Automata and Logics for Concurrent Systems: Realizability and Verification

Habilitation Thesis

Mémoire d'Habilitation à Diriger des Recherches

submitted by

Benedikt Bollig

June 2015

Abstract. Automata are a popular tool to make computer systems accessible to formal methods. While classical finite automata are suitable to model sequential boolean programs, models of concurrent systems involve several interacting processes and extend finite-state machines in various respects. This habilitation thesis surveys several such extensions, including pushdown automata with multiple stacks, communicating automata with fixed, parameterized, or dynamic communication topology, and automata running on words over infinite alphabets. We focus on two major questions of classical automata theory, namely realizability (asking whether a specification has an automata counterpart) and model checking (asking whether a given automaton satisfies its specification).

The defense took place on Tuesday, June 2, 2015, 2 p.m., at ENS Cachan, France. The jury members are:

Prof. Ahmed Bouajjani	Reviewer
Prof. Volker Diekert	Examiner
Prof. Paul Gastin	Examiner
Prof. Leonid Libkin	Reviewer
Prof. Anca Muscholl	Reviewer
Prof. Jean-Marc Talbot	Examiner

Author

Benedikt Bollig
Laboratoire Spécification et Vérification
CNRS & ENS Cachan
France
email: `bollig@lsv.ens-cachan.fr`

Thank you, Merci, Danke!

Many people have helped me over the last years, and I would like to use the opportunity to express my gratitude to all of them. The list of people I mention here cannot be exhaustive, but I am aware that many more crossed and influenced my road in one way or another.

My foremost thanks, and my sincerest appreciation, go to my mentor Paul Gastin. I am extremely grateful for his continuous support, advice, and encouragement. Since I defended my PhD, he has had, without any doubt, the most significant impact on my evolution as a researcher. When writing a definition or proof, the best guideline for me has always been to first think about how Paul would write it. And as if all that were not enough, he gave me the opportunity to work with such brilliant PhD students: I am so proud to count Aiswarya Cyriac, Benjamin Monmege, and Akshay Sundararaman among my family, in an academic sense and in terms of friendship. I would consider myself lucky if they have learned half as much from me as I have learned from them.

I am flattered that Ahmed Bouajjani, Volker Diekert, Paul Gastin, Leonid Libkin, Anca Muscholl, and Jean-Marc Talbot readily accepted to be a jury member or a reviewer of my habilitation thesis. I can only hope that this document fulfils the expectations that come with such a high-calibre committee.

I am very grateful to Benjamin Monmege for taking the trouble to read this thesis and for his many accurate, subtle, and pertinent comments.

During my years at LSV, I have been so fortunate to share my office with adorable and courteous people: Nicolas Markey, Graham Steel (together with a curious little cat whose name I do not know), and Claudine Picaronny have created an enjoyable and stimulating atmosphere every day.

I also spent a considerable amount of time in Christoph Haase's office. I sincerely thank Christoph for his support and encouragement, the various thought-provoking discussions we had on emerging trends in computer science, and many excellent (among some not so good) "Kalauers"[1].

[1] http://de.wikipedia.org/wiki/Kalauer

I would like to thank all my co-authors for joining me on this exciting journey that is research. Most results presented here would not have been possible without them. A special mention goes to Dietrich Kuske, Madhavan Mukund, K. Narayan Kumar, and Thomas Schwentick. Besides sharing their immense knowledge with me, they made some memorable research stays possible. Normann Decker, Peter Habermehl, Martin Leucker, and Daniel Thoma kindly let me take part in the LeMon project, which has been another invaluable and fruitful experience.

I could not imagine a better vehicle to undertake my research journey than the LSV, which is blessed with kind and talented people. I am particularly grateful to the MExICo team under the guidance of Stefan Haar, to Stefan Göller and Stefan Schwoon for enlightening and pleasurable conversations on computer science and football science, and to the remarkably efficient and friendly administrative and technical staff for their support.

I will never forget the warm welcome that I received when I joined the LSV ten years ago. I am very much indebted to Myrto Arapinis, Nathalie Bertrand, Patricia Bouyer-Decitre, Stéphanie Delaune, Marie Duflot-Kremer, Catherine Forestier, Steve Kremer, Nicolas Markey, Arnaud Sangnier, and Tali Sznajder, who have had a lasting impression on me, and certainly on the LSV as a whole.

My heartfelt thanks go to my beloved sister, I am so happy she is around. I am deeply grateful to my parents-in-law for their unfailing support and immense generosity. A special thought goes to my friends Darius Dlugosz, Carsten Kern, and Martin Leucker, who are far away and yet so close, and with whom all this started.

I dedicate this thesis to my mother, who has always been there for me, and to the memory of my father, who will always be right by my side, every step I take.

Le dernier mot est réservé pour toi, Virginie. Merci, infiniment, pour ton amour inconditionnel et pour Lucas, le plus beau cadeau qu'on ait pu me faire.

Benedikt Bollig
Cachan, May 2015

Contents

CHAPTER 1

Introduction

Computer systems are ubiquitous. They have pervaded almost every aspect of modern human life and are becoming more and more complex. A countless number of processes run in parallel and access data stored on servers that are physically distributed all over the globe. Indeed, mobile computing, ad-hoc networks, multicores are only a few of many omnipresent keywords that stand for a shift towards concurrent systems. To put it bluntly:

The world IS concurrent. It IS parallel.[1]

However, understanding concurrency is a challenging task. The interplay of several processes becomes unpredictable, and even carefully written and seemingly correct programs may eventually reveal serious bugs. Unfortunately, even subtle errors may entail serious damage to hardware, transport systems, business processes, electronic devices, etc.

This is where *formal methods* come into play. The term formal methods gathers tools and methods for a mathematically founded analysis of computer systems. They may be adopted at the early stages of system design, but also be applied to check an existing system against a requirements specification. This thesis aims at contributing to the mathematical foundations of concurrent systems, their formal modeling, specification, verification, and synthesis.

In the following, we first describe different models of concurrent systems to take into account applications in various domains. We will focus on *automata*, which are a versatile tool to model different phenomena of concurrency, yet, at the same time, offer a unifying view of all of them. We then describe high-level specification

[1] Joe Armstrong [Arm07].

formalisms (such as logics, regular expressions, and also automata). Though those are also equipped with a precise mathematical semantics, they allow for a more abstract, declarative view of a system. Two natural questions arise in this context, which are usually referred to as *realizability* and *model checking*:

Realizability: Given a requirements specification in terms of a formula (regular expression, automaton, respectively), is there an automaton that recognizes all, but only those, behaviors that satisfy the specification?

Model Checking: Given a system model in terms of an automaton, and a requirements specification in terms of a formula (regular expression, automaton, respectively), do all executions of the automaton satisfy the specification?

Both questions are actually closely related: various model-checking procedures rely on automata-theoretic techniques and the fact that a specification is "realizable" as an automaton. This thesis tries to contribute to both of these aspects of formal verification: realizability and model checking. In the following, we examine some characteristics of automata, specification formalisms, realizability, and model checking in more detail.

1.1 Automata Models and Their Behavior

Automata are a popular model of computer systems, making them accessible to formal methods and, in particular, synthesis and automated verification techniques such as model checking. While classical finite-state automata are suitable to model *sequential* boolean programs, models of *concurrent* systems, involving several interacting processes, extend finite-state machines in several respects. Roughly, we may classify a system (or a system model) according to the following characteristics:

Form of Communication. When there is only one process, we deal with a *sequential* system. When there are several processes, then interprocess communication may be accomplished, e.g., via *shared variables* or *message passing*. In this document, we consider boolean programs so that shared-variable communication usually gives rise to finite-state systems. Thus, classical methods are applicable for their analysis. On the other hand, message passing via a priori unbounded channels leads to undecidability of basic verification questions. However, putting some restrictions on the system (e.g., imposing a channel bound or restricting the system architecture) will allow us to infer positive results for both realizability and model checking.

In Figure 1.1, we illustrate the way of modeling the behavior that is generated by automata for the various types of communication. One single sequential process will produce sequences of events, as depicted on the left-hand side. One such execution can be seen as a word, i.e., the sequence of actions that we observe. More generally, it may be seen as a graph whose nodes are arranged chronologically, following a total order generated by the edge relation. Each node represents an *event*, and

with each event, we associate the *action* that it executes. For example, the actions a, b, c used in the figure may stand for writing or reading a variable, outputting the value of a variable, etc. The figure in the middle models an execution as it may be produced by a shared-memory system where processes share access to resources such as variables. This kind of graph is usually referred to as a *Mazurkiewicz trace*. In the example, we deal with two processes, sharing two events that both execute an action c. That is, c may stand for reading or writing a variable that is *shared* by both processes. Finally, in a message-passing environment, processes do not share resources in a synchronous manner, but use an asynchronous communication medium such as channels. The causal ordering implied by the channel policy may then be depicted as an additional edge between a send and the corresponding receive event. This is what happens in the graph on the right, which is called a *message sequence chart (MSC)*. In this example, we deal with actions a (for sending) and b (for receiving), while c may be some internal action.

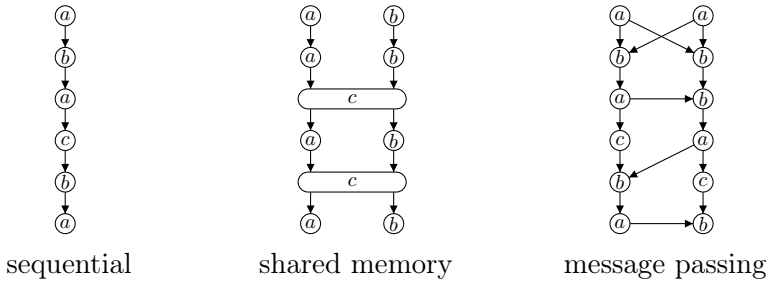


Figure 1.1: Forms of communication

As advocated in [AG14], modeling an execution of a concurrent system in terms of an acyclic (i.e., partially ordered) graph actually offers many advantages over the interleaving approach (which assumes an order on concurrent events). Since graphs visualize communication links between events, corresponding specification languages are usually more expressive and more intuitive.

System Architecture. The system architecture, connecting processes and arranging them in a certain way (e.g., as a pipeline or as a tree), may be static and fixed (i.e., known), or static but unknown, or it may change dynamically during a system execution. These three cases are schematically depicted in Figure 1.2. In the particular case where the topology is static but unknown, we deal with a *parameterized* setting. So, one will be interested in questions such as "Is the system correct no matter what the topology or number of processes is?" or "Can one transform a specification into a system that is correct for all system architectures?". More and more applications (let us just mention mobile computing) have dynamic architectures which allow processes to connect to/disconnect from processes at execution time. There has been a wide range of techniques for the verification of parameterized and dynamic systems. There are also close connections with the theory of words over infinite alphabets, where the infinitary part can be used to

model an unbounded supply of process identifiers.

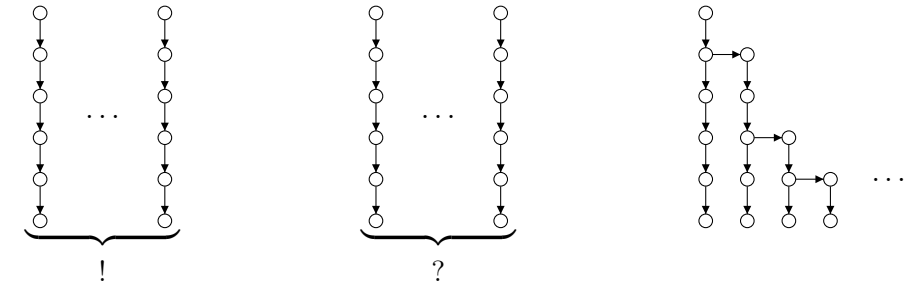


Figure 1.2: System architectures

Finite-State, Recursive, and Timed Processes. Processes themselves can be finite-state (i.e., be modeled as finite-state automata), recursive (i.e., be modeled as a pushdown automaton), or timed (i.e., be modeled as a timed automaton). Figure 1.3 illustrates three process behaviors, one for each case. The behavior of a recursive process is depicted as a *nested* word, where some (well-nested) edges connect the call and return position of a function call. A *timed* execution, on the other hand, provides additional time stamps telling us when an event has taken place.

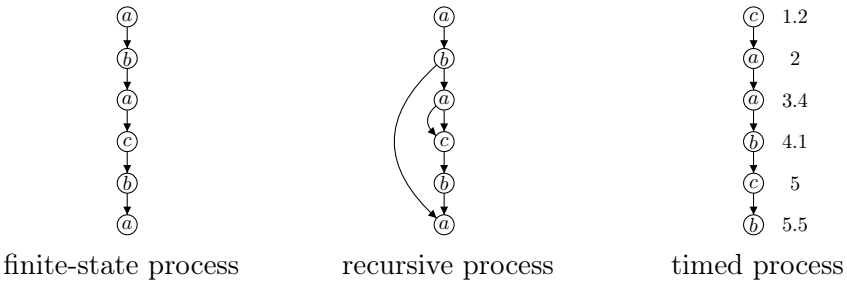


Figure 1.3: Type of a single process

In this thesis, we survey automata models for some combinations of the above characteristics. In particular, we cover automata for concurrent recursive shared-memory systems, parameterized and dynamic message-passing systems, and concurrent timed systems. In turn, these automata generalize the well-studied notions of Zielonka automata [Zie87] (towards recursion), communicating automata [BZ83] (towards dynamic and parameterized architectures), and timed automata [AD94] (towards concurrency).

1.2 Specification Formalisms

A specification describes the desired (or forbidden) behavior of a system. While automata serve as models of programs and, therefore, provide a low-level view of a system, requirements specifications are usually written in a high-level language, abstracting from implementation details and being closer to human way of thinking. Popular examples include regular expressions, monadic second-order (MSO) logic, as well as temporal logics. The expressive equivalence of finite automata and MSO logic, known as the Büchi-Elgot-Trakhtenbrot Theorem, constitutes one of the most beautiful cornerstones in the landscape of theoretical computer science [Büc60, Elg61, Tra62]. This can also be claimed for the coincidence of languages expressible in first-order (FO) logic and those definable in linear-time temporal logic (LTL), which is referred to as Kamp's Theorem [Kam68]. Beautiful on their own, those logical connections have led to practically usable verification tools [Hol03, HJJ+95].

The Büchi-Elgot-Trakhtenbrot Theorem and Kamp's Theorem have counterparts in concurrent settings [Tho90, TW97, DG02, DG06]. However, some classical results concerning the connection of specification formalisms with automata are not transferable. We will see that a generic MSO logic over graphs is, in general, too expressive to capture automata for static message-passing or recursive systems (while the second-order quantifier-alternation hierarchy collapses over words). Moreover, the transfer of temporal logics to richer structures and graphs than words is far from obvious. Indeed, there have been several proposals of temporal logics over Mazurkiewicz traces (see [GK07] for an overview), and a number of temporal logics have recently been considered for concurrent recursive processes [Ati10, ABKS12, LTN12]. Any such logic comes with tailored algorithms for its application in verification. However, it is fair to state that no canonical logic has been identified so far (unlike in the word case where LTL can certainly be considered the yardstick logic for linear-time model checking). Previously defined temporal logics for concurrent recursive systems do not even enjoy a theoretical justification as it is provided for the word case in terms of Kamp's Theorem.

For concurrent recursive programs, we will, therefore, consider *all* temporal logics, as defined in the book by Gabbay et al. [GHR94], in a unifying framework. Note that this approach has already led to a versatile framework for temporal logics over Mazurkiewicz traces [GK07, GK10]. We also provide automata-based specification formalisms (some of which can be considered or defined as regular expressions). The difference with *system* models is that automata specifications have a global state space offering, in a sense, a high-level view. As an example, automata with multiple pushdown stacks but one state space will serve as specifications for concurrent-system models where several pushdown automata, with separate state spaces, communicate via shared variables. In a dynamic setting, we introduce automata specifications with registers that can take values from infinite domains (for example, to model process identifiers). This will reveal new connections with the theory of formal languages over infinite alphabets.

1.3 Realizability

The realizability problem arises when we are given a specification that we would like to transform into an implementation in terms of an automata model. So, the first question to ask is if there is such an implementation at all. In the previous section, we already mentioned the Büchi-Elgot-Trakhtenbrot Theorem saying that, roughly speaking, all regular specifications are realizable. In Figure 1.4, three equivalent specifications are given (from top to bottom: LTL formula, regular expression, MSO formula) as well as an implementation in terms of a finite automaton.

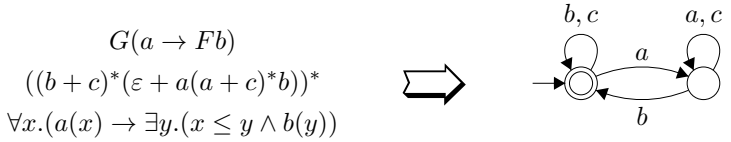


Figure 1.4: Realizability of sequential finite-state systems

While, in terms of regular languages, any specification has an implementation, the situation is more difficult when we move to the concurrent setting. Let us look at a well studied setting of concurrent shared-memory systems with finite-state processes. Suppose that we have two processes, 1 and 2, where 1 executes action a, and 2 executes action b. Consider Figure 1.5. Is it justified to say that the concurrent automaton on the right-hand side, with no communication between processes 1 and 2, is an implementation of the regular expression $\alpha = ab(ab)^*$? There are (at least) two arguments against it. Expression α implies that any execution performs as many a's as b's. But this is not realizable under the architecture that we consider (recall that 1 and 2 do not communicate). Moreover, b's have to be preceded by a's, which is not realizable for the same reason. Note that these problems are inherent to the specifications and will not vanish with a more clever implementation. So, what is the "right" formalism for the specification of those systems? Or, put differently, what are the valid specifications, when we assume that $ab(ab)^*$ is not realizable?

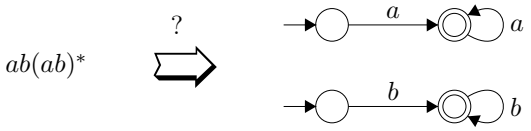


Figure 1.5: (Non-)realizability of concurrent finite-state systems

Consider Figure 1.6, and suppose that the system provides a third action, call it c, that is executed simultaneously by both processes. The concurrent automaton on the right-hand side might then be seen as an implementation of the regular expression on the left. First, the action c allows both processes to synchronize after each a-step (b-step, respectively), so that, in particular, there are as many a's as b's in any execution. Second, the order of executing a and b is not fixed anymore by the specification. Indeed, a classical result of Mazurkiewicz trace theory due

to Zielonka states that regular specifications that are *closed* under permutation of such *independent* actions can be effectively translated into a concurrent automaton [Zie87]. Note that the specification from Figure 1.5 is actually not closed in that sense. Similar results were obtained in the static message-passing setting [HMK+05, Kus03, GKM06]. In this thesis, we obtain a generalization of Zielonka's theorem for concurrent *recursive* programs, using Zielonka's result as a black box.

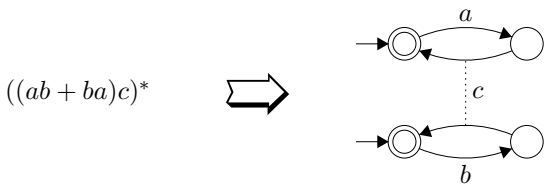


Figure 1.6: Realizability of concurrent finite-state systems

Recall that the realizability question amounts to asking whether a specification can be translated into an automaton. Thus, we would like to distinguish algorithmically between the specifications from Figure 1.5 (not realizable) and Figure 1.6 (realizable). The question is decidable for finite-state systems [Mus94, PWW98], but, unfortunately, it is undecidable in the recursive case, which follows from the undecidability of the nonemptiness problem [Ram00]. We will, therefore, identify sufficient decidable criteria that still guarantee realizability. The idea is to restrict the specifications. For example, we may impose a bound on the number of *context switches*, a notion introduced in [QR05], where each context allows only one process to call or return from a procedure. This amounts to *underapproximating* the complete system behavior. Note that the final implementation can exhibit executions that do not fit into this restriction anymore. We will come back to that point in the subsequent section about model checking.

Other possibilities are specification formalisms that do not distinguish between *linearizations* but are interpreted directly on a graph. Though this does not necessarily guarantee realizability, it already rules out design errors that can a priori be avoided. In this thesis, we consider MSO logic and temporal logic interpreted over behavior graphs (rather than their linearizations). Over some general classes of graphs (subsuming the behavior of message-passing systems and shared-memory systems), we provide results in the spirit of the Büchi-Elgot-Trakhtenbrot Theorem, showing an effective equivalence between automata and (the existential fragment of) MSO logic.

1.4 Model Checking

While realizability asks for an implementation of a given specification, *model checking* is the process of verifying the specification against a *given* system model. It has its root in the seminal works by Clarke & Emerson [CE81] and Queille & Sifakis [QS82]. Model checking is by now very well understood as far as finite-state sequential systems are concerned, and theoretical results have turned into tools

successfully employed in practice (e.g., [Hol03]). On the other hand, processes communicating via message passing through unbounded channels and shared-variable recursive processes have an undecidable control-state reachability problem [BZ83, Ram00]. However, similarly to the case of realizability, under- or overapproximating the behavior of a system will still allow us to check certain system properties. In other words, while exploring the set of possible behaviors, one restricts to a subset that is susceptible to containing the relevant critical executions.

For example, in a message-passing system, we may consider only those behaviors that do not exceed a predefined channel bound, and in a concurrent recursive setting, one may impose a bound on contexts (cf. previous section), each allowing only one process to evolve. The context restriction is all the more important (and practically justified) as many system errors can be detected within a few context-switches, as it was argued in [QR05]. Subsequently, other restrictions, relaxing the context-bound and being mutually incomparable, were defined and shown to have a decidable model-checking problem. They impose bounds on the number of *phases* [LMP07], *scopes* [LN11], or restrict to certain *ordered* excutions [BCCC96]. In [MP11], it was observed that all of them induce behaviors of bounded tree-width, giving a general explanation of decidability. Indeed, tree-automata techniques play an important role in model checking concurrent recursive programs.

Note that underapproximation techniques can be useful for verifying both *positive* (or, liveness) and *negative* (or, safety) specifications. A positive specification contains all the behaviors that a system should exhibit, while a negative specification represents faulty executions. If, now, the underapproximation admits all positive behaviors, then this is actually the case for the original system. Similarly, when the underapproximation intersects the negative specification, then this also applies to the system. Dually, an overapproximation can be used for the following reasoning. If it does not subsume all the positive behaviors, then the system does not exhibit all of them either. Moreover, if the overapproximation does not intersect with the negative behaviors, we can safely assume that the actual system will never exhibit any bad behavior. Overapproximation techniques are usually quite different from underapproximation methods. They have been applied successfully to message-passing systems (for example, assuming lossy channels) [AJ93], and concurrent recursive programs [BET03].

In this work, the focus is on *underapproximation* techniques for model checking parameterized message-passing systems and context-bounded recursive systems. Our results rely on very different approaches. In the first case, we exploit a realizability result for translating formulas into a system model (which is then compared to the actual system model via product construction). In the latter case, we exploit Hanf's theorem as well as aforementioned tree-automata techniques.

1.5 Contributions and Outline

The large majority of the results presented in this document have been obtained in collaboration with several co-authors, whom I gratefully acknowledge. All results

have been published in peer-reviewed conference proceedings and journals. Theorems that go back to publications that have been co-authored by myself after my PhD are highlighted in gray. Most proofs are omitted. However, we usually give the proof ideas and references to papers containing full proofs.

Next, we give a summary of the contributions along with the outline of this thesis. Recall that we deal with (a subset of)

$$\left\{\begin{array}{c}\text{realizability}\\ \text{model checking}\end{array}\right\} \text{ of } \left\{\begin{array}{c}\text{static}\\ \text{dynamic}\\ \text{parameterized}\end{array}\right\} \left\{\begin{array}{c}\text{finite-state}\\ \text{recursive}\\ \text{timed}\end{array}\right\} \left\{\begin{array}{c}\text{sequential}\\ \text{shared-memory}\\ \text{message-passing}\end{array}\right\}$$

systems. An attribution to chapters can be found in Table 1.1. Chapters are self-contained and can be read independently of each other, though we often point out analogies with other chapters.

Table 1.1: Systems covered by this document

Chapter 2	realizability/verification	static & fixed	recursive	shared-memory
Chapter 3	realizability/verification	parameterized	finite-state	message-passing
Chapter 4	realizability/verification	dynamic	finite-state	sequential
Chapter 5	realizability/verification	dynamic	finite-state	message-passing
Chapter 6	verification	static & fixed	timed	shared-memory

Chapter 2. Chapter 2 studies static recursive shared-memory systems. Both realizability and model checking are addressed. As specification formalisms, we consider (E)MSO logic and MSO-definable temporal logics, as well as (multiply) nested-word automata (NWAs), a kind of multi-stack automata with one single state space. As a model of an implementation, we introduce nested-trace automata (NTAs), which have distributed state spaces and generalize Zielonka automata to a recursive setting. Our main contributions in this chapter read as follows:

- We show that Zielonka's theorem carries over to concurrent recursive systems: any global specification (given by an NWA) that is closed under permutation of independent events can be translated into an equivalent NTA. As closure under independence rewriting is undecidable, we provide a variant of that theorem. It states that certain *bounded representations*, which can be considered as incomplete specifications, have a distributed counterpart in terms of an NTA. Here, "bounded" may refer to the number of contexts or phases. We show that the property of being a bounded representation is, in each case, decidable.

- We give logical characterizations of NTAs where behaviors are bounded. This extends work by Thomas [Tho90], who had established expressive equivalence

of MSO logic and (non-recursive) Zielonka automata. We complement our result by showing that MSO logic is strictly more expressive than NTAs over the domain of all nested words. Actually, this is obtained as a corollary from a characterization of NTAs in terms of EMSO logic and strictness of the monadic second-order quantifier-alternation hierarchy over multiply nested words.

- We determine the precise complexity of the emptiness problem for ordered NWAs. More precisely, the problem is shown to be 2-EXPTIME-complete.
- We show that model checking NTAs is decidable in elementary time for every MSO-definable temporal logic over nested words. Actually, the problem is solvable in $(n+2)$-EXPTIME where n is the maximal level of the MSO modalities in the monadic second-order quantifier-alternation hierarchy. We also establish an n-EXPSPACE lower bound for *some* MSO-definable temporal logic.

The chapter refers to the following references:

[ABH08] M. F. Atig, B. Bollig, and P. Habermehl. Emptiness of multi-pushdown automata is 2ETIME-complete. In *DLT'08*, volume 5257 of *LNCS*, pages 121–133. Springer, 2008.

[BCGZ14] B. Bollig, A. Cyriac, P. Gastin, and M. Zeitoun. Temporal logics for concurrent recursive programs: Satisfiability and model checking. *Journal of Applied Logic*, 2014. To appear.

[BGH09] B. Bollig, M.-L. Grindei, and P. Habermehl. Realizability of concurrent recursive programs. In *FoSSaCS'09*, volume 5504 of *LNCS*, pages 410–424. Springer, 2009.

[BKM13] B. Bollig, D. Kuske, and R. Mennicke. The complexity of model checking multi-stack systems. In *LICS'13*, pages 163–170. IEEE Computer Society Press, 2013.

[Bol08] B. Bollig. On the expressive power of 2-stack visibly pushdown automata. *Logical Methods in Computer Science*, 4(4:16), 2008.

Chapter 3. In this chapter, we provide a simple yet natural model of parameterized message-passing systems, which we call *parameterized communicating automata (PCAs)*. PCAs are a conservative extension of the classical communicating automata [BZ83]. They also have a static communication topology and accept message sequence charts (MSCs), which look like the graph depicted on the right in Figure 1.1. However, the precise communication topology is henceforth unknown so that it becomes a parameter of the system. Most work in the literature has focused on the *verification* of such systems, i.e., on the question "Does the given system satisfy some property independently of the topology or the number of processes?".

Here, we will initiate a study of the language theory of parameterized systems and, in particular, the realizability question. Combined with emptiness-checking algorithms, this eventually allows us to show decidability of model checking parameterized systems against MSO logic over MSCs, which is more expressive than the (temporal) logics that have been previously considered.

Our main contributions from this chapter can be summarized as follows:

- We provide several results in the spirit of the Büchi-Elgot-Trakhtenbrot Theorem, which roughly read as follows: Given a set of communication topologies $\mathfrak{T}$ (e.g., the class of pipelines, ranked trees, grids, or rings) and a high-level specification φ from some logic, there is a PCA that accepts precisely the models of φ when it is run on a topology from $\mathfrak{T}$.
- Inspired by the restrictions introduced for recursive systems, we study context-bounded verification of PCAs to overcome the inherent undecidability coming with that model. Using the previous logical characterization and an emptiness-checking algorithm, we obtain decidability of model checking of context-bounded PCAs.

The chapter refers to the following articles:

[BGK14] B. Bollig, P. Gastin, and A. Kumar. Parameterized communicating automata: Complementation and model checking. In *FSTTCS'14*, volume 29 of *Leibniz International Proceedings in Informatics*, pages 625–637. Leibniz-Zentrum für Informatik, 2014.

[BGS14] B. Bollig, P. Gastin, and J. Schubert. Parameterized verification of communicating automata under context bounds. In *RP'14*, volume 8762 of *LNCS*, pages 45–57. Springer, 2014.

[BK12] B. Bollig and D. Kuske. An optimal construction of Hanf sentences. *Journal of Applied Logic*, 10(2):179–186, 2012.

[Bol14] B. Bollig. Logic for communicating automata with parameterized topology. In *CSL-LICS'14*. ACM Press, 2014.

Chapter 4. This chapter considers sequential systems rather than concurrent ones. However, it lays some foundations that will be exploited later in a dynamic concurrent setting. In particular, it prepares some mechanism to deal with an unbounded number of processes and an infinite supply of process identifiers. Accordingly, finite automata are extended to run over infinite alphabets, whose elements are often referred to as *data values*. To do so, an automaton is equipped with registers as storing capabilities for data values. Many such models have been defined in the literature, each having its advantages and drawbacks. Here, we introduce two further models. One generalizes previously introduced models, one is a restriction, which we call session automata. More precisely, this chapter deals with the following issues:

- We consider the model-checking problem for session automata. Unlike previous approaches, which can only handle weak fragments of *data* MSO logic (featuring an equality predicate for data values), we allow the full MSO logic as a specification language. On the other hand, we restrict the automata model to gain decidability of model checking. Moreover, we show that session automata, unlike many other classes of register automata, form a robust language class: they are closed under boolean operations (complementation wrt. the class of certain *bounded* words) and have a decidable inclusion problem.

