverability graph, for each of its nodes and associate an effect-vector with

each edge of such an automaton. In this case, the presence of transfer transitions in the cycles of SD-TNs does not let us follow such a procedure directly. This is due to the fact that the effect of the transfer depends on the amount of tokens in the source places of the transfer and that is not a constant number.

In this case, first we compute the effect-vectors between two ω-AT-markings $\mathcal{M}, \mathcal{M}'$ in the coverability graph such that $\mathcal{M}'$ is reachable from $\mathcal{M}$. For any pair of ω-AT-markings $\mathcal{M}, \mathcal{M}' \in G$ we can effectively construct a semilinear set $\mathit{Effect}(\mathcal{M}, \mathcal{M}') \subseteq \mathbf{Z}^k$ which represents all possible effects of sequence of transitions of the form $\mathit{Seq}.\mathit{Trans}$ with $\mathcal{M} \longrightarrow_{\mathit{Seq}} \longrightarrow_{\mathit{Trans}} \mathcal{M}'$ where Seq is a sequence of transitions which does not contain Trans. This is done as follows. First, we compute the semilinear sets $\mathit{Effect}'(\mathcal{M}, X) \subseteq \mathbf{Z}^k$ for all $X \in G$ such that $X \longrightarrow_{\mathit{Trans}} \mathcal{M}'$ in the coverability graph $\mathcal{C}$ and $\mathcal{M} \xrightarrow{*} X$ without using Trans. The sets $\mathit{Effect}'(\mathcal{M}, X)$ are semilinear and effectively constructible, by computability of Presburger-arithmetic and its equivalence with semilinear languages (Theorem 6.24). This is due to the fact that $\mathcal{C}$ is a finite graph whose arcs are labelled with constant vectors in $\mathbf{Z}^k$ and the Parikh-image of regular languages is effectively semilinear. This means that one can consider $\mathcal{M}$ as the initial- and X as the final state of a finite automaton $\mathbb{A}$. Each edge in $\mathbb{A}$ is labelled by a unique symbol Λ and there is an associated effect-vector ζ for the effect of the transition by that edge. Let $\rho(x_1, \ldots, x_l)$ be the Presburger formula for the Parikh-image of $L(\mathbb{A})$ where l is the number of edges in the coverability graph. A valuation of the variable x_i for $i : 1 \leq i \leq l$ gives how many times the symbol Λ_i appears in a word in $L(\mathbb{A})$. Given k as the number of places in SD-TN, we have $\mathit{Effect}'(\mathcal{M}, X)$ given by a Presburger formula

$$\rho_X(y_1, \ldots, y_k) = \exists x_1 \ldots, x_l . \rho(x_1, \ldots, x_l) \wedge \bigwedge_{1 \leq i \leq k} y_i = \Sigma_{1 \leq j \leq l} x_j \zeta_j(i)$$

Secondly, we obtain $\mathit{Effect}(\mathcal{M}, \mathcal{M}')$ as a Presburger formula by introducing the effect of transfers ($Trans = (I, O, ST)$) as follows. Consider the set $\mathbf{X}$ containing ω-markings X such that $\mathcal{M} \xrightarrow{*} X \longrightarrow_{Trans} \mathcal{M}'$. For each $X \in \mathbf{X}$, we compute a Presburger formula

$$\rho''_X(z_1, \ldots, z_k) = \exists y_1, \ldots, y_k . \left(\rho_X(y_1, \ldots, y_k) \wedge \rho'_X(y_1, \ldots, y_k, z_1, \ldots, z_k)\right)$$

where $\rho'_X(y_1, \ldots, y_k, z_1, \ldots, z_k)$ is a conjunction of the following formulas.

- $\forall j, j' : (p_{j'}, p_j) \in ST . z_j = y_j + y_{j'} \ \wedge \ z_{j'} = 0$. Here, ST is from Def. 6.1. This corresponds to a transfer from place $p_{j'}$ to place p_j whenever $(p_{j'}, p_j) \in ST$.
- $\forall p_j \in I . z_j = y_j - 1 \ \wedge \ \forall p_j \in O . z_j = y_j + 1$. This corresponds to Petri net part of transfers, since I contains places from which there is an input arc to the transfer transition and O contains places from which there is an output arc to the transfer transitions.

– $\forall j.p_j \notin ST.p_j \notin I \cup O.z_j = y_j$. Here $p_j \notin ST$ is used to mean that there are no pairs $(p,q) \in ST$, such that $p_j = p$ or $p_j = q$. This means that there is no change in the number of tokens at the other places.

Finally the effect $\mathit{Effect}(\mathcal{M},\mathcal{M}') = \bigvee_{X\in\mathbf{X}} \rho''_X(z_1,\ldots,z_k)$. By Theorem 6.23, we can compute a semilinear set from the Presburger formula given above for $\mathit{Effect}(\mathcal{M},\mathcal{M}')$.

Now we construct a new finite graph $\mathcal{C}' = (G', \longrightarrow)$ as follows. $G' \subseteq G$ is the set of ω-AT-markings in G. For $\mathcal{M},\mathcal{M}' \in G'$ we have $\mathcal{M} \longrightarrow \mathcal{M}'$ in $\mathcal{C}'$ iff $\mathcal{M} \longrightarrow_{\mathit{Seq''}} \longrightarrow_{\mathit{Trans}} \mathcal{M}'$ in $\mathcal{C}$ where *Seq"* does not contain *Trans*. The arc between $\mathcal{M}$ and $\mathcal{M}'$ is labelled with (a symbolic Presburger-arithmetic representation of the semilinear set) $\mathit{Effect}(\mathcal{M},\mathcal{M}')$.

We check the following condition (Cond2).

$$\exists n \in \mathbb{N}.\, \mathcal{M}_0,\ldots,\mathcal{M}_n \in G'.\, \mathcal{M}_0 \longrightarrow \mathcal{M}_1 \longrightarrow \ldots \longrightarrow \mathcal{M}_n = \mathcal{M}_0.$$

$$\exists \overrightarrow{v_i} \in \mathit{Effect}(\mathcal{M}_i,\mathcal{M}_{i+1}).\, \sum_{i=0}^{n-1} \overrightarrow{v_i} \geq^m \overrightarrow{0}$$

Note that the $\mathcal{M}_i$ above do not need to be disjoint.

Now we show how to check the condition (Cond2). We transform the graph $\mathcal{C}'$, whose arcs are labelled with semilinear sets $\mathit{Effect}(\mathcal{M},\mathcal{M}')$ into a new equivalent graph $\mathcal{C}''$ whose arcs are labelled with constant vectors. Since $\mathit{Effect}(\mathcal{M},\mathcal{M}')$ is effectively semilinear, it can be represented as a finite union of linear sets of the form $L(\vec{u_i};\vec{w_i^1},\ldots,\vec{w_i^{n_i}})$ where $i : 1 \leq i \leq m$ and $m \geq 1$. $\mathcal{C}''$ contains the nodes of $\mathcal{C}'$ and some additional nodes:

– if there is an edge between two nodes $\mathcal{M},\mathcal{M}'$ labelled by $\mathit{Effect}(\mathcal{M},\mathcal{M}')$ (of the above form) in $\mathcal{C}'$, we add new nodes $\mathcal{M}'_i$ for $i : 1 \leq i \leq m$ in $\mathcal{C}''$.

Also, for any pair of nodes $\mathcal{M},\mathcal{M}'$ in C', labelled by $\bigcup_{1\leq i\leq m} L(\vec{u_i};\vec{w_i^1},\ldots,\vec{w_i^{n_i}})$, we have the following arcs in $\mathcal{C}''$. For each $i : 1 \leq i \leq m$, we have

– an edge from $\mathcal{M}$ to $\mathcal{M}'_i$, labelled by $\vec{u_i}$.

– edges from $\mathcal{M}'_i$ to $\mathcal{M}'_i$, labelled by $\vec{w_i^j}$ for $j : 1 \leq j \leq n_i$.

– an edge from $\mathcal{M}'_i$ to $\mathcal{M}'$, labelled by $\vec{0}$.

Let $\mathcal{C}'' = (G'', \longrightarrow)$ be the graph obtained in this way. We get immediately that the following condition (Cond3) holds for C'' iff (Cond2) holds for C'.
$\exists n \in \mathbb{N}.\, \mathcal{M}_0,\ldots,\mathcal{M}_n \in G''.$

$$(\mathcal{M}_0 \xrightarrow{v_0} \mathcal{M}_1 \xrightarrow{v_1} \ldots \xrightarrow{v_{n-1}} \mathcal{M}_n = \mathcal{M}_0) \;\wedge\; \sum_{i=0}^{n-1} \overrightarrow{v_i} \geq^m \overrightarrow{0}$$

The condition (Cond3) is decidable, since C'' is a finite graph and by Parikh's theorem [113] the Parikh-image of regular languages is effectively semilinear. (Just interpret C'' as a finite automaton and try out any $\mathcal{M}_0 \in G''$ as initial and final state.) Then we proceed as in Lemma 6.26. Thus (Cond2) is decidable.

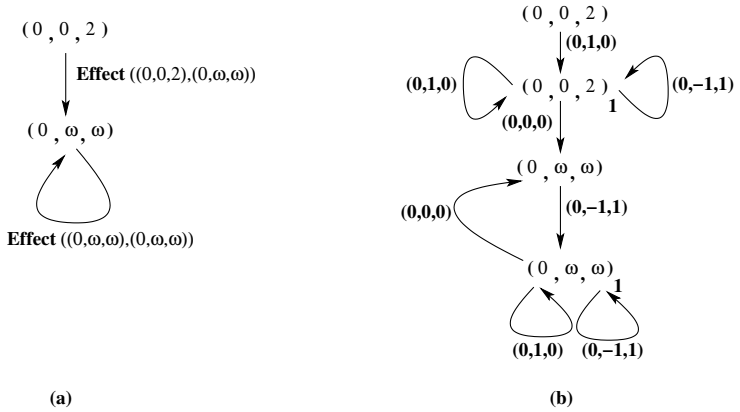

Figure 6.9: (a). Graph $\mathcal{C}'$ derived from $\mathcal{C}$ in Figure 6.8(b). (b) Graph $\mathcal{C}''$ derived from $\mathcal{C}'$.

Example 6.31. In Figure 6.9(a) we show $\mathcal{C}'$ obtained from $\mathcal{C}$ of Figure 6.8(b) with edges labelled by their Presburger-arithmetic representation. In Figure 6.9(b), finally we show the graph $\mathcal{C}''$ obtained from $\mathcal{C}'$ in Figure 6.9(a). □

Correctness of the above constructions:

To show the correctness of the above construction, we need the following lemma.

Lemma 6.32. (a) For every reachable marking M from the initial marking M_0 in an SD-TN, there is an ω-marking $M_{\mathcal{C}}$ in the coverability graph such that $M \leq^m M_{\mathcal{C}}$. (b) For every ω-marking $M_{\mathcal{C}}$ in $\mathcal{C}$, there is a marking M reachable from M_0 such that M is arbitrarily large in the places with ω in $M_{\mathcal{C}}$.

The proof of the above lemma is similar to the correctness proof of the Karp-Miller's algorithm for Petri nets.

Now we show the correctness of the above two constructions.

- Firstly, we show that (Cond1) is sufficient and necessary for the existence of an infinite M-computation π with finitely many transfers for some $M \leq^m M_0$.

  Suppose there is an infinite M-computation π with finitely many transfers. Then π has an infinite suffix π', starting at some marking M' which

uses only ordinary Petri net transitions. Since N' is obtained by removing transfer transitions, π' is an infinite M'-computation of N'. This implies that Cond1 holds for N' (Lemma 6.26). Since the coverability graph for N' is a subgraph of that for N, Cond1 also holds for N. On the other hand, from Lemma 6.26, we have that if Cond1 holds for N', then there is an infinite M'-computation. Since $M \xrightarrow{*} M'$, we have an infinite M-computation in N.

- Secondly, we show that (Cond2) is sufficient and necessary for the existence of an infinite M-computation with infinitely many transfers for some $M \leq^m M_0$.

  If Cond2 is satisfied (i.e., there is a sequence Seq of transitions with non-negative effect), then there exist markings $M \leq^m M_0$ where $M_0 \in \mathcal{C}$ and $M' \leq \mathcal{M}_0$ such that $M \xrightarrow{*} M'$ (by definition of $\mathcal{C}, \mathcal{C}', \mathcal{C}''$ and Lemma 6.32) such that M' is large enough to perform Seq once from M'. Now, Seq has a non-negative effect, therefore one can keep on repeating Seq resulting into an infinite M'-computation. This implies that there is an infinite M-computation.

  Now we show the other direction. Assume that there is some $M \in \mathbb{N}^k$ with $M \leq M_0$ and $M \in \mathit{INF}$ and some infinite M-computation π which uses Trans infinitely often. Thus it contains infinitely many AT-markings. Thus, by Dickson's Lemma [54], there is a computation where $M \xrightarrow{*} (\overrightarrow{0}, \overrightarrow{x_1}) \longrightarrow_{Seq} (\overrightarrow{0}, \overrightarrow{x_2})$ with $\overrightarrow{x_2} \geq^m \overrightarrow{x_1}$. Thus the total effect of the sequence Seq is non-negative. From Lemma 6.32, it follows that there exists an ω-AT-marking $\mathcal{M}_0 \in G$ with $\mathcal{M}_0 \geq^m (\overrightarrow{0}, \overrightarrow{x_2})$. In fact there exists a largest such $\mathcal{M}_0$ (as in case of Petri nets, see Lemma 6.26) such that we have $\mathcal{M}_0 \longrightarrow_{Seq} \mathcal{M}_0$ in C. So, $\mathit{Effect}(\mathcal{M}_0, \mathcal{M}_0) \geq^m \overrightarrow{0}$. The sequence Seq can be decomposed into $Seq = Seq_1 Seq_2 \dots Seq_n$ with $\mathcal{M}_i \longrightarrow_{Seq_i} \mathcal{M}_{i+1}$ for $1 \leq i \leq n-1$ and $\mathcal{M}_n = \mathcal{M}_0$. Here $\{\mathcal{M}_0, \dots, \mathcal{M}_n\}$ is the set of ω-AT markings visited in Seq. In other words, each Seq_i contains the transfer transition only once at the end. It follows that $\mathcal{M}_0 \longrightarrow \mathcal{M}_1 \dots \longrightarrow \mathcal{M}_n = \mathcal{M}_0$ is a cyclic path in C' and $\overrightarrow{v_i} \in \mathit{Effect}(\mathcal{M}_i, \mathcal{M}_{i+1})$ and $\sum_{i=0}^{n-1} \overrightarrow{v_i} = \mathit{Effect}(\mathcal{M}_0, \mathcal{M}_n) \geq^m \overrightarrow{0}$. Therefore the condition (Cond2) is satisfied.

Altogether we obtain that $M_0{\downarrow} \cap \mathit{INF} \neq \emptyset$ iff (Cond1) or (Cond2) is satisfied. Since both conditions are decidable, we obtain decidability of $M_0{\downarrow} \cap \mathit{INF} \neq \emptyset$. □

Lemma 6.33. For any SD-TN N' the set INF'_{min} can be effectively constructed.

Proof. Since INF is upward-closed, we can, by Lemma 6.30 and Theorem 6.25, construct the minimal elements of the set INF, i.e., the set INF_{min}. We obtain INF'_{min} by the restriction of INF_{min} to standard markings. □

6.4 Characterizing ZENO

Theorem 6.34. Let N be a TPN. The set *ZENO* is effectively constructible.

Proof. We first construct the SD-TN N' corresponding to N, according to Section 6.1.1. Then we consider the set Z from Def. 6.14. We have $ZENO = [\![Z]\!]^{\uparrow}$ by Lemma 6.15 and Lemma 6.16. The set Z is effectively constructible by Lemma 6.33, Definition 6.14, and Lemma 4.5. □

6.5 Zenoness-Problem for discrete-timed Petri nets

In this section, we discuss how to characterize ZENO for discrete-timed Petri nets. To do that first we describe how the semantics of a discrete-timed Petri net is different from that of a dense-timed Petri nets.

- Firstly, the ages of the token are natural numbers rather than real numbers.
- Secondly, timed transition takes only discrete steps.

Theorem 6.34 immediately implies the computability of *ZENO* for discrete-timed Petri nets (thus solving the open problem from [51]), because they can be encoded into dense-timed nets. The encoding is graphically described in Figure 6.10. The trick is to split the intervals on the input (output) arcs to several point intervals on a number of transitions.

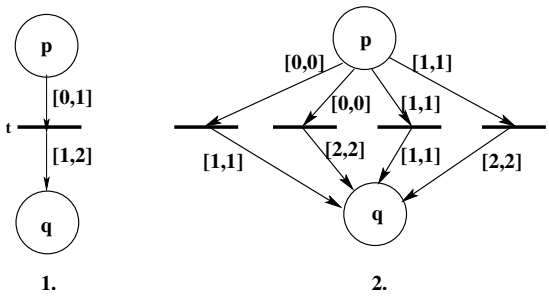

Figure 6.10: Simulating (1) t in TPN by (2) a set consisting of 4 transitions in 2.

Another more direct solution for discrete-timed nets would be to simply remove the transfer-transition from the net N' in Section 6.1.1. This modified construction would yield *ZENO* for the discrete-time case, because (unlike in the dense-time case) every infinite zeno-computation in a discrete-time net has an infinite suffix taking no time at all.

Chapter 7

Token Liveness, Boundedness and Repeated Reachability

In this chapter, we consider the following verification problems for TPNs.

Token Liveness: Markings in TPNs may contain tokens which cannot be used by any future computations of the TPN. Such tokens do not affect the behaviour of the TPN and are therefore called *dead tokens*. We give an algorithm to check, given a token and a marking, whether the token is dead (or alive). We do this by reducing the problem to the problem of *coverability* in TPNs (see Chapter 4, Lemma 4.5).

Token liveness for dense TPNs was left open in [51].

Boundedness: We consider the *boundedness problem* for TPNs: given a TPN and an initial marking, check whether the size of the reachable markings is finite. The decidability of this problem depends on whether we take dead tokens into consideration. In *syntactic boundedness* one considers dead tokens as part of the marking, while in *semantic boundedness* we disregard dead tokens; that is we check whether we can reach markings with unboundedly many live tokens. First we show decidability of syntactic boundedness. This is achieved through an extension of the Karp-Miller algorithm where each node represents a region (rather than a single marking). The underlying ordering on the nodes (regions) inside the Karp-Miller coverability graph is a *well quasi-ordering* (Lemma 4.4 from Chapter 4). This guarantees termination of the procedure.

Decidability of syntactic boundedness was shown for the simpler discrete case in [51], while the problem was left open for the dense case.

On the other hand, we show that semantic boundedness is undecidable (which was also left open in [51]) , using techniques similar to [123]. The same technique is the used to show the undecidability of *repeated reachability* (whether starting from a marking, there is a computation which visits a place infinitely often).

7.1 Token Liveness

First, we define the *liveness* of a token in a marking.

Let M be a marking in a TPN $N = (P, T, In, Out)$. A token in M is called *syntactically dead* if its age is $> max$. Recall that max is the largest integer

appearing on the arcs of TPN. It is trivial to decide whether a token is dead from a marking.

A token is called *semantically live* from a marking M, if we can fire a sequence of transitions starting from M which eventually consumes the token. Formally, given a token $p(x)$ and a marking M, we say that $p(x)$ can be *consumed* in M if there is a transition t satisfying the following properties:

- t is enabled in M.
- $In(t,p)$ is defined and $x \in In(t,p)$ (recall from Chapter 3 that In is a partial function from $T \times P$ to intervals).

Since dead tokens cannot influence the behavior of a TPN, one would like to abstract from them. For a marking M, we let $Live(M)$ be the multiset of live tokens in M.

Let N be a TPN with marking M. Then $Reach^l(M) := \{Live(M') \mid M \xrightarrow{*} M'\}$ is the set of reachable markings with dead tokens removed from them.

Definition 7.1. A token $p(x)$ in M is *semantically live* if there is a finite M-computation $\pi = M \longrightarrow M_1 \longrightarrow \cdots \longrightarrow M_r$ such that $p(x + \Delta(\pi))$ can be consumed in M_r.

Semantic liveness of tokens in TPN

Instance: A timed Petri net N with marking M and a token $p(x) \in M$.
Question: Is $p(x)$ live, i.e., $p(x) \in Live(M)$?

We show decidability of the semantic token liveness problem by reducing it to the *coverability problem* for TPNs (recall the definition of coverability problem from Chapter 5). Notice that the coverability problem is decidable for TPNs (Lemma 4.5).

Next we show how to translate an instance of the token liveness problem to the finitely many instances of the coverability problem.

Suppose that we are given a TPN $N = (P, T, In, Out)$ with marking M and a token $p(x) \in M$. We shall translate the question of whether $p(x) \in Live(M)$ into (several instances of) the coverability problem. To do that, we construct a new TPN N' by adding a new place p^* to the set P. The new place is not input or output of any transition. Either there is no transition in N which has p as its input place. Then it is trivial that $p(x) \notin Live(M)$. Otherwise, we consider all instances of the coverability problem defined on N' such that

- M_{init} contains a single marking $M - p(x) + p^*(x)$.
- M_{fin} is the set of markings of the form $[p_1(x_1), \ldots, p_n(x_n), p^*(x')]$ such that there is a transition t and
  - the set of input places of t is given by $\{p, p_1, \ldots, p_n\}$.
  - $x' \in In(t,p)$ and $x_i \in In(t,p_i)$ for each $i : 1 \leq i \leq n$.

In the construction above, we replace a token $p(x)$ in the initial marking by a token $p^*(x)$; we also replace a token $p(x')$ in the final marking where $x' \in In(t,p)$ by a token $p^*(x')$. The fact that the token in the question is not consumed in any predecessor of a marking in M_{fin}, is simulated by moving the token into the place p^* (in both the initial and final markings), since $p^* \notin P$ and not an input or output place in N'. Therefore, the token is live in M of N iff the answer to the coverability problem is 'yes'.

From Lemma 4.5 and the above construction, we get the following.

Theorem 7.2. The token liveness problem is decidable.

Example 7.3. In Figure 7.1, we show a marking $M = [Q(3), R(5)]$ in the TPN. The question is: whether the token $R(5)$ is in $Live(M)$. Here, $M_{init} = [Q(3), R^*(5)]$. There are two transitions t_1 and t_4 with R as input place. An upward-closed set of final markings is characterized by the following upward-closed set $\mathbb{C} \uparrow$ of regions where $\mathbb{C} = \{(\epsilon, R^*(5), \epsilon), (\epsilon, R^*(8), \epsilon)\}$. Finally we check whether there is a marking $M_0 \in M_{init}$ such that $M_0 \in [\![Pre^*(\mathbb{C})]\!]^{\uparrow}$. □

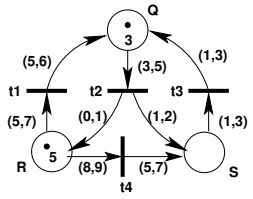

Figure 7.1: A marking in TPN N.

7.2 Syntactic Boundedness

In this section, we consider the boundedness problem for TPNs.

Given a marking M, we use $|M|$ to denote the number of elements in M. Also, abusing notations, we extend this notion to a set M of markings as follows : $|\mathsf{M}| = max(|M| : M \in \mathsf{M})$.

Syntactic Boundedness of TPN

Instance: A timed Petri net N with initial marking M_0.
Question: Is $|Reach(M_0)|$ finite ?

We give an algorithm similar to the Karp-Miller algorithm [91] for solving the syntactic boundedness problem for TPNs. The algorithm builds a tree, where each node of the tree is labeled with a region. We build the tree successively, starting from the root, which is labeled with $\mathcal{R}_{M_0}$: the unique region satisfied by M_0 (recall from Chapter 4). At each step we pick a leaf with label $\mathcal{R}$ and perform one of the following operations:

1. If $Post(\mathcal{R})$ is empty we declare the current node *unsuccessful* and close the node.

2. If there is another (already generated) node which is labeled with $\mathcal{R}$ then declare the current node *duplicate* and close the node.

3. If there is a predecessor of the current node labeled with $\mathcal{R}' \sqsubseteq \mathcal{R}$ then declare $|Reach(M_0)|$ infinite (the TPN is unbounded), and terminate the procedure.

4. Otherwise, declare the current node as an *interior* node, add the set of successors to it, each labeled with an element in $Post(\mathcal{R})$. This step is possible due to Lemma 5.17.

If the condition of step 3 is never satisfied during the construction of the tree, declare $Reach(M_0)$ finite (the TPN is bounded).

The proof of correctness of the above algorithm is similar to that of original Karp-Miller construction [91]. The termination of the algorithm is guaranteed due to the fact that the ordering $\sqsubseteq$ on the set of regions is a well-quasi-ordering (Lemma 4.4 in Chapter 4).

Theorem 7.4. Syntactic boundedness is decidable for TPNs.

From Theorem 7.4, it follows that termination (whether there is an infinite computation starting from an initial marking) is also decidable for TPNs.

7.3 Semantic Boundedness

In this section, we show that whether we can reach markings with unboundedly many live tokens is undecidable.

Semantic Boundedness of TPN

Instance: A timed Petri net N with initial marking M_0.
Question: Is $|Reach^l(M_0)|$ finite ?

Next we show the undecidability of semantic boundedness for dense-timed Petri nets, using slightly modified constructions of [123].

Theorem 7.5. Semantic Boundedness of TPN is undecidable.

The rest of this section is devoted to the proof of Theorem 7.5.

We show undecidability of semantic boundedness for dense-timed Petri nets through reduction from an undecidable problem for *lossy counter machines (LCMs)* .

7.3.1 Lossy Counter Machines

A *lossy counter machine (LCM)* is a tuple $L = (S, s_{init}, C, \mathtt{I})$, where S is a finite set of *states*, $s_0 \in S$ is the initial state, C is a finite set of counters and $\mathtt{I}$ is a finite set of *instructions*. An instruction is a triple of the form $(s_1, instr, s_2)$, where $s_1, s_2 \in S$ and *instr* is either an *increment* (of the form $++c$); a *decrement* (of the form $--c$); or a *reset* (of the form $c := 0$) for a counter

c in L. A *configuration* β of L is of the form (s, val), where $s \in S$ and val is a mapping from the set C of counters to the set $\mathbb{N}$ of natural numbers. The lossy counter machine L induces a transition relation $\rightsquigarrow$ on the set of configurations, such that $(s_1, val_1) \rightsquigarrow (s_2, val_2)$ iff one of the following conditions is satisfied:

1. $(s_1, ++c, s_2) \in \mathtt{I}$, $val_2(c) = val_1(c) + 1$ and $val_2(c') = val_1(c')$ if $c' \neq c$.
2. $(s_1, --c, s_2) \in \mathtt{I}$, $val_1(c) > 0$, $val_2(c) = val_1(c) - 1$ and $val_2(c') = val_1(c')$ if $c' \neq c$.
3. $(s_1, c := 0, s_2) \in \mathtt{I}$, $val_2(c) = 0$ and $val_2(c') = val_1(c')$ if $c' \neq c$.
4. $s_2 = s_1$, $val_2(c) = val_1(c) - 1$ for some $c \in C$, and $val_2(c') = val_1(c')$ if $c' \neq c$.

For a configuration β, a *β-computation* π of L is a sequence of configurations and transitions given by, $\beta_0 \rightsquigarrow \beta_1 \rightsquigarrow \beta_2, \ldots$, where $\beta_0 = \beta$. We say that a computation π of L is space-bounded iff there is an integer $n \in \mathbb{N}$ s.t. sum of values of the counters does not exceed n in each configuration of π.

An LCM is *space-bounded* (by k) if all its computations are space-bounded (by k). The following problems for LCMs are shown to be undecidable in [102].

Space-Boundedness for LCM

Instance: An LCM L with the initial configuration $\beta_0 = (s_0, val_0)$ such that for each counter c, $val_0(c) = 0$.
Question: Is L space-bounded ?

RSP-LCM

Instance: An LCM L with an initial configuration β and a state s of L.
Question: Is there a β-computation of L visiting q infinitely often ?

7.3.2 Proof of Theorem 7.5

It is well-known that space-boundedness for LCMs is undecidable. To show undecidability of semantic boundedness problem for TPNs, we reduce the problem of space-boundedness for LCM to it. Then Theorem 7.5 follows.

Construction

Given an LCM $L = (S, s_{init}, C, \mathtt{I})$, we construct a TPN $N = (P, T, In, Out)$ simulating L as follows. For each state $s \in S$ there is a place p in P. We use P_S to denote the set of places of N corresponding to S. For each counter $c \in C$, there is a place c in P. We use P_C to denote to the set of places corresponding to C. A configuration β of L is encoded by a marking M in N when the following conditions are satisfied.

- The state of β is defined in N by the element of P_S which contains a token. (The TPN N satisfies the invariant that there is at most one place in P_S which contains a token).

- The value of a counter c in β is defined in M by the number of tokens in place c which have ages equal to 0 (tokens which have ages more than 0 are considered to have been lost and do not affect the value of the counter).

Losses in L are simulated either by making the age of the token strictly greater than 0, or by firing a special *loss* transition which can always remove tokens from the places in P_C. Transitions in L are encoded by functions *In* and *Out* in N reflecting the above properties and are defined as follows.

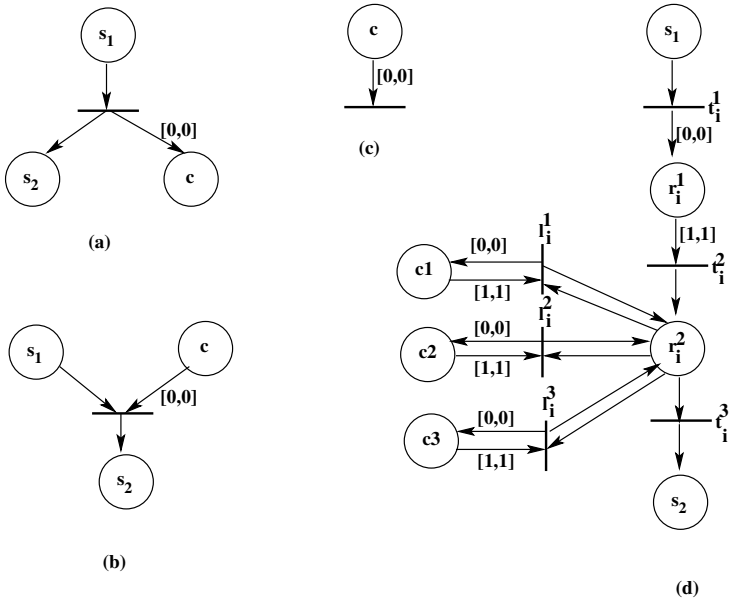

Figure 7.2: Simulating the operation of (a) increasing the counter, (b) decreasing the counter. (c) the loss of the counter. (d) resetting a counter.