- Moreover, we introduce class register automata, a "realistic" model of register automata. Though it has an undecidable emptiness problem, it allows one to reflect reading capabilities of automata (such as receive a message or return from a procedure) that will later be transferred to a concurrent setting. We show that the expressiveness of class register automata is between EMSO logic and MSO logic, solving the realizability problem for EMSO logic, and demonstrating that, somehow, class register automata recognize only "regular" languages over infinite alphabets.

The chapter covers the references indicated below. Note that, in the present manuscript, we present a restricted framework compared to the original publications. The latter deal with stacks, which we omit here for simplicity, and multi-dimensional data words, while we restrict to one data value per event.

[BCGNK12] B. Bollig, A. Cyriac, P. Gastin, and K. Narayan Kumar. Model checking languages of data words. In *FoSSaCS'12*, volume 7213 of *LNCS*, pages 391–405. Springer, 2012.

[BHLM14] B. Bollig, P. Habermehl, M. Leucker, and B. Monmege. A robust class of data languages and an application to learning. *Logical Methods in Computer Science*, 2014.

[Bol11] B. Bollig. An automaton over data words that captures EMSO logic. In *CONCUR'11*, volume 6901 of *LNCS*, pages 171–186. Springer, 2011.

Chapter 5. In this chapter, we study dynamic message-passing systems. Like Chapter 3, this extends classical communicating automata, but in an orthogonal way. Moreover, we consider different questions since the results from Chapter 3 do not carry over to the dynamic setting. We focus on realizability, but we also touch the model-checking problem. Our contributions are as follows:

- We introduce a dynamic version of communicating automata. Like the static variant, *dynamic communicating automata* (DCAs) feature concurrent processes that communicate via message exchange. However, new processes may be generated at runtime and the number of processes that participate in an

execution is a priori unbounded. Processes are identified by unique process identifiers (pids), which they can store in registers and pass to other processes via messages. In a natural way, DCAs recognize dynamic MSCs, which are MSCs extended by dynamic process creation.

- We study the realizability questions for *high-level MSCs* with registers, which are inspired by the model of session automata presented in Chapter 4. Now, dynamic process creation raises new problems and questions. In order for a language to be realizable, one has to guarantee that processes that communicate with one another know each other. In particular, there must be a way of communicating pids in such a way that a sender holds the pid of a receiver in one of its registers at the time of communication. To our knowledge, this question has not been considered before. Unfortunately, realizability (actually, we consider a variant called *implementability*) of high-level MSCs in terms of DCAs is undecidable. Therefore, we study a decidable sufficient criterion, which we call *executability*. We then define a subclass of high-level MSCs for which executability and implementability are equivalent. The proof of that result yields an effective construction of a DCA out of a high-level MSC, provided that the latter is executable.

The chapter covers the following references:

[BCH+13] B. Bollig, A. Cyriac, L. Hélouët, A. Kara, and T. Schwentick. Dynamic communicating automata and branching high-level MSCs. In *LATA'13*, volume 7810 of *LNCS*, pages 177–189. Springer, 2013.

[BH10] B. Bollig and L. Hélouët. Realizability of dynamic MSC languages. In *CSR'10*, volume 6072 of *LNCS*, pages 48–59. Springer, 2010.

Chapter 6. This chapter studies timed systems with a static communication topology. The model we introduce is kind of unusual, since it does not assume that clocks of the different processes evolve at the same speed. Indeed, it is often more appropriate to relax the assumption of perfectly synchronous clocks. So, we extend the classical notion of a timed automaton with an equivalence relation on the set of clocks. Roughly speaking, we assume that there is no way to predict the relative clock speed of clocks that are not equivalent (or, do not belong to the same process). Though relative clock speeds are not predictable, or, not robust, we are looking for a way to verify such *timed automata with independently evolving clocks*. The idea is to consider an under- and an overapproximation of the actual system behavior, which includes behaviors that are executable under *every* and, respectively, *some* evolution of local times. Note that we had already considered underapproximations in Chapters 2 and 3. However, the reason there to look at behavioral restrictions was undecidability of the general problem, rather than unpredictability. The chapter refers to the following article:

[ABG+14] S. Akshay, B. Bollig, P. Gastin, M. Mukund, and K. Narayan Kumar. Distributed timed automata with independently evolving clocks. *Fundamenta Informaticae*, 130(4):377–407, 2014.

Conclusion 7. In the last chapter, we conclude and give general directions and challenges for future work.

1.6 Further Contributions

Some of the author's contributions are perfectly in the scope of this document, but have not been considered in order to keep the presentation simple and homogenous. These contributions read as follows:

- We establish a logical characterization of Muller communicating automata, producing *infinite* executions. The result makes use of an extension of Hanf's Theorem to the infinitary quantifier [BK08].
- We also extend the logical characterization to a timed message-passing setting. Here, every process is an event-clock automaton (a subclass of timed automata). In addition, timing constraints can be imposed on the delay between sending and receiving a message. While timed automata accept *timed words*, event-clock message-passing automata accept *timed MSCs*, where each event comes with an additional time stamp. As a specification language, MSO logic is naturally extended by timing constraints [ABG13].
- We consider PDL logic for both realizability and model checking of message-passing systems. Concerning realizability, we show that every formula can be translated into a communicating automaton. Moreover, model checking communicating automata and high-level MSCs against PDL specifications are shown to be PSPACE-complete [BKM10].
- As an alternative approach to realizability, we sketch a learning framework for communicating automata. Based on Angluin's L*-algorithm, this approach allows one to infer message-passing systems simply by classifying MSCs as positive and negative [BK08].

The references refer to the following papers:

[ABG13] S. Akshay, B. Bollig, and P. Gastin. Event-clock message passing automata: A logical characterization and an emptiness checking algorithm. *Formal Methods in System Design*, 42(3):262–300, 2013.

[BK08] B. Bollig and D. Kuske. Muller message-passing automata and logics. *Information and Computation*, 206(9-10):1084–1094, 2008.

[BKKL10] B. Bollig, J.-P. Katoen, C. Kern, and M. Leucker. Learning communicating automata from MSCs. *IEEE Transactions on Software Engineering*, 36(3):390–408, 2010.

[BKM10] B. Bollig, D. Kuske, and I. Meinecke. Propositional dynamic logic for message-passing systems. *Logical Methods in Computer Science*, 6(3:16), 2010.

1.7 Notation

The set of natural numbers is denoted by $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ and the set of positive natural numbers by $\mathbb{N}_{>0} = \{1, 2, 3, \ldots\}$. For $n \in \mathbb{N}$, we let $[n] \stackrel{\text{def}}{=} \{1, \ldots, n\}$. An *alphabet* is any nonempty set. In this thesis, we will deal with both finite and infinite alphabets. Given an alphabet Σ, we denote by Σ^* the set of finite words (or, strings) over Σ. The empty word is denoted ε. Moreover, $\Sigma^+ \stackrel{\text{def}}{=} \Sigma^* \setminus \{\varepsilon\}$.

For two sets A and B, we let $[A \rightharpoonup B]$ be the set of partial mappings from A to B. Let $\mathsf{f} : A \to B$ be a total or partial function, $a \in A$, and $b \in B$. By $\mathsf{f}[a \mapsto b]$, we denote the function that maps a to b and coincides with f on all other arguments. We sometimes write f_a to denote $\mathsf{f}(a)$. We let $\mathrm{dom}(\mathsf{f})$ stand for the *domain* of f, i.e., the set of elements $a \in A$ such that $\mathsf{f}(a)$ is defined. We may identify f with the set $\{(a, \mathsf{f}(a)) \mid a \in \mathrm{dom}(\mathsf{f})\}$.

CHAPTER 2

Static Recursive Shared-Memory Systems

In this chapter, we consider *recursive* systems with a static and fixed (i.e., known) communication topology. A system is recursive if a process may suspend its activity and invoke (or, call) a subtask, before proceeding (or, returning). A stack is used to store the current configuration of the process, and to recover it, once the subtask has been completed. The programs we consider are *boolean*, i.e., possible variables range over a finite domain. In particular, the set of states and the alphabet of stack symbols are finite. Boolean programs, in turn, may arise as abstractions from programs with infinite-domain variables.

As discussed in the introduction, recursive systems may be defined in several equivalent ways. The classical definition is in terms of a pushdown automaton with explicit stacks. A run is then a sequence of configurations, which keep track of the current stack contents. Alternatively, automata may run on words enriched with nesting relations, one for each process. A nesting relation connects a call position with the corresponding return position. In that case, a transition of the automaton that performs a return will depend on the state reached after executing the corresponding call position. It is then sufficient to define a run as a simple sequence of states (without explicitly mentioning the stack contents). This is the approach adopted in this thesis.

2.1 (Multiply) Nested Words and Their Automata

Nested words [AM09a] extend a word (or, total order) by one or more nesting relations. They reflect executions of concurrent recursive programs, with the understanding that the nesting relation connects a function call with its corresponding return position. We assume that the event labeling tells us whether we deal with

a call or a return so that the nesting relation is actually uniquely given.

A *distributed (visibly pushdown) alphabet* is a tuple $\tilde{\Sigma} = (\Sigma, P, type, \delta)$ where Σ is a finite alphabet, P is a nonempty finite set of *processes*, the mapping $type : \Sigma \to \{\mathsf{call}, \mathsf{ret}, \mathsf{int}\}$ indicates the *type* of an action (call, return, internal), and $\delta : \Sigma \to 2^P$. The intuition is that $\delta(a)$ is the set of processes that are involved in the execution of $a \in \Sigma$. We require that $\delta(a) \neq \emptyset$ for all $a \in \Sigma$, and $|\delta(a)| = 1$ for all $a \in \Sigma_{\mathsf{call}} \cup \Sigma_{\mathsf{ret}}$. The latter condition implies that synchronization is achieved via internal actions only. Note that, however, the concrete distribution of internal actions only matters in Section 2.5, when we define *nested Mazurkiewicz traces*.

Definition 2.1 (nested word). *A* (multiply) nested word *over the distributed alphabet $\tilde{\Sigma}$ is a triple $(E, \lhd, \lambda)$ where*

- *E is a nonempty finite set of* events*,*
- *$\lambda : E \to \Sigma$ is the* event-labeling function*, and*
- *$\lhd = \lhd_{+1} \cup \lhd_{\mathsf{cr}} \subseteq E \times E$ is an acyclic edge relation (denoting direct successors or matching a call with a return)*

such that the properties 1.–4. below are satisfied. Hereby, for $\tau \in \{\mathsf{call}, \mathsf{ret}, \mathsf{int}\}$, we let $E_\tau \stackrel{\text{def}}{=} \{e \in E \mid \tau = type(\lambda(e))\}$. Moreover, given $p \in P$, set $E_p \stackrel{\text{def}}{=} \{e \in E \mid p \in \delta(\lambda(e))\}$.

1. *$\leqslant \stackrel{\text{def}}{=} \lhd^*$ is a total order on E (with strict part $<$), and $\lhd_{+1}$ is the direct successor relation of $\leqslant$,*
2. *$\lhd_{\mathsf{cr}}$ induces a bijection between E_{call} and E_{ret},*
3. *for all $(e, f) \in \lhd_{\mathsf{cr}}$, there is $p \in P$ such that $e \in E_p$ and $f \in E_p$, and*
4. *for all $(e_1, f_1), (e_2, f_2) \in \lhd_{\mathsf{cr}}$ and $p \in P$ such that $e_1 \in E_p$, $e_2 \in E_p$, and $e_1 < e_2 < f_1$, we have $f_2 < f_1$.* ♢

Condition 3. ensures that a call-return edge is restricted to one process, and condition 4. guarantees that calls and returns of one process follow a stack policy (in other words, $\lhd_{\mathsf{cr}} \cap (E_p \times E_p)$ is *well-nested* for every $p \in P$). Note that the sets $\lhd_{+1}$ and $\lhd_{\mathsf{cr}}$ are not necessarily disjoint.

The set of nested words over $\tilde{\Sigma}$ is denoted by $\mathsf{NW}(\tilde{\Sigma})$. When $\tilde{\Sigma}$ is clear from the context, we may just write NW. We do not distinguish isomorphic nested words.

The term *visibly* stresses the fact that an action uniquely determines the type and stack involved in its execution. Visibility actually reveals the nesting structure if only the sequence of labels from Σ is given. Let us formalize this and let $W = (E, \lhd, \lambda)$ be a nested word. Assume, without loss of generality, that $E = \{e_1, \ldots, e_n\}$ with $e_1 \lhd_{+1} e_2 \lhd_{+1} \ldots \lhd_{+1} e_n$. The *string* of W is defined as $string(W) \stackrel{\text{def}}{=} \lambda(e_1) \ldots \lambda(e_n) \in \Sigma^+$. Now, given any word $w \in \Sigma^+$, there is at most one (up to isomorphism) nested word W over $\tilde{\Sigma}$ such that $string(W) = w$.

If it exists, then we denote it by $\mathit{nested}(w)$ and we call w *well-nested*, since it is guaranteed that calls and returns are well bracketed, for any process/stack. Note that, for any nested word W, we have $\mathit{nested}(\mathit{string}(W)) = W$.

Example 2.2. Consider the distributed alphabet $\tilde{\Sigma} = (\Sigma, P, \mathit{type}, \delta)$ given by $\Sigma = \{a_1, b_1, a_2, b_2, c\}$, $P = \{1, 2\}$, $\mathit{type}(a_1) = \mathit{type}(a_2) = \mathsf{call}$, $\mathit{type}(b_1) = \mathit{type}(b_2) = \mathsf{ret}$, and $\mathit{type}(c) = \mathsf{int}$, as well as $\delta(a_1) = \delta(b_1) = \{1\}$, $\delta(a_2) = \delta(b_2) = \{2\}$, and $\delta(c) = \{1, 2\}$. Figure 2.1 shows the nested word $W_{2.1}$ over $\tilde{\Sigma}$. Straight edges represent the relation $\lhd_{+1}$, whereas curved edges represent $\lhd_{\mathsf{cr}}$. We have $\mathit{string}(W_{2.1}) = a_1 c a_2 a_1 c a_2 c b_1 c b_1 b_2 b_2$. ♢

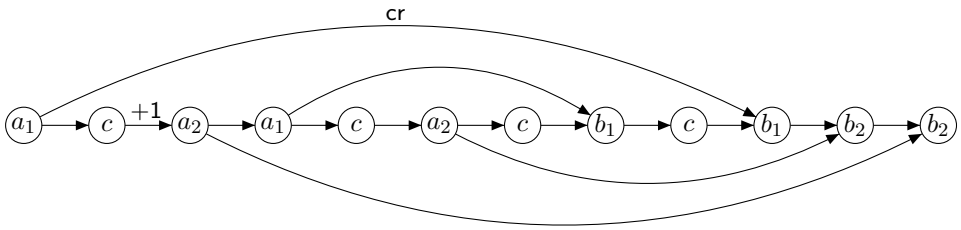


Figure 2.1: The multiply nested word $W_{2.1}$

Now, we consider automata that run on multiply nested words and are suitable as a model of recursive programs. As explained before, these automata run directly over nested words so that the stacks are only implicit.

Definition 2.3 (NWA). *A* nested-word automaton (NWA) *over* $\tilde{\Sigma}$ *is a tuple* $\mathcal{A} = (S, \Gamma, \Delta, \iota, F)$ *where*

- S *is the nonempty finite set of* states*,*
- Γ *is the nonempty finite set of* stack symbols*,*
- $\iota \in S$ *is the* initial state*,*
- $F \subseteq S$ *is the set of* final states*, and*
- $\Delta = \Delta_{\mathsf{call}} \uplus \Delta_{\mathsf{ret}} \uplus \Delta_{\mathsf{int}}$ *is the* transition relation*, partitioned into*
  - $\Delta_{\mathsf{call}} \subseteq S \times \Sigma_{\mathsf{call}} \times \Gamma \times S$*,*
  - $\Delta_{\mathsf{ret}} \subseteq S \times \Sigma_{\mathsf{ret}} \times \Gamma \times S$*, and*
  - $\Delta_{\mathsf{int}} \subseteq S \times \Sigma_{\mathsf{int}} \times S$*.* ♢

Reading an event e of a nested word, transition $(s, a, s') \in \Delta_{\mathsf{int}}$ lets the automaton move on from the current state s to the target state s' if e is labeled with letter a. The same applies to a transition $(s, a, A, s') \in \Delta_{\mathsf{call}} \cup \Delta_{\mathsf{ret}}$. However, in addition, this transition allows $\mathcal{A}$ to associate with a call event stack symbol A. The symbol can then be retrieved at the corresponding return position. In a sense, this is equivalent to reading a stack symbol previously pushed. More precisely, we require

that, whenever $e \lhd_{\mathsf{cr}} f$, the stack symbol chosen at e is the same as the stack symbol employed by the transition that is taken at position f.

Let $W = (E, \lhd, \lambda)$ be a nested word. Let us, again, assume that $E = \{e_1, \ldots, e_n\}$ with $e_1 \lhd_{+1} e_2 \lhd_{+1} \ldots \lhd_{+1} e_n$. A *run* of $\mathcal{A}$ on W is a pair (ρ, σ) of mappings $\rho : E \to S$ and $\sigma : E_{\mathsf{call}} \cup E_{\mathsf{ret}} \to \Gamma$ such that,

- for all $(e, f) \in \lhd_{\mathsf{cr}}$, we have $\sigma(e) = \sigma(f)$, and
- for every $i \in \{1, \ldots, n\}$ (letting $\rho(e_0) = \iota$),

$$\begin{cases} \big(\rho(e_{i-1}), \lambda(e_i), \rho(e_i)\big) \in \Delta_{\mathsf{int}} & \text{if } e_i \in E_{\mathsf{int}} \\ \big(\rho(e_{i-1}), \lambda(e_i), \sigma(e_i), \rho(e_i)\big) \in \Delta_{\mathsf{call}} \cup \Delta_{\mathsf{ret}} & \text{if } e_i \in E_{\mathsf{call}} \cup E_{\mathsf{ret}}\,. \end{cases}$$

The run (ρ, σ) is accepting if $\rho(e_n) \in F$. The set of multiply nested words for which there is an accepting run is denoted by $L(\mathcal{A})$.

We say that $\mathcal{A}$ is *deterministic* if

- for all $(s, a) \in S \times \Sigma_{\mathsf{call}}$, there is at most one pair $(A, s') \in \Gamma \times S$ such that $(s, a, A, s') \in \Delta_{\mathsf{call}}$,
- for all $(s, a, A) \in S \times \Sigma_{\mathsf{ret}} \times \Gamma$, there is at most one $s' \in S$ such that $(s, a, A, s') \in \Delta_{\mathsf{ret}}$, and
- for all $(s, a) \in S \times \Sigma_{\mathsf{int}}$, there is at most one $s' \in S$ such that $(s, a, s') \in \Delta_{\mathsf{int}}$.

Example 2.4. An NWA $\mathcal{A}_{2.2} = (\{s_0, \ldots, s_5\}, \{\$\}, \Delta, s_0, \{s_5\})$ over the distributed alphabet $\tilde{\Sigma}$ from Example 2.2 is given in Figure 2.2. Transitions are self-explanatory. For example, Δ_{int} contains (s_3, c, s_4), and Δ_{ret} contains $(s_4, b_2, \$, s_5)$, which is indicated by the edge from s_4 to s_5 with label $b_{2|\$}$. Note that $\mathcal{A}_{2.2}$ accepts those nested words that (i) contain a subsequence a_1a_2 or a_2a_1, (ii) have a call and a subsequent return phase, separated by some action c, and (iii) schedule returns of process 1 before those of process 2. The nested word over $\tilde{\Sigma}$ from Figure 2.1 is accepted by $\mathcal{A}_{2.2}$. Note that $\mathcal{A}_{2.2}$ is *not* deterministic. ♢

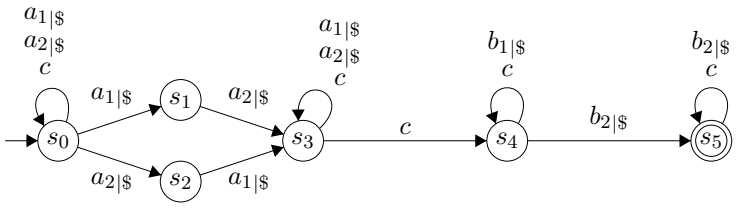


Figure 2.2: Nested-word automaton $\mathcal{A}_{2.2}$

Unfortunately, in the presence of at least two processes/stacks, most basic verification problems for NWAs are undecidable, such as nonemptiness:

Theorem 2.5 ([Ram00]). *The following problem is undecidable:*

INSTANCE: $\tilde{\Sigma}$; NWA $\mathcal{A}$ over $\tilde{\Sigma}$
QUESTION: $L(\mathcal{A}) \neq \emptyset$?

2.2 Contexts, Phases, Scopes, and Ordered Stacks

In the seminal paper [QR05], Qadeer and Rehof exploited the fact that errors of recursive programs typically occur already within a few *contexts* where a context refers to an execution involving only one process. Imposing a bound on the number of contexts indeed renders many verification problems decidable. In other words, instead of looking at all possible executions, we consider an underapproximation of the actual system behavior. This view is particularly suitable when checking *positive specifications*: if a system $\mathcal{A}$ should exhibit all behaviors given by a set *Good*, then it is sufficient to show that $Good \subseteq L(\mathcal{U})$ is true for a restricted version $\mathcal{U}$ of $\mathcal{A}$ such that $L(\mathcal{U}) \subseteq L(\mathcal{A})$. Thus, we are interested in restrictions $\mathcal{U}$ of $\mathcal{A}$ such that the problem $Good \subseteq L(\mathcal{U})$ is decidable and that we may adjust incrementally so as to converge to $L(\mathcal{A})$. On the other hand, given a *negative specification* (or, safety property) *Bad*, we can also draw conclusions from $L(\mathcal{U}) \cap Bad \neq \emptyset$, while showing complete absence of bad behaviors would require an overapproximation of the system behavior.

Next, we recall the notion of a context as well as other restrictions that have been defined in the literature. Let $W = (E, \lhd, \lambda) \in \mathsf{NW}(\tilde{\Sigma})$ be a nested word. An interval of W is a set $I \subseteq E$ such that $I = \emptyset$ or $I = [e, f] \stackrel{\text{def}}{=} \{g \in E \mid e \leq g \leq f\}$ for some $e, f \in E$. In a *context*, only one designated process is allowed to call or return, while a *phase* only restricts return operations:

- A *context* of W is an interval I such that $I \cap (E_{\mathsf{call}} \cup E_{\mathsf{ret}}) \subseteq E_p$ for some $p \in P$.
- A *phase* of W is an interval I such that $I \cap E_{\mathsf{ret}} \subseteq E_p$ for some $p \in P$.

Note that every context is a phase, while the converse does not hold in general.

Definition 2.6 (*k*-context word [QR05] and *k*-phase word [LMP07]). *Let $W = (E, \lhd, \lambda)$ be a nested word and $k \geq 1$. We say that W is a* k-context word (k-phase word) *if there are contexts (phases, respectively) $I_1, \ldots, I_k$ of W such that $E = I_1 \cup \ldots \cup I_k$.* ♢

While every k-context word is a k-phase word, but not the other way around, two orthogonal restrictions have been defined in terms of bounded scopes and ordered nested words, which we consider next. A scope-bounded word restricts the number of contexts between a push and the corresponding pop operation.

Definition 2.7 (*k*-scope word [LN11]). *Let $W = (E, \lhd, \lambda)$ be a nested word and $k \geq 1$. We call W a* k-scope word *if, for all $(e, f) \in \lhd_{\mathsf{cr}}$, there exist contexts $I_1, \ldots, I_k$ of W such that we have $[e, f] = I_1 \cup \ldots \cup I_k$.* ♢

Finally, in an ordered word, we refer to a total ordering $\preceq$ on P (with irreflexive part $\prec$). Then, a pop operation can only be performed by the process that is minimal among all processes with a nonempty stack.

Definition 2.8 (ordered word [BCCC96]). *Let $W = (E, \lhd, \lambda) \in \mathsf{NW}(\tilde{\Sigma})$. We call W an* ordered word *(wrt. $\preceq$) if, for all $p, p' \in P$, $e, f \in E_p$, and $f' \in E_{\mathsf{ret}} \cap E_{p'}$ such that $e \lhd_{\mathsf{cr}} f$ and $e < f' < f$, we have $p' \preceq p$.* ♢

Example 2.9 (continues Example 2.2). Figure 2.3 illustrates the concepts introduced above by means of the nested word $W_{2.1}$ from Figure 2.1. The shadowed areas are dedicated to process 2. Thus, $W_{2.1}$ is a 6-context word, a 5-scope word, and a 2-phase word. All these bounds are optimal. Moreover, $W_{2.1}$ is an ordered word under the assumption $1 \prec 2$ (but not for $2 \prec 1$). ♢

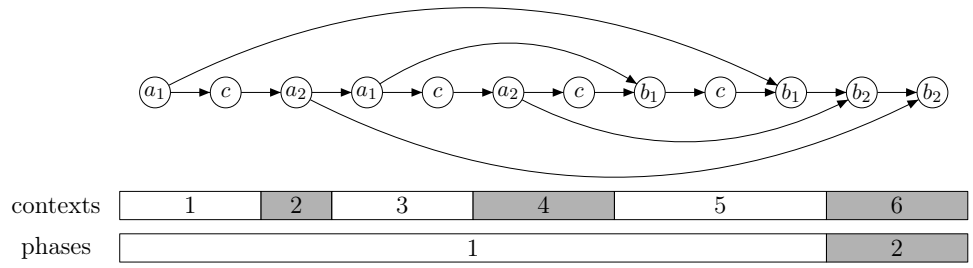


Figure 2.3: Illustration of context, phase, and scope

Let $\mathfrak{R} \stackrel{\text{def}}{=} \{k\text{-}\mathsf{cnt}, k\text{-}\mathsf{ph}, k\text{-}\mathsf{scp} \mid k \geq 1\} \cup \{\mathsf{ord}\}$ be the set of possible "restrictions". For $\theta \in \mathfrak{R}$, the set of θ-words is denoted by $\mathsf{NW}_\theta(\tilde{\Sigma})$. In particular, by $\mathsf{NW}_{\mathsf{ord}}(\tilde{\Sigma})$, we denote the set of ordered words, silently assuming an order on P. Moreover, for an NWA $\mathcal{A}$, we let $L_\theta(\mathcal{A}) \stackrel{\text{def}}{=} L(\mathcal{A}) \cap \mathsf{NW}_\theta(\tilde{\Sigma})$. When convenient, we omit the reference to $\tilde{\Sigma}$ and write, for example, $\mathsf{NW}_{k\text{-}\mathsf{ph}}$.

Example 2.10. Consider the NWA $\mathcal{A} = \mathcal{A}_{2.2}$ from Figure 2.2. We have that $L(\mathcal{A}) = L_{2\text{-}\mathsf{ph}}(\mathcal{A}) = L_{\mathsf{ord}}(\mathcal{A})$. On the other hand, $L_{k\text{-}\mathsf{cnt}}(\mathcal{A})$ and $L_{k\text{-}\mathsf{scp}}(\mathcal{A})$ are strictly included in $L(\mathcal{A})$, for all $k \geq 1$. ♢

Let us define the respective nonemptiness problems:

Problem 2.11. ORD-NONEMPTINESS:

INSTANCE:	$\tilde{\Sigma}$; NWA $\mathcal{A}$ (over $\tilde{\Sigma}$)
QUESTION:	$L_{\mathsf{ord}}(\mathcal{A}) \neq \emptyset$?

Problem 2.12. CONTEXT-NONEMPTINESS:

INSTANCE:	$\tilde{\Sigma}$; NWA $\mathcal{A}$; $k \geq 1$
QUESTION:	$L_{k\text{-}\mathsf{cnt}}(\mathcal{A}) \neq \emptyset$?

Problem 2.13. PHASE-NONEMPTINESS:

INSTANCE:	$\tilde{\Sigma}$; NWA $\mathcal{A}$; $k \geq 1$
QUESTION:	$L_{k\text{-}\mathsf{ph}}(\mathcal{A}) \neq \emptyset$?

Problem 2.14. SCOPE-NONEMPTINESS:

INSTANCE:	$\tilde{\Sigma}$; NWA $\mathcal{A}$; $k \geq 1$
QUESTION:	$L_{k\text{-}\mathsf{scp}}(\mathcal{A}) \neq \emptyset$?

Indeed, all these problems are decidable with varying complexities. We assume that the parameter k is given in unary. For a comparison between the effects of a unary and a binary encoding, see [BD13].

Theorem 2.15 ([QR05, LTKR08]). CONTEXT-NONEMPTINESS *is* NP*-complete.*

Theorem 2.16 ([LMP07, LMP08b]). PHASE-NONEMPTINESS *is* 2-EXPTIME-*complete.*

Theorem 2.17 ([LN11]). SCOPE-NONEMPTINESS *is* PSPACE-*complete.*

Theorem 2.18 ([ABH08]). ORD-NONEMPTINESS *is* 2-EXPTIME-*complete.*

In [MP11], a uniform argument for decidability of the above problems was given by Madhusudan and Parlato: for $\theta \in \{k\text{-}\mathsf{cnt}\,,\, k\text{-}\mathsf{ph} \mid k \geq 1\} \cup \{\mathsf{ord}\}$, the class of θ-words has bounded tree width. By Courcelle's theorem, this implies that nonemptiness of NWAs and even model-checking of NWAs against MSO properties is decidable. It was then shown that scope-bounded words have bounded tree width, too [LP12].

An alternative unifying approach is given by Cyriac, Gastin, and Narayan Kumar [CGNK12] in terms of the notion of *split-width*. Since split-width is tailored to nested words, it is simpler than tree width and allows for a more intuitive understanding of a class of bounded multi-threaded programs. Note that split-width also led to generalizations of the above-mentioned classes and other existing work on recursive message-passing systems [AGNK14b, Cyr14].