- An *increment* $\imath = (s_1, ++\, c, s_2)$ in $\mathtt{I}$ is simulated by a transition in T which is of the form in Figure 7.2(a). The transition moves a token from place s_1 to place s_2 and adds a token of age 0 to place c.
- A *decrement* $\imath = (s_1, --\, c, s_2)$ in $\mathtt{I}$ is simulated by a transition in T which is of the form in Figure 7.2(b). The transition moves a token from place s_1 to place s_2 and removes a token of age 0 from place c.
- For each place c in P_C there is a transition which is of the form in Figure 7.2(c). This transition removes the tokens of age 0 from a counter and thus simulates the lossiness of a counter.
- A *reset* instruction $\imath = (s_1, c := 0, s_2)$ has the most complicated simulation. The construction is shown in Figure 7.2(d). We use two intermediate places $r_\imath^1$ and $r_\imath^2$ for each reset instruction $\imath$. The transition $t_\imath^1$ is fired

by moving a token from place s_1 to place $r_\imath^1$. The token in $r_\imath^1$ has to stay there for a time equal to 1 and then transition $t_\imath^2$ is fired. If more time passes, then this token in $r_\imath^1$ will forever stay in place $r_\imath^1$ after which no tokens will ever reside in any place in P_S and thus the net will deadlock. The idea is that we reset the value of counter c to 0, by making the ages of all tokens in place c at least equal to 1. Observe that we simulate resetting the counter in L by resetting the counter in N. All tokens in each of the places in $P_C - \{c\}$ which had age 0 have now age equal to 1. Thus, in order to avoid resetting the values of the rest of the counters, we add, for each $c' \in C - \{c\}$ a new transition. In Figure 7.2(d), without loss of generality, we assume that $C - \{c\} = \{c_1, c_2, c_3\}$, and thus we add the transitions $\ell_\imath^1, \ell_\imath^2, \ell_\imath^3$. These transitions are used to *refresh* the ages of the tokens in the places in $P_C - \{c\}$. Now, if a token in place c_1 is equal to 1, and thus has become too old for firing other transitions (*increments* and *decrements*), it is replaced by a fresh token of age 0. Finally, when the transition $t_\imath^3$ is fired, the new control state will be s_2, and each token in place c will have an age which is at least one. The resulting marking will therefore correspond to the counter c having the value 0. The refreshing process for the counters c_1, c_2, c_3 will be stopped after firing $t_\imath^3$, since the token in $r_\imath^2$ will now be removed. Notice that some tokens in c_1, c_2, c_3 may be lost (i.e., may still have age greater or equal to 1), since the TPN has a lazy semantics and these tokens may not have been refreshed.

Consider a marking M of N and a configuration $\beta = (s, \mathit{val})$ of L. We say that M is an *encoding* of β if M contains a token in place s and the number of tokens with ages equal to 0 in place c is equal to $\mathit{val}(c)$ for each $c \in C$. Furthermore, all other places in M are empty. The token in place s of an encoding is *live*, otherwise net will deadlock. From the definition of encoding and the above contruction, it also follows that the other live tokens are of age 0 and they only reside in the places in P_C. This means that in an encoding, only the places in P_C may contain unboundedly many live tokens.

We also use the following notion of intermediate markings. A marking is called *intermediate* if it has a token in place $r_\imath^1$ ($r_\imath^2$) where $\imath$ is of the form $(s_1, c := 0, s_2)$ and there are no tokens in other intermediate places and in those belonging to P_S.

Suppose an intermediate marking M' is reached from a marking M during the execution. This can be the case only during the simulation of reset instructions of L. Let $\imath = (s_1, c := 0, s_2)$ be a reset instruction of L such that M' contains a token in $r_1^\imath$ or $r_2^\imath$. From the simulation, we know that during this transition (a) we do not refresh tokens in place c, (b) we may not always refresh tokens of age 1 in places $P_C \setminus \{c\}$. Furthermore, if the token is not refreshed, it will never be refreshed in the future. From this, it follows that the number of live tokens in the places $c \in P_C$ in M' is always smaller than those in M.

We derive N from L as described above. We define M_0 to be the encoding of γ_0.

Now, we prove that L is space-bounded from β_0 iff the number of live tokens in reached markings from M_0 of N are bounded.

Correctness of the above construction:

$\Rightarrow$:

First we show that if L is unbounded in space from β_0 then N generates from M_0 an unbounded number of "live" tokens. Consider a β_0-computation of L where $\beta_0 = (s_0, val_0)$ where for each counter c, $val_0(c) = 0$ and L is not space-bounded. We show that there is a M_0-computation π' of N, such that M_0 is an encoding of γ_0 and an unbounded number of live tokens are generated in π'.

From the property that the number of live tokens does not increase in intermediate markings (see above), it is clear that this direction of the proof follows if we prove the following.

Given two configurations β_1, β_2 of L such that $\beta_1 \rightsquigarrow \beta_2$ and a marking M which is an encoding of β_1, there is a sequence of transitions in N of the form $M = M_0 \longrightarrow_{Disc} M_1 \longrightarrow_{Time} \cdots \longrightarrow_{Disc} \longrightarrow_{Time} M_n = M'$ where $n \geq 1$ and the following holds.

- M' is an encoding of β_2.
- M_i is an intermediate marking for $0 < i < n$.

Since $\beta_1 \rightsquigarrow \beta_2$, we know that β_2 is derived from β_1, using one of the four possible types of transitions described for LCMs. We show the claim only for the least obvious case, namely when β_2 is derived from β_1 by executing a reset instruction $\imath$ which resets the value of a counter c for 0. The other cases can be explained in a similar manner. Let $\beta_1 = (s_1, val_1)$ and $\beta_2 = (s_2, val_2)$. We show that the marking M' is the encoding of β_2. Since M is an encoding of β_1, it means that place s_1 in M contains a token. From the construction described above (Figure 7.2(d)) we know that from M we can fire t_i^1 and produce a marking M_1 such that $M \longrightarrow_{t_i^1} M_1$ and $M_1 = M - s_1(x) + r_i^1(0)$. This means that $M_1(c'(0)) = M(c'(0))$ for each counter $c' \in P_C$.

Next we let time pass by one time unit and obtain a marking M_2 such that $M_1 \longrightarrow_{T=1} M_2$. This means that $M_2(c'(1)) = M_1(c'(0))$ for each counter $c' \in P_C$. Now we fire the transition t_i^2 from M_2 and this results in a marking M_3, i.e., $M_2 \longrightarrow_{t_i^2} M_3$ and $M_3 = M_2 - r_i^1(1) + r_i^2(x)$. Here, for each counter $c' \in P_C$, we have $M_3(c'(1)) = M_2(c'(1))$ and $M_3(c'(x)) = 0$ for all $x < 1$.

Then, suppose for a counter $c_1 \in P_C \setminus \{c\}$, $val_1(c_1) = n$. We fire the transition ℓ_i^1 n times and refresh all n tokens of age 1 in c_1 to age 0. Similarly we refresh all tokens of age 1 in other counters in $P_C \setminus \{c\}$. Notice that the refreshing phase always terminates due to bounded number of tokens and inability of N to refresh tokens of age different from 1 in c_1, c_2, c_3. The markings $M_1, M_2, \ldots$, etc in the above are all intermediate markings.

Now we fire the transition t_i^3 by moving the token from r_i^2 to s_2, yielding a marking M'. This means that for each counter $c' \in P_C \setminus \{c\}$, $M'(c'(0)) = M(c'(0))$. Furthermore, the ages of all tokens in place c are equal to 1, i.e., $M'(c(1)) = M(c(0))$. Also, it is the case that $M'(c(0)) = 0$. This implies that

M' is an encoding of β_2.

$\Leftarrow$:

Suppose that there is an infinite M-computation π of N such that the number of live tokens in the places in P_C is unbounded.

Consider the maximal subsequence π' of π, where each marking in π is an encoding of some configuration of L. The sequence π' exists (say, starts at M'') and due to the construction of N (from any intermediate-marking, the transitions can be fired only finitely many times), π' is infinite. In the following, we show that there is an infinite β-computation where M'' is an encoding of β and LCM is unbounded in space.

To prove this, it is enough to show that given two consecutive encodings M and M' in π' such that $M \xrightarrow{*} M'$ and a configuration β_1 which is encoded by M, there is a configuration β_2 such that $\beta_1 \xrightarrow{*} \beta_2$. Let $\beta_1 = (s_1, val_1)$.

Since $M \xrightarrow{*} M'$ we know that there are markings $M_0, \ldots, M_n$ such that $M = M_0 \longrightarrow M_1 \longrightarrow \cdots \longrightarrow M_n = M'$ where $n \geq 1$ and $M_1, \ldots, M_{n-1}$ are intermediate markings. There are two cases. Either $n = 1$ or $n > 1$.

If $n = 1$, i.e., $M \longrightarrow M'$, we know that M' can be derived from M by firing a

- discrete transition corresponding to one among Figures 7.2(a), 7.2(b) and 7.2(c). In this case, β_2 is obtained from β_1 by executing an increment/decrement/loss instruction of L.
- timed transition by letting time pass by $\delta > 0$. In this case, β_2 can be derived from β_1 by losing the current values of all the counters from β_1.

If $n > 1$, then M' is obtained from M by firing transitions corresponding to those in Figure 7.2(d). This means that $\imath = (s_1, c := 0, s_2)$ is an instruction in L, for some counter c. From the construction of Figure 7.2(d), we know that all tokens in place c will eventually have age at least 1. Furthermore, the ages of some of the tokens in $P_C - \{c\}$ may also exceed 1, since not all tokens need to be refreshed. We can derive β_2 from β_1 by first performing loss transitions corresponding to tokens which become too old followed by executing the instruction $(s_1, c := 0, s_2)$.

7.4 Repeated Reachability

In the following, we say that a computation $\pi = M_0 \longrightarrow M_1 \longrightarrow M_2 \longrightarrow \ldots$ of TPN *visits a place p infinitely often* if there are infinitely many i such that $M_i(p) \neq \emptyset$.

The *repeated reachability problem* for TPNs is defined as follows.

Repeated Reachability Problem (RPP-TPN)

Instance: A dense-timed Petri net N, a marking M of N and a place q of N.
Question: Is there an M-computation of N visiting q infinitely often ?

Proof. We reduce RSP-LCM to RPP-TPN by constructing a TPN N as in

Section 7.3.2. We let M be the encoding of β (recall the definition of RSP-LCM) and the place q to be the state s of LCM. By this construction, the state s is visited infinitely often in LCM if and only if the place p in TPN is visited infinitely often. The undecidability follows from this construction and undecidability of RSP-LCM. □

It is straightforward to show that RPP-TPN is reducible to the problem of checking whether there is a computation of the TPN along which a given transition is fired infinitely often. It follows that (action based) linear-time temporal logic (LTL) for TPNs is also undecidable.

7.4.1 Discrete-timed Petri nets

Finally, we consider the repeated reachability problem for discrete-timed Petri nets and show that this problem is also undecidable. Recall the definitions for discrete-timed Petri nets from Section 6.5.

The repeated reachability problem for discrete-timed Petri nets (RSP-DTPN) can be defined in a manner similar to the definition of RSP-TPN. Even in the case of discrete-timed nets, undecidability result follows from the undecidability of RSP-LCM and the construction in Section 7.3.2. This means that action-based LTL is also undecidable for discrete-timed Petri nets.

7.5 Related Work

- We already mentioned that the token-liveness problem and the syntactic boundedness problem for discrete-timed Petri nets were solved in [51], but they only gave a conjecture about the decidability of these problems and undecidability of semantic boundedness problem for dense-timed Petri nets.
- Syntactic boundedness problem for ordinary Petri nets [91] and transfer nets (Chapter 6) [55] are known to be decidable, while this problem is undecidable for reset nets [56], where a reset arc from a place just removes all the tokens from this place.
- Checking LTL formulae for standard (untimed) Petri nets [60] is decidable. In fact [35] shows that there is an effectively constructible representation of the set of markings satisfying a formula. We also recall that even small fragments of branching time logics are known to be undecidable for Petri nets [62, 103]. In fact, all state based temporal logics are undecidable for Petri nets [62].

Part II

Multi-Clock Timed Systems

Chapter 8

Timed Networks

A *timed network* (TN, in short) represents a family of systems, each consisting of a finite-state *controller*, together with finitely, but arbitrarily many *timed processes* (timed automata). A timed process operates on a finite number of real-valued clocks. This means that a TN operates on an unbounded number of clocks, and therefore its behaviour cannot be captured by that of a timed automaton [19].

The paper [9] showed decidability of the *controller state reachability problem* for TNs: given a state of the controller, is there a computation from an initial configuration leading to that state? This problem is relevant since it can be shown, using standard techniques, that checking large classes of safety properties can be reduced to controller state reachability (as shown in Chapter 4). The decidability result in [9] is given subject to the restriction that each timed process has a single clock. As an example, this allows automatic verification of a parameterized version of *Fischer's protocol* (see e.g. [95]). This protocol achieves mutual exclusion by defining timing constraints on an arbitrary set of processes each with one clock. The correctness of this protocol was shown by [9], regardless of the number of participating processes. The paper [9] leaves open the case of *multi-clock* TNs, i.e., TNs where each timed process may have several clocks. The question is then whether the decidability result of [9] can be extended to multi-clock systems. In this chapter, we answer this question negatively. In fact we show that it is sufficient to allow two clocks per process in order to get undecidability. The undecidability result is shown through a reduction from the classical reachability problem for 2-counter machines. The main ingredient in the undecidability proof is an encoding of counters which allows testing for zero. The encoding represents each counter by a linked list of processes, where ordering on elements of the list is reflected by ordering on clock values of the relevant processes, and where the link between two elements in the list is encoded by whether two clocks belong to the same process. The value of a counter is reflected by the length of the corresponding list.

8.1 Model

First we define *timed networks*: families of (infinitely many) systems each consisting of a *controller* and an arbitrary number of identical *timed processes*. The

controller is a finite state automaton while each process is a timed automaton [19], i.e., a finite-state automaton which operates on a finite number of local real-valued clocks $x_1, \ldots, x_K$. The values of all clocks are incremented continuously at the same rate. In addition, the network can change its configuration according to a finite number of *rules*. Each rule describes a set of transitions in which the controller and a fixed number of processes synchronize and simultaneously change their states. A rule may be conditioned on the local state of the controller, together with the local states and clock values of the processes. If the conditions for a rule are satisfied, then a transition may be performed where the controller and each participating process changes its state. Also, during a transition, a process may reset some of its clocks to 0.

Timed Networks

A *family of timed networks* (*timed network* for short) $\mathcal{N}$ with K clocks is a pair $(Q, \Re)$, where:

- Q is a finite set of *states*. The set Q is the union of two disjoint sets; the set Q^{ctrl} of *controller states*, and the set Q^{proc} of *process states*. These sets contain two distinguished *initial (idle)* states, namely $idle^c \in Q^{ctrl}$ and $idle^p \in Q^{proc}$.
- $\Re$ is a finite set of *rules* where each rule is of the form

$$\begin{bmatrix} q_0 \\ \rightarrow \\ q_0' \end{bmatrix} \quad \begin{bmatrix} q_1 \\ g_1 \rightarrow R_1 \\ q_1' \end{bmatrix} \quad \cdots \quad \begin{bmatrix} q_n \\ g_n \rightarrow R_n \\ q_n' \end{bmatrix}$$

  such that $q_0, q_0' \in Q^{ctrl}$, and for all $i : 1 \leq i \leq n$ we have: $q_i, q_i' \in Q^{proc}$, and $g_i \rightarrow R_i$ is a guarded command where g_i is a boolean combination of predicates of the form $k \ \triangleright \ x$ for $k \in \mathbb{N}$, $\triangleright \in \{=, <, \leq, >, \geq\}$, $x \in \{x_1, \ldots, x_K\}$ and $R_i \subseteq \{x_1, \ldots, x_K\}$.

Intuitively, the set Q^{ctrl} represents the states of the controller and the set Q^{proc} represents the states of the processes. A rule of the above form describes a set of transitions of the network. The rule is enabled if the state of the controller is q_0 and if there are n processes with states $q_1, \cdots, q_n$ whose clock values satisfy the corresponding guards. The rule is executed by simultaneously changing the state of the controller to q_0' and the states of the n processes to $q_1', \cdots, q_n'$, and resetting the clocks belonging to the sets $R_1, \ldots, R_n$.

For a guard g_i we write $g_i(y_1, \ldots, y_K)$ to denote the boolean expression which results from substituting the occurrences of $x_1, \ldots, x_K$ in g_i by $y_1, \ldots, y_K$ respectively.

Configurations A configuration γ of a timed network $(Q, \Re)$ with K clocks is a tuple of the form $(I, q, \mathcal{Q}, X)$, where I is a finite *index set*, $q \in Q^{ctrl}$, $\mathcal{Q} : I \rightarrow Q^{proc}$, and $X : \{1, \ldots, K\} \rightarrow I \rightarrow \mathbb{R}^{\geq 0}$.

Intuitively, the configuration γ refers to the controller whose state is q, and to $|I|$ processes, whose states are defined by $\mathcal{Q}$. The clock values of the processes

are defined by X. More precisely, for $k : 1 \leq k \leq K$ and $i \in I$, $X(k)(i)$ gives the value of clock x_k in the process with index i.

We use $|\gamma|$ to denote the number of processes in γ, i.e., $|\gamma| = |I|$. Also, we shall use X_k to denote the mapping $I \to \mathbb{R}^{\geq 0}$ such that $X_k(i) = X(k)(i)$.

Example 8.1. Figure 8.1 shows graphical representation of a configuration in a timed network with two clocks, given by $(\{1, 2, 3\}, q, \mathcal{Q}, X)$ where $\mathcal{Q}(1) = q_1, \mathcal{Q}(2) = q_2, \mathcal{Q}(3) = q_3$ and $X_1(1) = 0.1, X_1(2) = 0.5, X_1(3) = 5.0, X_2(1) = 2.3, X_2(2) = 1.4, X_2(3) = 0.6$. □

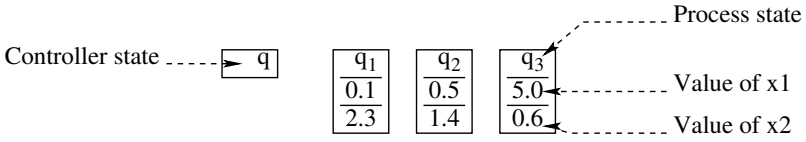

Figure 8.1: Graphical representation of a configuration in a timed network with two clocks.

Transition Relation The timed network $\mathcal{N}$ above induces a transition relation $\longrightarrow$ on the set of configurations. The relation $\longrightarrow$ is the union of a *discrete* transition relation $\longrightarrow_{Disc}$, representing transitions induced by the rules, and a *timed* transition relation $\longrightarrow_{Time}$ which represents passage of time.

The discrete relation $\longrightarrow_{Disc}$ is the union $\bigcup_{r \in \Re} \longrightarrow_r$, where $\longrightarrow_r$ represents a transition performed according to rule r. Let r be a rule of the form described in the above definition of timed networks. Consider two configurations $\gamma = (I, q, \mathcal{Q}, X)$ and $\gamma' = (I, q', \mathcal{Q}', X')$. We use $\gamma \longrightarrow_r \gamma'$ to denote that there is an injection $h : \{1, \ldots, n\} \to I$ such that for each $i : 1 \leq i \leq n$ and $k : 1 \leq k \leq K$ we have:

1. $q = q_0$, $\mathcal{Q}(h(i)) = q_i$, and $g_i(X_1(h(i)), \ldots, X_K(h(i)))$ holds. That is, the rule r is enabled.

2. $q' = q'_0$, and $\mathcal{Q}'(h(i)) = q'_i$. The states are changed according to r.

3. If $x_k \in R_i$ then $X'_k(h(i)) = 0$, while if $x_k \notin R_i$ then $X'_k(h(i)) = X_k(h(i))$. In other words, a clock is reset to 0 if it occurs in the corresponding set R_i. Otherwise its value remains unchanged.

4. $\mathcal{Q}'(j) = \mathcal{Q}(j)$ and $X'_k(j) = X_k(j)$, for $j \in I \setminus \text{range}(h)$, i.e., the process states and the clock values of the non-participating processes remain unchanged.

For a configuration $\gamma = (I, q, \mathcal{Q}, X)$ and $\delta \in \mathbb{R}^{\geq 0}$, we use $\gamma^{+\delta}$ to denote the configuration $(I, q, \mathcal{Q}, X')$ where $X'_k(j) = X_k(j) + \delta$ for each $j \in I$ and $k : 1 \leq k \leq K$. A *timed transition* is of the form $\gamma \longrightarrow_{T=\delta} \gamma'$ where $\gamma' = \gamma^{+\delta}$. Such a transition lets time pass by δ. We use $\gamma \longrightarrow_{Time} \gamma'$ to denote that $\gamma \longrightarrow_{T=\delta} \gamma'$ for some $\delta \in \mathbb{R}^{\geq 0}$.

We define $\longrightarrow$ to be $\longrightarrow_{Disc} \cup \longrightarrow_{Time}$ and use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\longrightarrow$. Notice that if $\gamma \longrightarrow \gamma'$ then the index sets of γ and γ' are identical and therefore $|\gamma| = |\gamma'|$. For a configuration γ and a controller state q, we use $\gamma \xrightarrow{*} q$ to denote that there is a configuration γ' of the form $(I', q', \mathcal{Q}', X')$ such that $\gamma \xrightarrow{*} \gamma'$ and $q' = q$. A γ-computation of TN from a configuration γ is a sequence $\gamma_0 \longrightarrow_{T=\delta_1} \longrightarrow_{r_1} \gamma_1 \ldots \longrightarrow_{T=\delta_n} \longrightarrow_{r_n} \gamma_n$ of configurations and transitions where $\gamma_0 = \gamma$ and $\delta_i \in \mathbb{R}^{\geq 0}$ for $i : 1 \leq i \leq n$ and $r_i \in \Re$. Given a computation π, we use $\pi(i)$ to denote the part of the computation $\gamma_0 \longrightarrow_{T=\delta_1} \longrightarrow_{r_1} \gamma_1 \ldots \longrightarrow_{T=\delta_i} \longrightarrow_{r_i} \gamma_i$ where $i \geq 0$.

8.2 Controller State Reachability

A configuration $\gamma_{init} = (I, q, \mathcal{Q}, X)$ is said to be *initial* if $q = idle^c$, $\mathcal{Q}(i) = idle^p$, and $X_k(i) = 0$ for each $i \in I$ and $k : 1 \leq k \leq K$. This means that an execution of a timed network starts from a configuration where the controller and all the processes are in their initial states, and the clock values are all equal to 0. Notice that there is an infinite number of initial configurations, namely one for each index set I.

Controller State Reachability Problem (TN(K)-Reach)
Instance A timed network $(Q, \Re)$ with K clocks and a controller state q_F.
Question Is there an initial configuration γ_{init} such that $\gamma_{init} \xrightarrow{*} q_F$?

The controller state reachability is relevant, since we recalled in Chapter 4, that using standard techniques in [132, 76], checking safety properties (expressed as regular languages) can be translated into instances of the problem. In [9], it was shown that TN(1)-Reach is decidable. In this chapter, we show that

Theorem 8.2. TN(2)-Reach is undecidable.

8.3 2-Counter Machines

First we recall the standard definition of counter machines Here, we assume that such a machine operates on two counters which we call c_1 and c_2.

A *two-counter machine* C is a tuple $(S, s_{init}, \{c_1, c_2\}, \mathtt{I})$ where S is a finite set of *local states* with a distinguished *initial local state* $s_{init} \in S$, and $\mathtt{I}$ is a finite set of *instructions* (recall from the definition of *lossy counter machines* from Chapter 7). An instruction $\imath$ is either an *increment*, a *decrement* or a *zero testing* instruction. Notice that C does not have a lossy transition. A *configuration* β of a two-counter machine is a triple (s, m_1, m_2), where $s \in S$ represents the local state, and $m_1, m_2 \in \mathbb{N}$ represent the values of the counters c_1 and c_2 respectively. The counter machine C induces a transition relation $\rightsquigarrow$ on the set of configurations, which is defined as usual using the standard interpretations of counter operations (see Chapter 7). We use $\overset{*}{\rightsquigarrow}$ to denote the reflexive transitive closure of $\rightsquigarrow$. In a similar manner to timed networks, we

use $\beta \overset{*}{\rightsquigarrow} s$ to denote that there is a configuration $\beta' = (s', m_1', m_2')$ such that $\beta \overset{*}{\rightsquigarrow} \beta'$ and $s' = s$. We define the *initial configuration* β_{init} to be $(s_{init}, 0, 0)$. The *control state reachability problem* for a 2-counter machines (CM-Reach) is: given local state s_F check whether $\beta_{init} \overset{*}{\rightsquigarrow} s_F$. The following result [108] is well-known.

Theorem 8.3. CM-Reach is undecidable.

In our correctness proof (Section 8.6), we use the relation $\overset{n}{\rightsquigarrow}$, with $n \geq 0$, on configurations, where $\beta \overset{n}{\rightsquigarrow} \beta'$ iff there is a sequence $\beta_0 \rightsquigarrow \beta_1 \rightsquigarrow \cdots \rightsquigarrow \beta_n$ with $\beta_0 = \beta$ and $\beta_n = \beta'$. The relation $\overset{n}{\rightsquigarrow}$ is extended to local states in a similar manner to $\overset{*}{\rightsquigarrow}$. Notice that $\overset{*}{\rightsquigarrow} = \cup_n \overset{n}{\rightsquigarrow}$.

8.4 Encoding of Configurations

We show undecidability of TN(2)-Reach through a reduction from CM-Reach. Given a counter machine $\mathsf{C} = (S, s_{init}, \{c_1, c_2\}, \mathtt{I})$, we shall derive a timed network $\mathcal{N}_\mathsf{C} = (Q_\mathsf{C}, \Re_\mathsf{C})$ with two clocks. In this section, we perform the first step in the reduction; namely we describe how to construct the set Q_C. Also, we describe how configurations of C are encoded as configurations of $\mathcal{N}_\mathsf{C}$. Finally, we introduce a special type of encodings, called *proper encodings*, which we use in our simulation of C.

States According to the model described in Section 8.1, the set Q_C will consist of two disjoint sets of states: the set Q_C^{ctrl} of controller states and the set Q_C^{proc} of process states. The set Q_C^{ctrl} contains three types of states:

1. The initial controller state $idle^c$.
2. *Local states of* C: all members of S have copies in Q_C^{ctrl}.
3. *Temporary states*: the set Q_C^{ctrl} contains a state $tmp^\imath$ for each increment instruction $\imath \in \mathtt{I}$. These states are used as intermediate states during the simulation of increments (Section 8.5). The set Q_C^{ctrl} also contains the state s'_{init} (recall that s_{init} is the initial local state of C). This state is used as an intermediate state in the initialization phase of the the simulation (Section 8.5).

The set Q_C^{proc} contains two types of states:

1. The initial process state $idle^p$.
2. Six states fst_1, mid_1, $last_1$, fst_2, mid_2, and $last_2$, used for encoding the two counters (as described below).

Encodings Each configuration β of C will be encoded by a set of configurations in $\mathcal{N}_\mathsf{C}$. The local state of β will be encoded by the controller state. Each counter will be modelled by a *counter encoding*. A counter encoding arranges a set of

processes as a circular list. The ordering among elements of the list is defined by the clock values. The length of the list reflects to the value of the counter. To define counter encodings, we shall use the six process states fst_1, mid_1, $last_1$ (used for encoding of c_1), and fst_2, mid_2, $last_2$ (used for encoding of c_2). The states fst_1 and $last_1$ are the states of the first and last processes in the list encoding the value of c_1. All processes in the middle of the list will be in state mid_1. The states fst_2, mid_2, and $last_2$ play similar roles in the encoding of c_2. Formally, a configuration $\gamma = (I, q, \mathcal{Q}, X)$ is said to be a c_1-*encoding* of *value* m if there is an injection h from the set $\{0, \ldots, m+1\}$ to I such that the following conditions are satisfied

- $\mathcal{Q}(h(0)) = fst_1$, $\mathcal{Q}(h(m+1)) = last_1$, and $\mathcal{Q}(h(i)) = mid_1$ for each $i : 1 \leq i \leq m$.
- $\mathcal{Q}(j) \in \{idle^p, fst_2, mid_2, last_2\}$ if $j \in I \setminus \text{range}(h)$.
- $X_1(h(i)) < X_1(h(i+1))$ for each $i : 0 \leq i \leq m$.
- $X_2(h(i)) = X_1(h((i+1) \mod (m+2)))$, for each $i : 0 \leq i \leq m+1$.

The first condition states that the processes which are part of a c_1-encoding are in one of the local states fst_1, mid_1, or $last_1$. The second condition states that the processes which are not part of a c_1-encoding are in one of the local states $idle^p, fst_2$, mid_2, or $last_2$. The third and the fourth conditions show how the processes which are part of a c_1-encoding are ordered as a circular list. The position of each process in the list is reflected by values of its clocks x_1 and x_2. More precisely, the ordering among the x_1 clocks reflects the positions of the processes in the list. Also, clock x_2 of each process (except the last process) is equal to clock x_1 of the next process. Finally, clock x_2 of the last process is equal to clock x_1 of the first process (giving the list a "circular" form). We use $Val_1(\gamma)$ to denote the value m of a c_1-encoding γ.