2.3 Monadic Second-Order Logic

We give a short account of monadic second-order (MSO) logic over nested words. The logic is built over countably infinite supplies $\{x, y, x_1, x_2, \ldots\}$ of *first-order* and $\{X, Y, X_1, X_2, \ldots\}$ of *second-order variables*. First-order variables are interpreted as events, second-order variables as sets of events. As usual, the predicates available in MSO logic depend on the signature of a structure, which is given in terms of the distributed alphabet $\tilde{\Sigma} = (\Sigma, P, \mathit{type}, \delta)$.

Definition 2.19. *The formulas from* $\mathrm{nwMSO}(\tilde{\Sigma})$ *are built according to the following grammar:*

$$\begin{aligned} \varphi \quad ::= \quad & a(x) \;\mid\; x \lhd_{+1} y \;\mid\; x \lhd_{\mathsf{cr}} y \;\mid\; x = y \;\mid\; x \in X \;\mid \\ & \neg\varphi \;\mid\; \varphi_1 \vee \varphi_2 \;\mid\; \exists x.\varphi \;\mid\; \exists X.\varphi \end{aligned}$$

where $a \in \Sigma$*,* x, y *are first-order variables, and* X *is a second-order variable.* ♢

Throughout this document, we use the usual abbreviations such as $\varphi_1 \wedge \varphi_2$ for $\neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\varphi \to \psi$ for $\neg\varphi \vee \psi$ and $\forall x.\varphi$ for $\neg\exists x.\neg\varphi$.

A formula $\varphi \in \mathrm{nwMSO}(\tilde{\Sigma})$ is interpreted over a nested word $W = (E, \lhd, \lambda) \in \mathsf{NW}(\tilde{\Sigma})$ wrt. a mapping $\mathcal{I}$. The purpose of the latter is to interpret free variables. It maps any first-order variable x to an event $\mathcal{I}(x) \in E$ and any second-order

variable X to a set of events $\mathcal{I}(X) \subseteq E$. We write $W \models_{\mathcal{I}} \varphi$ if formula φ is evaluated to true when the free variables of φ are interpreted according to $\mathcal{I}$. In particular, we have $W \models_{\mathcal{I}} a(x)$ if $\lambda(\mathcal{I}(x)) = a$, and $W \models_{\mathcal{I}} x \lhd_{\mathsf{cr}} y$ if $\mathcal{I}(x) \lhd_{\mathsf{cr}} \mathcal{I}(y)$. The remaining operators are as expected. When φ is a *sentence*, i.e., a formula without free variables, we may omit the indez $\mathcal{I}$ and simply write $W \models \varphi$. For an introduction to MSO logic on (classical) words and its semantics, we refer the reader to [Tho97].

The fragment $\mathrm{nwEMSO}(\tilde{\Sigma})$ contains the *existential MSO (EMSO) formulas*, which are of the form $\exists X_1 \ldots \exists X_n.\varphi$ where φ does not make use of second-order quantification.

2.4 The Language Theory of Nested-Word Automata

Let us examine closure properties of NWAs as well as their expressive power compared to MSO logic. One may call a language (or automata) class *robust* when it is effectively closed under all Boolean operations and has a decidable emptiness problem. A robust language class is amenable to verification tasks such as model checking, since the inclusion problem is decidable.

It is easy to see that NWAs are closed under union and intersection. This has to be seen in contrast to the fact that the class of context-free languages (CFLs) is not closed under intersection. The intuitive reason for this is that, in CFLs, actions are no longer visible. Indeed, augmenting words by the nesting structure makes sure that, while simulating two NWAs, the automaton for intersection performs push and pop operations at the same positions of the input word.

However, NWAs do not form a robust automata class, as their emptiness problem is undecidable (Theorem 2.5). This even applies when we restrict to two stacks. However, in the presence of at most two stacks, we can establish a logical characterization:

Theorem 2.20 ([Bol08]). *Suppose $|P| = 2$ and let $L \subseteq \mathsf{NW}(\tilde{\Sigma})$. The following statements are effectively equivalent:*

1. *There is an NWA $\mathcal{A}$ over $\tilde{\Sigma}$ such that $L(\mathcal{A}) = L$.*
2. *There is a sentence $\varphi \in \mathrm{nwEMSO}(\tilde{\Sigma})$ such that $L(\varphi) = L$.*

In [Bol08], it is also shown that the monadic quantifier-alternation hierarchy over nested words is infinite, by an encoding of *grids* into nested words. We obtain, as a corollary, that NWAs are not complementable:

Theorem 2.21 ([Bol08]). *Suppose $|P| = 2$. There is an NWA $\mathcal{A}$ over $\tilde{\Sigma}$ such that, for all NWAs $\mathcal{A}'$ over $\tilde{\Sigma}$, we have $L(\mathcal{A}') \neq \mathsf{NW}(\tilde{\Sigma}) \setminus L(\mathcal{A})$.*

We have seen that restricting the domain of nested words appropriately, renders the emptiness problem for NWAs decidable. It has been shown that above restrictions actually give rise to robust language classes. The next theorem states that, under any of the restrictions introduced above, NWAs are complementable:

Theorem 2.22 (complementability [LMP07, LNP14a, LNP14b]). *Let $\theta \in \mathfrak{R}$. Then, for all NWAs $\mathcal{A}$ over $\tilde{\Sigma}$, the following hold:*

1. *There exists an NWA $\mathcal{A}'$ over $\tilde{\Sigma}$ such that $L(\mathcal{A}') = \mathsf{NW}_\theta(\tilde{\Sigma}) \setminus L(\mathcal{A})$.*
2. *There exists an NWA $\mathcal{A}'$ over $\tilde{\Sigma}$ such that $L(\mathcal{A}') = \mathsf{NW}(\tilde{\Sigma}) \setminus L_\theta(\mathcal{A})$.*

For every $\theta \in \mathfrak{R}$, the proof more or less follows the following schema: A nested word $W \in \mathsf{NW}_\theta(\tilde{\Sigma})$ is encoded by means of an injective mapping as a ranked tree $\mathit{tree}(W)$. One constructs a tree automaton $\mathcal{B}$ recognizing $\mathit{tree}(\mathsf{NW}_\theta(\mathcal{A}))$ and a tree automaton $\mathcal{B}_\mathcal{A}$ recognizing $\mathit{tree}(L_\theta(\mathcal{A}))$. Since tree automata are complementable, there is a tree automaton $\mathcal{B}'_\mathcal{A}$ for the complement of $L(\mathcal{B}_\mathcal{A})$. From $\mathcal{B}'_\mathcal{A} \times \mathcal{B}$, one then extracts an NWA $\mathcal{A}'$ such that $L(\mathcal{A}') = \mathit{tree}^{-1}(L(\mathcal{B}'_\mathcal{A} \times \mathcal{B})) = \mathsf{NW}_\theta(\tilde{\Sigma}) \setminus L(\mathcal{A})$.

Note that, in Theorem 2.22, statement 1. is implied by 2. and vice versa by the following lemma:

Lemma 2.23. For all $\theta \in \mathfrak{R}$, there are deterministic NWAs $\mathcal{A}_\theta$ and $\mathcal{A}'_\theta$ over $\tilde{\Sigma}$ such that $L(\mathcal{A}_\theta) = \mathsf{NW}_\theta(\tilde{\Sigma})$ and $L(\mathcal{A}'_\theta) = \mathsf{NW}(\tilde{\Sigma}) \setminus \mathsf{NW}_\theta(\tilde{\Sigma})$.

Proof (sketch). The lemma is easily seen for contexts and phases. In the case of ordered words, one has to keep track of the stacks that currently have a stack symbol pushed. For k-scope words, one verifies that all call-return edges span over at most k contexts (some call-return edge spans over more that k contexts, respectively). ∎

While phase-bounded and ordered NWAs are in general not determinizable [LMP07], the other restrictions allow for the following theorem:

Theorem 2.24 ([LMP10a, LNP14a]). *Let $\theta \in \{k\text{-}\mathsf{cnt}, k\text{-}\mathsf{scp} \mid k \geq 1\}$. Then, for all NWAs $\mathcal{A}$ over $\tilde{\Sigma}$, there exists a deterministic NWA $\mathcal{A}'$ over $\tilde{\Sigma}$ such that $L(\mathcal{A}') = L_\theta(\mathcal{A})$.*

Moreover, all of the above restrictions give rise to a characterization in terms of MSO logic nwMSO($\tilde{\Sigma}$). The proof essentially uses Theorem 2.22, which reduces negation in MSO logic to complementation in NWAs.

Table 2.1: Closure, decidability, and expressiveness properties for NWA

	$\cup$	$\cap$	Compl.	Det.	Emptiness	Logic
RegL	yes	yes	yes	yes	NLOG	MSO [Büc60, Elg61, Tra62]
CFL	yes	no	no	no	P	$\exists$MatchFO [LST95]
1-NWA ($\|P\| = 1$)	yes	yes	yes	yes	P	MSO [AM09b]
2-NWA ($\|P\| = 2$)	yes	yes	no [Bol08]	no [LMP07]	undecidable	EMSO $\subsetneq$ MSO [Bol08]
k-context NWA	yes	yes	yes [LMP10a]	yes [LMP10a]	NP-c [QR05, LTKR08]	MSO [LMP10a]
k-phase NWA	yes	yes	yes [LMP07]	no [LMP07]	2-EXPTIME-c [LMP07, LMP08b]	MSO [LMP07]
k-scope NWA	yes	yes	yes [LNP14a]	yes [LNP14a]	PSPACE-c [LTN12]	MSO [LNP14a]
ordered NWA	yes	yes	yes [LNP14b]	no [LMP10a]	2-EXPTIME-c [ABH08]	MSO [LNP14b]

Theorem 2.25 ([LMP07, LNP14a, LNP14b]). *Let $\theta \in \mathfrak{R}$. Then, for all $L \subseteq \mathsf{NW}_\theta(\tilde{\Sigma})$, the following statements are equivalent:*

- *There is an NWA $\mathcal{A}$ over $\tilde{\Sigma}$ such that $L(\mathcal{A}) = L$.*
- *There is a sentence $\varphi \in \mathrm{nwMSO}(\tilde{\Sigma})$ such that $L(\varphi) = L$.*

From Theorems 2.15–2.18 and Theorem 2.25, one can infer that model checking bounded NWAs against MSO properties is decidable: Given $\theta \in \mathfrak{R}$, an NWA $\mathcal{A}$ over $\tilde{\Sigma}$, and a sentence $\varphi \in \mathrm{nwMSO}(\tilde{\Sigma})$, do we have $L_\theta(\mathcal{A}) \subseteq L(\varphi)$?

Table 2.1 summarizes decidability, closure, and expressiveness properties for NWA. In view of these results, it is justified to call NWA a robust automata class when considered over the classes of ordered words and k-context/phase/scope words. This makes them appealing in several verification settings as well as for language-theoretic considerations. Their robustness is underpinned by the results from the next section, which lift classical theorems from Mazurkiewicz trace theory to the recursive setting.

2.5 Nested Traces and Their Automata

Multiply nested words account for the interweaving of calls and returns performed by several processes. However, they do not reflect the inherent concurrency of events that are involved in distinct processes. A natural model of executions of concurrent recursive programs are nested traces, which combine nested words and Mazurkiewicz traces. Thus, we fix a distributed alphabet $\tilde{\Sigma} = (\Sigma, P, \mathit{type}, \delta)$ (cf.

Section 2.5). The definition of a nested trace is very similar to that of a nested word (cf. Definition 2.1 on page 18), but instead of just one direct-successor relation $\lhd_{+1}$, there is one such relation $\lhd_p$ for every process $p \in P$.

Definition 2.26 (nested trace [BGH09]). *A* nested trace *over the distributed alphabet $\tilde{\Sigma}$ is a triple $(E, \lhd, \lambda)$ where*

- *E is a nonempty finite set of* events,
- *$\lambda : E \to \Sigma$ is the* event-labeling function, *and*
- *$\lhd = \lhd_{\mathsf{cr}} \cup \bigcup_{p \in P} \lhd_p \subseteq E \times E$ is an acyclic edge relation*

such that properties 1.–4. below are satisfied. Here, the sets E_p, E_{call}, and E_{ret} are given as in Definition 2.1.

1. *For all $p \in P$, $\leqslant_p \stackrel{\text{def}}{=} \lhd_p^*$ is a total order on E_p (with strict part $<_p$), and $\lhd_p$ is the direct successor relation of $\leqslant_p$,*
2. *$\lhd_{\mathsf{cr}}$ induces a bijection between E_{call} and E_{ret},*
3. *for all $(e, f) \in \lhd_{\mathsf{cr}}$, there is $p \in P$ such that $e \in E_p$ and $f \in E_p$, and*
4. *for all $(e_1, f_1), (e_2, f_2) \in \lhd_{\mathsf{cr}}$ and $p \in P$ such that $e_1 \in E_p$, $e_2 \in E_p$, and $e_1 <_p e_2 <_p f_1$, we have $f_2 <_p f_1$.* ♢

The set of nested traces over $\tilde{\Sigma}$ is denoted by $\mathsf{NTr}(\tilde{\Sigma})$. Again, we do not distinguish isomorphic nested traces.

Example 2.27. Consider the distributed alphabet $\tilde{\Sigma}$ from Example 2.2 on page 19, with $P = \{1, 2\}$. A nested trace over $\tilde{\Sigma}$ is depicted in Figure 2.4. ♢

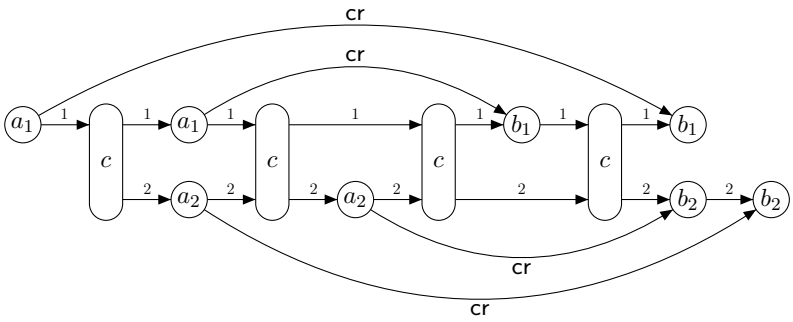


Figure 2.4: The nested trace $T_{2.4}$

Next, we define a distributed automata model running over nested traces, originally introduced in [BGH09]. Here, *distributed* means that every process has its own local state space and transition relation. More precisely, there is a transition relation for every action $a \in \Sigma$. A transition involving a can access the local state of any process from $\delta(a)$, and modify it. Since $a \in \Sigma_{\mathsf{call}} \cup \Sigma_{\mathsf{ret}}$ implies $|\delta(a)| = 1$, executing call and return actions updates the local state of only one single component.

Definition 2.28 (NTA [BGH09]). *A* nested-trace automaton (NTA) *over* $\tilde{\Sigma}$ *is a tuple* $\mathcal{C} = ((S_p)_{p\in P}, \Gamma, \Delta, \iota, F)$. *Here, the* S_p *are disjoint finite sets of* local states *(*S_p *containing the local states of process* p*), and* Γ *is the nonempty finite set of stack symbols. Given a set* $P' \subseteq P$*, let* $S_{P'}$ *denote the cartesian product* $\prod_{p\in P'} S_p$. *The tuple* $\iota \in S_P$ *is a* global initial state*, and* $F \subseteq S_P$ *is the set of* global final states. *Finally,* $\Delta = \Delta_{\mathsf{call}} \uplus \Delta_{\mathsf{ret}} \uplus \Delta_{\mathsf{int}}$ *is the* transition relation*, partitioned into*

- $\Delta_{\mathsf{call}} \subseteq \bigcup_{p\in P}(S_p \times \Sigma_{\mathsf{call}} \times \Gamma \times S_p)$,
- $\Delta_{\mathsf{ret}} \subseteq \bigcup_{p\in P}(S_p \times \Sigma_{\mathsf{ret}} \times \Gamma \times S_p)$, *and*
- $\Delta_{\mathsf{int}} \subseteq \bigcup_{a\in\Sigma_{\mathsf{int}}}(S_{\delta(a)} \times \{a\} \times S_{\delta(a)})$. ♢

Let $\mathbb{S} = \bigcup_{P'\subseteq P} S_{P'}$. For $s \in \mathbb{S}$ and $p \in P$, we let s_p be the p-th component of s (if it exists). A *run* of the NTA $\mathcal{C}$ on a nested trace $T = (E, \lhd, \lambda)$ will include a mapping $\rho : E \to \mathbb{S}$ such that $\rho(e) \in S_{\delta(\lambda(e))}$ for all $e \in E$. Intuitively, for process $p \in \delta(\lambda(e))$, the component $\rho(e)_p$ is the state that p reaches *after* executing e. Before we specify when ρ is actually a run, let us define another mapping $\rho^- : E \to \mathbb{S}$ that also satisfies $\rho^-(e) \in S_{\delta(\lambda(e))}$ for all $e \in E$. Intuitively, $\rho^-(e)$ collects the current source states of those processes that are involved in executing e. We let $\rho^-(e) = (s_p)_{p\in\delta(\lambda(e))}$ where

$$s_p = \begin{cases} \iota_p & \text{if } e \text{ is } \lhd_p\text{-minimal} \\ \rho(f)_p & \text{if } f \lhd_p e\,. \end{cases}$$

Recall that, hereby, $\rho(f)_p$ denotes the p-th component of $\rho(f) \in S_{\delta(\lambda(f))}$. This component indeed exists since $p \in \delta(\lambda(f))$. Now, we call the pair (ρ, σ), with $\sigma : E_{\mathsf{call}} \cup E_{\mathsf{ret}} \to \Gamma$, a *run* of $\mathcal{C}$ on T if,

- for all $(e, f) \in \lhd_{\mathsf{cr}}$, we have $\sigma(e) = \sigma(f)$, and
- for all $e \in E$, it holds

$$\begin{cases} \big(\rho^-(e)_p, \lambda(e), \sigma(e), \rho(e)_p\big) \in \Delta_{\mathsf{call}} & \text{if } e \in E_{\mathsf{call}} \cap E_p \\ \big(\rho^-(e)_p, \lambda(e), \sigma(e), \rho(e)_p\big) \in \Delta_{\mathsf{ret}} & \text{if } e \in E_{\mathsf{ret}} \cap E_p \\ \big(\rho^-(e), \lambda(e), \rho(e)\big) \in \Delta_{\mathsf{int}} & \text{if } e \in E_{\mathsf{int}}\,. \end{cases}$$

To determine if run (ρ, σ) is accepting, we look at the global final state reached at the end of a run. It collects, for all $p \in P$, the local state associated with the $\lhd_p$-maximal event, or ι_p if $E_p = \emptyset$. Formally, let $t = (t_p)_{p\in P} \in S_P$ be given by

$$t_p = \begin{cases} \rho(e)_p & \text{if } e \text{ is } \lhd_p\text{-maximal} \\ \iota_p & \text{if } E_p = \emptyset\,. \end{cases}$$

We call (ρ, σ) *accepting* if $t \in F$. Finally, we denote by $L(\mathcal{C})$ the set of nested traces over $\tilde{\Sigma}$ that come with an accepting run of $\mathcal{C}$.

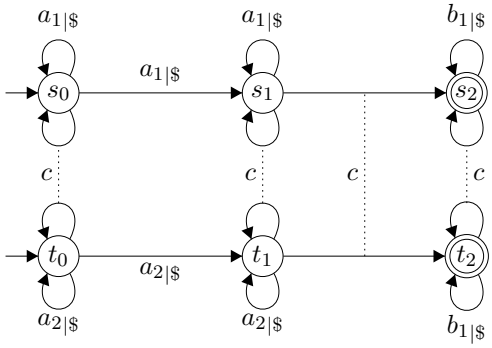


Figure 2.5: The NTA $\mathcal{C}_{2.5}$

Example 2.29. Again, we assume the distributed alphabet $\tilde{\Sigma}$ from Example 2.2, with $P = \{1, 2\}$. Consider the NTA $\mathcal{C}_{2.5} = ((S_p)_{p\in P}, \{\$\}, \Delta, \iota, F)$ over $\tilde{\Sigma}$ depicted in Figure 2.5. Its components are given by $S_1 = \{s_0, s_1, s_2\}$, $S_2 = \{t_0, t_1, t_2\}$, $\iota = (s_0, t_0)$, $F = \{(s_2, t_2)\}$. The new feature (compared to NWAs) are the synchronizing transitions from Δ_{int}, which contains $((s_i, s_i), c, (s_i, s_i))$ for all $i \in \{0, 1, 2\}$ as well as $((s_1, t_1), c, (s_2, t_2))$. Note that $\mathcal{C}_{2.5}$ is eventually forced to execute "concurrent" occurrences of a_1 and a_2: When one process moves on to s_1 or t_1, then executing c is no longer possible unless the other process catches up. The nested trace $T_{2.4}$ on page 27 is accepted by $\mathcal{C}_{2.5}$. Note that, there are indeed concurrent occurrences of a_1 and a_2. ♢

A special case of NTAs is given when $\Sigma = \Sigma_{\mathsf{int}}$. Then, an NTA is precisely an *asynchronous automaton* (also *Zielonka automaton*) [Zie87], and a nested trace is actually a *Mazurkiewicz trace*. However, according to our terminology, we rather call it a *trace automaton*. Since all actions from Σ have the same type, the language of a trace automaton may be seen as a set of Mazurkiewicz traces [DR95].

2.6 Realizability of NWA Specifications

Note that NTAs are a truly concurrent model as processes may move independently unless they perform synchronizing actions from Σ_{int}. On the other hand, NWAs possess one single state space, and the global control may enforce an order even on a priori independent actions. Yet, there are tight connections between NTAs and NWAs. To reveal that relation, we will first study more closely the relation between nested words and nested traces.

First, we can naturally associate with a nested trace $T = (E, \lhd, \lambda)$ over $\tilde{\Sigma}$ its linearizations, each of which fixes an ordering of uncomparable (wrt. $\leq$) events. Intuitively, a linearization is one possible scheduling policy of a nested trace. Formally, a *linearization* of T is any nested word of the form $W = (E, \lhd', \lambda) \in \mathsf{NW}(\tilde{\Sigma})$ such that $\leq \subseteq \leq'$. Note that this implies $\lhd_{\mathsf{call}} = \lhd'_{\mathsf{call}}$. For example, the nested word from Figure 2.1 on page 19 is a linearization of the nested trace from Figure 2.4 on page 27. By $\mathit{lin}(T)$, we denote the set of linearizations of T. Given

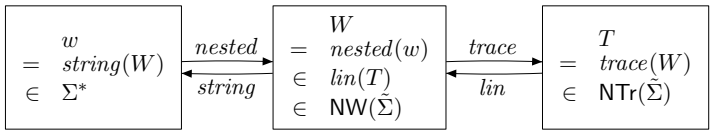


Figure 2.6: Relation between strings, nested words, and nested traces

$W \in \mathsf{NW}(\tilde{\Sigma})$, there is a unique nested trace $T \in \mathsf{NTr}(\tilde{\Sigma})$ such that $W \in \mathit{lin}(T)$. We denote this trace by $\mathit{trace}(W)$. This is extended to languages $L \subseteq \mathsf{NW}(\tilde{\Sigma})$, and we set $\mathit{trace}(L) = \{\mathit{trace}(W) \mid W \in L\}$. The relation between the mappings *string* and *nested* (from Section 2.1) as well as *lin* and *trace* is illustrated in Figure 2.6.

Since $\mathit{nested}(\mathit{string}(W)) = W$ for any nested word W, we may consider the set of linearizations of T as a string language over Σ. This will facilitate some definitions when we consider languages up to a congruence relation taking into account that some actions are independent (those that do not share processes), while others are not (those that have at least one process in common).

Given the distributed alphabet $\tilde{\Sigma}$, we define an *independence relation* $I_{\tilde{\Sigma}} \stackrel{\text{def}}{=} \{(a,b) \in \Sigma \times \Sigma \mid \delta(a) \cap \delta(b) = \emptyset\}$. Note that $I_{\tilde{\Sigma}}$ is irreflexive and symmetric. Its complement, the *dependence relation* $D_{\tilde{\Sigma}} \stackrel{\text{def}}{=} (\Sigma \times \Sigma) \setminus I_{\tilde{\Sigma}}$, is, therefore, reflexive and symmetric. With this, let $\sim_{\tilde{\Sigma}} \subseteq \Sigma^* \times \Sigma^*$ be the least congruence relation that satisfies $ab \sim_{\tilde{\Sigma}} ba$ for all $(a,b) \in I_{\tilde{\Sigma}}$. For example, if $\tilde{\Sigma}$ is the distributed alphabet from Example 2.2 on page 19, then $\{a_1a_2cb_1b_2\,,\ a_2a_1cb_1b_2\,,\ a_1a_2cb_2b_1\,,\ a_2a_1cb_2b_1\}$ is an equivalence class of $\sim_{\tilde{\Sigma}}$. The equivalence relation $\sim_{\tilde{\Sigma}}$ is lifted in the natural way to nested words: we let $W \sim_{\tilde{\Sigma}} W'$ if $\mathit{string}(W) \sim_{\tilde{\Sigma}} \mathit{string}(W')$. We say that $L \subseteq \mathsf{NW}(\tilde{\Sigma})$ is $\sim_{\tilde{\Sigma}}$*-closed* if we have $L = [L]_{\tilde{\Sigma}} \stackrel{\text{def}}{=} \{W \in \mathsf{NW}(\tilde{\Sigma}) \mid W \sim_{\tilde{\Sigma}} W' \text{ for some } W' \in L\}$.

We will consider an NWA $\mathcal{A}$ to be a specification of a system, and we are looking for a *realization* or *implementation* of $\mathcal{A}$, which is provided by an NTA $\mathcal{C}$ such that $L(\mathcal{C}) = \mathit{trace}(L(\mathcal{A}))$. Actually, specifications often have a "global" view of the system, and the difficult task is to *distribute* the state space onto the processes, which henceforth communicate in a restricted manner that conforms to the predefined system architecture $\tilde{\Sigma}$. Note that, unlike $\mathit{lin}(L(\mathcal{C}))$, the language $L(\mathcal{A})$ is not necessarily $\sim_{\tilde{\Sigma}}$-closed. However, $\mathcal{A}$ may yet be considered as an incomplete specification so that we can still ask for an NTA $\mathcal{C}$ such that $L(\mathcal{C}) = \mathit{trace}(L(\mathcal{A}))$.

Note that it is easy to come up with an NWA recognizing the linearizations of the nested traces for a given NTA. Here, the state space of the NWA is the cartesian product of the local state spaces.

Lemma 2.30. Let $\mathcal{C}$ be an NTA over $\tilde{\Sigma}$. There is an NWA $\mathcal{A}$ over $\tilde{\Sigma}$ such that $L(\mathcal{A}) = \mathit{lin}(L(\mathcal{C}))$.

We are, however, interested in the other direction of transforming a given NWA into an NTA. As a preparation, we now recall two well-known theorems from Mazurkiewicz trace theory. The first one, Zielonka's celebrated theorem, applies to distributed alphabets such that $\Sigma = \Sigma_{\mathsf{int}}$. It will later be lifted to general distributed alphabets.

Theorem 2.31 ([Zie87]). *Suppose $\Sigma = \Sigma_{\mathsf{int}}$. Let $\mathcal{A}$ be an NWA over $\tilde{\Sigma}$ such that $L(\mathcal{A})$ is $\sim_{\tilde{\Sigma}}$-closed. Then, there is an NTA $\mathcal{C}$ over $\tilde{\Sigma}$ such that $L(\mathcal{C}) = \mathit{trace}(L(\mathcal{A}))$.*

Note that the theorem actually yields a *deterministic* trace automaton (we omit the definition). The doubly exponential complexity (in the number of processes) of Zielonka's construction has been reduced to singly exponential [GM06, GGMW10].

Moreover, under the assumption $\Sigma = \Sigma_{\mathsf{int}}$, closure under $\sim_{\tilde{\Sigma}}$ is a decidable criterion:

Theorem 2.32 ([Mus94, PWW98]). *The following is* PSPACE*-complete:*

INSTANCE: $\tilde{\Sigma}$ such that $\Sigma = \Sigma_{\mathsf{int}}$; NWA $\mathcal{A}$ over $\tilde{\Sigma}$
QUESTION: Is $L(\mathcal{A})$ $\sim_{\tilde{\Sigma}}$-closed?

Note that, in these theorems, the NWA and the NTA do not employ any stack so that we actually deal with a finite and a trace automaton, respectively. However, one can lift Zielonka's theorem to *arbitrary* distributed alphabets:

Theorem 2.33 ([BGH09]). *Let $\mathcal{A}$ be an NWA over $\tilde{\Sigma}$ such that the language $L(\mathcal{A})$ is $\sim_{\tilde{\Sigma}}$-closed. There is an NTA $\mathcal{C}$ over $\tilde{\Sigma}$ such that $L(\mathcal{C}) = \mathit{trace}(L(\mathcal{A}))$.*

Proof (sketch). We interpret the given NWA $\mathcal{A} = (S, \Gamma, \Delta, \iota, F)$ as an NWA $\mathcal{B}$ over a new distributed alphabet $\tilde{\Omega} = (\Omega, P, \delta', \mathit{type}')$. Here, $\Omega = \Sigma \times \Gamma$ and, for all $(a, A) \in \Sigma \times \Gamma$, $\mathit{type}'((a, A)) = \mathsf{int}$ and $\delta'((a, A)) = \delta(a)$. A transition $(s, a, A, s') \in \Delta_{\mathsf{call}} \cup \Delta_{\mathsf{ret}}$ in $\mathcal{A}$ is considered as a transition $(s, (a, A), s')$ in the new NWA $\mathcal{B}$, and a transition $(s, a, s') \in \Delta_{\mathsf{int}}$ is translated to $(s, (a, A), s')$ with A arbitrary. One can now apply Theorem 2.31 to obtain, from $\mathcal{B}$, an NTA $\mathcal{C}$ over $\tilde{\Omega}$. Finally, we reinterpret $\mathcal{C}$ as an NTA $\mathcal{C}'$ over the original alphabet $\tilde{\Sigma}$. In particular, a call or return transition $(s, (a, A), s')$ becomes (s, a, A, s'). ∎

Remark 2.34. The size of $\mathcal{C}$ is at most doubly exponential in $|\mathcal{A}|$ and triply exponential in $|\Sigma|$, when we use the construction from [GM06].