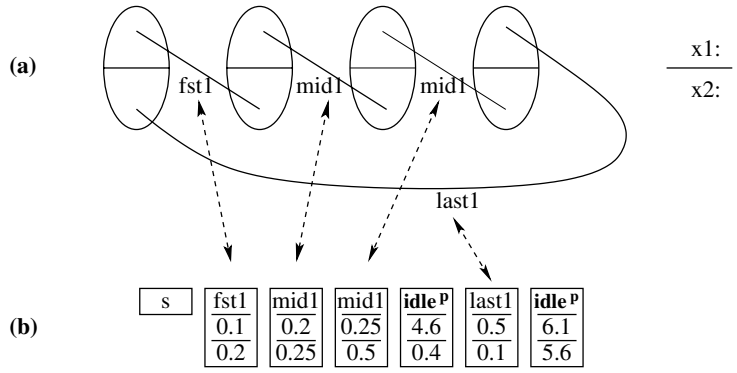

Figure 8.2: (a) Graphical representation of ordering among clocks in c_1-encodings. (b) a c_1-encoding satisfying the ordering on clocks described in (a).

Example 8.4. Figure 8.2(b) shows a c_1-encoding of value 2. Figure 8.2(a) shows a graphical representation of the ordering among clock values. In Section 8.5, we shall use such a graphical representation to explain the different steps in the simulation of C. Each ellipse contains clocks of equal values. Clocks in successive ellipses have increasing values, i.e., they are ordered from left (lower clock values) to right (higher clock values). The upper halves of the ellipses represent the x_1-clocks, while the lower halves represent the x_2-clocks. Each process is denoted by an edge whose end points are its two clocks. Such an edge is labelled by the current state of the process. □

A c_2-*encoding* and its value $Val_2(\gamma)$ are defined in a similar manner (replacing the states fst_1, mid_1, and $last_1$ by fst_2, mid_2, and $last_2$ in the encoding).

A configuration $\gamma = (I, q, \mathcal{Q}, X)$ is said to be an *encoding* if the following two conditions are satisfied:

- $q = s$ for some $s \in S$, i.e., q is the copy of a local state of C.
- γ is both a c_1- and a c_2-encoding.

If γ satisfies the above conditions (i.e. if γ is an encoding), we define the *signature* $sig(\gamma)$ of γ to be the triple (s, m_1, m_2), where $m_1 = Val_1(\gamma)$ and $m_2 = Val_2(\gamma)$. Intuitively, the triple (s, m_1, m_2) will correspond to a configuration of C. Notice that several (in fact infinitely many) configurations may have the same signature. However, all such configurations will have the same local states and the same orderings on clock values, and therefore will correspond to the same configuration in C.

Proper Encodings In our simulation of C we shall rely on a particular kind of encodings, called *proper encodings*. An encoding γ of the form $(I, q, \mathcal{Q}, X)$ is said to be *proper* if it satisfies the following condition:

- For each $i \in I$ with $\mathcal{Q}(i) \neq idle^p$, $0 < X_1(i), X_2(i) < 1$.

In other words, all clocks participating in the encoding have values between (not including) zero and one. Certain steps of the simulation (see the decrementing operation in Section 8.5) are not possible to carry out without an upper bound on clock values of the processes. Working with proper encodings guarantees such an upper bound (namely an upper bound of one).

8.5 Encoding of Transitions

In this section, we perform the second step in deriving the timed network $\mathcal{N}_{\mathsf{C}} = (Q_{\mathsf{C}}, \Re_{\mathsf{C}})$ from the counter machine $\mathsf{C} = (S, s_{init}, \{c_1, c_2\}, \mathtt{I})$. More precisely, we describe the set of rules $\Re_{\mathsf{C}}$. The set $\Re_{\mathsf{C}}$ contains the following rules:

8.5.1 Incrementing

For each instruction $\imath = (s_1, c_1{+}{+}, s_2)$ in C there are two rules in $\Re_{\mathsf{C}}$, namely

$$inc_1^{\imath} : \left[\begin{array}{c} s_1 \\ \rightarrow \\ tmp^{\imath} \end{array}\right] \left[\begin{array}{c} fst_1 \\ 0 < x_1 \rightarrow \{x_1\} \\ mid_1 \end{array}\right] \left[\begin{array}{c} idle^p \\ tt \rightarrow \{x_2\} \\ fst_1 \end{array}\right]$$

and the rule

$$inc_2^{\imath} : \left[\begin{array}{c} tmp^{\imath} \\ \rightarrow \\ s_2 \end{array}\right] \left[\begin{array}{c} fst_1 \\ 0 < x_2 \rightarrow \{x_1\} \\ fst_1 \end{array}\right] \left[\begin{array}{c} last_1 \\ tt \rightarrow \{x_2\} \\ last_1 \end{array}\right]$$

The total effect of the two rules is to increment the value of a c_1-encoding by adding one more process to the list. The rule $inc_1^{\imath}$ changes the state of the process which is currently first in the list to mid_1. This process will be placed in the second position in the new encoding. At the same time a new process is picked from the set of idle processes, and its state is changed to fst_1. The new process will be placed first in the list. Furthermore, the rule resets (and therefore equates) clock x_1 and x_2 respectively of the two above mentioned processes. This is done in order to maintain the invariant that clock x_2 of each process (except the last process) is equal to clock x_1 of the next process (recall the definition of an encoding from Section 8.4). The result of applying rule $inc_1^{\imath}$ on a c_1-encoding of value 2 is shown in Figure 8.3(b).

Rule $inc_2^{\imath}$ resets clock x_1 of the process which is now in state fst_1 and clock x_2 of the process which is last in the list. This is done in order to maintain (i) the invariant that clock x_1 of a process (here the first process) is smaller than clock x_1 of the next process; and (ii) the invariant that clock x_2 of the last process is equal to clock x_1 of the first process. The result of applying rule $inc_2^{\imath}$ is shown in Figure 8.3(c).

Some remarks about rules $inc_1^{\imath}$ and $inc_2^{\imath}$:

- After execution of $inc_1^{\imath}$, the controller will be in state $tmp^{\imath}$ and therefore $inc_2^{\imath}$ is the only rule which may eventually be enabled after execution of $inc_1^{\imath}$.
- The guard $0 < x_1$ in the definition of $inc_1^{\imath}$ is to guarantee that all clocks have positive values before the rule is applied. This makes sure that we avoid the scenario where we "accidentally" equate some clocks with the ones which are reset during the application of $inc_1^{\imath}$. The same reasoning applies to the guard $0 < x_2$ in the definition of the rule $inc_2^{\imath}$. Similar guards exist in the rest of the rules described in this section.
- After application of $inc_2^{\imath}$, the resulting encoding will not be proper, since clocks have just been reset and their values are now zero. We can re-create a proper encoding by letting time pass through a timed transition. Again, a similar reasoning is applicable to the rest of the rules described in this section.

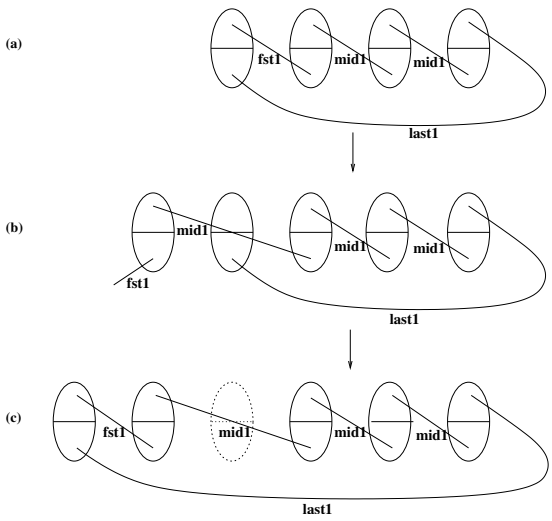

Figure 8.3: Simulating $(s_1, c_1{+}{+}, s_2)$ on a c_1-encoding

Also, for each instruction of the form $(s_1, c_2{+}{+}, s_2)$, there are two rules similar to the rules described above (replacing the states fst_1, mid_1, and $last_1$ by fst_2, mid_2 and $last_2$, respectively).

8.5.2 Decrementing

For each instruction $\imath = (s_1, c_1{-}{-}, s_2)$ in C there is a rule in $\Re_{\mathsf{C}}$, namely

$$dec^{\imath}: \left[\begin{array}{c} s_1 \\ \rightarrow \\ s_2 \end{array}\right] \left[\begin{array}{c} last_1 \\ x_1 = 1 \rightarrow \emptyset \\ idle^p \end{array}\right] \left[\begin{array}{c} fst_1 \\ 0 < x_1 \rightarrow \{x_1\} \\ fst_1 \end{array}\right] \left[\begin{array}{c} mid_1 \\ x_2 = 1 \rightarrow \{x_2\} \\ last_1 \end{array}\right]$$

The rule $dec^{\imath}$ decrements the value of a c_1-encoding by removing the last process of the list. More precisely, it changes the state of the last process to $idle^p$ (i.e. removes that process from the list), and changes the state of the process which is next last from mid_1 to $last_1$. In order to do that, we have to be able to identify the process which is next last in the list. Since all processes in the middle of the list are in state mid_1, we cannot identify the next last process simply by checking process states. Instead, we wait until the value of clock x_1 of the last process is equal to one. At that point of time, the process with clock x_2 equal to one is the next last process. Also, the rule resets (and therefore equates) clock x_1 of the first process and clock x_2 of the next last process (which will now become last in the list). Figure 8.4 shows the effect of applying the rule to a c_1-encoding.

Some remarks about the rule $dec^{\imath}$:

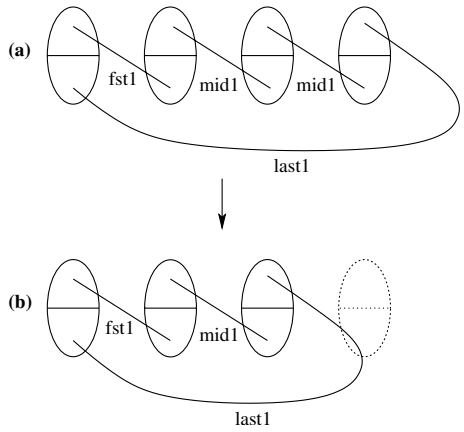

Figure 8.4: Simulating (s_1, c_1--, s_2) on a c_1-encoding

- Identifying the next last process (by waiting until some clocks are equal to one) uses the assumption that we start from a proper encoding. This implies that clocks of processes participating in the encoding have all values which are less than one. If this property is violated then the rule is not enabled (and will not become enabled through passage of time).

- The rule is not enabled in case the value of the c_1-encoding is equal to zero, since there will be no processes in state mid_1.

- Waiting for clock x_1 of the last process in the c_1-encoding to become equal to one may enforce clocks of processes in the c_2-encoding to become greater than one. More precisely, this happens if some clock in a process which is part of the c_2-encoding has a greater value than clock x_1 of the process which is currently in state $last_1$. After applying dec^ι, the value of such clocks will be greater than one, and therefore the resulting configuration will not be a *proper* encoding.

  Figure 8.5 illustrates this scenario. We consider a proper encoding (shown in Figure 8.5(a)) with signature $(s_1, 1, 1)$ such that clock x_1 of the process in state $last_1$ (0.35) is smaller than that of the process in state $last_2$ (0.65). In order to enable the rule dec^ι, we let time pass until clock x_1 of the process $last_1$ becomes equal to one (shown in Figure 8.5(b)). However, at this point of time, both clock x_2 of a process in state mid_2 and x_1 of the process in state $last_2$ have become larger than one (1.3). Therefore, after applying the dec^ι, we get an encoding (of value $(s_2, 0, 1)$) shown in Figure 8.5(c), which is not proper. This prevents any later application of decrementing and zero-testing rules.

  In order, to maintain the possibility of maintaining proper encodings in our simulation, we combine the rule dec^ι with the *rotation* rules described below.

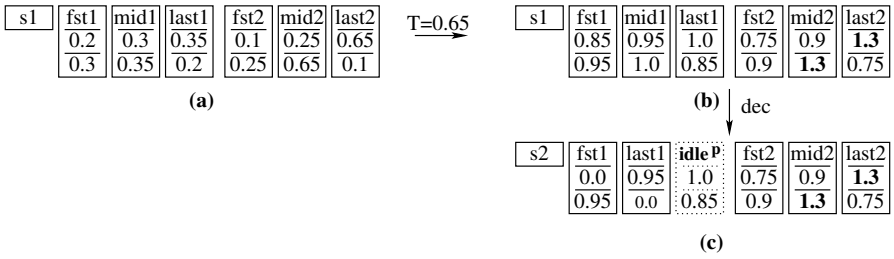

Figure 8.5: Decrementing may result in an improper encoding.

In a similar way to incrementing, there is also a rule corresponding to an instruction of the form $(s_1, c_2{-}{-}, s_2)$.

8.5.3 Rotation

To make it always possible to obtain a proper encoding after decrementing the value of a c_1- or a c_2-encoding (see the *decrementing* rule above), we add a set of *rotation* rules. More precisely, for each state $s \in S$, the set $\Re_{\mathsf{C}}$ contains the following two rules

$$rot_2^s : \begin{bmatrix} s \\ \rightarrow \\ s \end{bmatrix} \begin{bmatrix} fst_2 \\ 0 < x_1 \rightarrow \emptyset \\ mid_2 \end{bmatrix} \begin{bmatrix} last_2 \\ x_1 = 1 \rightarrow \{x_1\} \\ fst_2 \end{bmatrix} \begin{bmatrix} mid_2 \\ x_2 = 1 \rightarrow \{x_2\} \\ last_2 \end{bmatrix}$$

$$rotz_2^s : \begin{bmatrix} s \\ \rightarrow \\ s \end{bmatrix} \begin{bmatrix} fst_2 \\ (0 < x_1) \wedge (x_2 = 1) \rightarrow \{x_2\} \\ last_2 \end{bmatrix} \begin{bmatrix} last_2 \\ x_1 = 1 \rightarrow \{x_1\} \\ fst_2 \end{bmatrix}$$

Let us first explain the rule rot_2^s. The rule does not correspond to any instruction in C; nor does it change the signature of the encoding. In simulating C, we use the rotation rules in connection with decrementing. Recall that if $\imath = (s_1, c_1{-}{-}, s_2)$ then applying a rule $dec^\imath$ will not give a proper encoding in case the c_2-encoding has clocks with greater values than clock x_1 of the last process in the c_1-encoding (see Figure 8.5). The role of rot_2^s then is to decrement clock values of processes which are part of a c_2-encoding while preserving the signature of the whole encoding. More precisely, the rule rot_2^s moves the process which is in state $last_2$ and makes it first in the c_2-encoding. This amounts to a rotation of the list corresponding to the c_2-encoding. The rotation can be repeated until sufficiently many processes in the c_2-encoding have been moved. When there are no clocks in the c_2-encoding with greater clock values than clock x_1 of the last process in the c_1-encoding, the rotation stops and $dec^\imath$ can now safely be applied.

We illustrate the role of rot_2^s through Figure 8.6. In a similar manner to

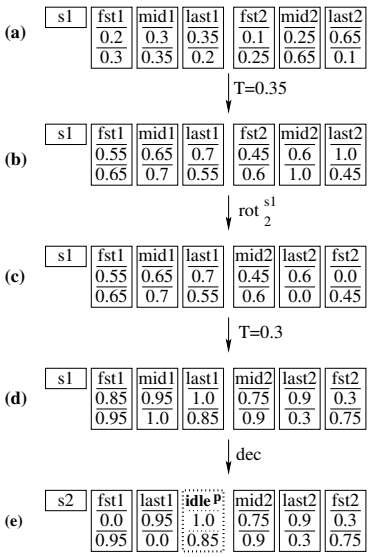

Figure 8.6: Decrementing preceded by rotation

Figure 8.5 we are interested in simulating a decrement instruction. However, instead of following the scenario of Figure 8.5, we now perform the following steps:

1. Wait for clock x_1 of the process in state $last_2$ to become equal to one (Figure 8.6(b)).
2. Apply the rule $rot_2^{s_1}$. This results in an encoding (shown in Figure 8.6(c)) where clock x_1 of the process in state $last_2$ (0.6) is smaller than that of the process in state $last_1$ (0.7).
3. Wait until clock x_1 of the process in state $last_1$ becomes one (Figure 8.6(d)).
4. Apply the *decrementing* rule. Notice that the resulting encoding (shown in Figure 8.6(e)) has all clock values less than one.

After the last step, we can perform a timed transition and obtain a proper encoding.

Also if, before applying dec^i, there is a clock in the c_2-encoding of the same value as clock x_1 of the process in state $last_1$, then we need to apply $rot_2^{s_2}$ once more after decrementing (this scenario does not occur in Figure 8.6, but is considered in the correctness proof).

Notice that we cannot apply the rotation rule in case the value of the c_2-encoding is zero. This is due to the fact that the rule requires at least one process in state mid_2. The rule $rotz_2^s$ has the same role as rot_2^s with the difference that it can be applied when the value of the c_2-encoding is zero.

There are also similar rules rot_1^s and $rotz_1^s$ which are used to rotate a c_1-encoding and which are used in connection with rules of the form $dec^\imath$ with $\imath = (s_1, c_2--, s_2)$.

8.5.4 Zero Testing

For each instruction $\imath = (s_1, c_1 = 0?, s_2)$ in C there is a rule in $\Re_\mathsf{C}$, namely

$$tst^\imath : \left[\begin{array}{c} s_1 \\ \rightarrow \\ s_2 \end{array}\right] \left[\begin{array}{c} fst_1 \\ (0 < x_1) \wedge (x_2 = 1) \rightarrow \{x_2\} \\ last_1 \end{array}\right] \left[\begin{array}{c} last_1 \\ x_1 = 1 \rightarrow \{x_1\} \\ fst_1 \end{array}\right]$$

The rule checks that the value of the encoding is zero by testing that there are no processes in state mid_1. This is done by verifying that the process which is next last in the list is the same as the process which is first in the list. We identify the next last process in a similar manner to the case with decrementing. More precisely, we wait until the value of clock x_1 of the last process is equal to one. At that moment, we check the process with clock x_2 equal to one, and check whether that process has a state equal to fst_1. Notice that, clock x_2 of the first process and clock x_1 of the last process are now both equal to one and the encoding is no more proper. In order to be able to obtain a proper encoding again, we reset both these clocks and interchange the states of the processes in states fst_1 and $last_1$ respectively.

Notice the similarity between the rules $tst^\imath$ and $rotz_1^s$.

Some remarks about the rule $tst^\imath$:

- The rule $tst^\imath$ is not enabled from an encoding γ with $sig(\gamma) = (s, m_1, m_2)$ and $m_1 \neq 0$, since, in such an encoding, values of clocks x_1 of the process in state $last_1$ and x_2 of the process in state fst_1 are different.
- Sometimes the rule $tst^\imath$ must be combined with the rotation rules according to the same scenarios explained for the *decrementing* rule.

In a similar way to the previous rules, there is also a rule corresponding to an instruction of the form $\imath = (s_1, c_2 = 0?, s_2)$.

8.5.5 Initialization

The initial phase consists of the following two rules.

$$init_1 : \left[\begin{array}{c} idle^c \\ \rightarrow \\ s'_{init} \end{array}\right] \left[\begin{array}{c} idle^p \\ tt \rightarrow \{x_2\} \\ fst_1 \end{array}\right] \left[\begin{array}{c} idle^p \\ tt \rightarrow \{x_2\} \\ fst_2 \end{array}\right] \left[\begin{array}{c} idle^p \\ tt \rightarrow \{x_1\} \\ last_1 \end{array}\right] \left[\begin{array}{c} idle^p \\ tt \rightarrow \{x_1\} \\ last_2 \end{array}\right]$$

$$init_2 : \begin{bmatrix} s'_{init} \\ \rightarrow \\ s_{init} \end{bmatrix} \begin{bmatrix} fst_1 \\ 0 < x_2 \rightarrow \{x_1\} \\ fst_1 \end{bmatrix} \begin{bmatrix} fst_2 \\ tt \rightarrow \{x_1\} \\ fst_2 \end{bmatrix} \begin{bmatrix} last_1 \\ tt \rightarrow \{x_2\} \\ last_1 \end{bmatrix} \begin{bmatrix} last_2 \\ tt \rightarrow \{x_2\} \\ last_2 \end{bmatrix}$$

The role of the initialization rules is to bring $\mathcal{N}_{\mathsf{C}}$ from its initial configuration (where the controller and all processes are idle) into a configuration which is an encoding of the initial configuration β_{init} of C. The rule $init_1$ takes the controller into the temporary state s'_{init}. It also picks four processes such that two processes become the first and last processes in the c_1-encoding (with value zero) and the other two processes become the first and last processes in the c_2-encoding (also with value zero). Clock x_2 of the first process and clock x_1 of the last process are reset. Rule $init_2$ changes the controller state to s_{init} and completes the creation of the c_1-encoding and c_2-encoding. This is done by first checking that some time has passed (through the guard $0 < x_2$), and then resetting both clock x_1 of the process which is now in state fst_1 (fst_2), and clock x_2 of the process which is now in state $last_1$ ($last_2$).

8.6 Correctness

In this section we prove the correctness of the construction described in Section 8.4 and Section 8.5.

Let $\mathsf{C} = (S, s_{init}, \{c_1, c_2\}, \mathtt{I})$ be a counter machine and let $\mathcal{N}_{\mathsf{C}} = (Q_{\mathsf{C}}, \Re_{\mathsf{C}})$ a timed network derived from C as described in Section 8.4 and Section 8.5. Let $\rightsquigarrow$ and $\longrightarrow$ be the transition relations induced by C and $\mathcal{N}_{\mathsf{C}}$ respectively.

In this section, we show that if s_F is a control state in C then the following holds.

Theorem 8.5. $\beta_{init} \overset{*}{\rightsquigarrow} s_F$ iff $\gamma_{init} \overset{*}{\longrightarrow} s_F$ for some initial configuration γ_{init} of $\mathcal{N}_{\mathsf{C}}$.

The proof of Theorem 8.5 is given in Subsections 8.6.1 and 8.6.2 each showing one direction of the equivalence.

In the proofs we need some definitions. Let $\mathcal{N} = (Q, \Re)$ be a timed network. We define $\Longrightarrow_r$ to denote $\longrightarrow_{Time} \circ \longrightarrow_r \circ \longrightarrow_{Time}$, i.e. $\Longrightarrow_r$ corresponds to performing a discrete transition according to the rule r, preceded and followed by a timed transition. We define $\Longrightarrow$ to be $\bigcup_{r \in \Re} \Longrightarrow_r$. For a set $\mathtt{R} \subseteq \Re$ of rules, we let $\gamma_1 \Longrightarrow_{\mathtt{R}} \gamma_2$ denote that $\gamma_1 \Longrightarrow_r \gamma_2$ for some $r \in \mathtt{R}$. We use $\overset{*}{\Longrightarrow}$, $\overset{*}{\Longrightarrow}_r$ and $\overset{*}{\Longrightarrow}_{\mathtt{R}}$ to denote the reflexive transitive closure of the respective relations.

We introduce a new type of encodings to describe the effect of the rule $inc_1^?$. For $m \geq 1$, a configuration $\gamma = (I, q, \mathcal{Q}, X)$ is said to be a *semi-c_1-encoding* of *value* m if there is an injection h from $\{0, \ldots, m+1\}$ to I such that the following conditions are satisfied

- $\mathcal{Q}$ is defined as in a c_1-encoding.
- $X_2(h(0)) = X_1(h(1)) < X_2(h(m+1)) < X_2(h(1))$.
- $X_1(h(i)) < X_1(h(i+1))$ for each $i : 2 \leq i \leq m$.
- $X_2(h(i)) = X_1(h(i+1))$, for each $i : 1 \leq i \leq m$.

A graphical representation of a semi-c_1-encoding is shown in Figure 8.3(b).

In a similar manner to a c_1-encoding, we use $Val_1(\gamma)$ to denote the value m of a semi-c_1-encoding γ.

A configuration $\gamma = (I, q, \mathcal{Q}, X)$ is said to be a *Type* 1 *semi-encoding* if it satisfies the following two conditions:

- $q = tmp^{\imath}$ for some increment (of the form $\imath = (s_1, c_1{+}{+}, s_2)$ or of the form $\imath = (s_1, c_2{+}{+}, s_2)$).
- γ is both a c_2-encoding and a semi-c_1-encoding.

In such a case, we define $sig(\gamma)$ of γ to be the triple $(tmp^{\imath}, m_1, m_2)$, where $m_1 = Val_1(\gamma)$ and $m_2 = Val_2(\gamma)$. Also, we define $next(\gamma)$ to be (s_2, m_1, m_2). Intuitively, $next(\gamma)$ is the signature of the configuration which occurs next time we perform a discrete transition (by rule $inc_2^{\imath}$) in our simulation.

The notion of a semi-encoding of Type 1 can be extended to a *proper* semi-encoding in the same manner as before, i.e. we require clocks of all processes which are not idle to have values strictly between zero and one. A (proper) semi-encoding of Type 2 is defined in a similar manner.

8.6.1 if-direction

The if-direction follows immediately from the following lemma.

Lemma 8.6. For any configuration $\gamma = (I, q, \mathcal{Q}, X)$ and initial configuration γ_{init} in $\mathcal{N}_{\mathsf{C}}$, if $\gamma_{init} \xrightarrow{*} \gamma$ then one of the following holds.

1. q is not a member of S, (i.e. q is either a temporary state or the state $idle^c$).
2. γ is an encoding such that $\beta_{init} \overset{*}{\rightsquigarrow} sig(\gamma)$.

Proof. Suppose that $\gamma_{init} \xrightarrow{*} \gamma$. If $\gamma_{init} \longrightarrow_{Time} \gamma$ then the result follows immediately. Otherwise, $\gamma_{init} \overset{*}{\Longrightarrow} \gamma$, i.e., there is a sequence

$$\gamma_{init} = \gamma_0 \Longrightarrow_{r_0} \gamma_1 \Longrightarrow_{r_1} \gamma_2 \Longrightarrow_{r_2} \cdots \Longrightarrow_{r_{n-1}} \gamma_n = \gamma$$

Let $\gamma_i = (I, q_i, \mathcal{Q}_i, X_i)$ for $i : 0 \leq i \leq n$. We notice that $q_0 = idle^c$. By definition of the rules, it must be the case that $r_0 = init_1$ and therefore $q_1 = s'_{init}$. In other words, both γ_0 and γ_1 satisfy the claim of the Lemma. Lemma 8.6 follows from the following property:

For each $2 \leq i \leq n$, it is the case that γ_i is either

- an encoding with $\beta_{init} \overset{*}{\rightsquigarrow} sig(\gamma_i)$; or
- a semi-encoding with $\beta_{init} \overset{*}{\rightsquigarrow} next(\gamma_i)$.

This property is shown using an induction on i. For the base case we observe that, by definition of the rules, it follows that $r_1 = init_2$ and therefore $sig(\gamma_2) = \beta_{init}$. For the induction step, we observe that, for each $i : 2 \leq i < n$, it follows from the rule definitions that one of the following cases is satisfied:

1. $r_i = inc_1^\imath$ for some $\imath = (s_1, c_1{+}{+}, s_2)$, γ_i is an encoding with $sig(\gamma_i) = (s_1, m_1, m_2)$, and γ_{i+1} is a Type 1 semi-encoding with $sig(\gamma_{i+1}) = (tmp^\imath, m_1 + 1, m_2)$.
2. $r_i = inc_2^\imath$ for some $\imath = (s_1, c_1{+}{+}, s_2)$, γ_i is a Type 1 semi-encoding with $sig(\gamma_i) = (tmp^\imath, m_1, m_2)$, and γ_{i+1} is an encoding with $sig(\gamma_{i+1}) = (s_2, m_1, m_2)$.
3. $r = dec^\imath$ for some $\imath = (s_1, c_1{-}{-}, s_2)$, γ_i is an encoding with $sig(\gamma_i) = (s_1, m_1, m_2)$, $m_1 > 0$, and γ_{i+1} is an encoding with $sig(\gamma_{i+1}) = (s_2, m_1 - 1, m_2)$.
4. $r = rot_1^s$ for some $s \in S$ and $sig(\gamma_i) = sig(\gamma_{i+1})$.
5. $r = rotz_1^s$ for some $s \in S$ and $sig(\gamma_i) = sig(\gamma_{i+1})$.
6. $r = tst^\imath$ for some $\imath = (s_1, c_1 = 0?, s_2)$, with $sig(\gamma_i) = (s_1, 0, m_2)$ and $sig(\gamma_{i+1}) = (s_2, 0, m_2)$.
7. Similar cases corresponding to instructions which change counter c_2.

□

8.6.2 only-if direction

The only-if-direction follows from the following lemma.

Lemma 8.7. If $\beta_{init} \overset{n}{\rightsquigarrow} s_F$ then $\gamma_{init} \overset{*}{\longrightarrow} s_F$, for each $n \geq 0$ and initial configuration γ_{init} of $\mathcal{N}$ with $|\gamma_{init}| \geq n + 4$.

The reason for the condition $|\gamma_{init}| \geq n + 4$ is that the sum of counter values never exceeds n in the path from β_{init} to s_F. Furthermore, each c_1- (or c_2)-encoding uses $m + 2$ processes for representing a counter value m. The lemma then states that the initial configuration, from which we start the simulation of the path from β_{init} to s_F, should be sufficiently large to incorporate all counter values which arise along that path.

Proof of Lemma 8.7

To show Lemma 8.7 we use some definitions.

Let $\gamma = (I, q, \mathcal{Q}, X)$ be a (semi-)encoding. Let $i \in I$ be the (unique) index such that $\mathcal{Q}(i) = last_1$. We define $Latest_1(\gamma) = X_1(i)$. In other words, $Latest_1(\gamma)$ is the highest among values of clocks belonging to processes which are part of the (semi-)c_1-encoding. We define $Latest_2(\gamma)$ in a similar manner, and define $Latest(\gamma) = max(Latest_1(\gamma), Latest_2(\gamma))$.