Actually, Theorem 2.33 is a corollary of a more general statement, which uses the notion of a lexicographic normal form and is also used to prove a different result on NWA realizability (Theorem 2.38 below).

Theorem 2.33 demonstrates that NWAs, though they have a global view of the system in terms of one single state space, are suitable specifications for NTAs provided they recognize a $\sim_{\tilde{\Sigma}}$-closed language. Unfortunately, it is in general undecidable if the language of a given NWA is $\sim_{\tilde{\Sigma}}$-closed. This can be easily shown by a reduction from the undecidable emptiness problem (cf. Theorem 2.5 on page 20).

Theorem 2.35. *The following problem is undecidable:*

INSTANCE: $\tilde{\Sigma}$; NWA $\mathcal{A}$ over $\tilde{\Sigma}$
QUESTION: Is $L(\mathcal{A})$ $\sim_{\tilde{\Sigma}}$-closed?

Therefore, we will consider restrictions to θ-words, for suitable $\theta \in \mathfrak{R}$. This will allow us to define decidable sufficient criteria for the transformation of an NWA into an NTA. We will state a Zielonka-like theorem that is tailored to this restriction. In the theorem, we require that an NWA *represents* the θ-words of a system, while the final implementation can produce executions that do not fit into the θ-restriction.

A θ-representation is a set of nested words that does not distinguish between *locally equivalent* θ-words. Here, two nested words $W, W' \in \mathsf{NW}(\tilde{\Sigma})$ are said to be locally equivalent, written $W \sim^{\mathrm{loc}}_{\tilde{\Sigma}} W'$, if there are $u, v \in \Sigma^*$ and $(a, b) \in I_{\tilde{\Sigma}}$ such that $\mathit{string}(W) = uabv$ and $\mathit{string}(W') = ubav$.

Definition 2.36. *Let $\theta \in \mathfrak{R}$ and $L \subseteq \mathsf{NW}(\tilde{\Sigma})$. We call L a* θ-representation *if $L \subseteq \mathsf{NW}_\theta(\tilde{\Sigma})$ and, for all $W, W' \in \mathsf{NW}_\theta(\tilde{\Sigma})$ such that $W \sim^{\mathrm{loc}}_{\tilde{\Sigma}} W'$, we have $W \in L$ iff $W' \in L$.* ♢

Example 2.37. For the NWA $\mathcal{A}_{2.2}$ from page 20, we have that $L(\mathcal{A}_{2.2})$ is both a 2-phase representation and an ordered representation. It is, however, neither a k-context nor a k-scope representation, for any $k \geq 1$. ♢

Next, we present our Zielonka theorem suited to θ-representations. Hereby, we require that θ be a member of the set $\mathfrak{R}^- \stackrel{\text{def}}{=} \{k\text{-}\mathsf{cnt}, k\text{-}\mathsf{ph} \mid k \geq 1\}$. We do not know if the following result holds for bounded-scope or ordered representations.

Theorem 2.38 ([BGH09]). *Let $\theta \in \mathfrak{R}^-$ and let $\mathcal{A}$ be an NWA over $\tilde{\Sigma}$ such that $L(\mathcal{A})$ is a θ-representation. Then, there is an NTA $\mathcal{C}$ over $\tilde{\Sigma}$ such that $L(\mathcal{C}) = \mathit{trace}(L(\mathcal{A}))$.*

The proof of Theorem 2.38 relies on the definition of a lexicographic normal form that allows one to apply Ochmańskis Theorem [Och95]. The crux is that reordering independent events in order to obtain the lexicographic normal form shall not affect membership in the given θ-representation. This, however, requires an extension of the underlying distributed alphabet. Note that the theorem was stated in [BGH09] only for the phase-restriction, but proving it for contexts is along the same lines.

Indeed, being a θ-representation is a decidable criterion (for all $\theta \in \mathfrak{R}$, including "bounded scope" and "ordered"). It is also decidable whether the θ-restriction of a given NWA is $\sim_{\tilde{\Sigma}}$-closed (which implies that it is a θ-representation).

Theorem 2.39. *The following problems are decidable (for every $\theta \in \mathfrak{R}$) in elementary time:*

INSTANCE:	$\tilde{\Sigma}$; NWA $\mathcal{A}$ over $\tilde{\Sigma}$
QUESTION 1:	Is $L_\theta(\mathcal{A}) \sim_{\tilde{\Sigma}}$-closed?
QUESTION 2:	Is $L_\theta(\mathcal{A})$ a θ-representation?

Proof (sketch). We will give the rough idea of the proof, which is inspired by [Mus94, PWW98] where similar problems are addressd in a stack-free setting. Consider first Question 1. Using Theorem 2.22, we transform the given NWA $\mathcal{A}$ into an NWA $\mathcal{A}'$ for the complement, i.e., such that $L(\mathcal{A}') = \mathsf{NW}(\tilde{\Sigma}) \setminus L_\theta(\mathcal{A})$. Afterwards, we build an NWA $\mathcal{B}$ recognizing all nested words $W \in \mathsf{NW}(\tilde{\Sigma})$ where $string(W)$ is of the form $uabv$ such that $(a, b) \in I_{\tilde{\Sigma}}$ and $ubav \in string(W')$ for some $W' \in L(\mathcal{A}')$. Then, $L_\theta(\mathcal{A})$ is *not* $\sim_{\tilde{\Sigma}}$-closed iff $L_\theta(\mathcal{B}) \neq \emptyset$. The latter is decidable due to Theorems 2.15–2.18. To solve Question 2, we build $\mathcal{A}'$ such that $L(\mathcal{A}') = \mathsf{NW}_\theta(\tilde{\Sigma}) \setminus L(\mathcal{A})$ (again, using Theorem 2.22). ∎

2.7 Realizability of MSO Specifications

On a higher level than NWAs, concurrent recursive programs can be specified in logic, be it MSO logic or temporal logic (for the latter, see the subsequent section). Similarly to the word case, the logic $\mathrm{ntMSO}(\tilde{\Sigma})$ is defined just as nwMSO but, instead of $\lhd_{+1}$, it features a collection of predicates $\lhd_p$, one for every process $p \in P$. Given a sentence $\varphi \in \mathrm{ntMSO}(\tilde{\Sigma})$, we denote by $L(\varphi)$ the set of nested traces T over $\tilde{\Sigma}$ such that $T \models \varphi$.

Again, we will recall a well known result for concurrent programs without stacks, and then lift it to the recursive setting.

Theorem 2.40 ([Tho90]). *Suppose $\Sigma = \Sigma_{\mathsf{int}}$ and let $L \subseteq \mathsf{NTr}(\tilde{\Sigma})$. The following statements are effectively equivalent:*

1. *There is an NTA $\mathcal{C}$ such that $L(\mathcal{C}) = L$.*
2. *There is a sentence $\varphi \in \mathrm{ntMSO}(\tilde{\Sigma})$ such that $L(\varphi) = L$.*

Note that this theorem cannot be lifted to nested structures without imposing any restriction. This is due to the fact that NWAs (and, therefore, NTAs) are not complementable (Theorem 2.21), while MSO logic is closed under negation. However, it will turn out that restricting to k-context/k-phase traces also helps in this case.

Of course, given a restriction $\theta \in \mathfrak{R}$, we first have to clarify what we mean by a *θ-trace*. There are at least two reasonable possibilities. For example, we may call a nested trace T a θ-trace if *all* linearizations of T are θ-words. Alternatively, we may require that *some* linearization of T is a θ-word. We will choose the latter, existential, definition, as it captures more nested traces. Note that there are similar options and definitions in the setting of MSCs (cf. Chapter 3).

Thus, we call $T \in \mathsf{NTr}(\tilde{\Sigma})$ a *θ-trace* if $lin(T) \cap \mathsf{NW}_\theta(\tilde{\Sigma}) \neq \emptyset$. We adopt other notations for nested words and let $\mathsf{NTr}_\theta(\tilde{\Sigma})$ be the set of θ-traces. Moreover, for an NTA $\mathcal{C}$, we let $L_\theta(\mathcal{C}) \stackrel{\text{def}}{=} L(\mathcal{C}) \cap \mathsf{NTr}_\theta(\tilde{\Sigma})$. For a sentence $\varphi \in \mathrm{ntMSO}(\tilde{\Sigma})$, the set $L_\theta(\varphi)$ is defined accordingly.

Example 2.41. The nested trace $T_{2.4}$ is a 2-phase trace (it admits the 2-phase linearization $W_{2.1}$), a 4-context trace, a 3-scope trace, and an ordered trace (since its linearization $W_{2.1}$ is ordered). ♢

As a preparation of a logical characterization of NTAs, we show that they are complementable for some restrictions of nested traces. This is an analogue of Theorem 2.22 for NWAs. As we rely on Theorem 2.38, we have to restrict to the set $\mathfrak{R}^- = \{k\text{-}\mathsf{cnt}\,,\, k\text{-}\mathsf{ph} \mid k \geq 1\}$, i.e., to bounded contexts or bounded phases.

Lemma 2.42. Let $\theta \in \mathfrak{R}^-$ and let $\mathcal{C}$ be an NTA over $\tilde{\Sigma}$. Then, there is an NTA $\mathcal{C}'$ such that $L(\mathcal{C}') = \mathsf{NTr}_\theta(\tilde{\Sigma}) \setminus L(\mathcal{C})$.

Proof. From the given NTA $\mathcal{C}$ be an NTA, using Lemma 2.30 and Theorem 2.22, we get an NWA $\mathcal{A}$ over $\tilde{\Sigma}$ such that $L(\mathcal{A}) = \mathsf{NW}_\theta(\tilde{\Sigma}) \setminus \mathit{lin}(L(\mathcal{C}))$. Observe that $L(\mathcal{A})$ is a θ-representation. By Theorem 2.38, there is an NTA $\mathcal{C}'$ over $\tilde{\Sigma}$ such that $L(\mathcal{C}') = \mathit{trace}(L(\mathcal{A}))$. One easily verifies that we actually have $L(\mathcal{C}') = \mathsf{NTr}_\theta(\tilde{\Sigma}) \setminus L(\mathcal{C})$. ∎

In particular, Lemma 2.42 implies that there is an NTA recognizing the set $\mathsf{NTr}_\theta(\tilde{\Sigma})$. The following theorem constitutes a generalization of Theorem 2.40 adapted to θ-traces, where $\theta \in \mathfrak{R}^-$.

Theorem 2.43 ([BGH09]). *The following implications are effective:*

1. *For every NTA $\mathcal{C}$ over $\tilde{\Sigma}$, there is a sentence $\varphi \in \mathrm{ntMSO}(\tilde{\Sigma})$ such that $L(\varphi) = L(\mathcal{C})$.*
2. *Let $\theta \in \mathfrak{R}^-$. For every sentence $\varphi \in \mathrm{ntMSO}(\tilde{\Sigma})$, there is an NTA $\mathcal{C}$ over $\tilde{\Sigma}$ such that $L(\mathcal{C}) = L_\theta(\varphi)$.*

Proof (sketch). The proof of 1. follows the standard construction for the translation of automata into logic. One guesses an assignment of states and stack symbols to events in terms of existentially quantified second-order variables. Then, a first-order kernel checks if the assignments actually correspond to an accepting run.

For the proof of 2., we proceed by structural induction. Hereby, the only critical case is negation, which will be taken care of by Lemma 2.42. For simplicity, we suppose that there are no free variables. To get an automaton for $\neg\varphi$, suppose that we already have an NTA $\mathcal{C}$ such that $L(\mathcal{C}) = L_\theta(\varphi)$. By Lemma 2.42, there is an NTA $\mathcal{C}'$ such that $L(\mathcal{C}') = \mathsf{NTr}_\theta(\tilde{\Sigma}) \setminus L(\mathcal{C})$. We have $L(\mathcal{C}') = L_\theta(\neg\varphi)$ so that we are done. ∎

2.8 Temporal Logic for Nested Words

In this section, we consider the model-checking problem for *phase-bounded* NWAs. Note that we can already infer, from the previous results, that it is decidable

whether all (phase-bounded) nested words accepted by a given NWA satisfy a given MSO formula. However, since the complexity is inherently nonelementary, it is worth looking at *temporal logics* for nested words.

There has been a whole bunch of papers considering the model-checking problem for temporal logics over (multiply) nested words [Ati10, BCGZ14, CGNK12, LTN12, ABKS12, BS14]. These works differ in the choice of the behavioral restriction described before (context-, phase-, scope-bounded, ordered), but also in the concrete temporal logic adopted for the model-checking task. While [Ati10, ABKS12] consider properties over strings such as classical LTL rather than nested words, [LTN12] introduces a temporal logic that allows one to identify related call and return positions of a given process and to distinguish between linear successors (referring to the word structure) and abstract successors (involving the nesting edges). However, as a matter of fact, there is so far no agreement on a canonical temporal logic for nested words, not even for those with one single nesting relation [AAB+08, AEM04].

We, therefore, consider the class of all temporal logics as defined in the book by Gabbay, Hodkinson, and Reynolds [GHR94], which subsumes virtually all existing formalisms. The unifying feature of these temporal logics is that their modalities are defined in MSO logic. In [BCGZ14], we showed that satisfiability and model checking for any MSO-definable temporal logic are decidable in EXPTIME when restricting to phase-bounded executions. We improved this in [BKM13] showing that the problems are still elementary if k is part of the input. This is an important issue, as an elementary procedure allows for a gradual adjustment of k at the cost of only an elementary blow-up. In the following, we will review the main results of [BKM13].

We fix a distributed alphabet $\tilde{\Sigma} = (\Sigma, P, type, \delta)$. For a natural number $m \in \mathbb{N}$, we call $\varphi \in \mathrm{nwMSO}(\tilde{\Sigma})$ an *m-ary modality* if its free variables consist of one first-order variable x and m set variables $X_1, \ldots, X_m$. The idea is that variable X_i is interpreted as the set of positions where the i-th argument of the modality holds.

Definition 2.44. *A* temporal logic *(over $\tilde{\Sigma}$) is given by a triple $\mathfrak{L} = (\mathcal{M}, arity, [\![-]\!])$ including*

- *a finite set $\mathcal{M}$ of* modality names*,*
- *a mapping $arity : \mathcal{M} \to \mathbb{N}$, and*
- *a mapping $[\![-]\!] : \mathcal{M} \to \mathrm{nwMSO}(\tilde{\Sigma})$ such that, for $M \in \mathcal{M}$ with $arity(M) = m$, $[\![M]\!]$ is an m-ary modality.* ♢

The syntax of $\mathfrak{L}$, i.e., the set of *formulas* $\varphi \in \mathrm{Form}(\mathfrak{L})$, is given by

$$\varphi \quad ::= \quad a \mid \neg\varphi \mid \varphi \vee \varphi \mid M(\underbrace{\varphi, \ldots, \varphi}_{arity(M)})$$

where a ranges over Σ and M ranges over $\mathcal{M}$.

The semantics of a formula $\varphi \in \mathrm{Form}(\mathfrak{L})$ wrt. a nested word $W = (E, \lhd, \lambda)$ over $\tilde{\Sigma}$ is defined inductively as a set $[\![\varphi]\!]_W \subseteq E$, containing the events of W that satisfy φ. Formally, $[\![-]\!]_W$ is given as follows:

- $[\![a]\!]_W \stackrel{\text{def}}{=} \{e \in E \mid \lambda(e) = a\}$
- $[\![\neg\varphi]\!]_W \stackrel{\text{def}}{=} E \setminus [\![\varphi]\!]_W$
- $[\![\varphi_1 \vee \varphi_2]\!]_W \stackrel{\text{def}}{=} [\![\varphi_1]\!]_W \cup [\![\varphi_2]\!]_W$
- $[\![M(\varphi_1, \ldots, \varphi_m)]\!]_W \stackrel{\text{def}}{=} \{e \in E \mid W \models_{\mathcal{I}_e} [\![M]\!]\}$
  where $\mathcal{I}_e$ is such that x is mapped to e, and X_i to $[\![\varphi_i]\!]_W$ for all $i \in \{1, \ldots, m\}$

Example 2.45. As mentioned above, a wide range of temporal logics have been defined for nested words and concurrent systems. Their *until* operator usually depends on what is considered a path between two word positions and, more specifically, on a notion of *successor*. In the classical setting of words without nesting relations, one naturally considers the direct successor following the linear order. The situation is less clear in the presence of one or more nesting relations. In [AAB+08], Alur et al. identify three different kinds of successors in *singly* nested words, namely the linear, call, and abstract successor. Each of them comes with a separate until operator.

Towards nested words with multiple nesting relations, Atig et al. consider only the linear successor [Ati10, ABKS12], while La Torre and Napoli also define modalities that correspond to linear, call, and abstract successors [LTN12]. As an example, we demonstrate how to deal with the *abstract until* in our framework. An abstract p-path in a nested word is a path that does not choose a linear direct successor *from* a call position or *to* a return position (wrt. $\lhd_{\mathsf{cr}} \cap (E_p \times E_p)$). The temporal-logic formula $\varphi \,\mathsf{U}^{\mathrm{a}}_p\, \psi$ for the abstract until says that, starting from x, there is an abstract p-path (represented by a second-order variable Y) to some event x'. Hereby, we require that φ is satisfied along the path ($Y \subseteq X_1$) until x' satisfies ψ ($x' \in X_2$). Formally, the binary modality $\mathsf{U}^{\mathrm{a}}_p$ is defined by

$$
\begin{aligned}
&[\![\mathsf{U}^{\mathrm{a}}_p]\!](x, X_1, X_2) = \\
&\exists Y. \exists x'. \Big(Y \subseteq X_1 \wedge x' \in X_2 \wedge \\
&\qquad\qquad \forall z. (z \in Y \vee z = x') \rightarrow \big(z = x \vee \exists y. (y \in Y \wedge \varphi_p(y, z))\big) \Big) .
\end{aligned}
$$

Hereby,

$$\varphi_p(y, z) \stackrel{\text{def}}{=} (p(y) \wedge y \lhd_{\mathsf{cr}} z) \vee (y \lhd_{+1} z \wedge \neg \mathsf{call}_p(y) \wedge \neg \mathsf{ret}_p(z))$$

where $p(x) \stackrel{\text{def}}{=} \bigvee_{\substack{a \in \Sigma \\ p \in \delta(a)}} a(x)$ and, for $\tau \in \{\mathsf{call}, \mathsf{ret}\}$, $\tau_p(x) \stackrel{\text{def}}{=} p(x) \wedge \bigvee_{\substack{a \in \Sigma \\ \mathit{type}(a) = t}} a(x)$.

Indeed, all the modalities considered in [AAB+08, Ati10, ABKS12, LTN12] are MSO-definable. However, they appear to be just a few of many other possibilities. For example, one may define an abstract path including two or more nesting relations, or include past-time counterparts of until modalities, which are not present

in [LTN12]. Such extensions can be realized in our framework by giving their definition in MSO. An elementary upper bound of the satisfiability and model-checking problem follows immediately from the result stated below, without changing anything in the decidability proof. ♢

Let $\mathfrak{L}$ be a temporal logic over $\tilde{\Sigma}$. For a temporal-logic formula $\varphi \in \mathrm{Form}(\mathfrak{L})$, we let $L(\varphi)$ be the set of nested words W such that $e \in [\![\varphi]\!]_W$ where e is the (unique) minimal event of W. Now, the corresponding model-checking problem is defined as follows:

Problem 2.46. NWA-Model-Checking$(\tilde{\Sigma}, \mathfrak{L})$:

Instance:	NWA $\mathcal{A}$ over $\tilde{\Sigma}$; $\varphi \in \mathrm{Form}(\mathfrak{L})$; $k \geq 1$ (encoded in unary)
Question:	Do we have $L_{k\text{-ph}}(\mathcal{A}) \subseteq L(\varphi)$?

For $n \geq 0$, let $\mathrm{M}\Sigma_n(\tilde{\Sigma})$ be the set of $\mathrm{nwMSO}(\tilde{\Sigma})$-formulas of the form

$$\exists \overline{X}_1 . \forall \overline{X}_2 \ldots \exists / \forall \overline{X}_n . \varphi$$

where φ is a first-order formula, i.e., it does not contain any second-order quantifier, and the $\overline{X}_i$ are blocks of second-order variables. A temporal logic $\mathfrak{L} = (\mathcal{M}, \mathit{arity}, [\![-]\!])$ is called $\mathrm{M}\Sigma_n(\tilde{\Sigma})$-*definable* if, for all $M \in \mathcal{M}$, we have $[\![M]\!] \in \mathrm{M}\Sigma_n(\tilde{\Sigma})$.

Theorem 2.47 ([BKM13]). *Let $n \geq 0$ and let $\mathfrak{L}$ be some $\mathrm{M}\Sigma_n(\tilde{\Sigma})$-definable temporal logic. Then,* NWA-Model-Checking$(\tilde{\Sigma}, \mathfrak{L})$ *is in* $(n+2)$-Exptime.

Theorem 2.48 ([BKM13]). *There is a distributed alphabet $\tilde{\Sigma}$ such that, for all $n \geq 1$, there is an $\mathrm{M}\Sigma_n(\tilde{\Sigma})$-definable temporal logic $\mathfrak{L}$ for which the problem* NWA-Model-Checking$(\tilde{\Sigma}, \mathfrak{L})$ *is n-Expspace-hard.*

Two key ideas are pursued in the proof of the upper bound. First, we translate, in polynomial time, a temporal logic formula into an MSO formula in a certain normal form. The construction is based on Hanf's locality theorem and independent of the number of phases. Second, we show that an MSO formula in normal form can be transformed into a tree automaton in $(n+1)$-fold exponential space. The tree automaton works on tree encodings of multiply nested words and can then be checked for emptiness. One of its key ingredients is a tree automaton recovering the direct successor relation of the encoded multiply nested words. We show that such an automaton can be computed in polynomial space, avoiding the generic doubly exponential construction given in [LMP07].

2.9 Perspectives

Most results presented in this chapter rely on the specific restriction of the domain of nested words to bounded contexts/phases, scopes, or to ordered words. It will be worthwhile to study a more generic setting by bounding the split-width. The model-checking question has been well-studied here as well [CGNK12, AGNK14b, Men14], but not much is known about realizability.

Note that some realizable specifications will inevitably yield implementations in terms of NWAs that are non-deterministic and suffer from deadlocks. One should, therefore, study classes of NWAs that are arguably more "realistic" meaning, in particular, that they are deterministic and deadlock-free [SEM03, ADGS13]. A natural question is then to ask for a specification formalism that guarantees such realistic implementations.

It also remains to study realizability in the realm of recursive processes *communicating through FIFO channels* [LMP08a, HLMS12]. The model-checking question is by now well understood, in particular thanks to the split-width technique [AGNK14b, Cyr14]. To the best of our knowledge, realizability questions for such communicating recursive processes have not been considered. The quest for *controllers*, whose study has been initiated in [AGNK14a], seems to be closely related.

CHAPTER 3

Parameterized Message-Passing Systems

While, in Chapter 2, the communication topology was static and fixed once for all in terms of a distributed alphabet $\tilde{\Sigma}$, we now consider systems that can be run on any topology from a (possibly infinite) class of topologies. Thus, the topology becomes a parameter of the system. We will study parameterized systems in the realm of message passing.

There has been a large body of literature on parameterized concurrent systems (e.g., [EN03, EK04, LMP10b, ABQ11, DSZ11, AHH13, EGM13, Esp14, AJKR14]), with a focus on verification: Does the given system satisfy a specification independently of the topology or the number of processes? A variety of different models have been introduced, covering a wide range of communication paradigms such as broadcasting, rendez-vous, token-passing, etc. However, it is justified to say that, so far, there is no such thing as a canonical or "robust" model of parameterized concurrent systems. In particular, expressiveness issues have not been considered. One reason for that may be that the semantics of parameterized systems is often described in terms of strings that represent interleavings of independent events. Since such interleavings blur the natural direct-successor relation of a single process, it is hard to find logical or algebraic formalisms over strings that match the expressive power of automata.

Similarly to nested traces from the previous chapter, the semantics that we present here does not rely on interleavings but keeps a maximum of information on an execution, such as a direct-successor relation for each process. This will finally allow us to come up with classes of parameterized message-passing systems that can be considered robust, since they satisfy at least one of the following properties:

(a) They have a decidable emptiness problem.

(b) They are closed under boolean operations.

(c) They enjoy a natural logical characterization.

In particular, we provide several Büchi-Elgot-Trakhtenbrot Theorems, which roughly read as follows:

> *Given a formula φ and a class $\mathfrak{T}$ of topologies, there is a system $\mathcal{A}$ such that, whenever $\mathcal{A}$ is run on a topology from $\mathfrak{T}$, it accepts exactly the behaviors that are a model of φ.*

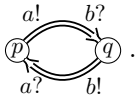
Depending on the logic and the class of topologies, such a result will require further restrictions on the behavior of a system, similar in spirit to the restrictions that we considered in Chapter 2 in the realm of multiply nested words.

3.1 Topologies

As we are in a parameterized setting, we assume that there are a *finite* but *unbounded* number of processes. We consider that processes are equipped with interfaces, through which they can talk to other processes. When interfaces of different processes are plugged together, we obtain a topology.

Though the number of processes may be unbounded, we will assume that there is a *fixed* nonempty finite set $\mathcal{N} = \{a, b, c, \ldots\}$ of *interface names* (or, simply, *interfaces*). When we consider pipelines or rings, then we may set $\mathcal{N} = \{a, b\}$ where a refers to the right neighbor and b to the left neighbor of a process. A pipeline with four processes is depicted in Figure 3.1 and a ring with five processes in Figure 3.2. They allow a process to execute an action $a!$, which sends a message to its right neighbor, or an action $b!$, which sends a message to the left neighbor. The complementary receive actions are $a?$ and $b?$, which receive from the right and left neighbor, respectively. For grids, we will need two more names, which refer to adjacent processes above and below (Figure 3.4). Ranked trees require an interface for each of the (boundedly many) children of a process, as well as pointers to the father process (see Figure 3.3 for a binary tree).

Actually, we assume that two processes p and q that are adjacent in a topology communicate through *channels*. More precisely, there are two FIFO channels between p and q, one for messages sent from p to q, and one for messages from q to p. Thus, an edge $p \overset{a}{\underset{b}{-}} q$ in the topology shall be understood as



.

As every process can only talk directly to $|\mathcal{N}|$ neighbors, topologies are structures of *bounded degree*. This excludes star topologies or unranked trees, but takes into account such natural cases as pipelines, ranked trees, rings, and grids.

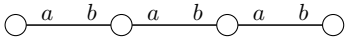


Figure 3.1: Pipeline

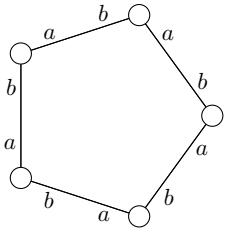


Figure 3.2: Ring

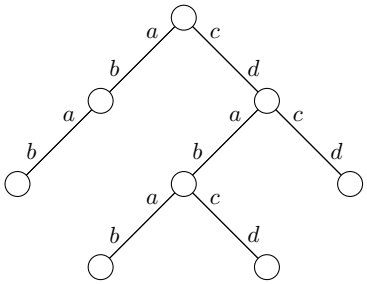


Figure 3.3: Tree topology

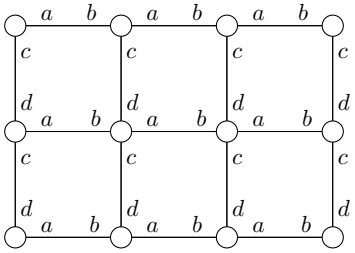


Figure 3.4: Grid

Definition 3.1 (topology). *A* topology *over* $\mathcal{N}$ *is a pair* $\mathcal{T} = (P, \longmapsto)$ *where* P *is the nonempty finite set of* processes *and* $\longmapsto \subseteq P \times \mathcal{N} \times \mathcal{N} \times P$ *is the* edge relation. *We write* $p \overset{a\ b}{\longmapsto} q$ *for* $(p, a, b, q) \in \longmapsto$, *which signifies that the* a*-interface of* p *points to* q, *and the* b*-interface of* q *points to* p. *We require that, whenever* $p \overset{a\ b}{\longmapsto} q$, *the following hold:*

(a) $p \neq q$ *(there are no self loops),*

(b) $q \overset{b\ a}{\longmapsto} p$ *(adjacent processes are mutually connected), and*

(c) *for all* $a', b' \in \mathcal{N}$ *and* $q' \in P$ *such that* $p \overset{a'\ b'}{\longmapsto} q'$, *we have* $a = a'$ *iff* $q = q'$ *(an interface points to at most one process, and two distinct interfaces point to distinct processes).* ♢

The automata and logics that we are going to define will not be able to distinguish isomorphic topologies. An exception is Section 3.5, where topologies are *fixed*.

Example 3.2. In this chapter, the focus will be on the specific topology classes that are illustrated in Figures 3.1–3.4. We define pipelines and rings as topologies over $\{a, b\}$, and binary trees and grids as topologies over $\{a, b, c, d\}$. However, whenever convenient, we assume that pipelines and rings are topologies over $\{a, b, c, d\}$ as well, though they do not make use of c and d. Note that several of the results in this chapter actually extend to more general classes.

- A *pipeline* is a topology over $\{a, b\}$ of the form $(\{1, \ldots, n\}, \longmapsto)$ where $n \geq 2$ and $\longmapsto = \{\, (i, a, b, i+1) \mid i \in [n-1] \,\} \cup \{\, (i+1, b, a, i) \mid i \in [n-1] \,\}$. Note that a pipeline is uniquely determined by its number of processes.

- A *ring* is like a pipeline where the endpoints are glued together. I.e., it is a topology of the form $(\{1,\dots,n\}, \longmapsto)$ where $n \geq 3$ and the edge relation is given by $\longmapsto = \{\,(i,a,b,(i \bmod n)+1) \mid i \in [n]\} \cup \{\,((i \bmod n)+1,b,a,i) \mid i \in [n]\}$. Like a pipeline, since we do not distinguish isomorphic topologies, a ring is uniquely determined by its number of processes. A *ring forest* is simply a disjoint union of rings.