Let $Delay_1(\gamma)$ be the size of the set $J \subseteq I$ such that $j \in J$ iff $\mathcal{Q}(j) \in \{fst_2, mid_2, last_2\}$ and $Latest_1(\gamma) < X_1(j)$. In other words, $Delay_1(\gamma)$ is the number of processes which are part of the c_2-encoding and which have clocks with values higher than any clock of a process which is part of the c_1-encoding. We define $Delay_2(\gamma)$ in a similar manner. Notice that it may be the case that both $Delay_1(\gamma) = 0$ and $Delay_2(\gamma) = 0$ (if the maximum clock values are equal in the c_1- and the c_2-encoding).

Lemma 8.7 follows immediately from the following lemma.

Lemma 8.8. For each $n \geq 0$ and initial configuration γ_{init}, if $\beta_{init} \overset{n}{\rightsquigarrow} \beta$ and $|\gamma_{init}| \geq n + 4$, then there exists a proper encoding γ such that $\gamma_{init} \overset{*}{\Longrightarrow} \gamma$ and $sig(\gamma) = \beta$.

Proof. We prove this lemma by induction on n.

In the base case ($n = 0$), we have $\beta = \beta_{init}$ and $|\gamma_{init}| \geq 4$. By the definition of $init_1$, this rule is enabled. Let γ_1 be such that $\gamma_{init} \longrightarrow_{init_1} \gamma_1$. Define $\gamma_2 = \gamma_1^{+\delta_1}$ with $0 < \delta_1 < 1$. We have $\gamma_1 \longrightarrow_{T=\delta_1} \gamma_2$. Rule $init_2$ is now enabled. Let γ_3 be such that $\gamma_2 \longrightarrow_{init_2} \gamma_3$. By definition of $init_2$, γ_3 is an encoding and $sig(\gamma_3) = \beta_{init}$. Let δ_2 be such that $0 < \delta_2 < 1 - Latest(\gamma_3)$. δ_2 exists by the definition of δ_1, $init_1$ and $init_2$. Let $\gamma_4 = \gamma_3^{+\delta_2}$. γ_4 is a proper encoding with $sig(\gamma_4) = \beta_{init}$. Notice that the transitions $\longrightarrow_{init_1}$ and $\longrightarrow_{init_2}$ are enabled only because $|\gamma_{init}| \geq 4$.

For the induction step, assume that $\beta_{init} \overset{n+1}{\rightsquigarrow} \beta$ and $|\gamma_{init}| \geq n + 5$. We know that there is a β_1 with $\beta_{init} \overset{n}{\rightsquigarrow} \beta_1 \rightsquigarrow \beta$. By the induction hypothesis, it follows that there is a proper encoding γ_1 such that $sig(\gamma_1) = \beta_1$ and $\gamma_{init} \overset{*}{\Longrightarrow} \gamma_1$. We need to show that there is a proper encoding γ with $sig(\gamma) = \beta$ and $\gamma_1 \overset{*}{\Longrightarrow} \gamma$. This follows from the following lemma. □

Lemma 8.9. Let β_1 and β_2 be configurations of C, where $\beta_1 \rightsquigarrow \beta_2$ and β_2 is of the form (s, m_1, m_2). Let γ_1 be a proper encoding such that $sig(\gamma_1) = \beta_1$ and $|\gamma_1| \geq m_1 + m_2 + 4$. There is a proper encoding γ_2 such that $sig(\gamma_2) = \beta_2$ and $\gamma_1 \overset{*}{\Longrightarrow} \gamma_2$.

The proof of Lemma 8.9 follows from Lemma 8.10, Lemma 8.11, Lemma 8.14, and Lemma 8.15:

- Lemma 8.10 and Lemma 8.11 state that an *increment* can be simulated by an application of the rule inc_1^ι followed by an application of the rule inc_2^ι.

- Lemma 8.14 states that a *decrement* can be simulated by the rule $dec^\imath$ possibly preceded and followed by a number of rotations. This lemma follows from Lemma 8.12 and Lemma 8.13.
- Lemma 8.15 deals with zero testing and is similar to Lemma 8.14.

The condition $|\gamma_1| \geq m_1 + m_2 + 4$ in the claim of Lemma 8.9 is relevant only in Lemma 8.10, since this is the only case where the value of a counter is increased.

Lemma 8.10. Consider an instruction $\imath = (s_1, c_1{+}{+}, s_2)$. Let γ_1 be a proper encoding with $sig(\gamma_1) = (s_1, m_1, m_2)$ and $|\gamma_1| \geq m_1 + m_2 + 5$. There is a proper semi-encoding γ_2 of Type 1 such that $sig(\gamma_2) = (tmp^\imath, m_1 + 1, m_2)$ and $\gamma_1 \Longrightarrow_{inc_1^\imath} \gamma_2$.

A similar result holds in case $\imath$ is of the form $(s_1, c_2{+}{+}, s_2)$.

Proof. Since $|\gamma_1| \geq m_1 + m_2 + 5$, there is at least one process in γ_1 whose state is $idle^p$ (we need $m_1 + 2$ processes for the c_1-encoding and $m_2 + 2$ processes for the c_2-encoding, which means that we have at least one process left to be in state $idle^p$). This together with the fact that γ_1 is a proper encoding implies that $inc_1^\imath$ is enabled from γ_1, i.e., there is configuration γ_3 with $\gamma_1 \longrightarrow_{inc_1^\imath} \gamma_3$. Define $\gamma_2 = \gamma_3^{+\delta}$ where $0 < \delta < 1 - Latest(\gamma_3)$. Such a δ exists by definition of $inc_1^\imath$ and since γ_1 is a proper encoding. By the definitions it follows that γ_2 is a proper semi-encoding of Type 1 with $sig(\gamma_2) = (tmp^\imath, m_1 + 1, m_2)$ and $\gamma_1 \longrightarrow_{inc_1^\imath} \gamma_3 \longrightarrow_{T=\delta} \gamma_2$. □

Lemma 8.11. Consider an instruction $\imath = (s_1, c_1{+}{+}, s_2)$. Let γ_1 be a proper semi-encoding of Type 1 with $sig(\gamma_1) = (tmp^\imath, m_1, m_2)$. There is a proper encoding γ_2 such that $sig(\gamma_2) = (s_2, m_1, m_2)$ and $\gamma_1 \Longrightarrow_{inc_2^\imath} \gamma_2$.

A similar result holds in case $\imath$ is of the form $(s_1, c_2{+}{+}, s_2)$.

Proof. Since γ_1 is a proper semi-encoding of Type 1, it follows that $inc_2^\imath$ is enabled from γ_1, i.e., there is a configuration γ_3 with $\gamma_1 \longrightarrow_{inc_2^\imath} \gamma_3$. Define $\gamma_2 = \gamma_3^{+\delta}$ where $0 < \delta < 1 - Latest(\gamma_3)$. Such a δ exists by definition of $inc_2^\imath$ and since γ_1 is proper semi-encoding. By the definitions it follows that γ_2 is a proper encoding with $sig(\gamma_2) = (s_2, m_1, m_2)$ and $\gamma_1 \longrightarrow_{inc_2^\imath} \gamma_3 \longrightarrow_{T=\delta} \gamma_2$. □

Lemma 8.12. Let γ_1 be a proper encoding with $Delay_1(\gamma_1) > 0$ and $sig(\gamma_1) = (s, m_1, m_2)$. There is a proper encoding γ_2 such that $Delay_1(\gamma_2) = Delay_1(\gamma_1) - 1$, $sig(\gamma_1) = sig(\gamma_2)$, and either $\gamma_1 \Longrightarrow_{rot_2^s} \gamma_2$ or $\gamma_1 \Longrightarrow_{rotz_2^s} \gamma_2$.

A similar result holds in case $Delay_2(\gamma_1) > 0$.

Proof. We distinguish between two cases, namely when $m_2 > 0$ and when $m_2 = 0$.

First, we assume that $m_2 > 0$. Define $\gamma_3 = \gamma_1^{+\delta_1}$ where $\delta_1 = 1 - Latest_2(\gamma_1)$. Such a δ_1 exists since γ_1 is a proper encoding. From the definition of δ_1 and the fact that $m_2 > 0$ it follows that rot_2^s is enabled from γ_3, i.e., there is a γ_4 with $\gamma_3 \longrightarrow_{rot_2^s} \gamma_4$. Define $\gamma_2 = \gamma_4^{+\delta_2}$ where $0 < \delta_2 < 1 - Latest(\gamma_4)$. Existence of δ_2 follows from the manner in which δ_1 is chosen, definition of the rule rot_2^s,

and since γ_1 is a proper encoding. By the definitions it follows that γ_2 is a proper encoding with $Delay_1(\gamma_2) = Delay_1(\gamma_1) - 1$, and $sig(\gamma_2) = sig(\gamma_1)$.

The case when $m_2 = 0$ is similar. Here we replace the rule $rot_2^{s_2}$ by the rule $rotz_2^{s_2}$, and obtain $\gamma_1 \longrightarrow_{T=\delta_1} \gamma_3 \longrightarrow_{rotz_2^s} \gamma_4 \longrightarrow_{T=\delta_2} \gamma_2$. □

Lemma 8.13. Consider an instruction $\imath = (s_1, c_1--, s_2)$. Let γ_1 be a proper encoding with $sig(\gamma_1) = (s_1, m_1, m_2)$ and $m_1 > 0$. If $Delay_1(\gamma_1) = 0$ then there is a proper encoding γ_2 such that $sig(\gamma_2) = (s_2, m_1 - 1, m_2)$ and one of the following holds.

1. If $Latest_1(\gamma_1) > Latest_2(\gamma_1)$ then $\gamma_1 \Longrightarrow_{dec^\imath} \gamma_2$.
2. If $Latest_1(\gamma_1) = Latest_2(\gamma_1)$ and $m_2 > 0$ then $\gamma_1 \Longrightarrow_{dec^\imath} \circ \Longrightarrow_{rot_2^{s_2}} \gamma_2$.
3. If $Latest_1(\gamma_1) = Latest_2(\gamma_1)$ and $m_2 = 0$ then $\gamma_1 \Longrightarrow_{dec^\imath} \circ \Longrightarrow_{rotz_2^{s_2}} \gamma_2$.

A similar result holds in case $\imath$ is of the form (s_1, c_2--, s_2).

Proof. Define $\gamma_3 = \gamma_1^{+\delta_1}$ where $\delta_1 = 1 - Latest_1(\gamma_1)$. Such a δ_1 exists since γ_1 is a proper encoding. From the definition of δ_1 and the fact that $m_1 > 0$ it follows that $dec^\imath$ is enabled from γ_3, i.e., there is a γ_4 with $\gamma_3 \longrightarrow_{dec^\imath} \gamma_4$. Now there are three cases depending on the values of $Latest_1(\gamma_1)$ and $Latest_2(\gamma_1)$ as follows:

1. If $Latest_1(\gamma_1) > Latest_2(\gamma_1)$ then define $\gamma_2 = \gamma_4^{+\delta_2}$ where $0 < \delta_2 < 1 - Latest(\gamma_4)$. Existence of δ_2 follows from the manner in which δ_1 is chosen, definition of the rule $dec^\imath$, and since γ_1 is a proper encoding. By the definitions it follows that γ_2 is a proper encoding with $sig(\gamma_2) = (s_2, m_1 - 1, m_2)$, and $\gamma_1 \longrightarrow_{T=\delta_1} \gamma_3 \longrightarrow_{dec^\imath} \gamma_4 \longrightarrow_{T=\delta_2} \gamma_2$.
2. If $Latest_1(\gamma_1) = Latest_2(\gamma_1)$ and $m_2 > 0$. It follows that clock x_1 of the last process in the c_2-encoding has value 1 in γ_4. Therefore, the rule $rot_2^{s_2}$ is enabled from γ_4, i.e., there is a γ_5 with $\gamma_4 \longrightarrow_{rot_2^{s_2}} \gamma_5$. Define $\gamma_2 = \gamma_5^{+\delta_2}$ where $0 < \delta_2 < 1 - Latest_1(\gamma_5)$. Existence of δ_2 follows from the manner in which t_1 is chosen, definitions of the rules $dec^\imath$ and $rot_2^{s_2}$, and since γ_1 is a proper encoding. By the definitions it follows that γ_2 is a proper encoding with $sig(\gamma_2) = (s_2, m_1 - 1, m_2)$, and $\gamma_1 \longrightarrow_{T=\delta_1} \gamma_3 \longrightarrow_{dec^\imath} \gamma_4 \longrightarrow_{rot_2^{s_2}} \gamma_5 \longrightarrow_{T=\delta_2} \gamma_2$.
3. If $Latest_1(\gamma_1) = Latest_2(\gamma_1)$, but $m_2 = 0$. The proof is similar to the previous case. Here, we use the rule $rotz_2^{s_2}$ instead of the rule $rot_2^{s_2}$ and obtain $\gamma_1 \longrightarrow_{T=\delta_1} \gamma_3 \longrightarrow_{dec^\imath} \gamma_4 \longrightarrow_{rotz_2^{s_2}} \gamma_5 \longrightarrow_{T=\delta_2} \gamma_2$.

□

From Lemma 8.12 and Lemma 8.13 we get the following.

Lemma 8.14. Consider an instruction $\imath = (s_1, c_1{-}{-}, s_2)$. Let γ_1 be a proper encoding with $sig(\gamma_1) = (s_1, m_1, m_2)$ and $m_1 > 0$. There is a proper encoding γ_2 such that $sig(\gamma_2) = (s_2, m_1 - 1, m_2)$ and

$$\gamma_1 \stackrel{*}{\Longrightarrow}_{\{rot_2^{s_1}, rotz_2^{s_1}\}} \circ \Longrightarrow_{dec^\imath} \circ \stackrel{*}{\Longrightarrow}_{\{rot_2^{s_2}, rotz_2^{s_2}\}} \gamma_2$$

A similar result holds in case $\imath$ is of the form $(s_1, c_2{-}{-}, s_2)$.

The proof of the following lemma is similar to the proof of Lemma 8.14.

Lemma 8.15. Consider an instruction $\imath = (s_1, c_1 = 0?, s_2)$. Let γ_1 be a proper encoding with $sig(\gamma_1) = (s_1, 0, m_2)$. Then there is a proper encoding γ_2 such that $sig(\gamma_2) = (s_2, 0, m_2)$ and

$$\gamma_1 \stackrel{*}{\Longrightarrow}_{\{rot_2^{s_1}, rotz_2^{s_1}\}} \circ \Longrightarrow_{tst^\imath} \circ \stackrel{*}{\Longrightarrow}_{\{rot_2^{s_2}, rotz_2^{s_2}\}} \gamma_2$$

A similar result holds in case $\imath$ is of the form $(s_1, c_2 = 0?, s_2)$.

8.7 Remark

The termination of the algorithm for TN(1)-Reach given in [9] depends on the theory of well-quasi-ordering. They give a backward analysis for single-clock timed networks, similar to that shown in Chapter 4. The paper [9] uses constraints similar to the regions shown for backward analysis for TPNs (Chapter 4). Backward analysis also needs that the ordering on the constraints for the model to be a well-quasi ordering. To use backward analysis for multi-clock timed networks, one might think of extending the constraint system of TN(1) by additionally requiring that clocks belonging to the same process have edges connecting them. In the following, we show an example of such a possible constraint for TN(2).

Example 8.16. Consider $max = 1$. Figure 8.7(a) shows such a constraint for TN(2) with three processes and Figure 8.7(b) shows a configuration satisfying such a constraint. The upper part of each circle in the constraint contains pairs of the form (process state, integral part of x_1 clock of the process) and its lower part contains pairs of the form (process state, integral part of x_2 clock of the process). Edges connect clocks belonging to the same process. □

Then the ordering $\sqsubseteq$ for TN(K) should satisfy the following requirement. If the pairs $p(k_1), p(k_2)$ in c_1 are mapped to pairs $p(k_3)$ and $p(k_4)$ respectively in c_2 and the pair $p(k_1)$ is connected to $p(k_2)$ by an edge, then there must be an edge between the pairs $p(k_3)$ and $p(k_4)$.

Now we show that there is an infinite sequence $\mathsf{c}_1 \not\sqsubseteq \mathsf{c}_2 \not\sqsubseteq \ldots$ of such constraints for TN(2) (shown in Figure 8.8) which violates the well-quasi-ordering property.

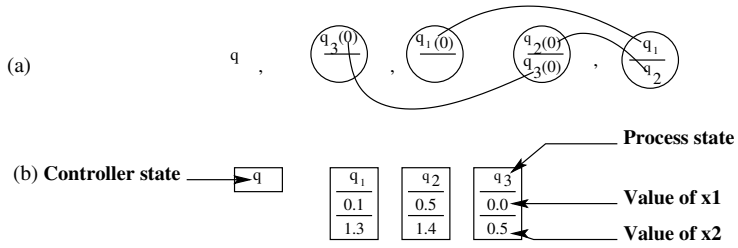

Figure 8.7: (a) A constraint for TN(2). (b) A configuration of TN(2) satisfying the constraint.

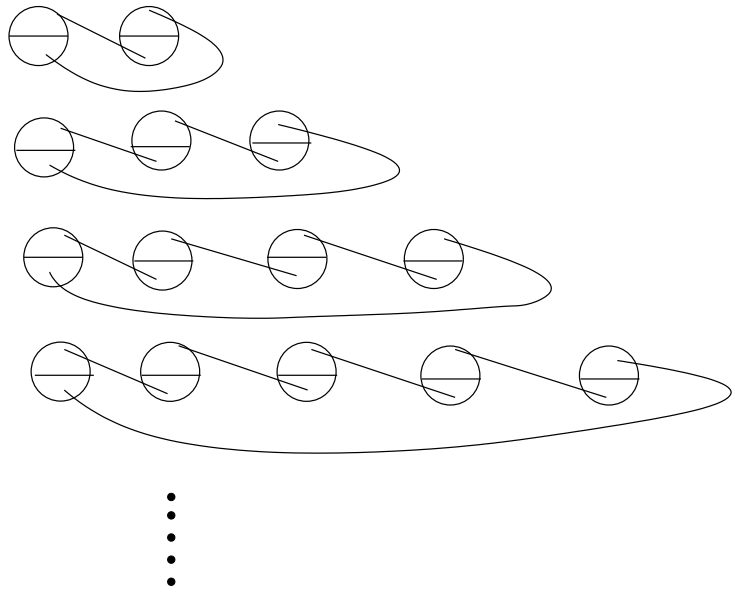

Figure 8.8: Graphical representation of an infinite non-well-quasi ordered sequence of constraints of TN(2).

8.8 Related Work

Most works on verification of parameterized systems consider the case where each component is a *finite-state system*. Applications include cache coherence protocols [52, 57], broadcast protocols [64], mutual exclusion protocols with linear topologies [94], etc.

In [24] a method is given for translating a timed automaton with several clocks into the parallel composition of a finite number of automata each operating on a single clock. This may give the impression that reachability problems for multi-clock TNs can in a similar way be reduced to corresponding problems for single-clock TNs. However, the construction given in [24] will not work in our case. The reason is that, due to the unbounded number of timed processes, it

is not possible to keep track of clocks belonging to the same process.

The works in [21, 20] consider timed automata which are parameterized in the the following sense: transitions are guarded with predicates which compare clocks (and counters) with parameters possibly ranging over infinite domains. The models used in these papers assume a finite number of clocks and are therefore orthogonal to the models considered in this thesis.

We assume a lazy behaviour for TNs. This means that we may choose to let time pass instead of performing discrete transitions, even if that makes these transitions disabled, due to some of the clocks becoming "too old". In fact, we can use the techniques in [89] to show that, in the case of urgent behaviour, the controller state reachability is undecidable even for single-clock TNs.

Chapter 9

Discrete-Timed Networks

In the last chapter, we showed the undecidability of controller state reachability problem for timed networks with just 2 clocks per process. In this chapter, we show that the problem becomes decidable when the clocks range over a discrete time domain. We consider the controller state reachability problem for *Discrete Timed Networks (DTNs)* : timed networks in which the clocks assume values from the set of natural numbers. The decidability result holds regardless of the number of clocks allowed inside each timed process. We show decidability using the theory introduced in [9, 14] for verification of well-quasi-ordered constraint systems. The idea of the proof is to define an ordering on configurations of the DTN. The ordering amounts to *counter abstraction*: for each configuration we count the number of processes which are in a given state and whose clocks are equal to some given values. We show that such an abstraction induces a well quasi-ordering.

9.1 Discrete Timed Networks (DTN)

The syntax of a DTN is the same as that of a TN (see Section 8.1). A configuration is also of the same form as in a TN. The behaviour of a DTN differs from that of a TN in two aspects, namely

- In a configuration $(I, q, \mathcal{Q}, X)$, the type of X is $\{1, \ldots, K\} \rightarrow I \rightarrow \mathbb{N}$, i.e., clocks have values which are natural numbers rather than reals.
- Timed transitions take only discrete steps, i.e., $\gamma_1 \longrightarrow_{T=t} \gamma_2$ if $\gamma_2 = \gamma_1^{+t}$ where $t \in \mathbb{N}$. Discrete transitions are defined in a similar manner to TN.

Thus, the transition relation $\longrightarrow$ defined in Section 8.1 is adapted to DTN as described above.

9.1.1 Ordering

We define an ordering $\sqsubseteq$ on the set of configurations Γ of a DTN $\mathcal{N}$ with K clocks as follows. To define $\sqsubseteq$, we first introduce a function

$$\natural \; : \; Q^{proc} \times \{0, \ldots, max + 1\}^K \; \rightarrow \; \Gamma \; \rightarrow \; \mathbb{N}$$

where max is the maximum natural number which occurs in the definitions of the rules in the DTN. Given $p \in Q^{proc}$, $m_1, \ldots, m_K \in \{0, \ldots, max+1\}$, and $\gamma = (I, q, \mathcal{Q}, X)$ we define $\natural(p, m_1, \ldots, m_K)(\gamma)$ to be the size of the largest set $I_1 \subseteq I$ such that for each index $i \in I_1$, $\mathcal{Q}(i) = p$ and for each $k : 1 \leq k \leq K$, one of the following conditions holds

- $X(k)(i) = m_k$ and $m_k \leq max$.
- $X(k)(i) > m_k$ and $m_k = max + 1$.

In other words, $\natural(p, m_1, \ldots, m_K)(\gamma)$ counts the number of processes in γ whose states are p and whose clock values are given by $m_1, \ldots, m_K$ respectively (we identify clock values larger than max). For $\gamma = (I, q, \mathcal{Q}, X)$ and $\gamma' = (I', q', \mathcal{Q}', X') \in \Gamma$, we use $\gamma \sqsubseteq \gamma'$ to denote that the following two conditions hold:

- $q = q'$
- $\natural(p, m_1, \ldots, m_K)(\gamma) \leq \natural(p, m_1, \ldots, m_K)(\gamma')$ for each $p \in Q^{proc}$, $m_1, \ldots, m_K \in \{0, \ldots, max+1\}$.

Thus, the ordering $\sqsubseteq$ is trivially computable. Well quasi-ordering of $\sqsubseteq$ follows from Dickson's Lemma [54]: according to the ordering, we can represent a configuration by a tuple *(controller state, vector of natural numbers)*, where each entry of the vector is indexed by a tuple of the form $(p, m_1, \ldots, m_K)$ and the set of states of the controller and the set of vectors of natural numbers are both well-quasi-ordered.

Lemma 9.1. The ordering $\sqsubseteq$ on the set of configurations of a DTN is a well-quasi-ordering.

The Problem DTN(K)-Reach is defined in the same manner as TN(K)-Reach except that the timed network $\mathcal{N}$ in the definition of the problem is now given a discrete interpretation as described above.

In this section we show the following.

Theorem 9.2. DTN(K)-Reach is decidable for each $K \in \mathbb{N}$.

The proof follows from the following observation. In Chapter 4, it is shown that the following conditions are sufficient for decidability of coverability problem for DTNs.

- A constraint c of TN is syntactically the same as a configuration of $\mathcal{N}$. However, $[\![\mathsf{c}]\!]^{\uparrow} = \{\mathsf{c}\} \uparrow$ is an upward closed set of configurations where $\{\mathsf{c}\} \uparrow$ is upward closed with respect to $\sqsubseteq$.
- For a constraint c , the set $Pre(\{\mathsf{c}\})$ is finite and computable (shown in Lemma 9.3 in the following). For a set $\mathbb{C}$ of constraints $Pre(\mathbb{C}) = \bigcup_{\mathsf{c} \in \mathbb{C}} Pre(\mathbb{C})$.

- The ordering $\sqsubseteq$ on the set of constraints is decidable (shown above) and it is a well-quasi-ordering (Lemma 9.1).

Lemma 9.3. For each constraint c, we can compute $Pre(\mathsf{c})$ as a finite set of constraints.

Proof. Given a constraint c, we compute $Pre(\mathsf{c})$ as follows. Since, $\longrightarrow = \longrightarrow_{Time} \cup \bigcup_{r \in \Re} \longrightarrow_r$, we compute $Pre(\mathsf{c}) = Pre_{\mathsf{T}}(\mathsf{c}) \cup \bigcup_{r \in \Re} Pre_{\mathsf{r}}(\mathsf{c})$ where $Pre_T(\mathsf{c})$ characterizes the set of configurations from which we reach $[\![\mathsf{c}]\!]^{\uparrow}$ through passage of time and $Pre_r(\mathsf{c})$ does the same through execution of a rule r.

For a constraint $\mathsf{c} = (\mathsf{I}, \mathsf{q}, \mathcal{Q}, \mathsf{X})$, we define $minc(\mathsf{c})$ to be the minimum value in the set $\{X_k(i) \mid 1 \leq k \leq K \ \wedge \ i \in I\}$, i.e, $minc(\mathsf{c})$ denotes the minimum clock value which occurs in c. We compute $Pre_T(\mathsf{c})$ as follows.

$$Pre_T(\mathsf{c}) = \left\{ \mathsf{c}^{-\delta} \mid 0 \leq \delta \leq minc(\mathsf{c}) \right\}$$

where $\mathsf{c}^{-\delta}$ is defined in a manner similar to the definition of $\gamma^{+\delta}$ (recall timed transitions in Chapter 8).

Next, we show how to compute Pre_r. Given a constraint $\mathsf{c} = \langle \mathsf{I}', \mathsf{q}'_0, \mathcal{Q}', \mathsf{X}' \rangle$ and a rule $r \in \Re$ of the form

$$\begin{bmatrix} q_0 \\ \rightarrow \\ q'_0 \end{bmatrix} \quad \begin{bmatrix} q_1 \\ g_1 \rightarrow R_1 \\ q'_1 \end{bmatrix} \quad \cdots \quad \begin{bmatrix} q_n \\ g_n \rightarrow R_n \\ q'_n \end{bmatrix}$$

we compute a finite set $\mathbb{C}$ of constraints such that $Pre_r(\mathsf{c}) = \mathsf{min}(\mathbb{C})$. We define $\mathbb{C}$ to be the set of all constraints of the form $\langle I, q_0, \mathcal{Q}, X \rangle$ such that there are three pairwise disjoint sets *changing*, *unchanged* and *guarding* of indices and the following holds.

- $I' = changing \cup unchanged$,
- $I = changing \cup unchanged \cup guarding$, and
- there is a bijection $h : changing \cup guarding \rightarrow \{1, \ldots, n\}$, which satisfies the following conditions.
  1. $\mathcal{Q}(j) = q_{h(j)}$, and the guard $g_{h(j)}(X_1(j), \ldots, X_K(j))$ holds. Furthermore, $X(k)(j) \in \{0, \ldots, max + 1\}$, for each $j \in (changing \cup guarding)$ and for each $k : 1 \leq k \leq K$.
  2. $\mathcal{Q}'(j) = q'_{h(j)}$ for $j \in changing$.
  3. $\mathcal{Q}'(j) = \mathcal{Q}(j)$ for $j \in unchanged$.
  4. For each $k : 1 \leq k \leq K$, $X'_k(j) = 0$ if $j \in changing$ and $x_k \in R_{h(j)}$.
  5. For each $k : 1 \leq k \leq K$ and for each j such that either $j \in changing$ and $x_k \notin R_{h(j)}$, or $j \in unchanged$, $X_k(j) = min(max + 1, X'_k(j))$.

Observe that

- The set *guarding* of indices with size m can correspond to infinitely many sets of the same size. We equate all of them modulo renaming. Therefore, the sets *changing*, *unchanged*, *guarding* are finite and effectively constructible.
- The conditions on the controller states are implicitly included by our notation, which requires the controller states of c and c' to be the controller states of r.
- The correctness of the above algorithm can be shown in a similar manner to the proofs in [9].

□

9.2 Complexity

Next we show the complexity of the coverability problem for DTNs using the results for lossy counter machines.

Theorem 9.4. $DTN(1)$-Reach has nonprimitive recursive complexity.

First we recall the definitions of lossy counter machines from Chapter 7. The problem LCM-Reach is defined in the same manner as CM-Reach.

Theorem 9.5. Coverability and termination for lossy counter machines have nonprimitive recursive complexity.

Using the technique in [125] for lossy channel systems, it is straightforward to prove Theorem 9.5 (as claimed in [125]).

Now we sketch the proof of Theorem 9.5. The idea is to construct a timed network simulating a LCM. The construction is very similar to that in Chapter 7 and omitted here. Additionally, we use an initialization rule similar to those in Chapter 8. We can show the correctness of this construction as in Chapter 8. This construction and Theorem 9.5 imply Theorem 9.4.

Remark: Notice that the algorithm for verifying parameterized systems given by [71] has double exponential time as the worst-case complexity in contrast with the non-primitive recursive complexity for analysing parameterized timed systems.

9.3 Discussions

We have shown decidability of the problem when clocks are interpreted over a discrete time domain.

The ordering we provide for proving decidability of DTN corresponds to an abstraction of configurations where we count the number of processes which are in a certain state and which have certain clock values. In a similar manner to

[71] we can view this abstraction as a "Petri net"-like model where each place corresponds to one combination of process states and clock values. In contrast to [71], the transitions in the abstract model do not correspond to those of a Petri net. The main difference is that a timed transition simultaneously moves all tokens from each place, corresponding to a certain clock value, to the place corresponding to the next clock value. Comparing to the model of *Transfer Nets* [68], a timed transition here corresponds to "parallel transfers", i.e. a set of transfers which are performed simultaneously. An alternative way to prove our decidability result would be to simulate a DTN by a transfer net. One ingredient in such a simulation is to simulate parallel transfers by sequences of transfer operations.

Chapter 10

Timed Networks : Syntactic Subclasses, Semantic Variants

In Chapter 8, we showed that the controller state reachability problem for multi-clock timed networks is undecidable for clocks ranging over a real-time domain. One may wonder what happens if we consider timed networks with clocks over dense-timed domain, but restrict their excessive expressive power due to their ability to differentiate points in time with infinite precision. In fact, this complaint has already been raised against the model of timed automata [19] by [80, 118]. This makes algorithmic analysis of timed automata hard in many cases. For instances, checking emptiness is PSPACE-complete, while checking universality is undecidable for this model. Two classes of methods have already been suggested to remedy this problem for timed automata:

- The use of *digitization techniques* [83]. The idea is to identify subclasses of timed automata for which verification problems can be reduced from the dense time domain to the discrete one. This either yields speeding-up of the verification problem, or implies decidability of problems which are undecidable in the general case. A class of timed automata which allow digitization is *closed timed automata* [80, 111, 110] in which only non-strict clock constraints are allowed (of the form $x \leq 3$ or $x \geq 2$).
- To restrict the model of timed automata so that checking exact equality of clock values is prohibited. This restriction can be achieved syntactically through the use of *open timed automata* [80, 111, 110]. In an open automaton only strict clock constraints are allowed (of the form $x < 3$ or $x > 2$). The restriction can also be achieved semantically by considering a *robust semantics* [80, 81, 110] where a computation is accepted to be valid if and only if neighbouring computations are also accepted. However, [110] shows that expressive powers of the timed automata under standard and robust semantics are incomparable.

The complaint about the excessive power of timed automata is equally valid in the case of timed networks. We consider subclasses of timed networks based on similar restrictions to the ones mentioned above.

In this chapter, first we introduce *closed timed networks* (CTNs) and *open timed networks* (OTNs). Using a similar idea to [111], we show that digitization is applicable to CTNs. This reduces controller state reachability for CTNs to

the same problem for discrete timed networks. The latter problem is shown to be decidable in Chapter 9. This result is of practical relevance since any timed network can be safely infinitesimally over-approximated by a closed timed network.

Furthermore, we show undecidability of controller state reachability for OTNs. The undecidability result is shown through a reduction from the reachability problem for 2-counter machines. The undecidability result strengthens the result in Chapter 8, in the sense that (i) it shows undecidability for a subclass of that in Chapter 8; and (ii) it uses an encoding which does not rely on using equality for clock values.

Finally, we consider *robust* semantics of timed networks by introducing timing fuzziness through semantic removal of equality testing. We show undecidability of controller state reachability for TNs under the robust semantics. This is achieved by reducing the problem for OTNs under the standard semantics to the problem for OTNs under the robust semantics.

10.1 Closed Timed Networks

In this section, we show that the controller state reachability problem for a subclass of timed networks, called *closed timed networks* is decidable.

Closed Timed Network A *closed timed network* is a timed network in which guarded commands in the rules may only contain a negation-free boolean combination of predicates of the form $k \leq x$ or $k \geq x$ where $x \in \{x_1, \ldots, x_K\}$.

We define the controller state reachability problem for closed timed networks with K clocks (CTN(K)-Reach) in the obvious manner.

Theorem 10.1. CTN(K)-Reach is decidable.

To prove Theorem 10.1, we reduce CTN(K)-Reach to the controller state reachability problem for *discrete* timed networks, which is already shown to be decidable in Chapter 9.

The rest of this section proves Theorem 10.1.

First we recall the digitization technique introduced in [83]. Let $\delta \in \mathbb{R}^+$ and let $0 \leq \varepsilon < 1$ be real numbers. If $\mathit{fract}(\delta) < \varepsilon$, let $[\delta]_\varepsilon = \lfloor \delta \rfloor$, otherwise $[\delta]_\varepsilon = \lceil \delta \rceil$. The $[.]_\varepsilon$ operator therefore shifts the value of a real number δ to the preceding or the following integer, depending on whether the fractional part of δ is less than ε or not.

From Theorem 9.2, we know that DTN(K)-Reach is decidable. To decide CTN(K)-Reach, we reduce the problem CTN(K)-Reach to the problem DTN(K)-Reach. Given a closed timed network $\mathcal{N}_1 = (Q, \Re)$, we shall consider a discrete timed network $\mathcal{N}_2 = (Q, \Re)$, which are syntactically identical. We show that for any initial configuration γ_{init} and a controller state s_F, there is an γ_{init}-computation in $\mathcal{N}_1$ which leads to the final state s_F iff there is a γ_{init}-computation leading to the final state s_F in the derived $\mathcal{N}_2$.

The direction from right to left is straightforward.

We give the proof for the other direction. Suppose $\gamma_{init} \xrightarrow{*} s_F$, i.e, there is a γ_{init}-computation π given by $\gamma_{init} = \gamma_0 \longrightarrow_{T=\delta_1} \gamma'_1 \longrightarrow_{r_1} \gamma_1 \longrightarrow_{T=\delta_2} \gamma'_2 \longrightarrow_{r_2} \gamma_2 \ldots \longrightarrow_{T=\delta_n} \gamma'_n \longrightarrow_{r_n} \gamma_n = \gamma$ in $\mathcal{N}_1$, where $\delta_i \geq 0$ for $i : 1 \leq i \leq n$. Let γ_j be of the form $(I, q_j, \mathcal{Q}_j, X_j)$.

Given any $\varepsilon : 0 \leq \varepsilon < 1$, we define a χ_0-computation in $\mathcal{N}_2$ such that $\chi_0 \longrightarrow_{T=\delta'_1} \chi'_1 \longrightarrow_{r_1} \chi_1 \longrightarrow_{T=\delta'_2} \chi'_2 \longrightarrow_{r_2} \chi_2 \ldots \longrightarrow_{r_n} \chi_n$ as follows.

Define $\chi_0 = \gamma_0$, and for $j : 1 \leq j \leq n$, define $\chi_j = \left(I, q_j, \mathcal{Q}_j, X'_j\right)$:

1. $X'_j(k)(i) = [\delta_1 + \cdots + \delta_j]_\varepsilon - [\delta_1 + \cdots + \delta_l]_\varepsilon$, where l is the largest natural number $\leq j$ such that the rule r_l resets the clock x_k.
2. $X'_j(k)(i) = [\delta_1 + \cdots + \delta_j]_\varepsilon$ if the clock x_k is never reset.

We define χ'_j in a similar manner to γ'_j. Furthermore, we define $\delta'_1 = [\delta_1]_\varepsilon$ and $\delta'_j = [\delta_1 + \cdots + \delta_j]_\varepsilon - [\delta_1 + \cdots + \delta_{j-1}]_\varepsilon$ for $j > 1$.

We show that π' is a computation in $\mathcal{N}_2$:

- From the definition of χ_{j-1}, χ'_j and δ'_j and the fact that $\gamma_{j-1} \longrightarrow_{T=\delta_j} \gamma'_j$, it is clear that $\chi_{j-1} \longrightarrow_{T=\delta'_j} \chi'_j$ for $j : 1 \leq j < n$. Notice that $\delta'_j \geq 0$.
- To show $\chi'_j \longrightarrow_{r_j} \chi_j$, we conclude first that r_j is enabled from χ'_j. This follows from the definition of χ'_j, the fact that r_j is enabled from γ'_j and the fact that, given $\xi_1, \xi_2, \varepsilon \in \mathbb{R}^{\geq 0}$ and $k \in \mathbb{N}$:
  - $\xi_1 - \xi_2 \leq k \implies [\xi_1]_\varepsilon - [\xi_2]_\varepsilon \leq k$.
  - $\xi_1 - \xi_2 \geq k \implies [\xi_1]_\varepsilon - [\xi_2]_\varepsilon \geq k$.

  From the definition of χ'_j, χ_j, enabling of the rule r_j from χ'_j and the fact that $\gamma'_j \longrightarrow_{r_j} \gamma_j$, it is clear that $\chi'_j \longrightarrow_{r_j} \chi_j$ for $j : 1 \leq j \leq n$.

Therefore π' is a computation in $\mathcal{N}_2$. Theorem 10.1 follows from this, the fact that each γ_j has the same controller state as χ_j and Theorem 9.2.

Example 10.2. Figure 10.1 shows graphical representation of a computation in a closed timed network with two clocks starting from a configuration given by $(\{1, 2, 3\}, q, \mathcal{Q}, X)$ where $\mathcal{Q}(1) = q_1, \mathcal{Q}(2) = q_2, \mathcal{Q}(3) = q_3$ and $X_k(j) = 0$ for $j \in \{1, 2, 3\}$ and $k \in \{1, 2\}$. Notice that r_0, r_1 and r_2 resets the clocks $(X_1(2), X_2(3))$, $(X_2(1))$ and $(X_1(1), X_2(1))$ respectively. Also, given $\varepsilon = 0.8$, we have $\delta'_0 = [1.5]_\varepsilon = 1$, $\delta'_1 = [1.8]_\varepsilon - [1.5]_\varepsilon = 2 - 1 = 1$ and $\delta'_2 = [3.9]_\varepsilon - [1.8]_\varepsilon = 4 - 2 = 2$. From this, it is easy to see the effect of the discrete transitions. □

Example 10.3. As in Figure 10.1, Figure 10.2 shows graphical representation of a computation in a closed timed network with two clocks starting from the same configuration given by $(\{1, 2, 3\}, q, \mathcal{Q}, X)$ with $q, \mathcal{Q}, X$ as before. We also consider the same time lapses and the same set of rules. However, in this case $\varepsilon = 0.2$, we have $\delta'_0 = [1.5]_\varepsilon = 2$, $\delta'_1 = [1.8]_\varepsilon - [1.5]_\varepsilon = 2 - 2 = 0$ and $\delta'_2 = [3.9]_\varepsilon - [1.8]_\varepsilon = 4 - 2 = 2$. Again, it is easy to see the effect of the discrete transitions. □

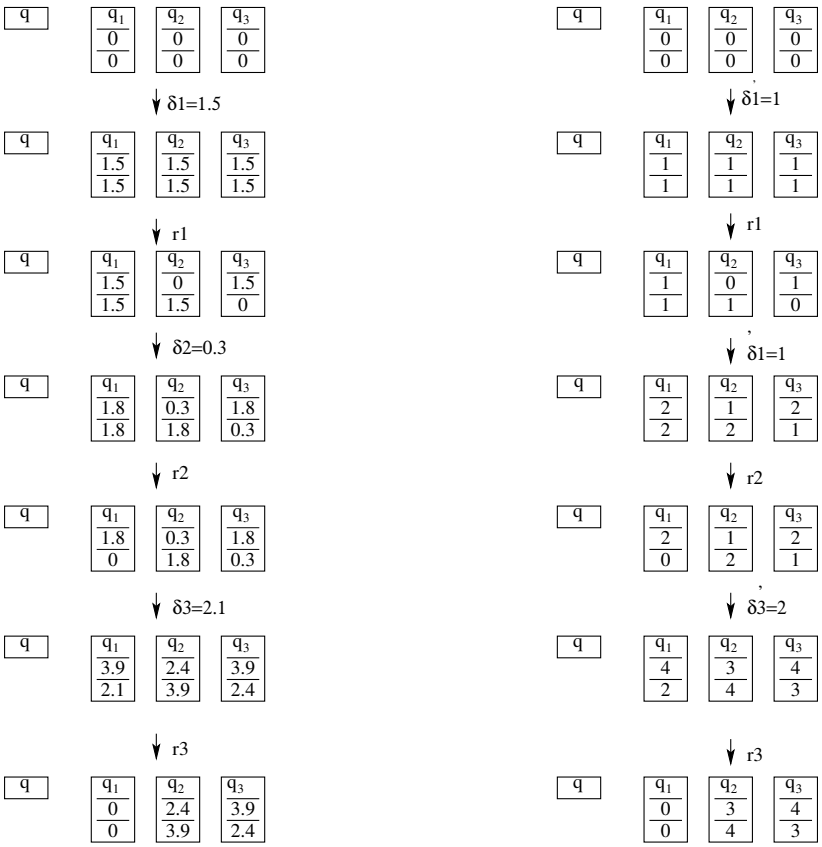

Figure 10.1: Simulating a computation of a CTN by a computation in a DTN for $\varepsilon = 0.8$.

10.2 Open Timed Networks

In this section, we strengthen the undecidability result of Chapter 8 by showing undecidability of the controller state reachability problem for a subclass of timed networks, namely *open timed networks*.

Open Timed Network An *open timed network* is a timed network in which guarded commands in the rules may only contain a negation-free boolean combination of predicates of the form $k < x$ or $k > x$ where $x \in \{x_1, \ldots, x_K\}$.

We define the controller state reachability problem for open timed networks with K clocks (OTN(K)-Reach) in the obvious manner.

Theorem 10.4. OTN(2)-Reach is undecidable.

Notice that Theorem 10.4 implies Theorem 8.2. But the encoding of transitions in 2-counter machine is more involved for OTNs.

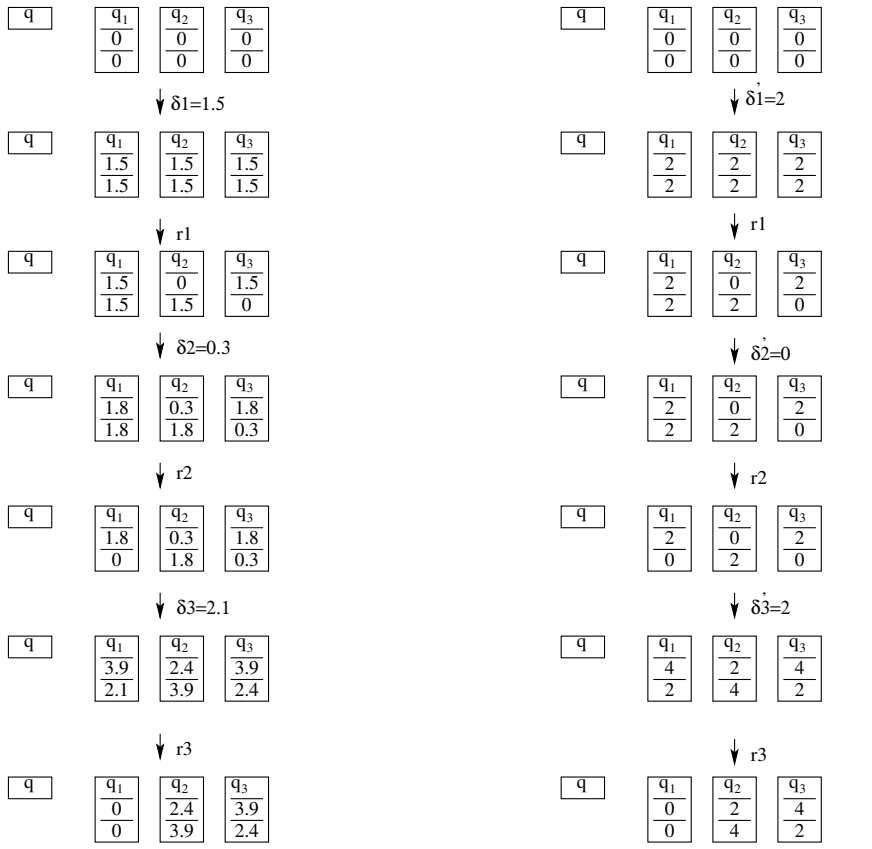

Figure 10.2: Simulating a computation of a CTN by a computation in a DTN for $\varepsilon = 0.2$.

10.2.1 Encoding of Configurations

We show undecidability of OTN(2)-Reach through a reduction from CM-Reach. Given a counter machine $\mathsf{C} = (S, s_{init}, \{c_1, c_2\}, \mathtt{I})$, we shall derive an open timed network $\mathcal{O}_\mathsf{C} = (Q_\mathsf{C}, \Re_\mathsf{C})$ with two clocks. First we describe how to construct the set Q_C. Then, we describe how configurations of C are encoded as configurations of $\mathcal{O}_\mathsf{C}$.

States According to the model described in Section 8.1, the set Q_C will consist of two disjoint sets of states: the set Q_C^{ctrl} of controller states and the set Q_C^{proc} of process states. As in Section 8.4, the set Q_C^{ctrl} contains a special state, called $idle^c$ and the *local states of* C. Furthermore, Q_C^{ctrl} also contains the *Temporary states* described below:

- three states $tmp_1^\imath$, $tmp_2^\imath$, $tmp_3^\imath$ for each increment instruction $\imath \in \mathtt{I}$,
- two states $tmp_1^\imath$, $tmp_2^\imath$ for each zero-testing instruction $\imath \in \mathtt{I}$,

- four states $tmp^s_{11}, tmp^s_{12}, tmp^s_{21}$ and tmp^s_{22}, for each controller state $s \in \mathsf{C}$, and
- three states $s^1_{init}, s^2_{init}, s^3_{init}$ (recall that s_{init} is the initial local state of C). These three states are used as intermediate states in the initialization phase of the simulation (Section 10.2.2).

The set Q^{proc}_{C} contains the two types of states described in Section 8.4 for $\mathcal{N}_{\mathsf{C}}$. Additionally, it contains a temporary state $fst_\imath$ for each increment instruction $\imath$. This state is used as an intermediate state in the simulation of incrementing instructions.

Encodings As in Section 8.4, each configuration β of C will be encoded by a set of configurations in $\mathcal{O}_{\mathsf{C}}$. A counter encoding in $\mathcal{O}_{\mathsf{C}}$ can be explained in a manner similar to that in $\mathcal{N}_{\mathsf{C}}$ in Section 8.4. The difference of this encoding with the encoding in Section 8.4 is roughly as follows. The encoding of Section 8.4 needs to have clock x_2 of each process (except the last one) exactly equal to clock x_1 of the next process. For OTNs, we need clock x_2 of each process to be strictly larger than clock x_1 of the next process (again, except the last process).

Formally, a configuration $\gamma = (I, q, \mathcal{Q}, X)$ is said to be a c_1-*encoding* of *value* m if there is an injection h from the set $\{0, \ldots, m+1\}$ to I such that the following conditions are satisfied.

- $\mathcal{Q}(h(0)) = fst_1$, $\mathcal{Q}(h(m+1)) = last_1$, and $\mathcal{Q}(h(i)) = mid_1$ for each $i : 1 \leq i \leq m$.
- $\mathcal{Q}(j) \in \{idle^p, fst_2, mid_2, last_2\}$ if $j \in I \setminus \text{range}(h)$.
- $X_1(h(i)) < X_2(h(i-1))$ for each $i : 1 \leq i \leq m+1$.
- $X_2(h(i)) < X_1(h((i+2)))$, for each $i : 0 \leq i \leq m-1$.
- $X_2(h(m+1)) < X_1(h(0)) < X_1(h(1))$.

The first two conditions are same as those for a c_1-encoding of value m in $\mathcal{N}_{\mathsf{C}}$ of Section 8.4 and can be explained in a similar manner. The last three conditions show how the processes which are part of a c_1-encoding are ordered as a circular list. The position of each process in the list is reflected by values of its clocks x_1 and x_2. More precisely, condition three says that except the first process, clock x_1 of each process in the list is strictly smaller than clock x_2 of the previous process. Condition four says that clock x_2 of each process is less than clock x_1 of the second process to its right (except the last two processes). Finally the last condition states that clock x_2 of the last process is strictly less than the clock x_1 of the first process, which is again strictly less than the clock x_1 of the second process. We use $Val_1(\gamma)$ to denote the value m of a c_1-encoding γ.

Example 10.5. Figure 10.3(a) shows a c_1-encoding of value 2. Figure 10.3(b) shows a graphical representation of the ordering among clock values. In Section 10.2.2, we shall use such a graphical representation to explain the different

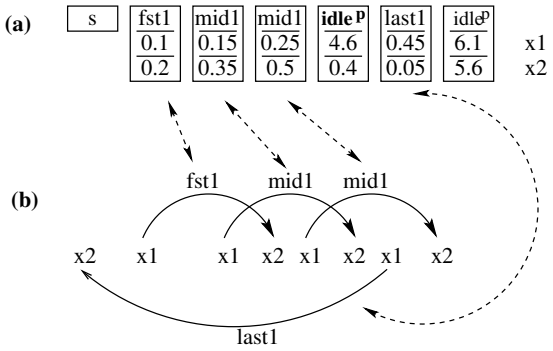

Figure 10.3: (a) a c_1-encoding. (b) Graphical representation of ordering among clocks in c_1-encodings.

steps in the simulation of C. Each process is denoted by an edge whose end points are its two clocks and the arrow is at clock x_2. Such an edge is labelled by the current state of the process. Clock values in the list are strictly increasing from left to right. □

A c_2-*encoding* and its value $Val_2(\gamma)$ are defined in a similar manner. Furthermore, the notion of *(proper) encoding* of $\mathcal{N}_\mathsf{C}$ carries over to the case of $\mathcal{O}_\mathsf{C}$.

10.2.2 Encoding of Transitions

Next we perform the second step in deriving the open timed network $\mathcal{O}_\mathsf{C} = (Q_\mathsf{C}, \Re_\mathsf{C})$ from the counter machine $\mathsf{C} = (S, s_{init}, \{c_1, c_2\}, \mathtt{I})$. More precisely, we describe the set of rules $\Re_\mathsf{C}$. The set $\Re_\mathsf{C}$ contains the following rules:

Incrementing For each instruction $\imath = (s_1, c_1{+}{+}, s_2)$ in $\mathtt{I}$ there are four rules in $\Re_\mathsf{C}$, namely

$$inc_1^\imath : \begin{bmatrix} s_1 \\ \rightarrow \\ tmp_1^\imath \end{bmatrix} \begin{bmatrix} last_1 \\ 0 < x_2 \rightarrow \emptyset \\ last_1 \end{bmatrix} \begin{bmatrix} idle^p \\ tt \rightarrow \{x_2\} \\ fst_\imath \end{bmatrix}$$

$$inc_2^\imath : \begin{bmatrix} tmp_1^\imath \\ \rightarrow \\ tmp_2^\imath \end{bmatrix} \begin{bmatrix} fst_\imath \\ 0 < x_2 \rightarrow \emptyset \\ fst_\imath \end{bmatrix} \begin{bmatrix} fst_1 \\ tt \rightarrow \{x_1\} \\ fst_1 \end{bmatrix}$$

$$inc_3^\imath : \begin{bmatrix} tmp_2^\imath \\ \rightarrow \\ tmp_3^\imath \end{bmatrix} \begin{bmatrix} fst_1 \\ 0 < x_1 \rightarrow \emptyset \\ mid_1 \end{bmatrix} \begin{bmatrix} fst_\imath \\ tt \rightarrow \{x_1\} \\ fst_1 \end{bmatrix}$$

$$inc_4^\imath : \begin{bmatrix} tmp_3^\imath \\ \rightarrow \\ s_2 \end{bmatrix} \begin{bmatrix} fst_1 \\ 0 < x_1 \rightarrow \emptyset \\ fst_1 \end{bmatrix} \begin{bmatrix} last_1 \\ tt \rightarrow \{x_2\} \\ last_1 \end{bmatrix}$$

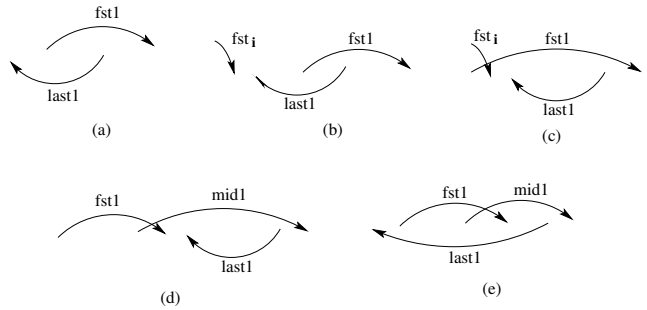

Figure 10.4: Simulating $(s_1, c_1{+}{+}, s_2)$ on a c_1-encoding

The total effect of the four rules is to increment the value of a c_1-encoding by adding one more process to the list. The rule inc_1^i picks a process in state $idle^p$ and changes its state to fst_i. The new process will be placed first in the list. Furthermore, the rule resets clock x_2 of the new process in state fst_i. The result of applying rule inc_1^i on a c_1-encoding of value 0 is shown in Figure 10.4(b).

Rule inc_2^i resets clock x_1 of the process which is now in state fst_1 and will be placed second in the list. This is done in order to maintain the invariant that clock x_1 of each process (except the first process) is strictly less than the clock x_2 of the previous process (recall the definition of an encoding from Section 10.2.1). The result of applying rule inc_2^i is shown in Figure 10.4(c).

Rule inc_3^i resets clock x_1 of the process which is now in state fst_i. It also changes the states of the processes in fst_1 and fst_i to states mid_1 and fst_1 respectively. This is done in order to maintain the invariant that clock x_1 of the first process is strictly less than the clock x_1 of the second process in the list (recall the definition of an encoding from Section 10.2.1). The result of applying rule inc_3^i is shown in Figure 10.4(d). Rule inc_4^i resets clock x_2 of the process which is now in state $last_1$. This is done in order to maintain the invariant that clock x_2 of the last process is strictly less than the clock x_1 of the first process in the list (recall the definition of an encoding from Section 10.2.1). The result of applying rule inc_4^i is shown in Figure 10.4(e).

Some remarks about rules inc_1^i,inc_2^i,inc_3^i and inc_4^i:

- After execution of inc_1^i (inc_2^i, inc_3^i resp.), the controller will be in state tmp_1^i (tmp_2^i, tmp_3^i resp.) and therefore inc_2^i (inc_3^i, inc_4^i resp.) is the only rule which may eventually be enabled after execution of inc_1^i (inc_2^i, inc_3^i resp.).

- The guard $0 < x_2$ in the definition of inc_1^i and inc_2^i and the guard $0 < x_1$ in the definition of the rules inc_3^i and inc_4^i can be explained in a manner similar to the guards in the incrementing rules of $\mathcal{N}_C$ (Section 8.5).

- After application of inc_4^i, the resulting encoding will not be proper. As

in Section 8.5, we can re-create a proper encoding by letting time pass through a timed transition.

Also, for each instruction of the form $(s_1, c_2{+}{+}, s_2)$, there are four rules similar to the rules described above (replacing the states fst_1, mid_1, and $last_1$ by fst_2, mid_2 and $last_2$, respectively).

Decrementing For each instruction $\imath = (s_1, c_1{-}{-}, s_2)$ in $\mathtt{I}$ there is a rule in $\Re_{\mathsf{C}}$, namely

$$dec^{\imath} : \left[\begin{array}{c} s_1 \\ \rightarrow \\ s_2 \end{array}\right] \left[\begin{array}{c} last_1 \\ (0 < x_2) \wedge (x_1 < 1) \rightarrow \emptyset \\ idle^p \end{array}\right] \left[\begin{array}{c} mid_1 \\ 1 < x_2 \rightarrow \{x_2\} \\ last_1 \end{array}\right]$$

The rule $dec^{\imath}$ decrements the value of a c_1-encoding by removing the last process of the list. More precisely, it changes the state of the last process to $idle^p$ (i.e. removes that process from the list), and changes the state of the process which is next last from mid_1 to $last_1$. In order to do that, we have to be able to identify the process which is next last in the list. Since all processes in the middle of the list are in state mid_1, we cannot identify the next last process simply by checking process states. Instead, we wait until the value of clock x_2 of the process (next last) in the state mid_1 is greater than one, but the clock x_1 of the process in $last_1$ is still less than 1. Figure 10.5 shows the effect of applying the rule to a c_1-encoding of value 2.

Figure 10.5: Simulating $(s_1, c_1{-}{-}, s_2)$ on a c_1-encoding

As in Section 8.5, decrementing may result in an improper encoding. In order, to keep the possibility of maintaining proper encodings in our simulation, we combine the rule $dec^{\imath}$ with the *rotation* rules described below.

Rotation For each state $s \in S$, the set $\Re_{\mathsf{C}}$ contains the following four rules

$$rot^s{}_{2,1} : \begin{bmatrix} s \\ \rightarrow \\ tmp^s_{21} \end{bmatrix} \begin{bmatrix} fst_2 \\ tt \rightarrow \{x_1\} \\ fst_2 \end{bmatrix} \begin{bmatrix} last_2 \\ 0 < x_2 \rightarrow \emptyset \\ last_2 \end{bmatrix}$$

$$rot^s{}_{2,2} : \begin{bmatrix} tmp^s_{21} \\ \rightarrow \\ tmp^s_{22} \end{bmatrix} \begin{bmatrix} fst_2 \\ 0 < x_1 \rightarrow \emptyset \\ mid_2 \end{bmatrix} \begin{bmatrix} mid_2 \\ 1 < x_2 \rightarrow \emptyset \\ last_2 \end{bmatrix} \begin{bmatrix} last_2 \\ x_1 < 1 \rightarrow \{x_1\} \\ fst_2 \end{bmatrix}$$

$$rot^s_{2,3} : \begin{bmatrix} tmp^s_{21} \\ \rightarrow \\ tmp^s_{22} \end{bmatrix} \begin{bmatrix} fst_2 \\ (0 < x_1) \wedge (1 < x_2) \rightarrow \emptyset \\ last_2 \end{bmatrix} \begin{bmatrix} last_2 \\ x_1 < 1 \rightarrow \{x_1\} \\ fst_2 \end{bmatrix}$$

$$rot^s{}_{2,4} : \begin{bmatrix} tmp^s_{22} \\ \rightarrow \\ s \end{bmatrix} \begin{bmatrix} fst_2 \\ 0 < x_1 \rightarrow \emptyset \\ fst_2 \end{bmatrix} \begin{bmatrix} last_2 \\ tt \rightarrow \{x_2\} \\ last_2 \end{bmatrix}$$

Recall that (from Section 8.5) if $\imath = (s_1, c_1{--}, s_2)$ then applying a rule $dec^\imath$ will not give a proper encoding in case the c_2-encoding has clocks with greater values than the clocks of the processes in the c_1-encoding. The role of $rot^s{}_{2,1}, rot^s{}_{2,2}, rot^s{}_{2,4}$ then is to decrement clock values of processes which are part of a c_2-encoding of positive value while preserving the signature of the whole encoding. The rule $rot^s{}_{2,3}$ is used instead of the rule $rot^s{}_{2,2}$ when we use the rotation of a c_2-encoding of value 0. More precisely,

- $rot^s_{2,1}$ resets the clock x_1 of the process in state fst_2. This process will be made second in the list by the next executed rule, i.e. $rot^s_{2,2}$.
- $rot^s_{2,2}$ resets the clock x_1 of the process in $last_2$ and makes this process first in the list. This rule also changes the state of the next last process to $last_1$ and the state of the second process (previously in fst_1) to mid_1. The identification of the next last process is the same as that in case of decrementing. The explanation of $rot^s_{2,3}$ is similar, but $rot^s_{2,3}$ is only applied if the c_2 encoding has value 0.
- $rot^s_{2,4}$ resets the clock x_2 of the process currently in state $last_2$.