- A *(binary-)tree topology* is a topology $(P, \longmapsto)$ over $\{a,b,c,d\}$ where P is a prefix-closed subset of $\{0,1\}^*$ such that $|P| \geq 2$, and $\longmapsto = \{\,(u,a,b,u0) \mid u, u0 \in P\} \cup \{\,(u0,b,a,u) \mid u, u0 \in P\} \cup \{\,(u,c,d,u1) \mid u, u1 \in P\} \cup \{\,(u1,d,c,u) \mid u, u1 \in P\}$.

- A *grid* is a topology of the form $([m] \times [n], \longmapsto)$ where $\max\{m,n\} \geq 2$ and

$$\begin{aligned} \longmapsto \;=\; & \{\,((i,j),a,b,(i,j+1)) \mid i \in [m] \text{ and } j \in [n-1]\} \\ \cup\; & \{\,((i,j+1),b,a,(i,j)) \mid i \in [m] \text{ and } j \in [n-1]\} \\ \cup\; & \{\,((i,j),c,d,(i+1,j)) \mid i \in [m-1] \text{ and } j \in [n]\} \\ \cup\; & \{\,((i+1,j),d,c,(i,j)) \mid i \in [m-1] \text{ and } j \in [n]\}. \end{aligned}$$ ♢

3.2 Message Sequence Charts

As already mentioned, we do not consider an interleaving semantics but rather keep a maximum of information about the type of an event and the causal dependencies between events. One possible behavior of a system is depicted as a *message sequence chart (MSC)*. An MSC consists of a topology (over the given set of interfaces) and a set of events, which represent the communication actions executed by a system. Events are located on the processes and connected by process and message edges, which reflect causal dependencies.

Definition 3.3 (MSC). *A* message sequence chart (MSC) *over $\mathcal{N}$ is a tuple $M = (P, \longmapsto, E, \lhd, \pi)$ where*

- *$(P, \longmapsto)$ is a topology over $\mathcal{N}$,*
- *E is the nonempty finite set of* events*,*
- *$\lhd \subseteq E \times E$ is the acyclic edge relation, which is partitioned into $\lhd_{\mathsf{proc}}$ and $\lhd_{\mathsf{msg}}$, and*
- *$\pi : E \to P$ determines the location of an event in the topology; for $p \in P$, we let $E_p \stackrel{\text{def}}{=} \{e \in E \mid \pi(e) = p\}$.*

We require that the following hold:

- *$\lhd_{\mathsf{proc}}$ is a union $\bigcup_{p \in P} \lhd_p$ where each $\lhd_p \subseteq E_p \times E_p$ is the direct-successor relation of some total order on E_p,*

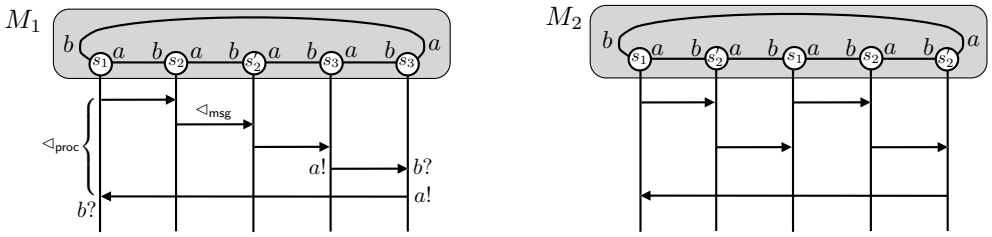


Figure 3.5: Two MSCs with a ring topology

- *there is a partition* $E = E_! \uplus E_?$ *such that* $\lhd_{\mathsf{msg}}$ *induces a bijection between* $E_!$ *and* $E_?$,
- *for all* $(e, f) \in \lhd_{\mathsf{msg}}$, *there are* $a, b \in \mathcal{N}$ *such that* $\pi(e) \overset{a\ b}{\longmapsto} \pi(f)$ *(communication is restricted to adjacent processes), and*
- *for all* $(e, f), (e', f') \in \lhd_{\mathsf{msg}}$ *such that* $\pi(e) = \pi(e')$ *and* $\pi(f) = \pi(f')$, *we have* $e \lhd^*_{\mathsf{proc}} e'$ *iff* $f \lhd^*_{\mathsf{proc}} f'$ *(FIFO).* ♢

Let $\Sigma \overset{\text{def}}{=} \{a! \mid a \in \mathcal{N}\} \cup \{a? \mid a \in \mathcal{N}\}$. We define a mapping $\ell_M : E \to \Sigma$ that associates with each event of the MSC M the type of action that it executes: for $(e, f) \in \lhd_{\mathsf{msg}}$ and $a, b \in \mathcal{N}$ such that $\pi(e) \overset{a\ b}{\longmapsto} \pi(f)$, we set $\ell_M(e) = a!$ and $\ell_M(f) = b?$.

The set of MSCs (over the fixed set $\mathcal{N}$) is denoted by MSC. Like for topologies, we usually do not distinguish isomorphic MSCs.

Example 3.4. Two example MSCs are depicted in Figure 3.5. As we will see, they represent executions of a token-ring protocol. In particular, their underlying topologies are rings (of size 5). The process labelings s_1, s_2, s'_2, s_3 can be ignored for the moment. The events are the endpoints of message arrows, which represent $\lhd_{\mathsf{msg}}$. Process edges are implicitly given; they connect successive events located on the same (top-down) process line. Finally, the mapping ℓ_M is illustrated on a few events. ♢

3.3 Underapproximation Classes

Similarly to multiply nested words (Chapter 2), we will run quickly into undecidability and non-complementability without imposing any restriction on the behavior of a parameterized system. There are several natural restrictions for MSCs. The simplest one is to assume a class of topologies $\mathfrak{T}$ among the topologies over $\mathcal{N}$ (e.g., the class of pipelines, trees, rings, grids, or rings) and to restrict ourselves to the set $\mathsf{MSC}_{\mathfrak{T}}$, defined as the set of MSCs $(P, \longmapsto, E, \lhd, \pi) \in \mathsf{MSC}$ such that $(P, \longmapsto) \in \mathfrak{T}$.

3.3.1 Channel-Bounded MSCs

Some of our (positive) results will deal with systems that have (existentially) k-bounded channels, for some $k \geq 1$ (see, e.g, [LM04, GKM06, GKM07]). Intuitively, an MSC is k-bounded if it can be scheduled in such a way that, along the execution, there are never more than k messages in each channel. Formally, we define boundedness via linearizations. A *linearization* of an MSC $M = (P, \longmapsto, E, \lhd, \pi)$ is any total order $\preceq \subseteq E \times E$ satisfying $\lhd^* \subseteq \preceq$. Then, $\preceq$ is called k-*bounded* if, for all $f \in E$ and all $p, q \in P$ and $a, b \in \mathcal{N}$ such that $p \overset{a\ b}{\longmapsto} q$, we have $|\{e \in E \mid e \preceq f,\ \pi(e) = p, \text{ and } \ell_M(e) = a!\}| - |\{e \in E \mid e \preceq f,\ \pi(e) = q, \text{ and } \ell_M(e) = b?\}| \leq k$. In other words, in any prefix of $\preceq$, there are no more than k pending messages, in every "channel" (p, q).

Definition 3.5 (k-bounded MSC). *For a natural number $k \geq 1$, an MSC is said to be* k-bounded *if it has some k-bounded linearization.* ♢

A stronger restriction is that of *rendez-vous communication*, where a send event and its receive event are executed simultaneously. We then say that an MSC is 0-bounded.

Definition 3.6 (0-bounded MSC). *An MSC $(P, \longmapsto, E, \lhd, \pi) \in \mathsf{MSC}$ is called* 0-bounded *if the graph $(E, \lhd \cup \lhd_{\mathsf{msg}}^{-1})$, does not admit a cycle that uses at least one $\lhd_{\mathsf{proc}}$-edge.* ♢

Intuitively, message edges in a 0-bounded MSC can always be drawn horizontally. For $k \in \mathbb{N}$, the set of k-bounded MSCs is denoted $\mathsf{MSC}_{\exists k}$. Note that, for all $k \in \mathbb{N}$, we have $\mathsf{MSC}_{\exists k} \subsetneq \mathsf{MSC}_{\exists k+1}$. The MSCs from Figure 3.5 are both 0-bounded.

3.3.2 Context-Bounded MSCs

We will see that even restricting to pipelines and 0-bounded MSCs will not be enough to obtain a "robust" class of parameterized automata that is closed under complementation. Inspired by the notion of a context bound in the realm of multiply nested words (cf. Chapter 2), we now introduce context-bounded MSCs.

Actually, the efficiency of distributed algorithms and protocols is usually measured in terms of two parameters: the number n of processes, and the number k of *contexts*. It is, therefore, natural to bound any of these parameters in order to turn parameterized communicating automata (as defined below) into a robust model. Note that bounding the number of processes is known as "cut-off" (see, e.g., [EN03, AKR+14]). A context, on the other hand, restricts communication of a process to patterns such as "send a message to each neighbor and receive a message from each neighbor". The trade-off between n and k is often in favor of a smaller k, so that it is all the more justified to impose a bound on k.

Here, we actually consider more relaxed definitions where, in every context, a process may perform an unbounded number of actions. In an *interface-context*, for example, a process can send and receive an arbitrary number of messages to/from a

fixed neighbor. A second context-type definition allows for arbitrarily many sends to all neighbors, or receptions from a fixed neighbor.

Recall that $\Sigma = \{a! \mid a \in \mathcal{N}\} \cup \{a? \mid a \in \mathcal{N}\}$. A word $w \in \Sigma^*$ is called an

- $(\mathsf{s}\oplus\mathsf{r})$-context if $w \in \{a! \mid a \in \mathcal{N}\}^*$ or $w \in \{a? \mid a \in \mathcal{N}\}^*$,
- $(\mathsf{s1}+\mathsf{r1})$-context if $w \in \{a!\,, b?\}^*$ for some $a, b \in \mathcal{N}$,
- $(\mathsf{s}\oplus\mathsf{r1})$-context if $w \in \{a! \mid a \in \mathcal{N}\}^*$ or $w \in \{b?\}^*$ for some $b \in \mathcal{N}$,
- intf-context if $w \in \{a!\,, a?\}^*$ for some $a \in \mathcal{N}$.

The context type $\mathsf{s1}\oplus\mathsf{r}$ ($w \in \{a!\}^*$ for some $a \in \mathcal{N}$ or $w \in \{b? \mid b \in \mathcal{N}\}^*$) is dual to $\mathsf{s}\oplus\mathsf{r1}$, and we only consider the latter case. All results for $\mathsf{s}\oplus\mathsf{r1}$ presented below are valid for $\mathsf{s1}\oplus\mathsf{r}$.

Let $k \geq 1$ be a natural number and $ct \in \{\mathsf{s}\oplus\mathsf{r}, \mathsf{s1}+\mathsf{r1}, \mathsf{s}\oplus\mathsf{r1}, \mathsf{intf}\}$ be a context type. We say that $w \in \Sigma^*$ is (k, ct)-*bounded* if there are $w_1, \ldots, w_k \in \Sigma^*$ such that $w = w_1 \cdot \ldots \cdot w_k$ and w_i is a ct-context, for all $i \in [k]$. To lift this definition to MSCs $M = (P, \longmapsto, E, \lhd, \pi)$, we define the projection $M|_p \in \Sigma^*$ of M to a process $p \in P$. Let $e_1 \lhd_{\mathsf{proc}} e_2 \lhd_{\mathsf{proc}} \ldots \lhd_{\mathsf{proc}} e_n$ be the unique process-order preserving enumeration of all events of E_p. We let $M|_p = \ell_M(e_1)\ell_M(e_2)\ldots\ell_M(e_n)$. In particular, $E_p = \emptyset$ implies $M|_p = \varepsilon$.

Definition 3.7 ((k, ct)-bounded MSC [BGS14]). *Let $k \geq 1$ be a natural number and $ct \in \{\mathsf{s}\oplus\mathsf{r}, \mathsf{s1}+\mathsf{r1}, \mathsf{s}\oplus\mathsf{r1}, \mathsf{intf}\}$. We say that an MSC $M = (P, \longmapsto, E, \lhd, \pi) \in \mathsf{MSC}$ is (k, ct)-*bounded *if $M|_p$ is (k, ct)-bounded, for all $p \in P$.* ♢

Let $\mathsf{MSC}_{(k,ct)}$ denote the set of all (k, ct)-bounded MSCs.

Example 3.8. As every process in the MSCs from Figure 3.5 executes only two events, both MSCs are $(2, ct)$ bounded, for all context types ct. Now, consider the set L of MSCs depicted in Figure 3.10 on page 55, where we assume that the dotted areas can be arbitrarily large. Contexts are depicted as rectangles. Every MSC in L is $(3, \mathsf{s}\oplus\mathsf{r1})$-bounded: processes 1, 2, and 3 use three contexts, while 4 and 5 use only one context. However, for all $k \geq 1$, there is an MSC in L that is not (k, intf)-bounded: process 4 switches unboundedly often between sending through a and sending through b. ♢

3.4 Communicating Automata

Next, we introduce our automata model of a parameterized message-passing system, which can be run on any topology over the fixed set $\mathcal{N}$. The idea is that each process can execute actions of the form $(a!, m)$, which emits a message m through interface a, or $(a?, m)$, which receives m from interface a.

Given a topology $\mathcal{T} = (P, \longmapsto)$ over $\mathcal{N}$, an automaton will actually run identical subautomata on processes of the same *type*. We define $type : P \to 2^{\mathcal{N}}$ by $type(p) \stackrel{\text{def}}{=} \{a \in \mathcal{N} \mid \text{there are } b \in \mathcal{N} \text{ and } q \in P \text{ such that } p \overset{a\ b}{\longmapsto} q\}$. Thus, $type(p)$ contains those interfaces of p that are connected to some other process in $\mathcal{T}$ (we assume that $\mathcal{T}$ is clear from the context).

Definition 3.9 (CA). *A* communicating automaton (CA) *over the set of interface names $\mathcal{N}$ is a tuple $\mathcal{A} = (S, \iota, Msg, \Delta, \mathcal{F})$ where*

- *S is the finite set of* states*,*
- *$\iota : 2^{\mathcal{N}} \to S$ assigns to every possible type in a topology an* initial state*,*
- *Msg is a nonempty finite set of* messages*,*
- *$\Delta \subseteq S \times (\Sigma \times Msg) \times (S \setminus \iota(2^{\mathcal{N}}))$ is the* transition relation*, and*
- *$\mathcal{F}$ is the* acceptance condition *(which will be specified below).* ♢

Here, $\iota(2^{\mathcal{N}}) \stackrel{\text{def}}{=} \{\iota(A) \mid A \subseteq \mathcal{N}\}$. Thus, processes do not return to initial states. This is a purely technical issue which allows us to consider several automata models in a unifying framework.

Let $M = (P, \longmapsto, E, \lhd, \pi)$ be an MSC. A run of $\mathcal{A}$ on M will be a mapping $\rho : E \to S$ satisfying some requirements. As in the previous chapter, $\rho(e)$ can be seen as the local state of $\pi(e)$ after executing e. To determine when ρ is a run, we define another mapping, $\rho^- : E \to S$, denoting the source states of a transition: whenever $f \lhd_{\mathsf{proc}} e$, we let $\rho^-(e) = \rho(f)$; moreover, if e is $\lhd_{\mathsf{proc}}$-minimal, we let $\rho^-(e) = \iota(type(\pi(e)))$. With this, we say that ρ is a *run* of $\mathcal{A}$ on M if, for all $(e, f) \in \lhd_{\mathsf{msg}}$, there are $a, b \in \mathcal{N}$ and a message $m \in Msg$ such that $\pi(e) \stackrel{a \quad b}{\longmapsto} \pi(f)$, $(\rho^-(e), (a!, m), \rho(e)) \in \Delta$, and $(\rho^-(f), (b?, m), \rho(f)) \in \Delta$.

Whether a run is accepting or not depends on the variant of CA and, in particular, its acceptance condition. Those will be defined below. In any case, acceptance will depend on the mapping $\lambda_{M,\rho} : P \to S$ that collects, for every process p the final state $\lambda_{M,\rho}(p)$ in which p terminates. Formally, $\lambda_{M,\rho}(p) = \iota(type(p))$ if $E_p = \emptyset$. Otherwise, $\lambda_{M,\rho}(p) = \rho(e)$ where e is the $\lhd_{\mathsf{proc}}$-maximal event from E_p.

Example 3.10. The CA $\mathcal{A}_{\text{token}}$ over $\{a, b\}$ from Figure 3.6 describes a simplified version of the IEEE 802.5 token-ring protocol. In the protocol, a single binary token, which can carry a value from $m \in \{0, 1\}$, circulates in a ring. Initially, the token has value 1. A process that has the token may emit a message and pass it along with the token to its a-neighbor. We will abstract the concrete message away and only consider the token value. Since all processes in a ring have type $\{a, b\}$, we have a single initial state s_0 for all of them, i.e., $\iota(\{a, b\}) = s_0$. Whenever a process receives the token from its b-neighbor, it will forward it to its a-neighbor, while

- leaving the token value unchanged (the process then ends in s_2 or s_3), or
- changing its value from 1 to 0, to signal that the message has been received (the process then ends in s_2').

We did not define the acceptance condition formally yet. However, note that it will allow us to require that (i) there is *exactly* one process that terminates in

state s_1 and (ii) no process terminates in a state from $\{t_1, t_2, t_3\}$ (i.e., no process stops halfways). With this condition, MSC M_1 from Figure 3.5 will be accepted by $\mathcal{A}_{\text{token}}$, while M_2 is not accepted, since it requires two processes to end in s_1, which violates the acceptance condition. ♢

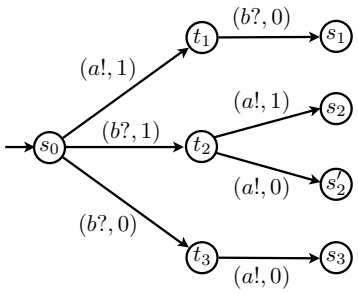


Figure 3.6: The PCA $\mathcal{A}_{\text{token}}$

3.5 Fixed-Topology Communicating Automata

Recall that we aim at Büchi-Elgot-Trakhtenbrot Theorems for (parameterized) CAs. To put the parameterized setting into perspective, we will first recall corresponding results in the case of fixed topologies.

Definition 3.11 (fixed-topology CA). *Let $\mathcal{T} = (P, \longmapsto)$ be a topology over $\mathcal{N}$. A* fixed-topology CA *over $\mathcal{T}$ is a CA $\mathcal{A} = (S, \iota, Msg, \Delta, \mathcal{F})$ over $\mathcal{N}$ where the acceptance condition $\mathcal{F}$ is a set of mappings from P into S.* ♢

Let $\mathcal{A} = (S, \iota, Msg, \Delta, \mathcal{F})$ be a fixed-topology CA over $\mathcal{T} = (P, \longmapsto)$. Moreover, suppose $M = (P, \longmapsto, E, \lhd, \pi) \in \mathsf{MSC}_{\{\mathcal{T}\}}$ is an MSC (with topology $\mathcal{T}$) and $\rho : E \to S$ a run of $\mathcal{A}$ on M. Then, ρ is *accepting* if $\lambda_{M,\rho} \in \mathcal{F}$. By $L(\mathcal{A})$, we denote the set of MSCs $M = (P, \longmapsto, E, \lhd, \pi) \in \mathsf{MSC}_{\{\mathcal{T}\}}$ such that there is an accepting run of $\mathcal{A}$ on M.

Once we fix a topology $\mathcal{T} = (P, \longmapsto)$ (over $\mathcal{N}$), a corresponding MSO logic comes naturally. In particular, it features a predicate $p(x)$ to express that some event is executed by process $p \in P$.

Definition 3.12. *For a topology $\mathcal{T} = (P, \longmapsto)$, the syntax of the logic $\mathrm{MSO}_{\mathcal{T}}$ is given by the grammar*

$$\begin{aligned}\varphi \;::=\; & p(x) \mid a!(x) \mid a?(x) \mid x \lhd_{\mathsf{proc}} y \mid x \lhd_{\mathsf{msg}} y \mid x = y \mid x \in X \mid \\ & \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \exists X.\varphi\end{aligned}$$

where $p \in P$ and $a \in \mathcal{N}$. ♢

As usual, we let $\mathrm{EMSO}_{\mathcal{T}}$ denote the set of formulas of the form $\exists X_1 \ldots \exists X_n.\varphi$ such that φ does not contain any second-order quantification. For an MSC $M =$

$(P, \longmapsto, E, \lhd, \pi) \in \mathsf{MSC}_{\{\mathcal{T}\}}$, a formula $\varphi \in \mathrm{MSO}_{\mathcal{T}}$, and an interpretation $\mathcal{I}$ of variables, the satisfaction relation $M \models_{\mathcal{I}} \varphi$ ($M \models \varphi$ if φ is a sentence) is defined as expected. For example, $M \models_{\mathcal{I}} p(x)$ if $\pi(\mathcal{I}(x)) = p$, $M \models_{\mathcal{I}} a!(x)$ if $\ell_M(\mathcal{I}(x)) = a!$, and $M \models_{\mathcal{I}} x \lhd_{\mathsf{msg}} y$ if $\mathcal{I}(x) \lhd_{\mathsf{msg}} \mathcal{I}(y)$. Note that, since the topology is fixed, $a!(x)$ and $a?(x)$ can actually be expressed using the predicates of the form $p(x)$ and $x \lhd_{\mathsf{msg}} y$. We let $L(\varphi)$ denote the set of MSCs $M \in \mathsf{MSC}_{\{\mathcal{T}\}}$ such that $M \models \varphi$.

We are now ready to present some previously known logical characterizations of fixed-topology CAs. Indeed, fixed-topology CAs over $\mathcal{T}$ and the logic $\mathrm{EMSO}_{\mathcal{T}}$ are expressively equivalent:

Theorem 3.13 ([BL06]). *Let $\mathcal{T}$ be a topology and $L \subseteq \mathsf{MSC}_{\{\mathcal{T}\}}$. The following statements are equivalent:*

1. *There is a fixed-topology CA $\mathcal{A}$ over $\mathcal{T}$ such that $L(\mathcal{A}) = L$.*
2. *There is a sentence $\varphi \in \mathrm{EMSO}_{\mathcal{T}}$ such that $L(\varphi) = L$.*

Proof (sketch). The direction 1. $\Rightarrow$ 2. is standard.

The proof of the transformation 2. $\Rightarrow$ 1. is based on Hanf's normal form, which states that any first-order formula (over structures of bounded degree) is logically equivalent to a boolean combination of the form "there are at least n occurrences of neighborhood N of radius r" (for a uniform $r \in \mathbb{N}$, which depends on the formula) [Han65, Lib04]. Here, the neighborhood of an event comprises all events that have distance at most r in the Gaifman graph $(E, \lhd \cup \lhd^{-1})$ of the underlying MSC. As MSCs are structures of bounded degree (every event has at most three neighbors), there are, up to isomorphism, only finitely many such neighborhoods. Now, to prove 2. $\Rightarrow$ 1., one first constructs a (fixed-topology) CA that "recognizes" neighborhoods of radius r. Another fixed-topology CA then counts these neighborhoods up to some threshold and determines, based on Hanf's normal form, if the original formula is satisfied. ∎

It was shown in [BK12] that the construction of Hanf's normal form can be done in triply exponential time. This implies that Theorem 3.13 is actually effective and that the translation of a formula into an automaton takes only elementary time.

Theorem 3.14 (cf. [BK12]). *The transformation of a fixed-topology CA from a formula as stated in Theorem 3.13 is effective and can be carried out in elementary time.*

It was also shown in [BL06] that fixed-topology CAs over the pipeline $\mathcal{T}$ with two processes are not closed under complementation so that $\mathrm{MSO}_{\mathcal{T}}$ is strictly more expressive. However, Genest, Kuske and Muscholl showed that the assumption of existentially bounded channels yields a robust model of message-passing systems:

Theorem 3.15 ([GKM06]). *Let $\mathcal{T}$ be a topology, $k \in \mathbb{N}$, and $L \subseteq \mathsf{MSC}_{\{\mathcal{T}\}} \cap \mathsf{MSC}_{\exists k}$. The following statements are equivalent:*

- *There is a fixed-topology CA $\mathcal{A}$ over $\mathcal{T}$ such that $L(\mathcal{A}) = L$.*
- *There is a sentence $\varphi \in \mathrm{MSO}_{\mathcal{T}}$ such that $L(\varphi) = L$.*

In particular, it was shown in [GKM06] that there is a fixed-topology CA $\mathcal{A}$ over $\mathcal{T}$ such that $L(\mathcal{A}) = \mathsf{MSC}_{\{\mathcal{T}\}} \cap \mathsf{MSC}_{\exists k}$.

Note that, when restricting MSCs further to universally-k-bounded MSCs (*all* linearizations are k-bounded), one gets a similar logical characterization and a class of CAs that is determinizable [HMK+05]. On the other hand, determinizability fails in the case of existentially bounded channels [GKM07].

3.6 Parameterized Communicating Automata (PCAs)

It is our aim to lift the above logical characterizations of fixed-topology CAs to a parameterized setting. Recall that, when we fixed a topology, the acceptance condition could directly speak about all of its processes (Definition 3.11). However, when we have to cope with a *class* of topologies, this is no longer possible. It is then natural to specify acceptance in terms of a formula from a tailored monadic second-order logic, which scans the final configuration reached by a system: the underlying topology together with the local final states in which the processes terminate. If S is the finite set of such local states, the formula thus defines a set of S-labeled topologies, i.e., structures $(P, \longmapsto, \lambda)$ where $(P, \longmapsto)$ is a topology and $\lambda : P \to S$. The corresponding logic $\mathrm{tMSO}(S)$ is given by the grammar

$$\mathcal{F} \ ::= \ u \overset{a\ b}{\longmapsto} v \mid \lambda(u) = s \mid u = v \mid u \in U \mid \neg\mathcal{F} \mid \mathcal{F} \vee \mathcal{F} \mid \exists u.\mathcal{F} \mid \exists U.\mathcal{F}$$

where $a, b \in \mathcal{N}$, $s \in S$, u and v are first-order variables (interpreted as processes), and U is a second-order variable (ranging over sets of processes). As usual, we assume an infinite supply of variables. Satisfaction $(P, \longmapsto, \lambda) \models \mathcal{F}$ for an S-labeled topology $\mathcal{T}$ and a sentence $\mathcal{F} \in \mathrm{tMSO}(S)$ is defined as expected.

Definition 3.16 (PCA). *A* parameterized communicating automaton (PCA, for short) *over $\mathcal{N}$ is a CA $\mathcal{A} = (S, \iota, Msg, \Delta, \mathcal{F})$ over $\mathcal{N}$ where the acceptance condition $\mathcal{F}$ is a sentence from* $\mathrm{tMSO}(S)$. ♢

Let $\mathcal{A} = (S, \iota, Msg, \Delta, \mathcal{F})$ be a PCA over $\mathcal{N}$, let $M = (P, \longmapsto, E, \lhd, \pi)$ be *any* MSC over $\mathcal{N}$, and suppose $\rho : E \to S$ is a run of $\mathcal{A}$ on M. Recall that the mapping $\lambda_{M,\rho} : P \to S$ yields, for each process p, the state in which p terminates. With this, ρ is accepting if $(P, \longmapsto, \lambda_{M,\rho}) \models \mathcal{F}$. By $L(\mathcal{A})$, we denote the set of MSCs M such that there is an accepting run of $\mathcal{A}$ on M.

A (syntactic) subclass of PCAs is obtained when the acceptance condition is only allowed to count terminal states of *non-idle* processes, up to some threshold:

Definition 3.17 (weak PCA). *A PCA* $\mathcal{A} = (S, \iota, Msg, \Delta, \mathcal{F})$ *is called* weak *if the acceptance condition* $\mathcal{F}$ *is a boolean combination of sentences of the form* $\exists^{\geq n} u.\lambda(u) = s$ *where* $n \in \mathbb{N}$ *and* $s \in S \setminus \iota(2^{\mathcal{N}})$. ♢

Here, the formula $\exists^{\geq n} u.\lambda(u) = s$ is an abbreviation of

$$\exists u_1, \ldots, u_n. \bigwedge_{1 \leq i < j \leq n} u_i \neq u_j \ \wedge \bigwedge_{1 \leq i \leq n} \lambda(u_i) = s$$

which says that there are at least n processes that terminate in state s. As $s \notin \iota(2^{\mathcal{N}})$, these processes have to be *non-idle*, meaning that they execute at least one event. Thus, unlike a PCA, a weak PCA is not allowed to talk about the topology but only about the *events* of an MSC. In particular, it cannot express that "the topology has five processes" but only "five processes are non-idle". This is a priori weaker, but also makes sense, since it reflects the intuition that a PCA accepts behaviors rather than topologies.

The set of PCAs over $\mathcal{N}$ is denoted by $\mathrm{PCA}_{\mathcal{N}}$, the class of weak PCAs over $\mathcal{N}$ by $\mathrm{wPCA}_{\mathcal{N}}$.

Example 3.18. We resume Example 3.10 on page 46 describing the CA $\mathcal{A}_{\text{token}}$ over $\{a, b\}$. To get a PCA, it remains to specify an appropriate acceptance condition $\mathcal{F}$. Recall that we required that (i) exactly one process terminates in s_1 and (ii) no process terminates in a state from $\{t_1, t_2, t_3\}$. Formally, this is achieved using

$$\mathcal{F} \ = \ \exists^{\geq 1} u.\lambda(u) = s_1 \wedge \neg\exists^{\geq 2} u.\lambda(u) = s_1 \wedge \bigwedge_{1 \leq i \leq 3} \neg\exists^{\geq 1} u.\lambda(u) = t_i \,.$$

As $\mathcal{F}$ is a boolean combination of statements of the form $\exists^{\geq n} u.\lambda(u) = s$ with $s \neq s_0$, the PCA $\mathcal{A}_{\text{token}}$ is actually a weak PCA. Consider again the MSCs M_1 and M_2 from Figure 3.5 on page 43. We have $M_1 \in L(\mathcal{A}_{\text{token}})$ and $M_2 \notin L(\mathcal{A}_{\text{token}})$. ♢

3.7 Logical Characterizations of PCAs

Since an MSC contains two types of varying objects, namely processes and events, it is natural to introduce a two-valued logic to reason about them. Accordingly, we have variables $u, v, w, \ldots$ and $U, V, W, \ldots$ to range over processes and sets of processes, respectively, as well as variables $x, y, z, \ldots$ and $X, Y, Z, \ldots$, ranging over events and sets of events, respectively.