This again amounts to a rotation of the list corresponding to the c_2-encoding of non-zero positive value. Figures 10.6(b), (c) and (d) graphically show the effect of applying these rules to a c_2-encoding of value 1 in Figure 10.6(a).

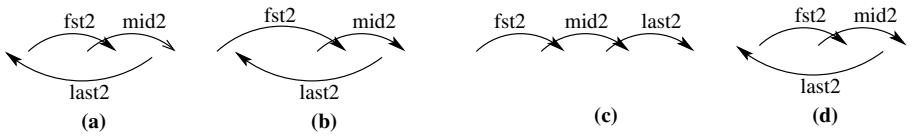

Figure 10.6: Rotation of c_2-encoding of value 1.

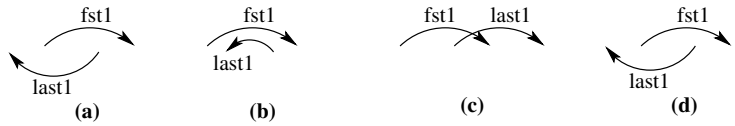

Figure 10.7: Simulating $(s_1, c_1?0, s_2)$ on a c_1-encoding of value zero.

There are also similar rules ${rot^s}_{1,1}$, ${rot^s}_{1,2}$, ${rot^s}_{1,3}$, and ${rot^s}_{1,4}$, which are used to rotate a c_1-encoding and which are used in connection with rules of the form $dec^\imath$ with $\imath = (s_1, c_2{-}{-}, s_2)$.

Zero Testing For each instruction $\imath = (s_1, c_1 = 0?, s_2)$ in $\mathtt{I}$ there are rules in $\Re_{\mathsf{C}}$, namely $tst^\imath_1, tst^\imath_2$ and $tst^\imath_3$. These three rules check that the value of the encoding is zero by testing that there are no processes in state mid_1. This is done by verifying that the process which is next last in the list is the same as the process which is first in the list in a manner similar to the decrementing rule.

For each instruction $\imath = (s_1, c_1 = 0?, s_2)$ in C there are three rules in $\Re_{\mathsf{C}}$, namely

$$tst^\imath_1 : \begin{bmatrix} s_1 \\ \rightarrow \\ tmp^\imath_1 \end{bmatrix} \begin{bmatrix} fst_1 \\ tt \rightarrow \{x_1\} \\ fst_1 \end{bmatrix} \begin{bmatrix} last_1 \\ 0 < x_2 \rightarrow \emptyset \\ last_1 \end{bmatrix}$$

$$tst^\imath_2 : \begin{bmatrix} tmp^\imath_1 \\ \rightarrow \\ tmp^\imath_2 \end{bmatrix} \begin{bmatrix} fst_1 \\ (0 < x_1) \wedge (1 < x_2) \rightarrow \emptyset \\ last_1 \end{bmatrix} \begin{bmatrix} last_1 \\ x_1 < 1 \rightarrow \{x_1\} \\ fst_1 \end{bmatrix}$$

$$tst^\imath_3 : \begin{bmatrix} tmp^\imath_2 \\ \rightarrow \\ s_2 \end{bmatrix} \begin{bmatrix} fst_1 \\ 0 < x_1 \rightarrow \emptyset \\ fst_1 \end{bmatrix} \begin{bmatrix} last_1 \\ tt \rightarrow \{x_2\} \\ last_1 \end{bmatrix}$$

The rules interchange the processes in states fst_1 and $last_1$ and reset the appropriate clocks to preserve the invariants of an encoding in a manner similar to the rotation rules described above. The explanation of the rules $tst^\imath_1, tst^\imath_2$ and $tst^\imath_3$ is in fact, similar to those for the rules $rot^s_{1,1}, rot^s_{1,3}$ and $rot^s_{1,4}$ respectively. Sometimes before applying the rules for zero-testing, one has to apply the rotation rules according to the same scenarios explained for the *decrementing* rule.

Also, there are similar rules in $\Re_{\mathsf{C}}$ for each instruction of the form $\imath = (s_1, c_2 = 0?, s_2)$.

Figure 10.7 shows the effect of applying the rule to a c_1-encoding of value 0.

Initialization The initial phase consists of the following four rules.

$$init_1 : \left[\begin{array}{c} idle^c \\ \rightarrow \\ s^1_{init} \end{array}\right] \left[\begin{array}{c} idle^p \\ tt \rightarrow \{x_2\} \\ fst_1 \end{array}\right] \left[\begin{array}{c} idle^p \\ tt \rightarrow \{x_2\} \\ fst_2 \end{array}\right]$$

$$init_2 : \left[\begin{array}{c} s^1_{init} \\ \rightarrow \\ s^2_{init} \end{array}\right] \left[\begin{array}{c} fst_1 \\ 0 < x_2 \rightarrow \emptyset \\ fst_1 \end{array}\right] \left[\begin{array}{c} idle^p \\ tt \rightarrow \{x_1\} \\ last_1 \end{array}\right] \left[\begin{array}{c} idle^p \\ tt \rightarrow \{x_1\} \\ last_2 \end{array}\right]$$

$$init_3 : \left[\begin{array}{c} s^2_{init} \\ \rightarrow \\ s^3_{init} \end{array}\right] \left[\begin{array}{c} fst_1 \\ tt \rightarrow \{x_1\} \\ fst_1 \end{array}\right] \left[\begin{array}{c} fst_2 \\ tt \rightarrow \{x_1\} \\ fst_2 \end{array}\right] \left[\begin{array}{c} last_1 \\ 0 < x_1 \rightarrow \emptyset \\ last_1 \end{array}\right]$$

$$init_4 : \left[\begin{array}{c} s^3_{init} \\ \rightarrow \\ s_{init} \end{array}\right] \left[\begin{array}{c} fst_1 \\ 0 < x_1 \rightarrow \emptyset \\ fst_1 \end{array}\right] \left[\begin{array}{c} last_1 \\ tt \rightarrow \{x_2\} \\ last_1 \end{array}\right] \left[\begin{array}{c} last_2 \\ tt \rightarrow \{x_2\} \\ last_2 \end{array}\right]$$

The role of the initialization rules is to bring $\mathcal{O}_{\mathsf{C}}$ from its initial configuration (where the controller and all processes are idle) into a configuration which is an encoding of the initial configuration β_{init} of C.

The first rule $init_1$ takes the controller into the temporary state s^1_{init}. It also picks two idle processes to be the first processes in the c_1-encoding and c_2-encoding (each with value zero). Clock x_2 of both the processes are reset. The second rule $init_2$ changes the controller state to s^2_{init} and picks two more processes to be the last processes in the c_1-encoding and c_2-encoding. This rule is enabled if the clock x_2 of the process fst_1 is strictly larger than 0. Also, clock x_1 of both the new processes are reset. The third rule $init_3$ is enabled when the clock x_1 of the process $last_1$ is greater than 0. and it changes the controller state to s^3_{init}. It also resets the clock x_1 of the processes in state fst_1 and fst_2 respectively. The fourth rule $init_4$ changes the controller state to s_{init}. It also resets the clock x_2 of the processes in state $last_1$ and $last_2$ respectively and completes the creation of the c_1-encoding and c_2-encoding of value zero. Finally, applying a timed transition yields a proper encoding. The effect of the rules $init_1, init_2, init_3$ and $init_4$ are illustrated through Figure 10.8 (only the c_1-encoding is shown).

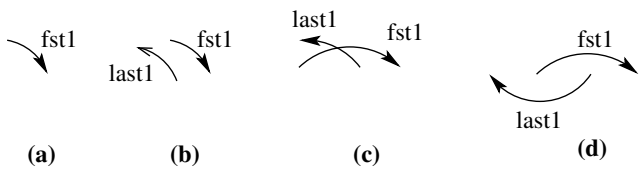

Figure 10.8: Initialization. Figure (d) shows a graphical representation of a c_1-encoding of value 0.

Correctness

Let $\mathsf{C} = (S, s_{init}, \{c_1, c_2\}, \mathtt{I})$ be a counter machine and let $\mathcal{O}_\mathsf{C} = (Q_\mathsf{C}, \Re_\mathsf{C})$ be an open timed network derived from C as described in Section 8.4 and Section 8.5. Let $\rightsquigarrow$ and $\longrightarrow$ be the transition relations induced by C and $\mathcal{O}_\mathsf{C}$ respectively.

If s_F is a control state in C then the following holds

Theorem 10.6. $\beta_{init} \overset{*}{\rightsquigarrow} s_F$ iff $\gamma_{init} \overset{*}{\longrightarrow} s_F$ for some initial configuration γ_{init} of $\mathcal{O}_\mathsf{C}$.

We show the proof of Theorem 10.6 in Appendix. Theorem 10.4 directly follows from Theorem 10.6.

10.3 Robust Timed Networks

In this section, we define the *robust* semantics of timed network. We show that the controller state reachability is undecidable for robust timed networks. Notice that the problem is decidable for robust timed automata.

First we define a *timed event* to be a pair (ξ, r) where ξ is the timestamp of the rule $r \in \Re$. A *timed trace* is a finite sequence of timed events with non-decreasing timestamps.

Given a computation $\gamma_{init} \longrightarrow_{T=\delta_1} \longrightarrow_{r_1} \gamma_1 \ldots \gamma_n \longrightarrow_{T=\delta_n} \longrightarrow_{r_n} \gamma_n$ of a timed network, there is an associated timed trace $tt(\pi) = \langle(\xi_1, r_1), \ldots, (\xi_n, r_n)\rangle$ where $\xi_1 = \delta_1$ and $\xi_i = \Sigma_{1 \leq j \leq i} \delta_j$. Consider a timed network N. Let Γ_{init} be the (infinite) set of initial configurations (defined as in Section 8.1) and let S_F be a set of final controller states. We define the *language* $L(N)$ of N as a set consisting of timed traces $tt(\pi)$ where π is a γ_{init}-computation with $\gamma_{init} \in \Gamma_{init}$ and π leads to some final controller state in S_F.

Next we define a metric D on the set of all timed traces of a timed network as follows. Given two timed traces $w = \langle(\xi_1, r_1), \ldots, (\xi_n, r_n)\rangle$ and $w' = \langle(\xi'_1, r'_1), \ldots, (\xi'_n, r'_n)\rangle$, let

- $D(w, w') = \infty$ if there is a $j : 1 \leq j \leq n$ such that $r_j \neq r'_j$.
- $D(w, w') = max\left\{|\xi_j - \xi'_j| \ : \ 1 \leq j \leq n\right\}$.

As argued in [80], any other 'reasonable' metric on timed traces of timed network yield the same topology as the metric D.

For the metric D, a timed trace w, and a positive real $\varepsilon \in \mathbb{R}^+$, we define the D-tube around w of diameter ε to be the set $\mathcal{T}(w, \varepsilon) = \{w' \mid D(w, w') < \varepsilon\}$ of all timed traces at a distance less than ε from w. A D-open set Op is a set of timed traces such that for all timed trace $w \in Op$, there is a positive real $\varepsilon \in \mathbb{R}^+$ with $\mathcal{T}(w, \varepsilon) \subseteq Op$. Thus, if a D-open set contains a computation π, then it also contains all computations in some neighbourhood of π. From now on, we shall omit reference to D and use 'open' to mean a D-open set.

Let the set of all timed traces be called TT. A set tts of timed traces is closed if its complement $TT - tts$ is open. The closure $\overline{tts}$ of a set tts of timed traces

is the least closed set containing tts and the interior tts^{int} is the greatest open set contained in tts. Given a set O, we use $[O]$ to denote the set $(\overline{O})^{int}$.

The language L of a timed network induces a *robust language* $[L] = (\overline{L})^{int}$. The set $[L]$ represents the set of timed traces of N under the robust semantics. Notice that a computation π is in $[L]$ if and only if there is some open neighbourhood $\mathcal{T}(\pi, \varepsilon)$ around π within some distance ε such that each timed trace in the neighbourhood is also included in the closure of the set L of computations of TN.

ROBUST REACHABILITY ($TN(K)$-ROBUST-REACH)

Instance: An open timed network $\mathcal{O} = (Q, \Re)$ with K clocks and a set S_F of controller states.

Question: Is $[L(\mathcal{O})] = \emptyset$?

We extend the theory of robust semantics for timed automata [80, 110] in a straightforward manner for timed networks and get the following lemmas. We consider timed networks where clock constraints are negation-free and disjunction-free. For any given timed network, one can easily obtain such an equivalent timed network.

Lemma 10.7. For every open timed network $\mathcal{O}$, $L(\mathcal{O})$ is an open set.

Proof. This proof is adapted from [80], where they show this lemma for timed automata. Consider an arbitrary timed trace $w \in L(\mathcal{O})$. Let $w = \langle(\xi_1, r_1), \ldots, (\xi_n, r_n)\rangle$. Since $w \in L(\mathcal{O})$, there is a computation $\pi = \gamma_{init} \longrightarrow_{T=\delta_1} \gamma'_1 \longrightarrow_{r_1} \gamma_1 \ldots \longrightarrow_{T=\delta_n} \gamma'_n \longrightarrow_{r_n} \gamma_n$, where $\delta_i = \xi_i - \xi_{i-1}$ for $i : 1 < i \leq n$ and $\delta_1 = \xi_1$. Let γ'_i be of the form $(I, q^i, \mathcal{Q}^i, X^i)$.

We will show that there is a ε such that $\mathcal{T}(w, \varepsilon) \subseteq L(\mathcal{O})$.

For each $i : 1 \leq i \leq n$,

- let ε_i be a real number smaller than the minimum of the distances $|K - X^i_k(j)|$ (where $j \in I$) such that there is a constant K and a guard $x_k < K$ or $x_k > K$ in the rule r_i and $g(X^i_k(j))$ is satisfied. (since all clock constraints are strict, these distances are strictly positive).

We define $\varepsilon := min\,\{\varepsilon_i/2 \mid 1 \leq i \leq n\}$.

Consider any timed trace $w' = \langle(\xi''_1, r_1), \ldots, (\xi''_n, r_n)\rangle$ where $D(w, w') < \varepsilon$. This means that $|\xi''_i - \xi_i| < \varepsilon$ for each i. We show that there is in fact a γ_{init}-computation $\gamma_{init} \longrightarrow_{T=\delta'_1} \beta'_1 \longrightarrow_{r_1} \beta_1 \ldots \longrightarrow_{T=\delta'_n} \beta'_n \longrightarrow_{r_n} \beta_n$, i.e., $w' \in L(\mathcal{O})$.

Let $\varepsilon'_i = |\xi''_i - \xi_i|$ for each i. Recall that $\varepsilon'_i < \varepsilon$. In π, consider a clock value $X^i_k(j)$. We show that if the clock participated in rule r_i, with its new valuation at time ξ'', it can still participate in the rule r_i. Either $X^i_k(j) = \xi_i$ if it was never reset or $X^i_k(j) = \xi_i - \xi_\ell$ if it was last reset by r_ℓ. Now, consider $\xi''_i - \xi''_\ell$. We know that $|\xi''_i - \xi_i| < \varepsilon$ and $|\xi''_\ell - \xi_\ell| < \varepsilon$. Then $\xi''_i - \xi''_\ell < (\xi_i + \varepsilon) - (\xi_\ell - \varepsilon) = 2 * \varepsilon = min(\varepsilon_i : 1 \leq i \leq n)$. Therefore, $g(X^i_k(j))$ is still true and we have $\beta'_i \longrightarrow_{r_i} \beta_i$.

The case when the clock is never reset before is handled in a similar manner.

Let $\delta'_i = \xi''_i - \xi''_{i-1}$ and $\delta'_1 = \xi''_1$. From the assumption that w' is a timed trace, i.e., $\xi''_{i-1} \leq \xi''_i$, it is clear that $\beta'_{i-1} \longrightarrow_{T=\delta'_i} \beta'_i$ for $i > 1$ and $\gamma_{init} \longrightarrow_{T=\delta'_1} \beta'_1$. This means that $w' \in L(\mathcal{O})$. This implies that $\mathcal{T}(w, \varepsilon) \subseteq L(\mathcal{O})$ and thus $L(\mathcal{O})$ is an open set. □

Lemma 10.8. For an open set Op, $Op = \emptyset$ iff $[Op] = \emptyset$.

Proof. Assume that $Op = \emptyset$. Then $\overline{Op} = \emptyset$. Then $(\overline{Op})^{int} = \emptyset$, i.e., $[Op] = \emptyset$.

Now we show the proof for the other direction. $Op \subseteq \overline{Op}$ and $Op^{int} \subseteq (\overline{Op})^{int}$. Since Op is open, $Op = Op^{int}$. Thus, $Op \subseteq (\overline{Op})^{int}$. Now, if $(\overline{Op})^{int} = \emptyset$, then from the above, it follows that $Op = \emptyset$. □

Lemma 10.9. For every open timed network $\mathcal{O}$, $L(\mathcal{O}) = \emptyset$ iff $[L(\mathcal{O})] = \emptyset$.

Proof. Since $L(\mathcal{O})$ is an open set by Lemma 10.7, the proof follows from Lemma 10.8. □

Now we show the following.

Theorem 10.10. TN(2)-Robust-Reach is undecidable.

Proof. The undecidability of OTN(2)-Reach (Theorem 10.4) means that it is undecidable for an open timed network $\mathcal{O}$ whether $L(\mathcal{O}) = \emptyset$. From this and Lemma 10.9, the theorem follows. □

10.4 Timed Networks with Finite and Bounded Variability

Considering the fact that any physically realisable system is subject to band-limitedness, one can impose a condition of band-limitedness on the timed networks. It will be quite appropriate to assume that there are only a bounded number of state changes in a fixed timed interval. In this section, we consider *multi-clock* timed networks in which either there are only (a) a finite number of state changes or (b) a bounded number of state changes in a unit interval. We show that the controller state reachability problem is still undecidable in the first case, while it is decidable in the second case.

The *variability* of a computation measures the maximum number of rules executed in a unit interval. Formally, the *variability* of a γ_{init}-computation $\pi = \gamma_0 \longrightarrow_{T=\delta_1} \gamma'_1 \longrightarrow_{r_1} \ldots \longrightarrow_{T=\delta_n} \gamma'_n \longrightarrow_{r_n} \gamma_n$ is given by the following expression :

$$var(\pi) = sup\,\{k+1 \mid \exists i.\Delta(\gamma_{i+k}) - \Delta(\gamma_i) < 1\}$$

where $\Delta(\pi(i)) = \Sigma_{1 \leq j \leq i}\delta_j$ for each $i \geq 1$ and $\Delta(\pi(0)) = 0$.

10.4.1 Timed Networks with Finite Variability

FINITE VARIABILITY PROBLEM

Instance: A timed network $\mathcal{N}$ with 2 clocks and a controller state q_F.
Question: Is there an initial configuration γ_{init} and a bound $k \in \mathbb{N}$ such that there is a computation π with $\gamma_{init} \xrightarrow{*} q_F$ and $var(\pi) < k$?

Theorem 10.11. Finite variability problem is undecidable.

Proof. The undecidability is shown by reducing TN(2)-Reach to the finite variability problem. If there is a γ_{init} from which q_F is reachable, it is reachable in a finite number of steps by a computation π given by $\gamma_{init} \xrightarrow{*} q_F$. Then $k = var(\pi)$. The other direction is trivial. □

10.4.2 Timed Networks with Bounded Variability

We say that a computation is of *bounded variability* k if the variability of the computation is less than or equal to k. Since number of elements in a rule is finite and there is a bound on the number of rules to be executed, it follows that there are only finite number of processes with clocks having ages less than max. This is due to the fact that one cannot refresh the ages of the clocks of an unbounded number of processes by executing only a bounded number of rules. It is easy to adapt the backward reachability algorithm of [9] for such timed networks.

10.5 Appendix

Proofs

In our correctness proof of Theorem 10.6, we use the relations $\rightsquigarrow$ on configurations of a counter machine C and $\Longrightarrow$ on the set of configurations of OTNs, as in the proof of Theorem 8.5 (Chapter 8).

First we introduce five new types of temporary encodings $c_1^{inc_1}$semi-encoding, $c_1^{inc_2}$semi-encoding, $c_1^{inc_3}$semi-encoding, $c_1^{rot_1}$semi-encoding and $c_1^{rot_2}$semi-encoding to describe the effect of the rules $inc_1^\imath$, $inc_2^\imath$, $inc_3^\imath$, $rot^s{}_{11}$ and $rot^s{}_{12}$ respectively. For $m \geq 1$, a configuration $\gamma = (I, q, \mathcal{Q}, X)$ is said to be

- a $c_1^{inc_1}$semi-encoding of *value* m if there is an injection h from $\{0, \ldots, m+1\}$ to I such that the following conditions are satisfied
  - $\mathcal{Q}(h(0)) = fst_\imath$ where $\imath$ is an increment instruction in C, $\mathcal{Q}(h(1)) = fst_1$, $\mathcal{Q}(h(i)) = mid_1$ for each $i : 2 \leq i \leq m$ and $\mathcal{Q}(h(m+1)) = last_1$.
  - $X_1(h(i)) < X_2(h(i-1))$ for each $i : 2 \leq i \leq m+1$.
  - $X_2(h(i)) < X_1(h((i+2)))$, for each $i : 1 \leq i \leq m-1$.
  - $X_2(h(0)) < X_2(h(m+1)) < X_1(h(1)) < X_1(h(2))$.

- a $c_1^{inc_2}$semi-encoding of *value* m if there is an injection h from $\{0,\ldots,m+1\}$ to I such that the following conditions are satisfied
  - $\mathcal{Q}$ is as defined for a $c_1^{inc_1}$semi-encoding.
  - $X_1(h(i)) < X_2(h(i-1))$ for each $i : 1 \leq i \leq m+1$.
  - $X_2(h(i)) < X_1(h((i+2)))$, for each $i : 1 \leq i \leq m-1$.
  - $X_2(h(0)) < X_2(h(m+1)) < X_1(h(2))$.
- a $c_1^{inc_3}$semi-encoding of *value* m if there is an injection h from $\{0,\ldots,m+1\}$ to I such that the following conditions are satisfied
  - $\mathcal{Q}$ is as defined for a c_1-encoding.
  - $X_1(h(i)) < X_2(h(i-1))$ for each $i : 1 \leq i \leq m+1$.
  - $X_2(h(i)) < X_1(h((i+2)))$, for each $i : 1 \leq i \leq m-1$.
  - $X_1(h(0)) < X_1(h(1))$.
  - $X_2(h(0)) < X_2(h(m+1)) < X_1(h(2))$.

A graphical representation of $c_1^{inc_1}$semi-encoding, $c_1^{inc_2}$semi-encoding and $c_1^{inc_3}$semi-encoding is shown in Figure 10.4(b), 10.4(c), and 10.4(d) respectively.

For $m \geq 0$, a configuration $\gamma = (I, q, \mathcal{Q}, X)$ is said to be

- a $c_1^{rot_1}$semi-encoding of *value* m if there is an injection h from $\{0,\ldots,m+1\}$ to I such that the following conditions are satisfied
  - $\mathcal{Q}$ is as defined for a c_1-encoding.
  - $X_1(h(i)) < X_2(h(i-1))$ for each $i : 1 \leq i \leq m+1$.
  - $X_2(h(i)) < X_1(h((i+2)))$, for each $i : 0 \leq i \leq m-1$.
  - $X_1(h(0)) < X_2(h(m+1)) < X_1(h(1))$.
- a $c_1^{rot_2}$semi-encoding of *value* m if there is an injection h from $\{0,\ldots,m+1\}$ to I such that the following conditions are satisfied
  - $\mathcal{Q}$ is as defined for a $c_1^{rot_1}$semi-encoding.
  - $X_1(h(i)) < X_2(h(i-1))$ for each $i : 1 \leq i \leq m+1$.
  - $X_2(h(i)) < X_1(h((i+2)))$, for each $i : 0 \leq i \leq m-1$.
  - $X_1(h(0)) < X_1(h(1))$.
  - $X_2(h(m)) < X_2(h(m+1))$.

Figure 10.6 illustrates the rotation of a c_1-encoding graphically and shows a graphical representation of $c_1^{rot_1}$semi-encoding and $c_1^{rot_2}$semi-encoding in Figure 10.6(b) and 10.6(c) respectively.

In a similar manner to a c_1-encoding, we use $Val_1(\gamma)$ to denote the value m of a $c_1^{inc_1}$semi-encoding ($c_1^{inc_2}$semi-encoding, $c_1^{inc_3}$semi-encoding, $c_1^{rot_1}$semi-encoding, $c_1^{rot_2}$semi-encoding) γ.

A configuration $\gamma = (I, q, \mathcal{Q}, X)$ is said to be a *Type* 1*a semi-encoding* if it satisfies the following two conditions:

- $q = tmp_1^\imath$ for some increment (of the form $\imath = (s_1, c_1{+}{+}, s_2)$).
- γ is both a c_2-encoding and a $c_1^{inc_1}$semi-encoding.

In such a case, we define $sig(\gamma)$ of γ to be the triple $(tmp_1^\imath, m_1, m_2)$, where $m_1 = Val_1(\gamma)$ and $m_2 = Val_2(\gamma)$. Also, we define $next(\gamma)$ to be (s_2, m_1, m_2). Intuitively, $next(\gamma)$ is the signature of the configuration which occurs after performing three discrete transitions (by rule $inc_2^\imath$, $inc_3^\imath$ and $inc_4^\imath$ in sequence) in our simulation.

A configuration $\gamma = (I, q, \mathcal{Q}, X)$ is said to be a *Type* 1*b semi-encoding* if it satisfies the following two conditions:

- $q = tmp_2^\imath$ for some increment (of the form $\imath = (s_1, c_1{+}{+}, s_2)$).
- γ is both a c_2-encoding and a $c_1^{inc_2}$semi-encoding.

We define the signature of a $c_1^{inc_2}$semi-encoding as in $c_1^{inc_1}$semi-encoding. Furthermore, we use $next(\gamma)$ to be (s_2, m_1, m_2). Intuitively, $next(\gamma)$ is the signature of the configuration which occurs next after performing two discrete transitions (by rules $inc_3^\imath$ and $inc_4^\imath$ in sequence) in our simulation.

Similarly, we define a *Type* 1*c semi-encoding*, it signature and the function *next* for such a semi-encoding.

A configuration $\gamma = (I, q, \mathcal{Q}, X)$ is said to be a *Type* 1*d semi-encoding* if it satisfies the following two conditions:

- $q = tmp_{11}^s$ for some controller state s in C or $q = tmp_1^\imath$ for some zero-testing instruction (of the form $\imath = (s_1, c_1?0, s_2)$).
- γ is both a c_2-encoding and a $c_1^{rot_1}$semi-encoding.

The signature for a $c_1^{rot_1}$semi-encoding is defined in a similar manner to a $c_1^{inc_1}$semi-encoding. Furthermore, the function $next(\gamma)$ is defined as (s, m_1, m_2) if $q = tmp_{11}^s$, $(s_2, 0, m_2)$ otherwise.

A configuration $\gamma = (I, q, \mathcal{Q}, X)$ is said to be a *Type* 1*e semi-encoding* if it satisfies the following two conditions:

- $q = tmp_{12}^s$ for some controller state s in C, or $q = tmp_2^\imath$ for some zero-testing instruction (of the form $\imath = (s_1, c_1?0, s_2)$).
- γ is both a c_2-encoding and a $c_1^{rot_2}$semi-encoding.

The signature and the function $next(\gamma)$ can be defined for a $c_1^{rot_2}$semi-encoding in a similar manner to a $c_1^{rot_1}$semi-encoding. In the following, sometimes we use semi-encoding to means semi-encodings of some Type. The notion of a (semi-)encoding can be extended to a *proper* (semi-)encoding in the same manner as before (Chapter 8), i.e. we require clocks of all processes which are not idle to have values strictly between zero and one.

Proof of Theorem 10.6

The if-direction follows immediately from the the following lemma.

Lemma 10.12. For any configuration $\gamma = (I, q, \mathcal{Q}, X)$ and initial configuration γ_{init} in $\mathcal{N}_{\mathsf{C}}$, if $\gamma_{init} \xrightarrow{*} \gamma$ then one of the following holds.

1. q is not a member of S, (i.e. q is either a temporary state or the state $idle^c$).
2. γ is an encoding such that $\beta_{init} \overset{*}{\rightsquigarrow} sig(\gamma)$.

The only-if-direction follows from the following lemma.

Lemma 10.13. If $\beta_{init} \overset{n}{\rightsquigarrow} s_F$ then $\gamma_{init} \xrightarrow{*} s_F$, for each $n \geq 0$ and initial configuration γ_{init} of $\mathcal{N}$ with $|\gamma_{init}| \geq n + 4$.

The reason for the condition $|\gamma_{init}| \geq n + 4$ is that the sum of counter values never exceeds n in the path from β_{init} to s_F. Furthermore, each c_1- (or c_2)-encoding uses $m + 2$ processes for representing a counter value m. The lemma then states that the initial configuration, from which we start the simulation of the path from β_{init} to s_F, should be sufficiently large to incorporate all counter values which arise along that path.

The proofs of Lemma 10.12 and Lemma 10.13 reflect the informal arguments provided together with each rule in Section 10.2.2.

Proof of Lemma 10.12

Suppose that $\gamma_{init} \xrightarrow{*} \gamma$. If $\gamma_{init} \longrightarrow_{Time} \gamma$ then the result follows immediately. Otherwise, $\gamma_{init} \overset{*}{\Longrightarrow} \gamma$, i.e., there is a sequence

$$\gamma_{init} = \gamma_0 \Longrightarrow_{r_0} \gamma_1 \Longrightarrow_{r_1} \gamma_2 \Longrightarrow_{r_2} \cdots \Longrightarrow_{r_{n-1}} \gamma_n = \gamma$$

Let $\gamma_i = (I, q_i, \mathcal{Q}_i, X_i)$ for $i : 0 \leq i \leq n$. We notice that $q_0 = idle^c$. By definition of the rules, it must be the case that $r_0 = init_1, r_1 = init_2, r_2 = init_3$ and therefore $q_1 = s^1_{init}$, $q_2 = s^2_{init}$ and $q_3 = s^3_{init}$. In other words, $\gamma_0, \ldots, \gamma_3$ satisfy the claim of the Lemma. Lemma 10.12 follows from the following property:

For each $4 \leq i \leq n$, it is the case that γ_i is either

- an encoding with $\beta_{init} \overset{*}{\rightsquigarrow} sig(\gamma_i)$; or
- a semi-encoding with $\beta_{init} \overset{*}{\rightsquigarrow} next(\gamma_i)$.

This property is shown using an induction on i. For the base case we observe that, by definition of the rules, it follows that $r_3 = init_4$ and therefore $sig(\gamma_4) = \beta_{init}$. For the induction step, we observe that, for each $i : 4 \leq i < n$, it follows from the rule definitions that one of the following cases is satisfied:

1. $r_i = inc_1^\imath$ for some $\imath = (s_1, c_1{+}{+}, s_2)$, γ_i is an encoding with $sig(\gamma_i) = (s_1, m_1, m_2)$, and γ_{i+1} is a Type $1a$ semi-encoding with $sig(\gamma_{i+1}) = (tmp_1^\imath, m_1 + 1, m_2)$.

2. $r_i = inc_2^\imath$ for some $\imath = (s_1, c_1{+}{+}, s_2)$, γ_i is a Type $1a$ semi-encoding with $sig(\gamma_i) = (tmp_1^\imath, m_1, m_2)$, and γ_{i+1} is a Type $1b$ semi-encoding with $sig(\gamma_{i+1}) = (tmp_2^\imath, m_1, m_2)$.

3. $r_i = inc_3^\imath$ for some $\imath = (s_1, c_1{+}{+}, s_2)$, γ_i is a Type $1b$ semi-encoding with $sig(\gamma_i) = (tmp_2^\imath, m_1, m_2)$, and γ_{i+1} is a Type $1c$ semi-encoding with $sig(\gamma_{i+1}) = (tmp_3^\imath, m_1, m_2)$.

4. $r_i = inc_4^\imath$ for some $\imath = (s_1, c_1{+}{+}, s_2)$, γ_i is a Type $1c$ semi-encoding with $sig(\gamma_i) = (tmp_3^\imath, m_1, m_2)$, and γ_{i+1} is an encoding with $sig(\gamma_{i+1}) = (s_2, m_1, m_2)$.

5. $r_i = dec^\imath$ for some $\imath = (s_1, c_1{-}{-}, s_2)$, γ_i is an encoding with $sig(\gamma_i) = (s_1, m_1, m_2)$, $m_1 > 0$, and γ_{i+1} is an encoding with $sig(\gamma_{i+1}) = (s_2, m_1 - 1, m_2)$.

6. $r_i = rot_{1,1}^s$ for some $s \in S$ and γ_i is an encoding with $sig(\gamma_i) = (s, m_1, m_2)$, and γ_{i+1} is a Type $1d$ semi-encoding with $sig(\gamma_{i+1}) = (tmp_{11}^s, m_1, m_2)$.

7. $r_i \in \left\{rot_{1,2}^s, rot_{1,3}^s\right\}$ for some $s \in S$ and γ_i is a Type $1d$ semi-encoding with $sig(\gamma_i) = (tmp_{11}^s, m_1, m_2)$, and γ_{i+1} is a Type $1e$ semi-encoding with $sig(\gamma_{i+1}) = (tmp_{12}^s, m_1, m_2)$.

8. $r_i = rot_{1,4}^s$ for some $s \in S$ and γ_i is a Type $1e$ semi-encoding with $sig(\gamma_i) = (tmp_{12}^s, m_1, m_2)$, and γ_{i+1} is an encoding with $sig(\gamma_{i+1}) = (s, m_1, m_2)$.

9. $r_i = tst_1^\imath$ for some $\imath = (s_1, c_1 = 0?, s_2)$, and γ_i is an encoding with $sig(\gamma_i) = (s_1, 0, m_2)$ is and γ_{i+1} is a Type $1d$ semi-encoding with $sig(\gamma_{i+1}) = (tmp_1^\imath, 0, m_2)$.

10. $r_i = tst_2^\imath$ for some $\imath = (s_1, c_1 = 0?, s_2)$, and γ_i is a Type $1d$ semi-encoding with $sig(\gamma_i) = (tmp_1^\imath, 0, m_2)$ is and γ_{i+1} is a Type $1e$ semi-encoding with $sig(\gamma_{i+1}) = (tmp_2^\imath, 0, m_2)$.

11. $r_i = tst_3^\imath$ for some $\imath = (s_1, c_1 = 0?, s_2)$, and γ_i is a Type $1e$ semi-encoding with $sig(\gamma_i) = (tmp_2^\imath, 0, m_2)$ is and γ_{i+1} is an encoding with $sig(\gamma_{i+1}) = (s_2, 0, m_2)$.

12. Similar cases corresponding to instructions which change counter c_2.

Proof of Lemma 10.13

To show Lemma 10.13 we use some definitions.

Let $\gamma = (I, q, \mathcal{Q}, X)$ be a configuration in our simulation. We define $Latest_1(\gamma) = max(X_2(i) : i \in I \wedge \mathcal{Q}(i) \in \{fst_1, mid_1\})$. In other words,

$Latest_1(\gamma)$ is the highest among values of clocks belonging to processes which are part of the (semi-)c_1-encoding. We define $Latest_2(\gamma)$ in a similar manner, and define $Latest(\gamma) = max(Latest_1(\gamma), Latest_2(\gamma))$. We also define $Next2Latest_1(\gamma) = max(X_k(i) : k \in \{1,2\} \wedge i \in I \wedge \mathcal{Q}(i) \in \{fst_1, mid_1, last_1\} \wedge X_k(i) < Latest_1(\gamma))$. In other words, $Next2Latest_1(\gamma)$ is the next highest among values of clocks belonging to processes which are part of the (semi-)c_1-encoding. We define $Next2Latest_2(\gamma)$ in a similar manner.

Let $Delay_1(\gamma)$ be the size of the set consisting of clock values of the form $X_i(j)$ where $i \in \{1,2\}, j \in I, \mathcal{Q}(j) \in \{fst_2, mid_2, last_2\}$ and $Latest_1(\gamma) < X_i(j)$. In other words, $Delay_1(\gamma)$ is the number of clocks which are part of the c_2-encoding and which have values higher than any clock of a process which is part of the c_1-encoding. We define $Delay_2(\gamma)$ in a similar manner. Notice that it may be the case that both $Delay_1(\gamma) = 0$ and $Delay_2(\gamma) = 0$ (if the maximum clock values are equal in the c_1- and the c_2-encoding).

We define another temporary encoding: *almost proper* c_1-encoding which is a c_1 encoding with one process having index $i \in I$ s.t $\mathcal{Q}(i) = mid_1$ and $X_2(i) > 1$ while all processes with index $j \in I$, $\mathcal{Q}(j) \neq idle^p$ and $j \neq i$ satisfies $0 < X_1(j), X_2(j) < 1$. Also, $0 < X_1(i) < 1$. An almost proper encoding of Type 1 is an almost proper c_1-encoding and a proper c_2-encoding. Similarly, we define an almost proper c_2-encoding and an almost proper encoding of Type 2. We also extend this notion of almost proper encodings to almost proper semi-encodings.

Lemma 10.13 follows immediately from the following lemma.

Lemma 10.14. For each $n \geq 0$ and initial configuration γ_{init}, if $\beta_{init} \overset{n}{\rightsquigarrow} \beta$ and $|\gamma_{init}| \geq n+4$, then there exists a proper encoding γ such that $\gamma_{init} \overset{*}{\Longrightarrow} \gamma$ and $sig(\gamma) = \beta$.

Proof. We prove this lemma by induction on n.

In the base case ($n = 0$), we have $\beta = \beta_{init}$ and $|\gamma_{init}| \geq^m 4$. By the definition of $init_1$, this rule is enabled. Let γ_1 be such that $\gamma_{init} \longrightarrow_{init_1} \gamma_1$. Define $\gamma_2 = \gamma_1^{+t_1}$ with $0 < t_1 < 1$. We have $\gamma_1 \longrightarrow_{T=t_1} \gamma_2$. Rule $init_2$ is now enabled. Let γ_3 be such that $\gamma_2 \longrightarrow_{init_2} \gamma_3$. Define $\gamma_4 = \gamma_3^{+t_2}$ with $0 < t_2 < 1 - Latest(\gamma_3)$. We have $\gamma_3 \longrightarrow_{T=t_2} \gamma_4$. Rule $init_3$ is now enabled. Let γ_5 be such that $\gamma_4 \longrightarrow_{init_3} \gamma_5$. Define $\gamma_6 = \gamma_5^{+\delta_3}$ with $0 < \delta_3 < 1 - Latest(\gamma_5)$. We have $\gamma_5 \longrightarrow_{T=\delta_3} \gamma_6$. Rule $init_4$ is now enabled. Let γ_7 be such that $\gamma_6 \longrightarrow_{init_4} \gamma_7$. By definition of $init_4$, γ_7 is an encoding and $sig(\gamma_7) = \beta_{init}$. Let δ_4 be such that $0 < \delta_4 < 1 - Latest(\gamma_7)$. δ_4 exists by the definition of δ_1, δ_2, δ_3, $init_1$, $init_2$, $init_3$ and $init_4$. Let $\gamma_8 = \gamma_7^{+\delta_4}$. γ_8 is a proper encoding with $sig(\gamma_8) = \beta_{init}$. Notice that the transitions $\longrightarrow_{init_1}$, $\longrightarrow_{init_2}$, $\longrightarrow_{init_3}$ and $\longrightarrow_{init_4}$ are enabled only because $|\gamma_{init}| \geq 4$.

For the induction step, assume that $\beta_{init} \overset{n+1}{\rightsquigarrow} \beta$ and $|\gamma_{init}| \geq n+5$. We know that there is a β_1 with $\beta_{init} \overset{n}{\rightsquigarrow} \beta_1 \rightsquigarrow \beta$. By the induction hypothesis, it follows that there is a proper encoding γ_1 such that $sig(\gamma_1) = \beta_1$ and $\gamma_{init} \overset{*}{\Longrightarrow} \gamma_1$. We need to show that there is a proper encoding γ with $sig(\gamma) = \beta$ and $\gamma_1 \overset{*}{\Longrightarrow} \gamma$. This follows from the following lemma. □

Lemma 10.15. Let β_1 and β_2 be configurations of C, where $\beta_1 \rightsquigarrow \beta_2$ and β_2 is of the form (s, m_1, m_2). Let γ_1 be a proper encoding such that $sig(\gamma_1) = \beta_1$ and $|\gamma_1| \geq m_1 + m_2 + 4$. There is a proper encoding γ_2 such that $sig(\gamma_2) = \beta_2$ and $\gamma_1 \stackrel{*}{\Longrightarrow} \gamma_2$.

The proof of Lemma 10.15 follows from Lemma 10.16, Lemma 10.17, Lemma 10.18, Lemma 10.19, Lemma 10.25, and Lemma 10.26:

- Lemma 10.16, Lemma 10.17, Lemma 10.18 and Lemma 10.19 state that an *increment* can be simulated by an application of the rule $inc_1^\imath$ followed by an application of the rule $inc_2^\imath$, followed by an application of rule $inc_3^\imath$, followed by an application of rule $inc_4^\imath$.
- Lemma 10.25 states that a *decrement* can be simulated by the rule $dec^\imath$ preceded and followed by a number of rotations. This lemma follows from Lemma 10.20, Lemma 10.21, Lemma 10.22, Lemma 10.23, and Lemma 10.24.
- Lemma 10.26 deals with zero testing and is similar to Lemma 10.25.

The condition $|\gamma_1| \geq m_1 + m_2 + 4$ in the claim of Lemma 10.15 is relevant only in Lemma 10.16, since this is the only case where the value of a counter is increased.

Lemma 10.16. Consider an instruction $\imath = (s_1, c_1{+}{+}, s_2)$. Let γ_1 be a proper encoding with $sig(\gamma_1) = (s_1, m_1, m_2)$ and $|\gamma_1| \geq m_1 + m_2 + 5$. There is a proper semi-encoding γ_2 of Type 1*a* such that $sig(\gamma_2) = (tmp_1^\imath, m_1 + 1, m_2)$ and $\gamma_1 \Longrightarrow_{inc_1^\imath} \gamma_2$.
A similar result holds in case $\imath$ is of the form $(s_1, c_2{+}{+}, s_2)$.

Proof. Since $|\gamma_1| \geq m_1 + m_2 + 5$, there is at least one process in γ_1 whose state is $idle^p$ (we need $m_1 + 2$ processes for the c_1-encoding and $m_2 + 2$ processes for the c_2-encoding, which means that we have at least one process left to be in state $idle^p$). This together with the fact that γ_1 is a proper encoding implies that $inc_1^\imath$ is enabled from γ_1, i.e., there is configuration γ_3 with $\gamma_1 \longrightarrow_{inc_1^\imath} \gamma_3$. Define $\gamma_2 = \gamma_3^{+\delta}$ where $0 < \delta < 1 - Latest(\gamma_3)$. Such a δ exists by definition of $inc_1^\imath$ and since γ_1 is a proper encoding. By the definitions it follows that γ_2 is a proper semi-encoding of Type 1*a* with $sig(\gamma_2) = (tmp_1^\imath, m_1 + 1, m_2)$ and $\gamma_1 \longrightarrow_{inc_1^\imath} \gamma_3 \longrightarrow_{T=\delta} \gamma_2$. □

Lemma 10.17. Consider an instruction $\imath = (s_1, c_1{+}{+}, s_2)$. Let γ_1 be a proper semi-encoding of Type 1*a* with $sig(\gamma_1) = (tmp_1^\imath, m_1, m_2)$ and $|\gamma_1| \geq m_1 + m_2 + 4$. There is a proper semi-encoding γ_2 of Type 1*b* such that $sig(\gamma_2) = (tmp_2^\imath, m_1, m_2)$ and $\gamma_1 \Longrightarrow_{inc_2^\imath} \gamma_2$.
A similar result holds in case $\imath$ is of the form $(s_1, c_2{+}{+}, s_2)$.

Proof. The fact that γ_1 is a proper semi-encoding of Type 1*a* implies that $inc_2^\imath$ is enabled from γ_1, i.e., there is configuration γ_3 with $\gamma_1 \longrightarrow_{inc_2^\imath} \gamma_3$.

Define $\gamma_2 = \gamma_3^{+\delta}$ where $0 < \delta < 1 - \mathit{Latest}(\gamma_3)$. Such a δ exists by definition of $\mathit{inc}_2^\imath$ and since γ_1 is a proper encoding. By the definitions it follows that γ_2 is a proper semi-encoding of Type 1b with $\mathit{sig}(\gamma_2) = (\mathit{tmp}_2^\imath, m_1, m_2)$ and $\gamma_1 \longrightarrow_{\mathit{inc}_2^\imath} \gamma_3 \longrightarrow_{T=\delta} \gamma_2$. □

Lemma 10.18. Consider an instruction $\imath = (s_1, c_1{+}{+}, s_2)$. Let γ_1 be a semi-encoding of Type 1b with $\mathit{sig}(\gamma_1) = (\mathit{tmp}_2^\imath, m_1, m_2)$. There is a proper semi-encoding γ_2 of Type 1c such that $\mathit{sig}(\gamma_2) = (\mathit{tmp}_3^\imath, m_1, m_2)$ and $\gamma_1 \Longrightarrow_{\mathit{inc}_3^\imath} \gamma_2$.
A similar result holds in case $\imath$ is of the form $(s_1, c_2{+}{+}, s_2)$.

Proof is similar to that of Lemma 10.17.

Lemma 10.19. Consider an instruction $\imath = (s_1, c_1{+}{+}, s_2)$. Let γ_1 be a proper semi-encoding of Type 1c with $\mathit{sig}(\gamma_1) = (\mathit{tmp}_3^\imath, m_1, m_2)$. There is a proper encoding γ_2 such that $\mathit{sig}(\gamma_2) = (s_2, m_1, m_2)$ and $\gamma_1 \Longrightarrow_{\mathit{inc}_3^\imath} \gamma_2$.
A similar result holds in case $\imath$ is of the form $(s_1, c_2{+}{+}, s_2)$.

Proof. Since γ_1 is a proper semi-encoding of Type 1c, it follows that $\mathit{inc}_4^\imath$ is enabled from γ_1, i.e., there is a configuration γ_3 with $\gamma_1 \longrightarrow_{\mathit{inc}_4^\imath} \gamma_3$. Define $\gamma_2 = \gamma_3^{+\delta}$ where $0 < \delta < 1 - \mathit{Latest}(\gamma_3)$. Such a δ exists by definition of $\mathit{inc}_4^\imath$ and since γ_1 is proper semi-encoding of Type 1c. By the definitions it follows that γ_2 is a proper encoding with $\mathit{sig}(\gamma_2) = (s_2, m_1, m_2)$ and $\gamma_1 \longrightarrow_{\mathit{inc}_4^\imath} \gamma_3 \longrightarrow_{T=\delta} \gamma_2$. □

Lemma 10.20. Let γ_1 be an (almost) proper c_2-encoding with $\mathit{Delay}_1(\gamma_1) > 0$ and $\mathit{sig}(\gamma_1) = (s, m_1, m_2)$. There is an (almost) proper semi-encoding of Type 2d, γ_2 such that $\mathit{Delay}_1(\gamma_1) - \mathit{Delay}_1(\gamma_2) \in \{0, 1\}$, $\mathit{sig}(\gamma_2) = (\mathit{tmp}_{21}^s, m_1, m_2)$, and $\gamma_1 \Longrightarrow_{\mathit{rot}_{2,1}^s} \gamma_2$.
A similar result holds in case $\mathit{Delay}_2(\gamma_1) > 0$.

Proof. Now, γ_1 is an (almost) proper c_2-encoding and by definition of the rule $\mathit{rot}_{2,1}^s$, $\mathit{rot}_{2,1}^s$ is enabled from γ_1 and there is a γ_3 with $\gamma_1 \longrightarrow_{\mathit{rot}_{2,1}^s} \gamma_3$. Define $\gamma_2 = \gamma_3^{+\delta}$ where $0 < \delta < 1 - \mathit{Latest}(\gamma_3)$ if γ_1 is a proper encoding, and $0 < \delta < 1 - \mathit{max}(\mathit{Next2Latest}_2(\gamma_3), \mathit{Latest}_1(\gamma_3))$ otherwise. Existence of δ follows from the definition of the rule $\mathit{rot}_{2,1}^s$, and the fact that γ_1 is an (almost) proper c_2-encoding. By the definitions it follows that γ_2 is an (almost) proper semi-encoding of Type 2d with

- $\mathit{Delay}_1(\gamma_2) = \mathit{Delay}_1(\gamma_1) - 1$ if $\mathit{Latest}_2(\gamma_1)$ is strictly smaller than the value of the clock x_1 of the process in state fst_2.
- $\mathit{Delay}_1(\gamma_2) = \mathit{Delay}_1(\gamma_1)$ otherwise.

with $\mathit{sig}(\gamma_2) = (\mathit{tmp}_{21}^s, m_1, m_2)$ and $\gamma_1 \longrightarrow_{\mathit{rot}_{2,1}^s} \gamma_3 \longrightarrow_{T=\delta_2} \gamma_2$. □

Lemma 10.21. Let γ_1 be an (almost) proper semi-encoding of Type 2d with $\mathit{Delay}_1(\gamma_1) > 0$ such that $\mathit{sig}(\gamma_1) = (\mathit{tmp}_{21}^s, m_1, m_2)$. There is an almost proper semi-encoding of Type 2e, γ_2 such that $\mathit{Delay}_1(\gamma_1) - \mathit{Delay}_1(\gamma_2) \in \{0, 1\}$, $\mathit{sig}(\gamma_1) = (\mathit{tmp}_{22}^s, m_1, m_2)$, and either $\gamma_1 \Longrightarrow_{\mathit{rot}_{2,2}^s} \gamma_2$ or $\gamma_1 \Longrightarrow_{\mathit{rot}_{2,3}^s} \gamma_2$.

A similar result holds in case $Delay_2(\gamma_1) > 0$.

Proof. We distinguish between two cases, namely when $m_2 > 0$ and when $m_2 = 0$.

First, we assume that $m_2 > 0$. Now there are two cases.

- γ_1 is a proper semi-conding of Type 2d. To apply $rot^s_{2,2}$, we need to first define δ_1 such that $1 - Latest(\gamma_1) < \delta_1 < 1 - max(Next2Latest_2(\gamma_1), Latest_1(\gamma_1))$ and $\gamma_3 = \gamma_1^{+\delta_1}$. δ_1 exists due to the fact that γ_1 is a proper semi-encoding of Type 2d and by definition of the rule $rot^s_{2,1}$. From the value of δ_1 and the fact that $m_2 > 0$ it follows that $rot^s_{2,2}$ is enabled from γ_3, i.e., there is a γ_4 with $\gamma_3 \longrightarrow_{rot^s_{2,2}} \gamma_4$.
- γ_1 is an almost proper semi-conding of Type 2d. Now, $rot^s_{2,2}$ is enabled from γ_1, i.e., there is a γ_4 with $\gamma_1 \longrightarrow_{T=0} \longrightarrow_{rot^s_{2,2}} \gamma_4$.

Define $\gamma_2 = \gamma_4^{+\delta_2}$ where $0 < \delta < 1 - max(Next2Latest_2(\gamma_4), Latest_1(\gamma_4))$. Existence of δ_2 follows from the definition of the rule $rot^s_{2,2}$, existence of δ_1 and the fact that γ_1 is an (almost) proper semi-encoding of Type 2*d*. By the definitions it follows that γ_2 is an almost proper semi-encoding of Type 2*e* with

- $Delay_1(\gamma_2) = Delay_1(\gamma_1)$ if the clock x_1 of the process in $last_2$ is smaller than or equal to $Latest_1(\gamma_1)$, in other words, if $Delay_1(\gamma_1) = 1$,
- $Delay_1(\gamma_2) = Delay_1(\gamma_1) - 1$, otherwise.

and $sig(\gamma_2) = (tmp^s_{22}, m_1, m_2)$.

The case when $m_2 = 0$ is similar. Here we replace the rule $rot^s_{2,2}$ by the rule $rot^s_{2,3}$, and obtain $\gamma_1 \longrightarrow_{T=\delta_1} \gamma_3 \longrightarrow_{rot^s_{2,3}} \gamma_4 \longrightarrow_{T=\delta_2} \gamma_2$. □

Lemma 10.22. Let γ_1 be an almost proper semi-encoding of Type 2*e* with $Delay_1(\gamma_1) > 0$ such that $sig(\gamma_1) = (tmp^s_{22}, m_1, m_2)$. There is a proper encoding γ_2 such that $Delay_1(\gamma_2) = Delay_1(\gamma_1) - 1$, with $sig(\gamma_1) = (s, m_1, m_2)$, and $\gamma_1 \Longrightarrow_{rot^s_{2,4}} \gamma_2$.

A similar result holds in case $Delay_2(\gamma_1) > 0$.

Proof. From the fact that γ_1 is an almost proper semi-encoding of Type 2*e* and the fact that $m_2 > 0$ it follows that $rot^s_{2,4}$ is enabled from γ_1, i.e., there is a γ_3 with $\gamma_1 \longrightarrow_{rot^s_{2,4}} \gamma_3$. Define $\gamma_2 = \gamma_3^{+\delta_1}$ where $0 < \delta_1 < 1 - Latest(\gamma_3)$. Existence of δ_1 follows from the definition of the rule $rot^s_{2,4}$, and the fact that γ_1 is an almost proper semi-encoding of Type 2*e*. By the definitions it follows that γ_2 is a proper encoding with $Delay_1(\gamma_2) = Delay_1(\gamma_1) - 1$ (due to the fact that the largest clock in the semi-encoding of Type 2e is reset and $Delay_1(\gamma_1) > 0$), and $sig(\gamma_2) = (s, m_1, m_2)$. □

Lemma 10.23. Let γ_1 be an almost proper semi-encoding of Type 2*e* with $Delay_1(\gamma_1) > 0$ such that $sig(\gamma_1) = (tmp^s_{23}, m_1, m_2)$. There is a proper encoding γ_2 such that $Delay_1(\gamma_2) = Delay_1(\gamma_1) - 1$, with $sig(\gamma_1) = (s, m_1, m_2)$, and $\gamma_1 \Longrightarrow_{rot^s_{2,4}} \gamma_2$.

A similar result holds in case $Delay_2(\gamma_1) > 0$.

Proof of this lemma is similar to that of the previous one.

Lemma 10.24. Consider an instruction $\imath = (s_1, c_1{-}{-}, s_2)$. Let γ_1 be a proper encoding with $sig(\gamma_1) = (s_1, m_1, m_2)$ and $m_1 > 0$. If $Delay_1(\gamma_1) = 0$ then there is a proper encoding γ_2 such that $sig(\gamma_2) = (s_2, m_1 - 1, m_2)$ and one of the following holds.

1. If $Latest_1(\gamma_1) > Latest_2(\gamma_1)$ then $\gamma_1 \Longrightarrow_{dec^\imath} \gamma_2$.
2. If $Latest_1(\gamma_1) = Latest_2(\gamma_1)$ and $m_2 > 0$ then $\gamma_1 \Longrightarrow_{dec^\imath} \circ \Longrightarrow_{rot^{s_2}_{2,1}} \circ \Longrightarrow_{rot^{s_2}_{2,2}} \circ \Longrightarrow_{rot^{s_2}_{2,4}} \gamma_2$.
3. If $Latest_1(\gamma_1) = Latest_2(\gamma_1)$ and $m_2 = 0$ then $\gamma_1 \Longrightarrow_{dec^\imath} \circ \Longrightarrow_{rot^{s_2}_{2,1}} \circ \Longrightarrow_{rot^{s_2}_{2,3}} \circ \Longrightarrow_{rot^{s_2}_{2,4}} \gamma_2$.

A similar result holds in case $\imath$ is of the form $(s_1, c_2{-}{-}, s_2)$.

Proof. Define $\gamma_3 = \gamma_1^{+\delta_1}$ where

- $1 - Latest_1(\gamma_1) < \delta_1 < 1 - max(Next2Latest_1(\gamma_1), Latest_2(\gamma_1))$ if $Latest_1(\gamma_1) > Latest_2(\gamma_1)$.
- $1 - Latest_1(\gamma_1) < \delta_1 < 1 - max(Next2Latest_1(\gamma_1), Next2Latest_2(\gamma_1))$ if $Latest_1(\gamma_1) = Latest_2(\gamma_1)$.

Such a δ_1 exists since γ_1 is a proper encoding. From the definition of δ_1 and the fact that $m_1 > 0$ it follows that $dec^\imath$ is enabled from γ_3, i.e., there is a γ_4 with $\gamma_3 \longrightarrow_{dec^\imath} \gamma_4$. Now there are three cases depending on the values of $Latest_1(\gamma_1)$ and $Latest_2(\gamma_1)$ as follows:

1. If $Latest_1(\gamma_1) > Latest_2(\gamma_1)$ then define $\gamma_2 = \gamma_4^{+\delta_2}$ where $0 < \delta_2 < 1 - Latest(\gamma_4)$. Existence of δ_2 follows from the manner in which δ_1 is chosen, definition of the rule $dec^\imath$, and since γ_1 is a proper encoding. By the definitions it follows that γ_2 is a proper encoding with $sig(\gamma_2) = (s_2, m_1 - 1, m_2)$, and $\gamma_1 \longrightarrow_{T=\delta_1} \gamma_3 \longrightarrow_{dec^\imath} \gamma_4 \longrightarrow_{T=\delta_2} \gamma_2$.
2. If $Latest_1(\gamma_1) = Latest_2(\gamma_1)$ and $m_2 > 0$. γ_1 is a proper c_2-encoding, but γ_4 is an almost proper c_2-encoding with $sig(\gamma_4) = (s_2, m_1 - 1, m_2)$, From this fact, it is clear that the rule $rot^{s_2}_{2,1}$ is enabled from γ_4, i.e., there is a γ_5 with $\gamma_4 \longrightarrow_{rot^{s_2}_{2,1}} \gamma_5$. Define $\gamma_6 = \gamma_5^{+\delta_2}$ where $0 < \delta_2 < 1 - max(Next2Latest_2(\gamma_5), Latest_1(\gamma_5))$. Existence of δ_2 follows from the manner in which δ_1 is chosen, definitions of the rules $dec^\imath$ and $rot^{s_2}_{2,1}$, and since γ_1 is a proper encoding. By the definitions it follows that γ_6 is an almost proper semi-encoding of Type 2d with $sig(\gamma_6) = \left(tmp^{s_2}_{2,2}, m_1 - 1, m_2\right)$. From the definition of $dec^\imath$ and the condition that $Latest_1(\gamma_1) = Latest_2(\gamma_1)$, it follows that the largest clock value of the processes in the c_2-encoding has value larger than 1 in γ_6. Therefore, the

rule $rot^{s_2}_{2,2}$ is enabled from γ_6, i.e., there is a γ_7 with $\gamma_6 \longrightarrow_{rot^{s_2}_{2,2}} \gamma_7$. Define $\gamma_8 = \gamma_7^{+\delta_3}$ where $0 < \delta_3 < 1 - max(Next2Latest_2(\gamma_7), Latest_1(\gamma_7))$. Existence of δ_3 follows from the manner in which δ_1, δ_2 are chosen, definitions of the rules $rot^{s_2}_{2,2}$, and since γ_6 is an almost proper semi-encoding of Type 2*d*. By the definitions it follows that γ_8 is an almost proper semi-encoding of Type 2*e* with $sig(\gamma_8) = \left(tmp^{s_2}_{2,2}, m_1 - 1, m_2\right)$. Therefore, the rule $rot^{s_2}_{2,4}$ is enabled from γ_8, i.e., there is a γ_9 with $\gamma_8 \longrightarrow_{rot^{s_2}_{2,4}} \gamma_9$. Define $\gamma_2 = \gamma_9^{+\delta_4}$ where $0 < \delta_4 < 1 - Latest(\gamma_9)$. By the definitions it follows that γ_2 is a proper encoding with $sig(\gamma_2) = (s_2, m_1 - 1, m_2)$. and $\gamma_1 \longrightarrow_{T=\delta_1} \gamma_3 \longrightarrow_{dec^\imath} \gamma_4 \longrightarrow_{rot^{s_2}_{2,1}} \gamma_5 \longrightarrow_{T=\delta_2} \gamma_6 \longrightarrow_{rot^{s_2}_{2,2}} \gamma_7 \longrightarrow_{T=\delta_3} \gamma_8 \longrightarrow_{rot^{s_2}_{2,4}} \gamma_9 \longrightarrow_{T=\delta_4} \gamma_2$.