Definition 3.19. *The syntax of the logic* $\mathrm{MSO}_{\mathcal{N}}$ *(over the fixed set of interface names* $\mathcal{N}$*) is given as follows:*

$$\begin{aligned} \varphi \ ::= \ & a!(x) \mid a?(x) \mid a \in \mathit{type}(\pi(x)) \mid x \lhd_{\mathsf{proc}} y \mid x \lhd^*_{\mathsf{proc}} y \mid x \sim y \mid x \lhd_{\mathsf{msg}} y \mid \\ & x = y \mid x \in X \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \exists X.\varphi \mid \psi \\ \psi \ ::= \ & x@u \mid u \overset{a \quad b}{\longmapsto} v \mid u = v \mid u \in U \mid \exists u.\varphi \mid \exists U.\varphi \end{aligned}$$

where $a, b \in \mathcal{N}$. ♢

All predicates are more or less self-explanatory apart from $x \sim y$, saying that x and y are located on the same process, and $x@u$, saying that x is located on u. Note that $\sim$ can be expressed in terms of $\lhd^*_{\mathsf{proc}}$, since $x \sim y$ is equivalent to $(x \lhd^*_{\mathsf{proc}} y) \vee (y \lhd^*_{\mathsf{proc}} x)$. However, some of our results hold for logic fragments that include $\sim$ but discard $\lhd^*_{\mathsf{proc}}$. For the same reason, we include the predicate $a \in \mathit{type}(\pi(x))$, which is actually expressible using the formulas of type ψ.

There are indeed several natural fragments of $\mathrm{MSO}_{\mathcal{N}}$. When we do not allow formulas of the form ψ, then we obtain *weak* MSO, denoted $\mathrm{wMSO}_{\mathcal{N}}$. The fragment $\mathrm{wEMSO}_{\mathcal{N}}$ of $\mathrm{wMSO}_{\mathcal{N}}$ consists of the (ψ-free) formulas of the form $\exists X_1 \ldots \exists X_n.\varphi$ where φ does not contain any second-order quantification. We will also refer to the fragment $\mathrm{wEMSO}_{\mathcal{N}}(\lhd_{\mathsf{proc}}, \sim, \lhd_{\mathsf{msg}})$ of $\mathrm{wEMSO}_{\mathcal{N}}$ where the predicate $\lhd^*_{\mathsf{proc}}$ is not allowed.

Let us formally define when $M \models_{\mathcal{I}} \varphi$ for an MSC $M = (P, \longmapsto, E, \lhd, \pi)$, a formula $\varphi \in \mathrm{MSO}_{\mathcal{N}}$, and an interpretation function $\mathcal{I}$, which maps

- a first-order event variable x to an event $\mathcal{I}(x) \in E$,
- a first-order process variable u to a process $\mathcal{I}(u) \in P$,
- a second-order event variable X to a set $\mathcal{I}(X) \subseteq E$, and
- a second-order process variable U to a set $\mathcal{I}(U) \subseteq P$.

We proceed by induction, but consider only some cases:

- $M \models_{\mathcal{I}} a!(x)$ if $\ell_M(\mathcal{I}(x)) = a!$,
- $M \models_{\mathcal{I}} a?(x)$ if $\ell_M(\mathcal{I}(x)) = a?$,
- $M \models_{\mathcal{I}} a \in \mathit{type}(\pi(x))$ if $a \in \mathit{type}(\pi(\mathcal{I}(x)))$,
- $M \models_{\mathcal{I}} x \sim y$ if $\pi(\mathcal{I}(x)) = \pi(\mathcal{I}(y))$,
- $M \models_{\mathcal{I}} x@u$ if $\pi(\mathcal{I}(x)) = \mathcal{I}(u)$,
- $M \models_{\mathcal{I}} u \overset{a\ b}{\longmapsto} v$ if $\mathcal{I}(u) \overset{a\ b}{\longmapsto} \mathcal{I}(v)$.

The other formulas are interpreted as expected. When φ is a sentence, i.e., it does not have free variables, then satisfaction does not depend on the interpretation function so that we write $M \models \varphi$ instead of $M \models_{\mathcal{I}} \varphi$. In that case, the set of MSCs $M \in \mathsf{MSC}$ such that $M \models \varphi$ is denoted by $L(\varphi)$.

Example 3.20. We continue the token-ring protocol from Examples 3.10 and 3.18. Recall that $\mathcal{N} = \{a, b\}$. We would like to express that there is a process that emits a message and gets an acknowledgment that results from a sequence of forwards through interface a. We first let $\mathrm{fwd}(x, y) \equiv x \overset{a\ b}{\longmapsto} y \wedge \exists z.(x \lhd_{\mathsf{proc}} z \lhd_{\mathsf{msg}} y)$ where $x \overset{a\ b}{\longmapsto} y$ is a shorthand for $\exists u.\exists v.(x@u \wedge y@v \wedge u \overset{a\ b}{\longmapsto} v)$. It is well known that the transitive closure of the relation induced by $\mathrm{fwd}(x, y)$ is definable in $\mathrm{MSO}_{\mathcal{N}}$, too. Let $\mathrm{fwd}^+(x, y)$ be the corresponding formula. It expresses that there is a

sequence of events leading from x to y that alternately takes process and message edges, hereby following the causal order. With this, the desired formula is

$$\varphi \;=\; \exists x, y, z.(x \lhd_{\mathsf{proc}} y \wedge x \lhd_{\mathsf{msg}} z \wedge x \overset{a\ b}{\longmapsto} z \wedge \mathrm{fwd}^{+}(z, y)) \in \mathrm{MSO}_{\mathcal{N}}\,.$$

Consider again Figure 3.5 on page 43. We have $M_1 \models \varphi$ and $M_2 \not\models \varphi$, as well as $L(\mathcal{A}_{\text{token}}) \subseteq L(\varphi)$. ♢

We now turn to logical characterizations of PCAs. We first consider weak PCAs, which turn out to be closely related to $\mathrm{wEMSO}_{\mathcal{N}}$. However, note that restrictions on the logic and the topologies are necessary [Bol14]: over $\mathcal{N} = \{a, b, c, d\}$, there is a *weak* first-order formula without $\lhd^{*}_{\mathsf{proc}}$ and $\sim$ that is not realizable by a weak PCA on the class of ring forests. Thus, we are left with a small margin for positive results. However, the following theorem is an analogon to Theorem 3.13, which was stated there for fixed topologies:

Theorem 3.21 ([Bol14]). *Suppose $\mathcal{N} = \{a, b, c, d\}$. Let $\mathfrak{T}$ be any of the following topology classes: pipelines, trees, grids, or rings. Moreover, let $L \subseteq \mathsf{MSC}_{\mathfrak{T}}$. The following statements are equivalent:*

- *There is $\mathcal{A} \in \mathrm{wPCA}_{\mathcal{N}}$ such that $L(\mathcal{A}) \cap \mathsf{MSC}_{\mathfrak{T}} = L$.*
- *There is $\varphi \in \mathrm{wEMSO}_{\mathcal{N}}(\lhd_{\mathsf{proc}}, \sim, \lhd_{\mathsf{msg}})$ such that $L(\varphi) \cap \mathsf{MSC}_{\mathfrak{T}} = L$.*

We leave open if Theorem 3.21 holds for the full logic $\mathrm{wEMSO}_{\mathcal{N}}$. The answer is positive iff Theorem 3.13 still holds when we include the predicate $\lhd^{*}_{\mathsf{proc}}$, but this is an open problem. However, we can safely include $\lhd^{*}_{\mathsf{proc}}$ when we restrict to existentially bounded MSCs:

Theorem 3.22 ([Bol14]). *Suppose $\mathcal{N} = \{a, b, c, d\}$. Let $\mathfrak{T}$ be any of the following topology classes: pipelines, trees, grids, or rings. Moreover, let $k \in \mathbb{N}$ and $L \subseteq \mathcal{M} \overset{\text{def}}{=} \mathsf{MSC}_{\mathfrak{T}} \cap \mathsf{MSC}_{\exists k}$. The following statements are equivalent:*

- *There is $\mathcal{A} \in \mathrm{wPCA}_{\mathcal{N}}$ over $\mathcal{N}$ such that $L(\mathcal{A}) \cap \mathcal{M} = L$.*
- *There is a sentence $\varphi \in \mathrm{wEMSO}_{\mathcal{N}}$ such that $L(\varphi) \cap \mathcal{M} = L$.*

Proof (sketch). Like in the proof of Theorem 3.13, we make use of a normal form of first-order logic. However, due to the predicate $\lhd^{*}_{\mathsf{proc}}$, the structures we consider do not have bounded degree anymore. Fortunately, there is a normal form due to Schwentick and Barthelmann [SB99] that does not rely on that assumption (like Gaifman's normal form which, however, seems to be more difficult to apply in

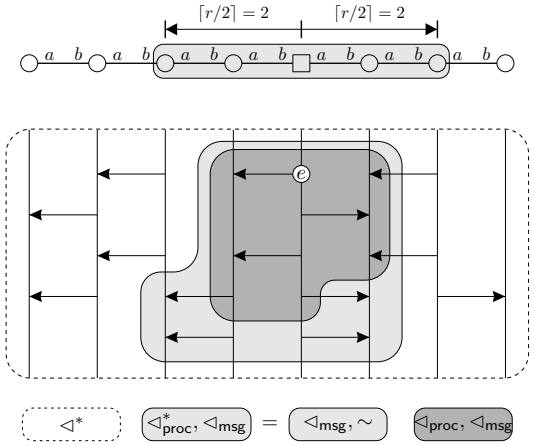


Figure 3.7: Neighborhoods of e with radius $r = 3$

our setting). Their normal form essentially (modulo existential quantification) says that *all* events satisfy a formula φ that only talks about a neighborhood of a certain radius r. Unlike in Hanf's normal form, the size of a neighborhood is unbounded. This is illustrated in Figure 3.7 for several signatures. However, every neighborhood spans over a bounded sub-topopoloy (unless we include a predicate $\lhd^*$ in the logic that refers to the full partial order). Topologies, in turn, are structures of bounded degree so that there are finitely many such sub-topologies. For sub-topology $\mathcal{T}$, using Theorem 3.15 on page 49 by Genest, Kuske, and Muscholl, we can construct a fixed-topology CA $\mathcal{A}_\mathcal{T}$ that evaluates φ on $\mathcal{T}$. The weak PCA we are looking for now glues these finitely many fixed-topology CAs together. More precisely, every process guesses a sub-topology $\mathcal{T}$ and starts $\mathcal{A}_\mathcal{T}$. To make sure that the guess was correct, it communicates it to any communication partner. As neighbors also have to start their own fixed-topology CA, processes have to simulate several of them simultaneously. However, a weak PCA is not able to detect topology neighborhoods by itself. It needs some help from the underlying class of topologies. Informally speaking, we require that paths that form a cycle in some topology form a cycle everywhere, in any other topology from the underlying class. This is satisfied by the classes of pipelines, grid, and trees. For rings, some additional work is needed to get the result. Actually the theorem holds for any *unambiguous* class of topologies (see [Bol14] for details). Note that this proof works similarly for Theorem 3.21. ∎

Let us now turn to (full) PCAs. Again, there is no chance to get an MSO characterization even when we restrict to 0-bounded and certain context-bounded MSCs. Actually, PCAs are not closed under complementation:

Theorem 3.23 ([BGK14]). *Suppose $\mathcal{N} = \{a, b\}$. Let $ct \in \{\mathsf{s}{\oplus}\mathsf{r}, \mathsf{s1}{+}\mathsf{r1}\}$, let $\mathfrak{T}$ be the set of pipelines, and set $\mathcal{M} \stackrel{\text{def}}{=} \mathsf{MSC}_\mathfrak{T} \cap \mathsf{MSC}_{\exists 0} \cap \mathsf{MSC}_{(1,ct)}$. There is $\mathcal{A} \in \mathrm{PCA}_\mathcal{N}$ such that, for all $\mathcal{A}' \in \mathrm{PCA}_\mathcal{N}$, we have $L(\mathcal{A}') \cap \mathcal{M} \neq \mathcal{M} \setminus L(\mathcal{A})$.*

Proof (sketch). Even under the context-type restrictions $\mathsf{s}\oplus\mathsf{r}$ and $\mathsf{s1}{+}\mathsf{r1}$, MSCs over pipelines can encode grid-like structures (cf. Figures 3.8 and 3.9). For grids, there have been similar results, showing that *graph acceptors* are not complementable [Tho96, MST02]. By the grid encoding, the non-complementability result can be transferred to our setting. ∎

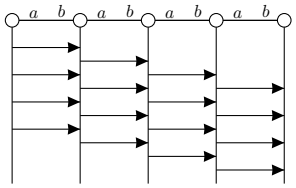


Figure 3.8: Grid encoding for $\mathsf{s1}{+}\mathsf{r1}$

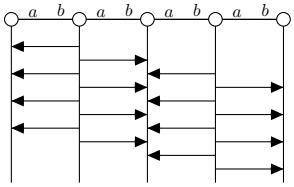


Figure 3.9: Grid encoding for $\mathsf{s}\oplus\mathsf{r}$

The next theorem does not rely on a particular class of topologies. However, recall that we fixed some finite set $\mathcal{N}$ of interface names.

Theorem 3.24 ([BGK14]). *Let $k \geq 1$ and $ct \in \{\mathsf{s}\oplus\mathsf{r1}, \mathsf{intf}\}$. Moreover, let $L \subseteq \mathcal{M} \stackrel{\text{def}}{=} \mathsf{MSC}_{\exists 0} \cap \mathsf{MSC}_{(k,ct)}$. The following statements are equivalent:*

- *There is $\mathcal{A} \in \mathrm{PCA}_{\mathcal{N}}$ such that $L(\mathcal{A}) \cap \mathcal{M} = L$.*
- *There is a sentence $\varphi \in \mathrm{MSO}_{\mathcal{N}}$ such that $L(\varphi) \cap \mathcal{M} = L$.*

Proof (sktech). Again, the construction of a formula from an automaton is rather standard, though we have to cope with a two-sorted logic.

The other direction is by induction on the structure of the formula. In particular, negation translates to complementability of (context-bounded) PCAs, which is the key technical issue in the proof. Complementation relies on a disambiguation construction. From the given PCA $\mathcal{A}$, one constructs another PCA that is unambiguous and equivalent to $\mathcal{A}$ on $\mathsf{MSC}_{(k,ct)}$. Unambiguous here means that every MSC comes with exactly one run (accepting or not). The main idea for this construction is to divide the events of an MSC from $\mathsf{MSC}_{(k,ct)}$ into *zones*, as illustrated in Figure 3.10. We only illustrate the more complicated case $ct = \mathsf{s}\oplus\mathsf{r1}$ (the case intf is similar). An *interval* is a set of *successive* (i.e., not interrupted) events that belong to one and the same process. Then, a zone consists of an interval of successive send events of one process, together with the corresponding receive events, as long as the latter form intervals themselves on the corresponding processes.

One can show that, for each MSC in $\mathsf{MSC}_{(k,\mathsf{s}\oplus\mathsf{r1})}$, there is a zone partitioning such that every process traverses at most $K = k \cdot (|\mathcal{N}|^2 + 2|\mathcal{N}| + 1)$ different zones. The crucial point is now that, on $\mathsf{MSC}_{(k,\mathsf{s}\oplus\mathsf{r1})}$, zones can be computed deterministically by a PCA. During that procedure, a sending process will maintain a summary of all possible "global transitions" induced by a zone, by simulating all possible transitions

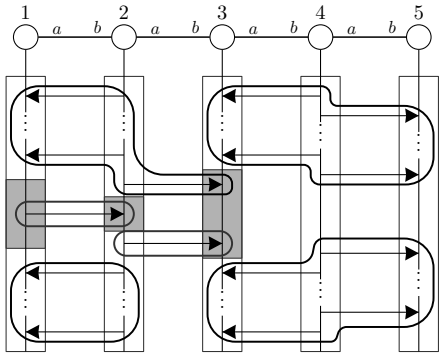


Figure 3.10: s⊕r1-Contexts and Zones

of the receiving processes. The acceptance condition will then check in terms of a tMSO formula whether the summaries can be glued together towards an accepting run. Once we have an unambiguous PCA, complementation is accomplished just by negating the acceptance condition. ∎

Interestingly, determinization procedures have been used to obtain complementability and MSO characterizations for context-bounded and scope-bounded multi-pushdown automata [LMP10a, LNP14a] (cf. Chapter 3). A pattern that we share with these approaches is that of computing *summaries* in a deterministic way. Overall, however, we have to use quite different techniques, which is due to the fact that, in our model, processes evolve asynchronously.

3.8 Model Checking

From the previous section, we conclude that context-bounded PCAs with rendezvous communication form a robust automata model: they are closed under all boolean operations and enjoy a logical characterization in terms of an unrestricted MSO logic. But there is more: the emptiness problem is decidable for some topology classes, which is stated in the following theorem.

Theorem 3.25 ([BGS14]). *Suppose* $\mathcal{N} = \{a, b, c, d\}$. *Let* $ct \in \{\mathsf{s}\oplus\mathsf{r1}, \mathsf{intf}\}$, *and let* $\mathfrak{T}$ *be any of the following topology classes: pipelines, trees, or rings. The following problem is decidable:*

INSTANCE: $\mathcal{A} \in \mathrm{PCA}_{\mathcal{N}}$; $k \geq 1$

QUESTION: $L(\mathcal{A}) \cap \mathsf{MSC}_{\mathfrak{T}} \cap \mathsf{MSC}_{\exists 0} \cap \mathsf{MSC}_{(k,ct)} \neq \emptyset$?

Proof (sketch). The proof relies on the zone partitioning that we used for Theorem 3.24. Consider the case of pipelines. Roughly speaking, we construct a finite automaton (running over pipelines) that guesses a run of the given PCA in terms

of zones and state summaries. As we can assume that every process traverses a bounded number of zones, we actually deal with a finite automaton over a finite alphabet. The finite automaton can now check if its guess actually corresponds to an accepting run. Thus, it accepts all those pipelines that allow for a context-bounded run of the given PCA. Emptiness of that exponentially big (in k) finite automaton can be checked in PSPACE. Accordingly, for tree topologies, we construct a tree automaton. In the case of rings, we essentially use the construction for pipelines. However, some additional work is needed to rule out cyclic dependencies that would contradict the run definition. ∎

Over pipelines and rings, the decision problem from Theorem 3.25 is PSPACE-complete when the acceptance condition is presented as a finite automaton (instead of an MSO formula). Over trees, it is EXPTIME-complete when the acceptance condition is given in terms of a tree automaton. In all cases, we assume that the context bound k is encoded in unary.

From Theorems 3.24 and 3.25, we deduce that context-bounded model checking of PCAs against MSO properties is decidable:

Theorem 3.26 ([BGS14, BGK14]). *Let $ct \in \{\mathsf{s}{\oplus}\mathsf{r1}, \mathsf{intf}\}$ and let $\mathfrak{T}$ be any of the following topology classes over $\mathcal{N} = \{a, b, c, d\}$: pipelines, trees, or rings. The following problem is decidable:*

INSTANCE: $\mathcal{A} \in \mathrm{PCA}_{\mathcal{N}}$; $\varphi \in \mathrm{MSO}_{\mathcal{N}}$; $k \geq 1$

QUESTION: $L(\mathcal{A}) \cap \mathsf{MSC}_{\mathfrak{T}} \cap \mathsf{MSC}_{\exists 0} \cap \mathsf{MSC}_{(k,ct)} \subseteq L(\varphi)$?

Let us conclude by comparing the above model-checking result with related work. Maybe the most interesting feature in view of previous results on model checking parameterized systems is that our MSO logic is *unrestricted*, while other approaches rely on dropping temporal operators such as the next modality from LTL [EN03, AJKR14, JB14]. However, as demonstrated in Example 3.20, next modalities make a lot of sense once the behavior of a parameterized system is modeled as a partial order or MSC rather than a string.

In general, there are several orthogonal ways to get decidability of model checking in a parameterized setting. For example, nonemptiness of PCAs becomes decidable over rings when tokens are unary (i.e., $|Msg| = 1$) [EN03, AJKR14]. Note that our token-ring protocol from Example 3.10 on page 46 assumes a binary token $Msg = \{0, 1\}$, for which nonemptiness is already undecidable in general. However, as we have seen, we obtain decidability when restricting to a bounded number of contexts. Point-to-point communication yields an undecidable model, too, unless processes (i) choose a communication partner nondeterministically [AKR+14], (ii) use broadcasting [DSZ10, DSZ11], (iii) restrict the topologies in a suitable manner (e.g., bounded depth) [Mey08, EGM13, AKR+14], or (iv) bound the number of contexts (our approach).

3.9 Perspectives

The results presented in this chapter can only be called a first step towards a "regular" language theory of parameterized concurrent systems.

Note that Theorems 3.25 and 3.26 apply to concrete "regular" classes of topologies, namely pipelines, trees, and rings. It will be interesting to see if this can be extended to classes of bounded tree/clique width. This is the approach from [AKR+14], though for topologies of unbounded degree.

One should actually try to extend our results to automata models that take account of communication topologies of unbounded degree such as star topologies or, more generally, unranked trees. Such automata may be equipped with registers so that processes can remember, at any time, some of their neighbors [DST13, BCH+13] (see also Chapters 4 and 5). Moreover, many distributed protocols are able to exchange and compare process identifiers. Actually, the logic $\mathrm{MSO}_{\mathcal{N}}$ is so powerful that it is capable of tracing back the origin of a pid in such a system. This will possibly allow us to verify also leader-election protocols and alike. Similar ideas will be applied in Chapter 4 to automata with registers, which can store values from an infinite alphabet (such as pids) and compare them for equality.

Finally, there seem to be close connections with the area of distributed algorithms, which should be explored further. Indeed, there are algorithms that evaluate a given process architecture wrt. logical specifications [GW10] or construct a map of an *anonymous graph* [CDK10], which is similar to what a (weak) PCA does in the proof of Theorem 3.22.

CHAPTER 4

Dynamic Sequential Systems

Like the previous chapter, this chapter also deals with systems whose behavior involves an unbounded number of processes. However, the systems modeled here may be considered *dynamic*, since there will be a feature that allows them to create fresh process identifiers. We will first consider sequential systems, meaning that there is a global control to which all participating processes have access. Consequently, our system model will have one single state space. A study of a model of dynamic *concurrent* systems can be found in Chapter 5.

An action that is performed by a dynamic sequential system is taken from an infinite alphabet. The alphabet is the cartesian product of a finite part A, denoting the *type* or *label* of an action, and an infinite part D, denoting the pid of the process executing the action. In a more general setting, the elements of the infinite set are often referred to as *data values*. As our systems are sequential, a behavior may be described as a string over $A \times D$, which is commonly called a *data word*. Similarly to words over a distributed (visibly) alphabet (cf. Chapter 2), which induce nesting relations, a data word comes with a natural graph structure in terms of edges between two successive word positions as well as between two position that carry the same data value.

Recall that we are interested in modeling systems as automata. The study of automata over infinite alphabets has its origins in the seminal work by Kaminski and Francez [KF94]. Their *finite-memory automata* (more commonly called *register automata*) equip finite-state machines with registers in which data values can be stored and be reused later. Register automata preserve some of the nice properties of finite automata: they have a decidable emptiness problem and are closed under union and intersection. On the other hand, they are neither determinizable nor closed under complementation. There are actually several variants of register

automata, which all have the same expressive power but differ in the complexity of decision problems (see [DL09]).

Subsequently, many more automata models have been considered, aiming at a good balance between expressivity and decidability [NSV04, DL09, KZ10, BL10]. Motivated by the study of logics over data words, Bojanczyk et al. introduced *data automata* [BDM$^+$11]. While register automata are a one-way model, data automata read a data word twice: a finite-state transducer first scans its projection onto A, outputting a word of equal length over some finite alphabet A'. A second device, a finite automaton over A', then checks every projection to positions with the same data value.

Data automata are expressively equivalent to the one-way model of *class memory automata* [BS10]. A class memory automaton does not use registers either. Instead, when reading a word position with, say, some data value d, it can access the state taken after executing the *prior* position with d. This is very much in the spirit of nested-word automata where, at some return position, one can access the state (or, stack symbol) associated with the corresponding call (cf. Chapter 2). We come back to that issue in Section 4.3. Class memory automata [BS10] feature a notion of *freshness*. A transition of a class memory automaton explicitly allows a data value to be declared as fresh, i.e., to occur for the first time in the history of a run. Freshness is an important notion in programming languages when names are supposed to be unique. In our context, it is crucial, since we require that processes are identifiable by unique pids. Freshness for register automata was introduced in [Tze11]. Like ordinary register automata, *fresh-register automata* preserve some of the good properties of finite automata. However, they inherit negative results of register automata and have, for example, an undecidable universality/equivalence problem.

In this chapter, we are aiming at a *robust* automata model for data words, i.e., a model that is closed under complementation (in a restricted sense), has a decidable equivalence problem, and enjoys a logical characterization.[1] Moreover, it should be suitable as a model of dynamic sequential systems. In particular, this means that we are looking for a one-way model so that (variants of) data automata, alternating automata, or pebble automata are out of the question. In Section 4.1, we propose *session automata*. Like register automata, session automata are a syntactical restriction of fresh-register automata, but in an orthogonal way. While register automata drop the feature of *global freshness* (referring to the history) and keep a local variant (referring to the registers), session automata discard local freshness, while keeping the global one. In Section 4.3, we will then look at a more general model that unifies several of the above-mentioned models but still comes with a solution to the realizability problem.

[1] In [BHLM14], we were also aiming at an application in the context of automata learning, where decidability of equivalence is crucial. However, this issue is beyond the scope of the thesis.

4.1 (Fresh-)Register Automata and Session Automata

Let, in the following, A be a finite alphabet and D be an infinite alphabet. We will assume that A and D are disjoint.

For a set $\mathcal{R}$, we let $\mathcal{R}^{\circledast} \stackrel{\text{def}}{=} \{r^{\circledast} \mid r \in \mathcal{R}\}$, $\mathcal{R}^{\odot} \stackrel{\text{def}}{=} \{r^{\odot} \mid r \in \mathcal{R}\}$, and $\mathcal{R}^{\uparrow} \stackrel{\text{def}}{=} \{r^{\uparrow} \mid r \in \mathcal{R}\}$. Below, we introduce automata with a set of registers $\mathcal{R}$. Transitions will be labeled with an element from $\mathcal{R}^{\circledast} \cup \mathcal{R}^{\odot} \cup \mathcal{R}^{\uparrow}$, which determines a register and the operation that is performed on it. More precisely, $r^{\circledast}$ writes a globally fresh value into r, $r^{\odot}$ writes a locally fresh value into r, and $r^{\uparrow}$ uses the value that is currently stored in r. For $\pi \in \mathcal{R}^{\circledast} \cup \mathcal{R}^{\odot} \cup \mathcal{R}^{\uparrow}$, we let $\mathsf{reg}(\pi) = r$ if $\pi \in \{r^{\circledast}, r^{\odot}, r^{\uparrow}\}$. Similarly,

$$\mathsf{op}(\pi) = \begin{cases} \circledast & \text{if } \pi \text{ is of the form } r^{\circledast} \\ \odot & \text{if } \pi \text{ is of the form } r^{\odot} \\ \uparrow & \text{if } \pi \text{ is of the form } r^{\uparrow}\,. \end{cases}$$

Definition 4.1 (fresh-register automaton, cf. [Tze11]). *A* fresh-register automaton (FRA) *over A and D is a tuple $\mathcal{A} = (S, \mathcal{R}, \iota, F, \Delta)$ where*

- *S is the nonempty finite set of* states*,*
- *$\mathcal{R}$ is the nonempty finite set of* registers*,*
- *$\iota \in S$ is the* initial state*,*
- *$F \subseteq S$ is the set of* final states*, and*
- *Δ is a finite set of* transitions*.*

A transition is a tuple of the form $(s, (a, \pi), s')$ where $s, s' \in S$ are the source and target state, respectively, $a \in A$, and $\pi \in \mathcal{R}^{\circledast} \cup \mathcal{R}^{\odot} \cup \mathcal{R}^{\uparrow}$. We call (a, π) the transition label. ♢

For a transition $(s, (a, \pi), s') \in \Delta$, we also write $s \xrightarrow{(a,\pi)} s'$. When taking this transition, the automaton moves from state s to state s' and reads a symbol of the form $(a, d) \in A \times D$. If $\pi = r^{\uparrow} \in \mathcal{R}^{\uparrow}$, then d_i is the data value that is currently stored in r. If $\pi = r^{\circledast} \in \mathcal{R}^{\circledast}$, then d is some *globally fresh* data value that has not been read in the *whole* history of the run; d is then written into register r. Finally, if $\pi = r^{\odot} \in \mathcal{R}^{\odot}$, then d is some *locally fresh* data value that is *currently* not stored in the registers; it will henceforth be stored in register r.

Let us formally define the semantics of $\mathcal{A}$. A configuration is a triple $\gamma = (s, \tau, U)$ where $s \in S$ is the current state, $\tau : \mathcal{R} \rightharpoonup D$ is a partial mapping denoting the current register assignment, and $U \subseteq D$ is the set of data values that have been used so far. We say that γ is *final* if $s \in F$. As usual, we define a transition relation over configurations and let $(s, \tau, U) \xrightarrow{(a,d)} (s', \tau', U')$, where $(a, d) \in A \times D$, if there is a transition $s \xrightarrow{(a,\pi)} s'$ such that the following hold:

1. $\begin{cases} d = \tau(\mathsf{reg}(\pi)) & \text{if } \mathsf{op}(\pi) = \uparrow \\ d \notin \tau(\mathcal{R}) & \text{if } \mathsf{op}(\pi) = \odot \\ d \notin U & \text{if } \mathsf{op}(\pi) = \circledast\,, \end{cases}$

2. $\mathrm{dom}(\tau') = \mathrm{dom}(\tau) \cup \{\mathsf{reg}(\pi)\}$ and $U' = U \cup \{d\}$,

3. $\tau'(\mathsf{reg}(\pi)) = d$ and $\tau'(r) = \tau(r)$ for all $r \in \mathrm{dom}(\tau) \setminus \{\mathsf{reg}(\pi)\}$.

A *run* of $\mathcal{A}$ on a data word $(a_1, d_1) \dots (a_n, d_n) \in (A \times D)^*$ is a sequence

$$\gamma_0 \xrightarrow{(a_1,d_1)} \gamma_1 \xrightarrow{(a_2,d_2)} \dots \xrightarrow{(a_n,d_n)} \gamma_n$$

for suitable configurations $\gamma_0, \dots, \gamma_n$ with $\gamma_0 = (\iota, \emptyset, \emptyset)$. The run is *accepting* if γ_n is a final configuration. The *language* $L(\mathcal{A}) \subseteq (A \times D)^*$ of $\mathcal{A}$ is then defined as the set of data words for which there is an accepting run. Note that FRAs cannot distinguish between data words that are equivalent up to permutation of data values: for $w, w' \in (\Sigma \times D)^*$, we write $w \approx w'$ if $w = (a_1, d_1) \dots (a_n, d_n)$ and $w' = (a_1, d'_1) \dots (a_n, d'_n)$ such that, for all $i, j \in [n]$, we have $d_i = d_j$ iff $d'_i = d'_j$. For instance, $(a, 4)(b, 2)(b, 4) \approx (a, 2)(b, 5)(b, 2)$. We call $L \subseteq (\Sigma \times D)^*$ a *data language* if, for all $w, w' \in (\Sigma \times D)^*$ such that $w \approx w'$, we have $w \in L$ iff $w' \in L$. In particular, $L(\mathcal{A})$ is a data language for every FRA $\mathcal{A}$.

We obtain natural subclasses of fresh-register automata when we restrict the transition labels $(a, \pi) \in A \times (\mathcal{R}^{\circledast} \cup \mathcal{R}^{\odot} \cup \mathcal{R}^{\uparrow})$ in the transitions.

Definition 4.2 (register automaton [KF94]). *A* register automaton (RA) *is an FRA* $(S, \mathcal{R}, \iota, F, \Delta)$ *where every transition label is from* $A \times (\mathcal{R}^{\odot} \cup \mathcal{R}^{\uparrow})$. ♢

Definition 4.3 (session automaton [BHLM14]). *A* session automaton (SA) *is an FRA* $(S, \mathcal{R}, \iota, F, \Delta)$ *where every transition label is from* $A \times (\mathcal{R}^{\circledast} \cup \mathcal{R}^{\uparrow})$. ♢

The following example will demonstrate that RAs and SAs are incomparable wrt. expressive power, and that FRAs are strictly more expressive than both restrictions.

Example 4.4. Consider the set of labels $A = \{\mathsf{req}, \mathsf{ack}\}$ and the set of data values $D = \mathbb{N}$, representing an infinite supply of pids. We model a simple (sequential) system where processes can approach a server and make a request, indicated by req, and where the server can acknowledge these requests, indicated by ack. More precisely, $(\mathsf{req}, d) \in A \times D$ means that the process with pid d performs a request, which is acknowledged when the system executes (ack, d). Consider the following data languages:

- $L_1 =$ "there are at most two open requests at a time"
- $L_2 =$ "a process waits for acknowledgment before its next request"
- $L_3 =$ "every acknowledgment is preceded by a request"
- $L_4 =$ "requests are acknowledged in the order they are received"

- L_5 = "every process makes at most one request"
- L_6 = "all requests take place before all acknowledgments"

Figure 4.1 depicts an RA that recognizes the data language $L_1 \cap \ldots \cap L_4$. That is, it models a server that can store two requests at a time and will acknowledge them in the order they are received). For example, it accepts the data word $(\mathsf{req}, 8)(\mathsf{req}, 4)(\mathsf{ack}, 8)(\mathsf{req}, 3)(\mathsf{ack}, 4)(\mathsf{req}, 8)(\mathsf{ack}, 3)(\mathsf{ack}, 8)$. Note that a transition label containing $r_i^{\odot} \vee r_i^{\uparrow}$ actually refers to two transitions, one using $r_i^{\odot}$ and one using $r_i^{\uparrow}$

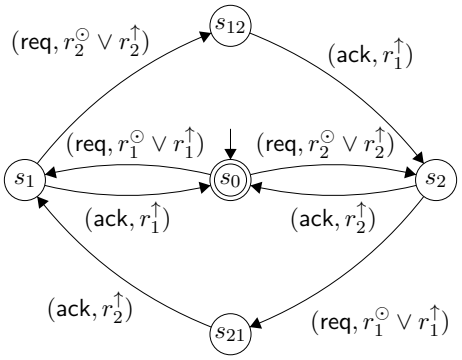


Figure 4.1: Register automaton $\mathcal{A}_{4.1}$ for $L_1 \cap \ldots \cap L_4$

When, in addition, we want to guarantee that every process makes at most one request, we need the freshness operator. Figure 4.1 depicts the SA $\mathcal{A}_{4.2}$ recognizing $L_1 \cap \ldots \cap L_5$. We obtain $\mathcal{A}_{4.2}$ from $\mathcal{A}_{4.1}$ by replacing any occurrence of $r_i^{\odot} \vee r_i^{\uparrow}$ with $r_i^{\circledast}$. While $(\mathsf{req}, 8)(\mathsf{req}, 4)(\mathsf{ack}, 8)(\mathsf{req}, 3)(\mathsf{ack}, 4)(\mathsf{req}, 8)(\mathsf{ack}, 3)(\mathsf{ack}, 8)$ is no longer contained in $L(\mathcal{A}_{4.2})$, $(\mathsf{req}, 8)(\mathsf{req}, 4)(\mathsf{ack}, 8)(\mathsf{req}, 3)(\mathsf{ack}, 4)(\mathsf{ack}, 3)$ is still accepted. Note that $L(\mathcal{A}_{4.2})$ is indeed not recognizable by a (non-fresh) RA.

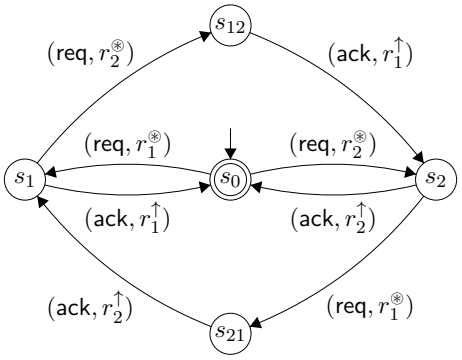


Figure 4.2: Session automaton $\mathcal{A}_{4.2}$ for $L_1 \cap \ldots \cap L_5$

To separate FRAs from RAs and SAs, simply consider data language L_5. Clearly, it is not recognized by any RA nor by any SA, while an FRA recognizing it is given in Figure 4.3 (where r is the only register).

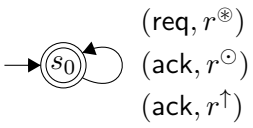


Figure 4.3: Fresh-register automaton $\mathcal{A}_{4.3}$ for L_5

However, it is easily seen that there is no FRA that recognizes $L_3 \cap \ldots \cap L_6$. To address this weakness of FRAs, we will later introduce an automata model that actually captures that language (Section 4.3). ♢

From Example 4.4, we deduce the following:

Corollary 4.5. FRAs are strictly more expressive than both RAs and SAs, while SAs and RAs are incomparable wrt. expressive power.

4.2 Closure Properties of Session Automata

Closure properties of SAs and decidability will be established by means of a normal form of a data word. The crucial observation is that, in an SA, data equality in a data word only depends on the transition labels that generate it. In this section, we suppose that the set of registers of a session automaton is of the form $\mathcal{R} = \{1, \ldots, k\}$ so that we can assume a natural total ordering on it. In the following, let $\Omega \stackrel{\text{def}}{=} (\mathbb{N}_{>0})^{\circledast} \cup (\mathbb{N}_{>0})^{\uparrow}$.

Suppose an SA reads a sequence $u = (a_1, \pi_1) \ldots (a_n, \pi_n) \in (A \times \Omega)^*$ of transition labels. We call u a *symbolic word*. It "produces" a data word iff a register is initialized before it is used. Formally, we say that u is *well-formed* if, for all positions $j \in [n]$ with $\mathsf{op}(\pi_j) = \uparrow$, there is $i \in [n]$ such that $\pi_i = \mathsf{reg}(\pi_j)^{\circledast}$. Let $\mathsf{WF} \subseteq (A \times \Omega)^*$ be the set of all well-formed words. With $u \in (A \times \Omega)^*$, we can associate an equivalence relation $\sim_u \subseteq [n] \times [n]$, letting $i \sim_u j$ iff

- $\mathsf{reg}(\pi_i) = \mathsf{reg}(\pi_j)$, and
- $i \leq j$ and there is no position $k \in \{i+1, \ldots, j\}$ such that $\pi_k = \mathsf{reg}(\pi_i)^{\circledast}$, or $j \leq i$ and there is no position $k \in \{j+1, \ldots, i\}$ such that $\pi_k = \mathsf{reg}(\pi_i)^{\circledast}$.

If u is well-formed, then the data values of any data word $w = (a_1, d_1) \ldots (a_n, d_n)$ that is "accepted via" u conform with the equivalence relation $\sim_u$. That is, we have $d_i = d_j$ iff $i \sim_u j$. We call w a *concretization* of u. Let $\mathit{data}(u)$ denote the set of all concretizations of u. This is extended to sets $L \subseteq (A \times \Omega)^*$ of well-formed words, and we set $\mathit{data}(L) \stackrel{\text{def}}{=} \bigcup_{u \in L \cap \mathsf{WF}} \mathit{data}(u)$. Note that, here, we first filter the well-formed words before applying the operator. Now, let $\mathcal{A} = (S, \mathcal{R}, \iota, F, \Delta)$ be an SA. In the obvious way, we may consider $\mathcal{A}$ as a finite automaton over $A \times (\mathcal{R}^{\circledast} \cup \mathcal{R}^{\uparrow})$. We then obtain a regular language $L_{\mathit{symb}}(\mathcal{A}) \subseteq (A \times \Omega)^*$. It is not difficult to verify that $L(\mathcal{A}) = \mathit{data}(L_{\mathit{symb}}(\mathcal{A}))$.

Though we have a symbolic representation of data languages recognized by SAs, it is in general difficult to compare their languages, since quite different symbolic words may give rise to the same concretizations. For example, we have $\mathit{data}((a,1^{\circledast})(a,1^{\circledast})(a,1^{\uparrow})) = \mathit{data}((a,1^{\circledast})(a,2^{\circledast})(a,2^{\uparrow}))$. However, we can associate, with every data word, a (symbolic) normal form, producing the same set of concretizations. Intuitively, the normal form uses the first (according to the natural total order) register whose current data value is not used anymore. In the above example, $(a,1^{\circledast})(a,1^{\circledast})(a,1^{\uparrow})$ would be in symbolic normal form: the data value stored at the first position in register 1 is not reused so that, at the second position, register 1 has to be overwritten. For that reason, $(a,1^{\circledast})(a,2^{\circledast})(a,2^{\uparrow})$ is not in symbolic normal form. In contrast, $(a,1^{\circledast})(a,2^{\circledast})(a,2^{\uparrow})(a,1^{\uparrow})$ is in normal form, since register 1 is read at the end of the word.

Let $w = (a_1,d_1)\dots(a_n,d_n) \in (A \times D)^*$ be a data word and $d \in \{d_1,\dots,d_n\}$. By $\mathit{first}(d)$, we denote the position j where d occurs for the first time, i.e., such that $d_j = d$ and there is no $k < j$ such that $d_k = d$. Accordingly, we define $\mathit{last}(d)$ to be the last position where d occurs. We define the symbolic normal form $\mathit{snf}(w) \stackrel{\text{def}}{=} (a_1,\pi_1)\dots(a_n,\pi_n) \in (A \times \Omega)^*$ of w inductively, along with sets $\mathit{Free}(i) \subseteq \mathbb{N}$ indicating the registers that are reusable after executing position $i \in [n]$. Setting $\mathit{Free}(0) = \mathbb{N}_{>0}$, we define

$$\pi_i = \begin{cases} \min(\mathit{Free}(i-1))^{\circledast} & \text{if } i = \mathit{first}(d_i) \\ \mathsf{reg}(\pi_{\mathit{first}(d_i)})^{\uparrow} & \text{otherwise}\,, \end{cases}$$

and

$$\mathit{Free}(i) = \begin{cases} \mathit{Free}(i-1) \setminus \min(\mathit{Free}(i-1)) & \text{if } i = \mathit{first}(d_i) \neq \mathit{last}(d_i) \\ \mathit{Free}(i-1) \cup \{\mathsf{reg}(\pi_i)\} & \text{if } i = \mathit{last}(d_i) \\ \mathit{Free}(i-1) & \text{otherwise}\,. \end{cases}$$

We canonically extend snf to data languages L, setting $\mathit{snf}(L) = \{\mathit{snf}(w) \mid w \in L\}$.

Example 4.6. Let $w = (a,8)(b,4)(a,8)(c,3)(a,4)(b,3)(a,9)$. Then, we have that $\mathit{snf}(w) = (a,1^{\circledast})(b,2^{\circledast})(a,1^{\uparrow})(c,1^{\circledast})(a,2^{\uparrow})(b,1^{\uparrow})(a,1^{\circledast})$. ♢

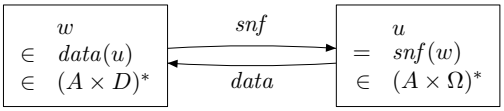


Figure 4.4: Relation between data words and symbolic normal forms

The relation between the mappings data and snf is illustrated in Figure 4.4. One easily verifies that $L = \mathit{data}(\mathit{snf}(L))$, for all data languages L. Therefore, equality of data languages reduces to equality of their symbolic normal forms:

Lemma 4.7. Let L and L' be data languages. Then, $L = L'$ iff $\mathit{snf}(L) = \mathit{snf}(L')$.

Of course, symbolic normal forms may use any number of registers so that the set of symbolic normal forms is a language over an infinite alphabet as well. However, given an SA $\mathcal{A}$, the symbolic normal forms that represent the language $L(\mathcal{A})$ do with a bounded (i.e., finite) number of registers. Indeed, an important notion in the context of SAs is the *bound* of a data word. Intuitively, the bound of $w = (a_1, d_1) \dots (a_n, d_n) \in (A \times D)^*$ is the minimal number of registers that an SA needs to execute w. Or, in other words, the bound is the maximal number of overlapping *scopes*, where a scope is an interval delimiting the occurrences of one particular data value. Formally, a scope of w is any set $I \subseteq \{1, \dots, n\}$ such that there is $d \in \{d_1, \dots, d_n\}$ with $I = \{\mathit{first}(d), \dots, \mathit{last}(d)\}$. Given a natural number $k \geq 1$, we say that w is *k-bounded* if every position $i \in \{1, \dots, n\}$ is contained in at most k scopes. One can verify that a data word is k-bounded iff $\mathit{snf}(w)$ is a word over the alphabet $A \times \Omega_k$ where $\Omega_k \stackrel{\text{def}}{=} \{1, \dots, k\}^{\circledast} \cup \{1, \dots, k\}^{\uparrow}$.

Let DW_k denote the set of B-bounded data words. A data language L is *k-bounded* if $L \subseteq \mathsf{DW}_k$. Note that the set of all data words is not k-bounded, for any k.

Theorem 4.8 ([BHLM14]). *Let $\mathcal{A} = (S, \mathcal{R}, \iota, F, \Delta)$ be an SA with $\mathcal{R} = \{1, \dots, k\}$. Then, $L(\mathcal{A})$ is k-bounded. Moreover, $\mathit{snf}(L(\mathcal{A}))$ is a regular language over the finite alphabet $A \times \Omega_k$. A corresponding automaton can be effectively computed. Its number of states is at most exponential in k and linear in $|S|$.*

Using Theorem 4.8, we obtain the following closure properties of SA:

Theorem 4.9 ([BHLM14]). *We have the following closure properties:*

- *SAs are closed under union and intersection.*
- *SAs are complementable for k-bounded data words: Given an SA $\mathcal{A}$ with k registers, there is an SA $\mathcal{B}$ with k registers such that $L(\mathcal{B}) = \mathsf{DW}_k \setminus L(\mathcal{A})$.*

Note that, while FRA are not complementable wrt. the set of all data words, they are complementable for k-bounded data words. The reason is that, given an FRA $\mathcal{A}$, one can construct an SA $\mathcal{B}$ such that $L(\mathcal{B}) = L(\mathcal{A}) \cap \mathsf{DW}_k$.

4.3 Class Register Automata

Example 4.4 on page 62 revealed a certain weakness of FRAs concerning their expressiveness and, therefore, modeling power. In this section, we will present an automata model that captures the language $L_3 \cap \ldots \cap L_6$ in question. Despite its expressive power (which entails undecidability of its emptiness problem), it

has a natural mode of operation, taking into account the local control flow of a process. Moreover, it captures previously defined automata (as well as others from the literature) as special cases. Our model is named *class register automata*, since it combines register automata and class memory automata [BS10]. Though basic decision problems are undecidable for that model, we will later see that its expressive power lies between EMSO and MSO logic so that it is still, in a sense "regular".

Similarly to nested words and message sequence charts (cf. Chapters 2 and 3), our new automata model will access a graph structure that we add on top of a data word. Actually, we associate with a data word $w = (a_1, d_1) \dots (a_n, d_n) \in (A \times D)^*$ two relations, $\lhd_{+1} \subseteq [n] \times [n]$ and $\lhd_{\oplus 1} \subseteq [n] \times [n]$. We let $\lhd_{+1} = \{(i, i+1) \mid i \in [n-1]\}$. Moreover, for all $i, j \in [n]$, we let $i \lhd_{\oplus 1} j$ iff $i < j$, $d_i = d_j$, and there is no $k \in \{i+1, \dots, j-1\}$ such that $d_i = d_k$. Thus, $i \lhd_{\oplus 1} j$ denotes the fact that i and j are *successive* positions carrying the same data value.

Example 4.10. In Figure 4.5, we illustrate the two relations associated with a data word w over $\Sigma = \{a, b\}$ and $D = \mathbb{N}$. Straight edges represent the relation $\lhd_{+1}$, while curved edges represent $\lhd_{\oplus 1}$. ♢

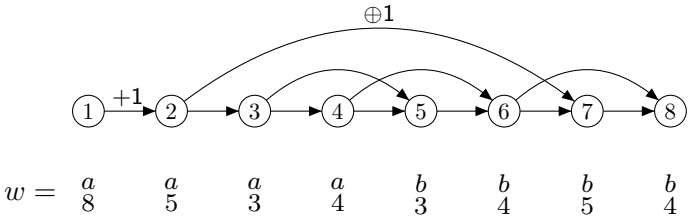


Figure 4.5: The relations $\lhd_{+1}$ and $\lhd_{\oplus 1}$

A run of a class register automaton on a data word $w = (a_1, d_1) \dots (a_n, d_n)$ proceeds as follows. Like an FRA, it comes with a finite set of registers, say $\mathcal{R}$. Like in an FRA, data values may be written into these registers and be reused later. The main difference is that, when reading a position j in the data word, the automaton can access the contents of some register r both

- at position i with $i \lhd_{+1} j$ (denoted r^-), and
- at position i with $i \lhd_{\oplus 1} j$ (denoted $r^{\ominus}$).

This looks all the more reasonable, since $\lhd_{\oplus 1}$ reflects the control flow of one single process. Moreover, it is in the spirit of nested-word automata (cf. Chapter 2). So, let $\mathcal{R}^- \stackrel{\text{def}}{=} \{r^- \mid r \in \mathcal{R}\}$ and $\mathcal{R}^{\ominus} \stackrel{\text{def}}{=} \{r^{\ominus} \mid r \in \mathcal{R}\}$. Actually, a transition will be constrained by a *guard*, which is a boolean combination of atoms $\theta_1 = \theta_2$ where $\theta_1, \theta_2 \in \{\mathbb{c}\} \cup \mathcal{R}^- \cup \mathcal{R}^{\ominus}$. Here, $\mathbb{c}$ stands for "current data value". The set of guards (over $\mathcal{R}$) is denoted by $\mathit{Guard}(\mathcal{R})$. When executing a transition, registers may obtain new data values: either the data value currently read, or one of the data values previously stored in registers, or a "guessed" data value: any

data value that is in the *ℓ-neighborhood* (as defined below) of the given position, where $\ell \in \mathbb{N}$ is a radius. This is specified by an *update*, which is a partial function $\nu : \mathcal{R} \rightharpoonup \mathcal{R}^- \cup \mathcal{R}^\ominus \cup \mathbb{N}$. The set of updates is denoted by $Upd(\mathcal{R})$.

Definition 4.11 (CRA [Bol11]). *A* class register automaton (CRA) *over A and D is a tuple $\mathcal{A} = (S, \mathcal{R}, \Delta, \iota, F, G)$ where*

- *S is the nonempty finite set of* states*,*
- *$\mathcal{R}$ is the nonempty finite set of* registers*,*
- *$\iota \in S$ is the* initial state*,*
- *$F \subseteq S$ is the set of* local final states*,*
- *$G \subseteq S$ is the set of* global final states*, and*
- *$\Delta \subseteq 2^S \times S \times Guard(\mathcal{R}) \times A \times Upd(\mathcal{R}) \times S$ is the finite set of* transitions*.* $\diamondsuit$

We call $\mathcal{A}$ *non-guessing* if every update is a partial function $\nu : \mathcal{R} \rightharpoonup \mathcal{R}^- \cup \mathcal{R}^\ominus \cup \{0\}$. This means that each "guess" is restricted to the current data value.

Again, we define the semantics of $\mathcal{A}$ via the notion of a *configuration*. Here, a configuration is a pair $\gamma = (s, \tau)$ where $s \in S$ is the current state and $\tau : \mathcal{R} \rightharpoonup D$ is a partial mapping denoting the current register assignment. We do not need to keep track of the set of used data values, since a run will be defined as a *mapping* of word positions to configurations. As before, configuration γ is *final* if $s \in F$. It is *initial* if $s = \iota$ and τ is undefined everywhere.

Let $(a_1, d_1) \dots (a_n, d_n) \in (A \times D)^*$ be a data word. For $i \in [n]$ and $\ell \in \mathbb{N}$, let $\mathfrak{D}_\ell(i)$ denote the set $\{d_j \mid j \in [n]$ has distance at most ℓ from i in the (undirected) graph $([n], \lhd_{+1} \cup \lhd_{\oplus 1} \cup \lhd_{+1}^{-1} \cup \lhd_{\oplus 1}^{-1})$ of those data values that are in the *ℓ-neighborhood* of i in w. A *run* of $\mathcal{A}$ is a sequence of configurations $\gamma_0 \gamma_1 \dots \gamma_n$ satisfying some properties. First, γ_0 has to be initial. Suppose $\gamma_i = (s_i, \tau_i)$. Then, we require that, for all $i \in [n]$, there are a guard $\Phi \in Guard(\mathcal{R})$, an update $\nu \in Upd(\mathcal{R})$, and $T \subseteq S$ such that

- $(T, s_{i-1}, \Phi, a_i, \nu, s_i) \in \Delta$,
- if there is no $j \in [n]$ such that $j \lhd_{\oplus 1} i$, then we have both $T = \emptyset$ and $\nu(r) \notin \mathcal{R}^\ominus$ for all $r \in \mathcal{R}$,
- $j \lhd_{\oplus 1} i$ implies $s_j \in T$, for all $j \in [n]$ (note that there is at most one such j),
- guard Φ is evaluated to true on the basis of its atoms: $\theta_1 = \theta_2$ is true iff $val_i(\theta_1) = val_i(\theta_2) \in D$ where
  - $val_i(\mathbb{c}) = d_i$,
  - $val_i(r^-) = \tau_{i-1}(r)$ (i.e., if $\tau_{i-1}(r)$ is undefined, then so is $val_i(r^-)$), and

– $val_i(r^{\ominus}) = \begin{cases} \tau_j(r) & \text{if } j \lhd_{\oplus 1} i \\ \text{undefined} & \text{if there is no } j \text{ such that } j \lhd_{\oplus 1} i \end{cases}$

(note that, in the latter two cases, the valuation may be undefined and, therefore, not be in D so that $\theta_1 = \theta_2$ is not satisfied),

- for all $r \in \mathcal{R}$, we have $\begin{cases} \tau_i(r) = \tau_{i-1}(\hat{r}) & \text{if } \nu(r) = \hat{r}^- \\ \tau_i(r) = \tau_j(\hat{r}) & \text{if } j \lhd_{\oplus 1} i \text{ and } \nu(r) = \hat{r}^{\ominus} \\ \tau_i(r) \in \mathfrak{D}_\ell(i) & \text{if } \nu(r) = \ell \in \mathbb{N} \\ \tau_i(r) \text{ undefined} & \text{if } \nu(r) \text{ undefined.} \end{cases}$

The run is *accepting* if $s_n \in G$ and, for all $i \in [n]$ such that i is $\lhd_{\oplus 1}$-maximal, $s_i \in F$. As usual, the set of data words that admit an accepting run of $\mathcal{A}$ is denoted by $L(\mathcal{A})$. Clearly, $L(\mathcal{A})$ is a data language.

Example 4.12. Recall from Example 4.4 on page 62 that there is no FRA recognizing the language defined as the intersection of the following data languages over $A = \{\mathsf{req}, \mathsf{ack}\}$ and $D = \mathbb{N}$:

- $L_3 =$ "every acknowledgment is preceded by a request"
- $L_4 =$ "requests are acknowledged in the order they are received"
- $L_5 =$ "every process makes at most one request"
- $L_6 =$ "all requests take place before all acknowledgments"

Formally, the language in question is

$$L = \{w \in (A \times D)^* \mid w \approx (\mathsf{req}, 1) \dots (\mathsf{req}, n)(\mathsf{ack}, 1) \dots (\mathsf{ack}, n) \text{ for some } n \in \mathbb{N}\}.$$

For example, $(\mathsf{req}, 8)(\mathsf{req}, 5)(\mathsf{req}, 3)(\mathsf{req}, 4)(\mathsf{ack}, 8)(\mathsf{ack}, 5)(\mathsf{ack}, 3)(\mathsf{ack}, 4) \in L$.

The (non-guessing) CRA $\mathcal{A}_{4.6}$ from Figure 4.6 recognizes L, i.e., $L(\mathcal{A}_{4.6}) = L$. We have $F = \{s_2\}$ and $G = \{s_0, s_2\}$, i.e., s_2 is the only local final state (indicated by the outgoing arrow), and s_0 and s_2 are the global final states. The updates are written in the form of an assignment. In a run, state s_1 is taken after executing requests, while s_2 is assigned to acknowledgment positions. Moreover, the CRA has two registers, r_1 and r_2. Suppose it reads $(\mathsf{req}, d_1) \dots (\mathsf{req}, d_n)(\mathsf{ack}, d_{n+1}) \dots (\mathsf{ack}, d_{2n})$. In the request phase, the current data values are respectively stored in register r_1. Moreover, r_2 takes the data values at the $\lhd_{+1}$-predecessor position. When entering the second phase (going from s_1 to s_2), the guard $r_2^{\ominus} \neq r_2^{\ominus}$ will ensure that position $n+1$ is the unique $\lhd_{\oplus 1}$-predecessor position where r_2 is undefined. Thus, $d_1 = d_{n+1}$. Then, the loop in s_2 makes sure that every position $n+i$ with $i \geq 2$ matches the request position where the contents of r_2 equals that of r_1 at position $n+i-1$. Since s_2 is the only local final state, every request has to be followed by a matching acknowledgment. Altogether, we have $d_1 \dots d_n = d_{n+1} \dots d_{2n}$. ♢

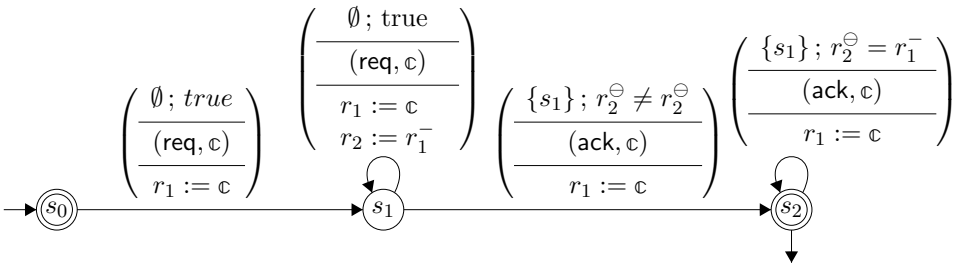


Figure 4.6: (Non-guessing) class register automaton $\mathcal{A}_{4.6}$

Note that some interesting automata models over data words can be identified as a special case of CRAs.

- A *class memory automaton (CMA)* [BS10] is a CRA $\mathcal{A} = (S, \mathcal{R}, \Delta, \iota, F, G)$ such that, for all transitions $(T, s, \Phi, a, \nu, s') \in \Delta$, we have that $\Phi = \text{true}$ and ν is undefined everywhere. In other words, registers are not used at all, while states of $\lhd_{\oplus 1}$ can still be accessed.
- A *quasi-FRA* is a non-guessing CRA $\mathcal{A} = (S, \mathcal{R}, \Delta, \iota, F, G)$ such that $F = S$ and, for all transitions $(T, s, \Phi, a, \nu, s') \in \Delta$,

  1. $T = S$ or $T = \emptyset$,
  2. if $T = S$, then $(\emptyset, s, \Phi, a, \nu, s') \in \Delta$,
  3. for all atoms $\theta_1 = \theta_2$ occurring in Φ, we have $\theta_1, \theta_2 \in \{\mathbb{c}\} \cup \mathcal{R}^-$, and
  4. for all registers $r \in \mathcal{R}$, $\nu(r)$ is undefined or contained in $\mathcal{R}^- \cup \{0\}$.