3. If $Latest_1(\gamma_1) = Latest_2(\gamma_1)$, but $m_2 = 0$. The proof is similar to the previous case. Here, we use the rule $rot^{s_2}_{2,3}$ instead of the rule $rot^{s_2}_{2,2}$ in the above and obtain $\gamma_1 \longrightarrow_{T=\delta_1} \gamma_3 \longrightarrow_{dec^\imath} \gamma_4 \longrightarrow_{rot^{s_2}_{2,1}} \gamma_5 \longrightarrow_{T=\delta_2} \gamma_6 \longrightarrow_{rot^{s_2}_{2,3}} \gamma_7 \longrightarrow_{T=\delta_3} \gamma_8 \longrightarrow_{rot^{s_2}_{2,4}} \gamma_9 \longrightarrow_{T=\delta_4} \gamma_2$.

□

From Lemma 10.20, Lemma 10.21, Lemma 10.22, Lemma 10.23, and Lemma 10.24 we get the following.

Lemma 10.25. Consider an instruction $\imath = (s_1, c_1--, s_2)$. Let γ_1 be a proper encoding with $sig(\gamma_1) = (s_1, m_1, m_2)$ and $m_1 > 0$. There is a proper encoding γ_2 such that $sig(\gamma_2) = (s_2, m_1 - 1, m_2)$ and

$$\gamma_1 \circ \stackrel{*}{\Longrightarrow}_{rot_2^{s_1} \cup rotz_2^{s_1}} \Longrightarrow_{dec^\imath} \circ \stackrel{*}{\Longrightarrow}_{rot_2^{s_2} \cup rotz_2^{s_2}} \gamma_2$$

where for a controller state s, we define $rot_2^s = \left\{rot^s_{2,1}, rot^s_{2,2}, rot^s_{2,4}\right\}$, and $rotz_2^s = \left\{rot^s_{2,1}, rot^s_{2,3}, rot^s_{2,4}\right\}$.

A similar result holds in case $\imath$ is of the form (s_1, c_2--, s_2).

Lemma 10.26. Consider an instruction $\imath = (s_1, c_1 = 0?, s_2)$. Let γ_1 be a proper encoding with $sig(\gamma_1) = (s_1, 0, m_2)$.

Then there is a proper encoding γ_2 such that $sig(\gamma_2) = (s_2, 0, m_2)$ and

$$\gamma_1 \stackrel{*}{\Longrightarrow}_{rot_2^{s_1} \cup rotz_2^{s_1}} \circ \Longrightarrow_{tst_1^\imath} \circ \Longrightarrow_{tst_2^\imath} \circ \Longrightarrow_{tst_3^\imath} \circ \stackrel{*}{\Longrightarrow}_{rot_2^{s_2} \cup rotz_2^{s_2}} \gamma_2$$

where for a controller state s, $rot_2^s, rotz_2^s$ are as defined in Lemma 10.25.

A similar result holds in case $\imath$ is of the form $(s_1, c_2 = 0?, s_2)$.

The proof is similar to that for Lemma 10.25.

Chapter 11

Conclusions

A *parameterized timed system* is a *family* of timed systems each consisting of a finite, but arbitrary number of timed processes (so-called timed automata). The aim of this thesis was to investigate model checking of different properties for parameterized timed systems. The main contribution of this thesis is summarized in the following.

Part I

Petri nets are well-established graphical tools for modelling and analysing concurrent systems. Concurrent real-time systems are usually modelled by timed extensions of Petri nets, so-called *timed Petri nets*. In the first part of this thesis, we show how to solve a number of problems for timed Petri nets.

Coverability

The coverability problem for TPNs ask whether a set of final states can be covered from a set of initial states. Since forward analysis is necessarily incomplete, we provide a semi-algorithm augmented with an acceleration technique in order to make it terminate more often on practical examples.

Next we mention a number of interesting directions for future research in this context.

1. We showed how to accelerate with respect to single discrete transition interleaved with timed transitions. A remaining challenge is to extend the technique and consider accelerations of *sequences* of discrete transitions. It is not clear to us whether such accelerations are computable in the first place.

2. We assume a lazy behaviour of TPNs. It is well-known that checking safety properties is undecidable for TPNs with *urgent* behaviours even if the net is safe (bounded). Therefore, designing acceleration techniques is of particular interest for urgent TPNs. Notice that downward closure is no longer an exact abstraction if the behaviour is urgent.

3. We use *region generators* for symbolic representation. It is necessary to investigate designing efficient data structures (e.g., *zone generators* corresponding to a large number of region generators). *Zones* are widely used in existing tools for verification of timed automata [99, 135]. Intuitively, a zone generator will correspond to a state in each minimized automaton in Figures 5.10, 5.11 and 5.12.

4. We aim at developing generic methods for building downward closed languages, in a similar manner to the methods developed for building upward closed languages in [5]. This would give a general theory for forward analysis of infinite state systems, in the same way the work in [5] is for backward analysis. Simple regular expressions of [2] and the region generators of this paper are examples of data structures which might be developed in a systematic manner within such a theory.

5. It will also be interesting to explore the coverability problem for multi-clock TPNs where each token is equipped with a finite number of real-valued clocks.

6. In a recent work by Raskin, et.al [70], a general algorithmic schema called "Expand, Enlarge and Check" is given from which new algorithms for the coverability problem for well-structured transition systems can be constructed. The authors of [70] show how to verify transfer nets and lossy channel systems by forward analysis. We intend to investigate the application of their method enhanced with our acceleration scheme for our timed Petri net model.

Zenoness

In Chapter 6, we show how to characterize the set of states (zeno-markings) from which infinite computations of finite durations start. To solve the zenoness problem, we defined a new class of untimed Petri nets which is more general than standard Petri nets, but which is a subclass of transfer nets [56]. For these nets, we gave a method to compute a characterization of the set of markings from which there are infinite computations. This is interesting in itself since, for general transfer nets, such a characterization is not computable. In fact, for general transfer nets, the coverability tree may not be finite in general.

We also showed that this result holds both for dense-timed and discrete-timed Petri nets.

The algorithm (Chapter 6) for TPNs considers only one clock per token. We want to investigate whether this algorithm still works for TPNs with multiple clocks per token. This algorithm uses the coverability analysis which may become undecidable when multiple clocks per token is considered (such a result is shown for a variant of TPNs in Chapter 8). Therefore, the conjecture is that in case of TPNs with multiple clocks per token, it will not be possible to have a characterization of markings from which zeno computations start. However, it will also be worthwhile to see whether we can still have such a characterization for discrete-timed Petri nets with multiple clocks per token.

Token Liveness, Boundedness and Repeated Reachability

In Chapter 7 of this thesis, we solve two problems, namely token-liveness (whether a token in a marking is going to be consumed in future) and syntactic boundedness (whether the space required by the reachability set is bounded, while one does not omit dead tokens from the reachable states) for TPNs. We also give undecidability results for semantic boundedness (boundedness while considering only live part of the reachability set), and repeated reachability problem (whether a place is visited infinitely often) for TPNs.

For all the problems studied so far for TPNs, the decidability results coincide for dense-time and discrete time (although the proofs for dense-time are harder), assuming that each token is equipped with just one clock. However, if we consider TPNs with *two* clocks per token, we already mentioned that there is a decidability gap between the dense-time and the discrete time domain. The coverability problem becomes undecidable for dense-time TPNs with only *two* clocks per token (Chapter 8), while it remains decidable for discrete TPNs with any finite number of clocks per token (Chapter 9). It is therefore worth investigating whether this more general class induces a similar gap also for the token-liveness and syntactic boundedness problems.

Part II

In the second part of the thesis, we consider another model for timed systems: so-called *timed networks*. Roughly, a timed network consists of a controller and an arbitrary number of timed processes (timed automata) running in parallel. TPNs are popular in the verification community for their graphical formalism. However, we believe that TNs are sometimes a more natural choice for modelling parameterized timed systems. This is due to the fact that timed networks let us keep the clock values of different participating process in a transition unchanged.

We considered controller-state reachability of timed networks for a number of different syntactic subclasses and semantic variants. We give a summary of Part II in the following.

- In Chapter 8, we have shown undecidability of controller state reachability for timed networks with at least two clocks per timed process. This result is surprising for the following reason. Firstly, it was known from [9] that the problem is decidable for timed networks restricted to one clock per process. Second, it is well-known [24] that for timed automata, one can construct parallel timed automata with single clocks from a multi-clock timed automaton.

- Despite the undecidability result for multi-clock dense-timed networks, we show that the problem is decidable (Chapter 9) if the clocks range over a discrete-time domain.

- Inspired by different syntactic subclasses and semantic variants of timed

automata, we also considered similar restrictions for timed networks. It turns out that if one considers closed timed networks, (CTNs) which supports only non-strict clock constraints, the controller-state reachability problem becomes decidable. This can be useful, since often we can over-approximate a general timed network by a timed network without strict clock constraints and analyse it. However, the problem remains undecidable for open timed networks (OTNs) with only strict clock constraints.

- We also investigated robust semantics for timed networks. It turns out that controller state reachability for this model is undecidable even for robust semantics. The result is shown by reducing the controller state reachability problem for OTNs under standard semantics to the problem for OTNs under robust semantics. We have left open the controller state reachability problem for CTNs with robust semantics.

For timed networks, we only considered safety properties. Liveness properties have been shown to be undecidable for single-clock TNs in [9].

There are several classes of protocols which can be modelled as multi-clock TNs, such as the parameterized versions of the protocols in [27] and [101]. This means that, despite our undecidability result, it is interesting to design semi-algorithms for multi-clock TNs. One direction for future work is to design acceleration techniques which are sufficiently powerful to handle such classes of protocols.

It will also be worthwhile to investigate the problems considered in Chapter 6 and Chapter 7 for the above subclasses of timed networks.

Implementation Effort

We are currently building a compiler which translates parameterized protocols to TPN models. This will enable us to create a package for verification of parameterized protocols. The TPN models generated for Fischer's protocol and Lynch and Shavit's protocol in this thesis are outcomes of this compiler. However, we intend to enhance the specification language and verify properties like safety, zenoness, syntactic boundedness, token-liveness, etc.

Bibliography

[1] P. Abdulla, J. Deneux, P. Mahata, and A. Nylén. Forward reachability analysis of timed petri nets. In *Proc. FORMATS-FTRTFT'04*, volume 3253 of *Lecture Notes in Computer Science*, pages 343–362, 2004.

[2] P. A. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *Proc.* 10^{th} *Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 305–318, 1998.

[3] P. A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parameterized system verification. In *Proc.* 11^{th} *Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145, 1999.

[4] P. A. Abdulla and K. Čerāns. Simulation is decidable for one-counter nets. In *Proc. CONCUR '98,* 9^{th} *Int. Conf. on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 253–268, 1998.

[5] P. A. Abdulla, K. Čerāns, B. Jonsson, and T. Yih-Kuen. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160:109–127, 2000.

[6] P. A. Abdulla, J. Deneux, P. Mahata, and A. Nylén. Forward reachability analysis of timed petri nets. Technical Report 2003-056, Dept. of Information Technology, Uppsala University, Sweden, 2003. http://user.it.uu.se/~pritha/Papers/tpn.ps.

[7] P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.

[8] P. A. Abdulla and B. Jonsson. Verifying networks of timed processes. In Bernhard Steffen, editor, *Proc. TACAS '98,* 4^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 298–312, 1998.

[9] P. A. Abdulla and B. Jonsson. Model checking of systems with many identical timed processes. *Theoretical Computer Science*, 290(1):241–264, 2003.

[10] P. A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular tree model checking. In *Proc.* 14^{th} *Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, 2002.

[11] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model checking made simple and efficient. In *Proc. CONCUR 2002,* 13^{th} *Int. Conf. on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130, 2002.

[12] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Algorithmic improvements in regular model checking. In *Proc.* 15^{th} *Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 236–248, 2003.

[13] P. A. Abdulla and A. Nylén. Better is better than well: On efficient verification of infinite-state systems. In *Proc. LICS' 00* 16^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 132–140, 2000.

[14] P. A. Abdulla and A. Nylén. Timed Petri nets and BQOs. In *Proc. ICATPN'2001: 22nd Int. Conf. on application and theory of Petri nets*, volume 2075 of *Lecture Notes in Computer Science*, pages 53 –70, 2001.

[15] P. A. Abdulla and A. Nylén. Undecidability of ltl for timed petri nets. In *INFINITY 2002, 4th International Workshop on Verification of Infinite-State Systems*, 2002.

[16] R. Alur. *Techniques for Automatic Verification of Real-Time Systems.* PhD thesis, Dept. of Computer Sciences, Stanford University, 1991.

[17] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proc. LICS' 90,* 5^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 414–425, Philadelphia, 1990.

[18] R. Alur and D. Dill. Automata for modelling real-time systems. In *Proc. ICALP '90*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335, 1990.

[19] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[20] R. Alur, T. Henzinger, and M. Vardi. Parametric real-time reasoning. In *Proc.* 25^{th} *ACM Symp. on Theory of Computing*, LNCS, pages 592–601, 1993.

[21] A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *Proc. CAV'00*, volume 1855 of *LNCS*, 2000.

[22] K. Apt and D.C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22:307–309, 1986.

[23] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In Berry, Comon, and Finkel, editors, *Proc.* 13^{th} *Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 221–234, 2001.

[24] E. Asarin, O. Maler, and P. Caspi. Timed regular expressions. *The Journal of ACM*, 49(2):172–206, 2002.

[25] T. Ball, B. Cook, V. Levin, and S.K. Rajamani. Slam and static driver verifier: technology transfer of formal methods inside microsoft. In *Proc. of Integrated Formal Methods '04*, 2004.

[26] M. Ben-Ari. *Mathematical Logic for Computer Science*. Prentice Hall, 1993.

[27] J. Bengtsson, W. O. D. Griffioen, K.J. Kristoffersen, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In *Proc. CAV'96*, volume 1102 of *LNCS*, pages 244–256, 1996.

[28] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New Generation of UPPAAL. In *Int. Workshop on Software Tools for Technology Transfer*, June 1998.

[29] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems automatically and compositionally. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, 1998.

[30] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. on Software Engineering*, 17(3):259–273, 1991.

[31] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In Alur and Henzinger, editors, *Proc.* 8^{th} *Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer Verlag, 1996.

[32] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of fifo-channel systems with nonregular sets of configurations. In *Proc. ICALP '97,* 24^{th} *International Colloquium on Automata, Lnaguages, and Programming*, volume 1256 of *Lecture Notes in Computer Science*, 1997.

[33] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla, editors, *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer Verlag, 2000.

[34] A. Bouajjani and R. Mayr. Model checking lossy vector addition systems. In *Proc. of STACS'99*, volume 1563 of *LNCS*. Springer Verlag, 1999.

[35] A. Bouajjani and R. Mayr. Model checking lossy vector addition systems. In *Symp. on Theoretical Aspects of Computer Science*, volume 1563 of *Lecture Notes in Computer Science*, pages 323–333, 1999.

[36] Ahmed Bouajjani and Tayssir Touili. Extrapolating Tree Transformations. In *Proc.* 14^{th} *Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, 2002.

[37] F. D. J. Bowden. Modelling time in Petri nets. In *Proc. Second Australian-Japan Workshop on Stochastic Models*, 1996.

[38] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.

[39] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit vrifications. *IEEE Trans. on Computer Aided Design and Integrated Circuits and Systems*, 13(13):401–424, April 1994.

[40] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. LICS' 90,* 5^{th} *IEEE Int. Symp. on Logic in Computer Science*, 1990.

[41] O. Burkart and B. Steffen. Composition, decomposition, and model checking of pushdown processes. *Nordic Journal of Computing*, 2(2):89–125, 1995.

[42] K. Čerāns. Deciding properties of integral relational automata. In Abiteboul and Shamir, editors, *Proc. ICALP '94,* 21^{st} *International Colloquium on Automata, Lnaguages, and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 35–46. Springer Verlag, 1994.

[43] E. Clarke, M. Talupur, T. Touili, and H. Veith. Verification by network decomposition. In *Proc. of the 15th International Conference on Concurrency Theory (Concur' 04)*, pages 276–291, 2004.

[44] E. M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Trans. on Programming Languages and Systems*, 16(5), Sept. 1994.

[45] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.

[46] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Proc. of the 5th Workshop on Computer-Aided Verification*, pages 450–462, 1993.

[47] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking (3rd Edition).* The MIT Press, 2001.

[48] E.M. Clarke, D.E. Long, and K.L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In *Proc. of the 9th International Symposium on Computer Hardware Description Languages and Their Applications*, pages 281–295, 1989.

[49] R. Cleaveland, J. Parrow, and B. Steffen. A semantics-based tool for the verification of finite-state systems. In Brinksma, Scollo, and Vissers, editors, *Protocol Specification, Testing, and Verification IX*, pages 287–302. North-Holland, 1989.

[50] J. E. Coolahan and N. Roussopoulos. Timing requirements for time driven system using augmented petri nets. In *IEEE Transactions on Software Engineering*, volume SE9, 1983.

[51] D. de Frutos Escrig, V. Valero Ruiz, and O. Marroquín Alonso. Decidability of properties of timed-arc Petri nets. In *ICATPN 2000*, number 1825, pages 187–206, 2000.

[52] G. Delzanno. Automatic verification of cache coherence protocols. In *Proc. CAV'00*, volume 1855 of *LNCS*, pages 53–68, 2000.

[53] G. Delzanno and J. F. Raskin. Symbolic representation of upward-closed sets. In *Proc. TACAS '00,* 6^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 426–440, 2000.

[54] L. E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *Amercan Journal of Mathematics*, 35:413–422, 1913.

[55] C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. of ICALP'98*, volume 1443 of *LNCS*. Springer Verlag, 1998.

[56] C. Dufourd, P. Jančar, and Ph. Schnoebelen. Boundedness of Reset P/T Nets. In *Proc. of ICALP'99*, volume 1644 of *LNCS*. Springer Verlag, 1999.

[57] E.A. Emerson and V. Kahlon. Model checking guarded protocols. In *Proc. LICS'03*, 2003.

[58] E.A. Emerson and K.S. Namjoshi. Reasoning about rings. In *Proc.* 22^{th} *ACM Symp. on Principles of Programming Languages*, 1995.

[59] E.A. Emerson and A.P. Sistla. Symmetry and model checking. In *Proc. of the 5th Workshop on Computer-Aided Verification*, pages 463–478, 1993.

[60] J. Esparza. On the decidability of model checking for several mu-calculi and Petri nets. In *CAAP '94*, volume 787 of *Lecture Notes in Computer Science*, pages 115–129. Springer Verlag, 1994.

[61] J. Esparza. Petri nets, commutative context-free grammers, and basic parallel processes. In *Proc. Fundementals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 221–232, 1995.

[62] J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.

[63] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. of LICS'99*. IEEE, 1999.

[64] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. LICS'99*, 1999.

[65] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *Proc. TACAS '96,* 2^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 87–106. Springer Verlag, 1996.

[66] A. Finkel. Decidability of the termination problem for completely specified protocols. *Distributed Computing*, 7(3), 1994.

[67] A. Finkel, S. Purushothaman Iyer, and G. Sutre. Well-abstracted transition systems. In *Proc. CONCUR 2000,* 11^{th} *Int. Conf. on Concurrency Theory*, pages 566–580, 2000.

[68] A. Finkel, J.-F. Raskin, M. Samuelides, and L. Van Begin. Monotonic extensions of petri nets: Forward and backward search revisited. In *Proc. Infinity'02*, 2002.

[69] A. Finkel and P. Schnoebelen. Fundamental structures in well-structured infinite transition sytems. In *Proc. LATIN '98*, volume 1380 of *Lecture Notes in Computer Science*, pages 102–118, 1998.

[70] G. Geeraerts, J.F. Raskin, and L.B. Begin. Expand, enlarge, and check. In *Proc. FSTTCS '04*, volume 3328 of *Lecture Notes in Computer Science*, pages 287–298, 2004.

[71] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.

[72] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A unified high-level Petri net formalism for time-critical systems. *IEEE Trans. on Software Engineering*, 17(2):160–172, 1991.

[73] S. Ginsburg. *The Mathematical Theory of Context-free Languages.* McGraw-Hill, 1966.

[74] S. Ginsburg and E. Spanier. Semigroups, Presburger formulas, and languages. *Pacific J. of Mathematics*, 16:285–296, 1966.

[75] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proc. Workshop on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, pages 332–342. Springer Verlag, 1991.

[76] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[77] J.C. Godskesen. *Timed Modal Specifications.* PhD thesis, Aalborg University, 1994.

[78] S. Graf and B. Steffen. Compositional minimazation of finite state processes, 1990.

[79] O. Grumberg and D.E. Long. Model checking and modular verification. In J.C.M. Baseten and J.F. Groote, editors, *Proc. CONCUR '91, Theories of Concurrency: Unification and Extension*, volume 527 of *Lecture Notes in Computer Science*, pages 250–265, Amsterdam, Holland, 1991. Springer Verlag.

[80] V. Gupta, T. Henzinger, and R. Jagadesan. Robust timed automata. In *In Proc. of HART' 97*, volume 1201 of *Lecture Notes in Computer Science*, pages 331–345, 1997.

[81] T. Henzinger and J. Raskin. Robust undecidability of timed and hybrid systems. In *Proc. of HSCC' 00*, volume 1790 of *Lecture Notes in Computer Science*, pages 145–159, 2000.

[82] T.A. Henzinger. Hybrid automata with finite bisimulations. In *Proc. ICALP '95,* 22^{nd} *International Colloquium on Automata, Lnaguages, and Programming*, 1995.

[83] T.A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks. In *Proc. ICALP' 92*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558, 1992.

[84] G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc.*, 2:326–336, 1952.

[85] M. A. Holliday and M. K. Vernon. A generalized timed petri net model for performance analysis. In *IEEE Transactions on Software Engineering*, volume SE13, 1987.

[86] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[87] P. Jančar. Bisimulation equivalence is decidable for one-counter processes. In *Proc. ICALP '97,* 24^{th} *International Colloquium on Automata, Lnaguages, and Programming*, pages 549–559, 1997.

[88] P. Jančar and F. Moller. Checking regular properties of Petri nets. In *Proc. CONCUR '95,* 6^{th} *Int. Conf. on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 348–362. Springer Verlag, 1995.

[89] N. D. Jones, L. H. Landweber, and Y. E. Lyen. Complexity of some problems in Petri nets. *Theoretical Computer Science*, (4):277–299, 1977.

[90] B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, 107(2):272–302, Dec. 1993.

[91] R.M. Karp and R.E. Miller. Parallel program schemata. *Journal of Computer and Systems Sciences*, 3(2):147–195, May 1969.

[92] Shmuel Katz and Doron Peled. An efficient verification method for parallel and distributed programs. In *Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 489–507, 1988.

[93] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proc. 9^{th} Int. Conf. on Computer Aided Verification*, volume 1254, pages 424–435, Haifa, Israel, 1997. Springer Verlag.

[94] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256:93–112, 2001.

[95] K.J. Kristoffersen, F. Larroussinie, K. G. Larsen, P. Pettersson, and W. Yi. A compositional proof of a real-time mutual exclusion protocol. In *Proc. TAPSOFT '97*, LNCS, 1997.

[96] S.K. Lahiri and R.E. Bryant. Indexed predicate discovery for unbounded system verification. In *Proc. 16th International Conference on Computer-Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 135–147, 2004.

[97] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Symbolic techniques for parametric reasoning about counter and clock systems. In *Proc. TACAS '01, 7^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.

[98] K.G. Larsen. Modal specifications. In Sifakis, editor, *Proc. Workshop on Computer Aided Verification*, volume 407 of *Lecture Notes in Computer Science*, pages 232–246. Springer Verlag, 1989.

[99] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1-2), 1997.

[100] N. A. Lynch and N. Shavit. Timing-based mutual exclusion. In *IEEE Real-Time Systems Symposium*, pages 2–11, 1992.

[101] I. A. Mason and C. L. Talcott. Simple network protocol simulation within maude. In *ENTCS*, volume 36. Elsevier, 2001.

[102] R. Mayr. Undecidable problems in unreliable computations. In *Theoretical Informatics (LATIN'2000)*, number 1776 in Lecture Notes in Computer Science, 2000.

[103] R. Mayr. Decidability of model checking with the temporal logic ef. *Theoretical Computer Science*, 256:31–62, 2001.

[104] R. Mayr. Undecidable problems in unreliable computations. *TCS*, 297(1-3):337–354, 2003.

[105] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[106] K.L. McMillan. A technique of a state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.

[107] P. Merlin and D.J. Farber. Recoverability of communication protocols - implications of a theoretical study. *IEEE Trans. on Computers*, COM-24:1036–1043, Sept. 1976.

[108] M. Minsky. Recursive unsolvabitity of post's problem of tag and other topics in the theory of turing machines. *Ann. of Math.*, 74:437–455, 1961.

[109] M. Nielson, V. Sassone, and J. Srba. Towards a distributed time for petri nets. In *Proc. ICATPN'01*, volume 2075 of *Lecture Notes in Computer Science*, pages 23–31, 2001.

[110] J. Ouaknine and J. Worrell. Revisiting digitization, robustness and decidability for timed automata. In *Proc. of LICS' 03*, pages 198–207. IEEE Computer Society Press, 2003.

[111] J. Ouaknine and J. Worrell. Universality and language inclusion for open and closed timed automata. In *Proc. of HSCC' 03*, volume 2623 of *Lecture Notes in Computer Science*, 2003.

[112] W. Overman. *Verification of Concurrent Systems: Function and Timing*. PhD thesis, UCLA, Aug. 1981.

[113] R. J. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, 1966.

[114] D. Peled. Combining partial order reductions with on-the-fly model checking. *Formal Aspects of Computing*, 8:39–64, 1996.

[115] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962.

[116] A. Pnueli. In transition from global to modular temporal reasoning about programs. Technical Report CS-85-05, Weizmann institute, May 1985.

[117] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proc. TACAS '01, 7^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031, pages 82–97, 2001.

[118] A. Puri. Dynamical properties of timed automata. In *Proc. FTRTFT'98*, volume 1486 of *Lecture Notes in Computer Science*, pages 210–227, 1998.

[119] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *5th International Symposium on Programming, Turin*, volume 137 of *Lecture Notes in Computer Science*, pages 337–352. Springer Verlag, 1982.

[120] J.P. Queille and J. Sifakis. A temporal logic to deal with fairness in transition systems. In *Proc. 23^{rd} Annual Symp. Foundations of Computer Science*, pages 217–225, 1982.

[121] C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using petri nets. In *IEEE Transactions on Software Engineering*, volume SE6, 1980.

[122] R. Razouk and C. Phelps. Performance analysis using timed Petri nets. In *Protocol Testing, Specification, and Verification*, pages 561–576, 1985.

[123] V. Valero Ruiz, F. Cuartero Gomez, and D. de Frutos Escrig. On nondecidability of reachability for timed-arc Petri nets. In *Proc. 8th International Workshop on Petri Nets and Performance Models*, pages 188–196, 1999.

[124] F. B. Schneider, B. Bloom, and K. Marzullo. Putting time into proof outlines. In de Bakker, Huizing, de Roever, and Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, 1992.

[125] Ph. Schnoebelen. Verifying lossy channel systems has nonprimitive recursive complexity. *Information Processing Letters*, 83(5):251–261, 2002.

[126] P. D. Scotts and T. W. Pratt. Hierarchical modelling of software systems with timed petri nets. In *1st International Workshop on Timed Petri Nets, Torino, Italy*, July 1985.

[127] A.U. Shankar. An introduction to assertional reasoning for concurrent systems. *Computing Surveys*, 25(3):225–262, Sept. 1993.

[128] I. Suzuki. Proving properties of a ring of finite-state machines. In *Information Processing Letters*, volume 28, pages 213–214, 1988.

[129] S. Tripakis. Verifying progress in times systems. In *Proc. ARTS '99*, pages 299–314, 1999.

[130] R. Valk and M. Jantzen. The Residue of Vector Sets with Applications to Decidability Problems in Petri Nets. *Acta Informatica*, 21:643–674, 1985.

[131] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer-Verlag, 1990.

[132] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS'86*, pages 332–344. IEEE Computer Society Press, 1986.

[133] F. Wang. Formal verification of timed systems: A survey and perspective. 92(8):1283–1305, 2004.

[134] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic (extended abstract). In *Proc.* 13^{th} *ACM Symp. on Principles of Programming Languages*, pages 184–193, Jan. 1986.

[135] S. Yovine. Kronos: A verification tool for real-time systems. *Journal of Software Tools for Technology Transfer*, 1(1-2), 1997.

Index