  Condition 1. implies that the concrete state at a $\lhd_{\oplus 1}$-predecessor cannot be accessed anymore. However, one may specify that a data value is globally fresh ($T = \emptyset$). By Condition 2., one cannot say that a data value is "non-fresh". Conditions 3. and 4. prevent the automaton from accessing registers at $\lhd_{\oplus 1}$-predecessors.
- A *quasi-RA* is a quasi-FRA $\mathcal{A} = (S, \mathcal{R}, \Delta, \iota, F, G)$ such that, for all transitions $(T, s, \Phi, a, \nu, s') \in \Delta$, both $(\emptyset, s, \Phi, a, \nu, s')$ and (S, s, Φ, a, ν, s') are contained in Δ. In other words, it cannot be enforced anymore that a data value is globally fresh.

Actually, many variants of *register automata* have been defined in the literature. In the configuration of an RA, a register assignment is always an injective mapping, which is relaxed in quasi-RAs. As a consequence, the latter use *guards* to compare data values. Quasi-RAs are similar to the model presented in [CHJ+11] in the context of automata learning. That model also uses guards to compare register contents. Transitions in the model from [Seg06], on the other hand, are guarded by a subset R of registers. The meaning is that R is exactly the set of registers that contain the data value that is currently read. Though syntactically different, those models are expressively equivalent, and the same holds for quasi-FRAs and

FRAs. Note that, however, the complexity of decision problems do not carry over. For example, nonemptiness for quasi-RAs is PSPACE-complete, while it is NP-complete for RAs (cf. Section 4.5). The expressive power of the various automata models over data words (and logics as presented in the next section) is illustrated in Figure 4.7. In particular, it was shown in [BS10] that CMAs are strictly more expressive than RAs. The fact that CMAs are strictly weaker than CRAs is witnessed by the language from Example 4.12. In fact, a simple pumping argument shows that this language cannot be recognized by a CMA.

In the next section, we will establish that the expressive power of CRAs is between EMSO and MSO logic (with a predicate for comparing data values for equality), solving the realizability problem for EMSO logic. Note that, in the article [Bol11], we introduced CRAs in a more general framework, allowing some flexibility in the choice of the signature (rather than only $\lhd_{+1}$ and $\lhd_{\oplus 1}$). This allowed us to treat dynamic communicating automata, which will be introduced in Chapter 5, as a special case.

4.4 Automata vs. Logic over Data Words

We will now consider MSO logic over data words. It is quite natural to extend classical MSO logic over words with a binary predicate $x \sim y$ to express that the data values at positions x and y coincide. Alternatively, one may include a predicate $x \lhd_{\oplus 1} y$ relating two *successive* positions with the same data values. In MSO logic, one will be expressible in terms of the other.

Definition 4.13. *The set* $\mathrm{dMSO}(A, D)$ *of* data MSO formulas *(over A and D) is built from the following grammar:*

$$\varphi \quad ::= \quad a(x) \;\mid\; x \lhd_{+1} y \;\mid\; x \lhd_{\oplus 1} y \;\mid\; x = y \;\mid\; x \in X \;\mid\; \neg\varphi \;\mid\; \varphi_1 \vee \varphi_2 \;\mid\; \exists x.\varphi \;\mid\; \exists X.\varphi$$

where $a \in A$, x, y are first-order variables, and X is a second-order variable. ♢

The semantics of $\mathrm{dMSO}(A, D)$ is as expected. Let us only pick some cases: for a data word $w = (a_1, d_1) \dots (a_n, d_n) \in (A \times D)^*$ and positions $i, j \in [n]$, we write

- $w, i \models a(x)$ if $a_i = a$,
- $w, i, j \models x \lhd_{+1} y$ if $i + 1 = j$, and
- $w, i, j \models x \lhd_{\oplus 1} y$ if $d_i = d_j$ and, for all $k \in \{i+1, \dots, j-1\}$, we have $d_i \neq d_k$.

For a sentence $\varphi \in \mathrm{dMSO}(A, D)$, we let $L(\varphi)$ denote the set of data words w such that $w \models \varphi$. Like for automata, $L(\varphi)$ is a data language.

We will often write dMSO instead of $\mathrm{dMSO}(A, D)$, since we fixed A and D.

The predicate $x \sim y$ is defined as an abbreviation for $(x \lhd^*_{\oplus 1} y) \vee (y \lhd^*_{\oplus 1} x)$, where, in turn, $\lhd^*_{\oplus 1}$ is the dMSO-definable reflexive transitive closure of $\lhd_{\oplus 1}$.

The fragments dFO and dEMSO are defined as expected. Moreover, dEMSO_2 will denote the fragment of dEMSO that makes use of only two first-order variables. Again, we may explicitly mention all the binary predicates (apart from $x = y$ and $x \in X$) that we allow. For example, the logic $\mathrm{dEMSO}(\lhd_{+1}, \lhd_{\oplus 1}, \sim)$ includes the binary predicate $\sim$, which is per se not expressible in $\mathrm{dEMSO} = \mathrm{dEMSO}(\lhd_{+1}, \lhd_{\oplus 1})$.

Example 4.14. We give a dFO formula φ over $A = \{\mathsf{req}, \mathsf{ack}\}$ and $D = \mathbb{N}$ such that $L(\varphi)$ equals $\{w \in (A \times D)^* \mid w \approx (\mathsf{req}, 1) \dots (\mathsf{req}, n)(\mathsf{ack}, 1) \dots (\mathsf{ack}, n)$ for some $n \in \mathbb{N}\}$ from Example 4.12. Recall that there is a CRA recognizing $L(\varphi)$, but no CMA and no FRA can accept $L(\varphi)$. We define φ as the conjunction $\varphi_1 \wedge \dots \wedge \varphi_4$ where

$$\varphi_1 = \forall x.\exists y.(\mathsf{req}(x) \to \mathsf{ack}(y) \wedge x \lhd_{\oplus 1} y)$$

says that every request is acknowledged (before the same process sends another request),

$$\varphi_2 = \forall x.\exists y.(\mathsf{ack}(x) \to \mathsf{req}(y) \wedge y \lhd_{\oplus 1} x)$$

says that every acknowledgment has a corresponding request,

$$\varphi_3 = \neg\exists x.\exists y.(\mathsf{ack}(x) \wedge \mathsf{req}(y) \wedge x \lhd_{+1} y)$$

says that all requests precede all acknowledgments, and

$$\varphi_4 = \forall x.\forall y.\left(\begin{array}{c}\mathsf{req}(x) \wedge \mathsf{req}(y) \wedge x \lhd_{+1} y \\ \to \exists x'.\exists y'.(\ \mathsf{ack}(x') \wedge \mathsf{ack}(y') \wedge x \lhd_{\oplus 1} x' \lhd_{+1} y' \wedge y \lhd_{\oplus 1} y'\ )\end{array}\right)$$

guarantees that two (successive) requests are acknowledged in the order they were received. ♢

There have been some logical characterizations of automata over data words. A prominent one considers *data automata* [BDM+11], which were shown to be expressively equivalent to CMAs in [BS10].

Theorem 4.15 ([BDM+11]). *CMAs are expressively equivalent to the data logic* $\mathrm{dEMSO}_2(\lhd_{+1}, \lhd_{\oplus 1}, \sim, <)$.

A logical characterization of RAs has been obtained in [CLP11] via a semantical restriction:

Definition 4.16 ([CLP11]). *The set* rgMSO *of* rigid guarded MSO *formulas contains any* dMSO *sentence of the form* $\exists X_1 \dots \exists X_m.\varphi$ *where* φ *is built from the grammar*

$$\begin{array}{rcl}\varphi & ::= & a(x) \mid \alpha(x, y, X_1, \dots, X_m) \wedge x \sim y \mid x = y \mid x \in X \mid \\ & & \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \exists x.\varphi \mid \exists X.\varphi\end{array}$$

with α *being generated by the same grammar (without quantification over the variables* $X_1, \dots, X_m$*) such that* $\min(x, y)$ *is uniquely determined from both* $\max(x, y)$ *and* $X_1, \dots, X_n$ *(in the expected manner).* ♢

Theorem 4.17 ([CLP11]). *RAs are expressively equivalent to the logic* rgMSO.

Further automata models and logics have been considered in the literature. For example, a logical characterization has been established for *weak data automata*, which are a semantical restriction of data automata and expressively equivalent to the logic $\mathrm{dEMSO}_2(\lhd_{+1}, \sim)$ [KST12]. It seems that a logical characterization for FRAs has not been considered so far.

Now, we turn to SAs and CRAs. We first identify a fragment of $\mathrm{dMSO}(A, D)$ that is expressively equivalent to SAs.

Definition 4.18. *A* session MSO (sMSO) *formula is a* dMSO *sentence of the form*

$$\exists X_1 \ldots \exists X_m.(\alpha \wedge \forall x.\forall y.(x \sim y \;\leftrightarrow\; \beta))$$

where α and β are from $\mathrm{dMSO}(\lhd_{+1})$, i.e., they are classical MSO formulas containing neither $\lhd_{\oplus 1}$ nor $\sim$. ♢

Example 4.19. The sentence

$$\varphi_1 = \forall x.\forall y.(x \sim y \leftrightarrow x = y)$$

is an sMSO formula. Its semantics $L(\varphi_1)$ is the set of data words in which every data value occurs at most once. Moreover,

$$\varphi_2 = \forall x.\forall y.(x \sim y \leftrightarrow \mathrm{true})$$

is an sMSO formula, and $L(\varphi_2)$ is the set of data words where all data values coincide. As a last example, let

$$\varphi_3 = \exists X.\forall x.\forall y.(x \sim y \leftrightarrow (\neg\exists z.z \in X \wedge (x < z \leq y \vee y < z \leq x))).$$

Then, $L(\varphi_3)$ is the set of 1-bounded data words. Intuitively, the variable X represents the set of word positions where a fresh data value is introduced. ♢

Theorem 4.20 ([BHLM14]). *Let $L \subseteq (A \times D)^*$. The following are equivalent:*

- *There is an SA $\mathcal{A}$ such that $L(\mathcal{A}) = L$.*
- *There is a sentence $\varphi \in$ sMSO such that $L(\varphi) = L$.*

Proof (sketch). The construction of a formula from an SA follows the classical scheme: by means of existential second-order quantifiers, we guess an assignment of transitions to word positions. In the first-order part α, it is then verified if this assignment corresponds to an accepting run. Moreover, formula β checks if data

equality in the data word at hand corresponds to the symbolic word produced by the automaton.

For the other direction, the crucial observation is that every sMSO formula defines a k-bounded data language where we can choose k to be the number of states of a deterministic finite (word) automaton that recognizes the models of β (where free variables are handled by an extension of the alphabet A). ∎

Thanks to the closure of SA under complementation for bounded data words, we can establish another logical characterization in terms of *full* dMSO logic. However, we have to restrict in advance to the class of k-bounded data words, for some k. Recall that this is in the spirit of the logical characterizations in Chapters 2 and 3 where, similarly, one has to constrain the domain of nested words and MSCs to obtain logical characterizations in terms of MSO logic.

Theorem 4.21. *Let $k \geq 1$ and $L \subseteq \mathsf{DW}_k$. The following statements are (effectively) equivalent:*

- *There is an SA $\mathcal{A}$ such that $L(\mathcal{A}) = L$.*
- *There is a sentence $\varphi \in$ dMSO such that $L(\varphi) = L$.*

This implies that, considered over k-bounded data words, the formalisms dMSO logic, CRA, FRA, and SA all have the same expressive power, while RAs are strictly weaker. Next, we turn to the relation between logic and CRAs. Recall that, basic decision problems being undecidable, we were interested in the realizability problem. It is solved positively for dEMSO specifications:

Theorem 4.22 ([Bol11]). *For every sentence $\varphi \in$ dEMSO, there is a CRA $\mathcal{A}$ such that $L(\mathcal{A}) = L(\varphi)$.*

Proof (sketch). We adopt the technique from [BL06] (Theorem 3.13 on page 48). The idea is to construct an automaton that computes, in a data word, the sphere around a given event. The theorem then follows by Hanf's normal form. Using registers, the technique from [BL06] can indeed be adapted to deal with data words. Note that the guessing mode is necessary to be able to anticipate neighborhoods before verifying them. ∎

Moreover, one can show that, despite their expressive power, CRAs recognize only data languages that are definable in dMSO:

Theorem 4.23 ([Bol11]). *For every CRA $\mathcal{A}$, there is a sentence $\varphi \in$ dMSO such that $L(\varphi) = L(\mathcal{A})$.*

Figure 4.7 summarizes and compares the expressive power of the various formalisms considered in this chapter. Here, $\longrightarrow$ means "strictly included" and $\dashrightarrow$ means "included" (strict inclusion being an open problem).

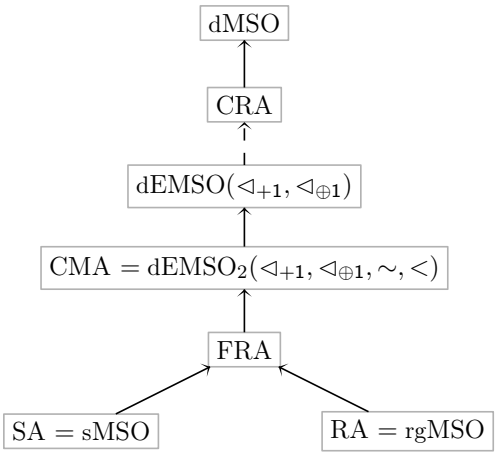


Figure 4.7: A hierarchy of automata and logics over data words

4.5 Decision Problems

Let us turn to decision problems, namely nonemptiness, inclusion, and model checking. For a class $\mathcal{C} \in \{$FRA,RA,SA,CRA,CMA$\}$ of automata over data words and a logic fragment $\mathcal{L}$, we consider the following problems[2]:

Problem 4.24. NONEMPTINESS($\mathcal{C}$):

INPUT:	Automaton $\mathcal{A}$ from $\mathcal{C}$
QUESTION:	$L(\mathcal{A}) \neq \emptyset$?

Problem 4.25. INCLUSION($\mathcal{C}$):

INPUT:	Automata $\mathcal{A}$ and $\mathcal{B}$ from $\mathcal{C}$
QUESTION:	$L(\mathcal{A}) \subseteq L(\mathcal{B})$?

Problem 4.26. MODELCHECKING($\mathcal{C}, \mathcal{L}$):

INPUT:	Automaton $\mathcal{A}$ from $\mathcal{C}$; $\varphi \in \mathcal{L}$
QUESTION:	$L(\mathcal{A}) \subseteq L(\varphi)$?

We start with results that had been known for FRAs and RAs:

Theorem 4.27 ([KF94, Tze11]). *The problem* NONEMPTINESS(FRA) *and the problem* NONEMPTINESS(RA) *are both* NP*-complete.*

[2]Recall that A and D are fixed. For the forthcoming results, it actually does not make a difference if A is part of the input or any fixed finite alphabet.

The NP upper bound directly carries over to SAs. The nonemptiness problem for SAs is actually NP-complete, too. The lower bound is shown by a reduction from 3-CNF-SAT (cf. [BCH+13]).

Theorem 4.28. NONEMPTINESS(SA) *is* NP*-complete.*

Note that the problem becomes considerably harder when we move up in the hierarchy of automata for data words.

Theorem 4.29 ([BDM+11]). NONEMPTINESS(CMA) *is decidable, and as hard as Petri-net reachability.*

As a corollary from Theorem 4.29 and the *effectiveness* of Theorem 4.15, one obtains that satisfiability of $\mathrm{dEMSO}_2(\lhd_{+1}, \lhd_{\oplus 1}, \sim, <)$ is decidable. It is easy to see that dropping the two-variable restriction renders the satisfiability problem undecidable. By Theorem 4.22, this means that nonemptiness for CRAs is undecidable. Thus, the gain in expressive power that CRAs provide comes at the expense of an undecidable nonemptiness problem:

Theorem 4.30. NONEMPTINESS(CRA) *is undecidable.*

Next, we consider the inclusion problem. Unlike the nonemptiness problem, it is already undecidable for the simple model of RAs:

Theorem 4.31 ([KF94, Tze11]). INCLUSION(RA) *is undecidable.*

Therefore, the inclusion problem is undecidable for any other stronger automata model such as FRAs and CMAs. However, we can establish decidability for SAs, which is a corollary from Theorem 4.8:

Theorem 4.32 ([BHLM14]). INCLUSION(SA) *is decidable.*

The situation in model checking is similar to that of inclusion when we choose the full dMSO logic as a specification. It is undecidable for RAs, which follows from undecidability of the satisfiability problem for dMSO.

Theorem 4.33. MODELCHECKING(RA,dMSO) *is undecidable.*

To obtain decidability, one therefore has to restrict the logic. From Theorems 4.29 and 4.15, it follows that the model-checking problem of CMAs against formulas from the logic $\mathrm{dFO}_2(\lhd_{+1}, \lhd_{\oplus 1}, \sim, <)$ is decidable.

Theorem 4.34 ([BDM+11]). ModelChecking(CMA,dFO$_2(\lhd_{+1}, \lhd_{\oplus 1}, \sim, <)$) *is decidable.*

For the full dMSO logic, model checking is decidable for SAs, which follows from Theorems 4.21 and 4.32:

Theorem 4.35 ([BCGNK12]). ModelChecking(SA,dMSO) *is decidable.*

Note that the model-checking problem was actually solved in [BCGNK12] for a more powerful model than SAs, including stacks.

4.6 Perspectives

Though we saw already a certain number of different automata models running on data words, our study is far from being exhaustive. There are actually many more interesting models such as alternating automata [DL09] or walking automata [MMP13]. The latter, however, are rather "recognizers" than "generators" of behaviors so that they cannot be seen as system models. In this chapter, we introduced the model of CRAs that captures, in a unifying framework, many of those models from the literature that can actually be seen as system models. This, however, comes at the price of undecidability of basic verification questions. Still, it would be interesting to identify subclasses of CRAs that extend other existing models but allow one to model realistic protocols such as leader-election algorithms. To this end, one may resort to symbolic techniques or to well-quasi orderings as a technique to establish decidability for infinite-state systems. The latter have actually been successfully applied in the realm of data words [Fig10].

CHAPTER 5

Dynamic Message-Passing Systems

In this chapter, we consider *dynamic* message-passing systems. In a sense, it combines Chapters 3 and 4, which considered *static* message-passing systems and, respectively, dynamic *sequential* systems. Processes will communicate, via message passing, in a dynamic environment, which allows them to create new processes during runtime. This framework is inspired by concurrent programming languages such as Erlang. More precisely, each process has a unique process identifier (pid). It also disposes of a finite number of registers, in which it can store pids of other processes. Now, a process can communicate with all processes that it knows, i.e., whose pids are present in its registers. The knowledge topology is subject to changes, though. When sending a message, a process may include, in the message, processes that it knows. In turn, the receiving process may overwrite some of its registers using the pids attached to the message. Moreover, a process can spawn a new process, which obtains a unique, fresh pid (cf. fresh-register automata from Chapter 4). The latter is henceforth stored in one of the registers of the spawning process, which can then propagate the new pid to other processes.

In the following, we first extend the notion of message sequence charts (MSCs, cf. Chapter 3) towards a dynamic version, and then introduce dynamic communicating automata (DCAs) as an extension of the static (fixed-topology) version from Chapter 3. While communicating automata accept MSCs, DCAs will accept *dynamic* MSCs. On the specification side, we introduce a conservative extension of the session automata from the previous chapter to handle dynamic MSCs. These *register message sequence graphs* (rMSGs) allow the specifier a global view of the system. In this chapter, we study mainly the realizability question. Given an rMSG, is there a DCA recognizing the same set of behaviors? Since our dynamic framework properly extends the static setting, these questions are undecidable. For this reason, we will consider *executability*, a non-trivial sufficient criterion for

implementability (essentially, realizability modulo message refinement), which we then show to be decidable. Essentially, an rMSG is executable if, in each scenario, a process p that sends a message to another process, say, with pid q, "knows" q. Here, knowledge refers to the possibility of passing, during the execution, the process identity of q to p. In a second step, we discover a restriction of rMSGs, for which executability and implementability coincide.

A first step towards MSCs over an evolving set of processes was made in [LMM02], where MSO model checking is shown decidable for *fork-and-join MSC grammars*. Our rMSGs are similar to these grammars, but take into account pids as message contents and distinguish messages and process creation. Moreover, (implementable) subclasses can be identified more easily. Nevertheless, several of our results apply to the formalism from [LMM02] once the latter is adjusted to our setting. In [BGP08], an MSC semantics was given for the π-calculus, but the problems studied there are very different from ours and do not distinguish between a specification and an implementation.

5.1 Dynamic Message Sequence Charts

We modify MSCs from Chapter 3 (Definition 3.3 on page 42) to deal with process creation. Let P be a nonempty (finite or infinite) set of processes. Later, P may be instantiated with a finite set of registers or the infinite set $\mathbb{P} \stackrel{\text{def}}{=} \mathbb{N}$ of pids. A process can perform send and receive actions, but it is also allowed to spawn new processes. Moreover, it may attach processes/pids to a message. Thus, we fix a finite ranked alphabet[1] A. For any set B, let $A\langle B\rangle \stackrel{\text{def}}{=} \{a(b_1,\ldots,b_n) \mid a \in A, n = \mathit{arity}(a), \text{ and } b_1,\ldots,b_n \in B\}$. Then, the set of *messages* is given as $A\langle P\rangle$.

The following definition is similar to Definition 3.3. However, since communication topologies are henceforth dynamic, an MSC does not include a topology but certain spawn events. We will first introduce partial MSCs, which feature an arbitrary number of "existing" processes (i.e., those that are not spawned). Those processes are analogous to free variables in logic formulas.

Definition 5.1 (partial MSC). *A* partial MSC *over A and P is a tuple $M = (E, \lhd, \pi, \mu)$ where*

- *E is a nonempty finite set of* events*,*
- *$\lhd = \lhd_{\mathsf{new}} \uplus \lhd_{\mathsf{proc}} \uplus \lhd_{\mathsf{msg}}$ is the acyclic edge relation,*
- *$\pi : E \to P$ maps each event to a process—for $p \in P$, we let $E_p \stackrel{\text{def}}{=} \{e \in E \mid \pi(e) = p\}$, and*
- *$\mu : \lhd_{\mathsf{msg}} \to A\langle P\rangle$ maps each message edge to a message.*

We require that the following hold:

[1] A *ranked alphabet* is an alphabet A that comes with a mapping $\mathit{arity} : A \to \mathbb{N}$.

1. $\lhd_{\mathsf{proc}}$ *is a union* $\bigcup_{p \in P} \lhd_p$ *where each* $\lhd_p \subseteq E_p \times E_p$ *is the direct-successor relation of some total order on* E_p*,*

2. *there is a partition* $E = E_! \uplus E_? \uplus E_{\mathsf{spawn}} \uplus E_{\mathsf{start}}$ *such that*

   - $\lhd_{\mathsf{msg}}$ *induces a bijection between* $E_!$ *and* $E_?$*,*
   - $\lhd_{\mathsf{new}}$ *induces a bijection between* E_{spawn} *and* E_{start}*,*
   - $\lhd_{\mathsf{msg}}$ *and* $\lhd_{\mathsf{new}}$ *are subsets of* $\bigcup_{p \neq q}(E_p \times E_q)$*,*
   - *there is no event* $e \in E_{\mathsf{start}}$ *that has a* $\lhd_{\mathsf{proc}}$*-predecessor, and*
   - *for all* $(e, f), (e', f') \in \lhd_{\mathsf{msg}}$ *such that* $\pi(e) = \pi(e')$ *and* $\pi(f) = \pi(f')$*, we have* $e \lhd^*_{\mathsf{proc}} e'$ *iff* $f \lhd^*_{\mathsf{proc}} f'$ *(FIFO).* ♢

   The set of partial MSCs over A *and* P *is denoted by* $\mathsf{pMSC}(A, P)$*.*

Note that the partition $E = E_! \uplus E_? \uplus E_{\mathsf{spawn}} \uplus E_{\mathsf{start}}$ is uniquely determined by the fact that $\lhd$ is a partition $\lhd_{\mathsf{new}} \uplus \lhd_{\mathsf{proc}} \uplus \lhd_{\mathsf{msg}}$. We let $\mathit{Pids}(M) \stackrel{\text{def}}{=} \{p \in P \mid E_p \neq \emptyset\}$. By $\mathit{Free}(M) \stackrel{\text{def}}{=} \{p \in \mathit{Pids}(M) \mid E_{\mathsf{start}} \cap E_p = \emptyset\}$, we denote the set of *free* processes of M. Intuitively, free processes of a partial MSC M have not been initiated in M. Moreover, $\mathit{Bnd}(M) \stackrel{\text{def}}{=} \mathit{Pids}(M) \setminus \mathit{Free}(M)$ denotes the set of *bound* processes.

For every $p \in \mathit{Pids}(M)$, there are a unique minimal and a unique maximal event in the total order $(E_p, \lhd^*_p)$, which we denote by $\min_p(M)$ and $\max_p(M)$, respectively. By $\mathit{MsgPar}(M)$, we denote the set of processes $p \in P$ that occur as a parameter in some message, i.e., such that there is $a(p_1, \ldots, p_n) \in \mu(\lhd_{\mathsf{msg}})$ with $p \in \{p_1, \ldots, p_n\}$.

Now, we can define (complete rather than partial) dynamic MSCs:

Definition 5.2 (MSC). *A* dynamic MSC *(or, simply, MSC) is a partial MSC* $M = (E, \lhd, \pi, \mu)$ *such that* $\mathit{Free}(M)$ *is a singleton set.* ♢

Note that the partial order $(E, \lhd^*)$ associated with M has a unique minimal event, which we denote by $\mathit{init}(M)$. The set of MSCs over A and P is denoted by $\mathsf{dMSC}(A, P)$.

The notion of free processes in a partial MSC is important in the context of automata, where we start with a fixed number of processes. Actually, we suppose that, at the beginning of an execution, we start with *one single* free process, though we could extend the framework to an arbitrary fixed number of initial processes.

Example 5.3. An MSC M over $A = \{a, b, c\}$ and $\mathbb{P}$ (recall that $\mathbb{P} = \mathbb{N}$) is shown in Figure 5.1. Here, $\mathit{arity}(a) = 1$ and $\mathit{arity}(b) = \mathit{arity}(c) = 0$. We have $\mathit{Free}(M) = \{8\}$ and $\mathit{Bnd}(M) = \{3, 4, 5\}$. ♢

As usual, we do not distinguish isomorphic structures. Moreover, none of the formalisms that we are going to introduce next will be able to distinguish between MSCs that differ only in pids. Given a partial MSC M, we denote by $[M]$ the set of partial MSCs that can be obtained from M by renaming of pids (in the expected manner—we omit the formal definition). This is extended to sets L, and we let $[L]$

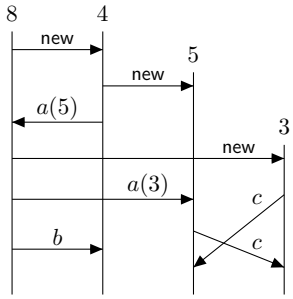


Figure 5.1: An MSC

denote the union of sets $[M]$ with M ranging over L. When $L = [L]$ holds, then we say that L is *closed*. Note that a closed set of MSCs is in the spirit of a data language (cf. Chapter 4), which is a "closed" set of data words.

5.2 Dynamic Communicating Automata

Next, we present our model of an implementation of a dynamic message-passing system. As mentioned before, we essentially deal with communicating automata, where processes execute send and receive actions (cf. Definition 3.9 on page 46). However, processes can henceforth spawn new processes and exchange pids along with messages.

5.2.1 The Model

Let us give the formal definition of dynamic communicating automata.

Definition 5.4 (DCA [BCH+13]). *A* dynamic communicating automaton (DCA) *over the ranked alphabet A (and $\mathbb{P}$) is a tuple $\mathcal{A} = (S, \mathcal{R}, \iota, F, \Delta)$ where*

- *S is the nonempty finite set of* states
- *$\iota \in S$ is the* initial state,
- *$F \subseteq S$ is the set of* final states,
- *$\mathcal{R}$ is the nonempty finite set of* registers, *and*
- *Δ is the set of* transitions.

A transition is of the form (s, α, s') where $s, s' \in S$, and α is an action, possibly

- *$r := \mathsf{spawn}(t, r')$ where $r, r' \in \mathcal{R}$ and $t \in S$,*
- *$r!a(r_1, \ldots, r_n)$ where $r \in \mathcal{R}$ and $a(r_1, \ldots, r_n) \in A\langle \mathcal{R} \uplus \{\mathsf{self}\}\rangle$, or*
- *$r?a(r_1, \ldots, r_n)$ where $r \in \mathcal{R} \uplus \{*\}$, and $a(r_1, \ldots, r_n) \in A\langle \mathcal{R} \uplus \{-\}\rangle$ such that $r_i = r_j \in \mathcal{R}$ implies $i = j$.*

As usual, we write a transition $(s, \alpha, s') \in \Delta$ *as* $s \xrightarrow{\alpha} s'$. ♢

The basic feature of a DCA is the transmission of pids via messages. When a process executes $r!a(r_1, \ldots, r_n)$, it sends a message to the process whose pid is stored in register r. The message consists of label a as well as $n = \mathit{arity}(a)$ many pids stored in registers $r_1, \ldots, r_n$ (or the sender's pid if $r_i = \mathsf{self}$). Executing $r?a(r_1, \ldots, r_n)$, a process receives a message from the process whose pid is stored in r (selective receive) or, in case $r = *$, from any process (non-selective receive). The message must be of the form $a(p_1, \ldots, p_n) \in A\langle\mathbb{P}\rangle$. In the resulting configuration, the receiving process updates its local registers $r_1, \ldots, r_n$ to $p_1, \ldots, p_n$, respectively, unless $r_i = -$, in which case p_i is neglected. Finally, a process executing action $r := \mathsf{spawn}(t, r')$ spawns a new process, whose fresh pid is henceforth stored in register r. The new process starts in state t. Its registers are a copy of the registers of the spawning process, except for r', which is set to the pid of the spawning process.

Example 5.5. An example DCA is depicted in Figure 5.2. It realizes a peer-to-peer protocol. Hereby, $A = \{\mathsf{ack}, \mathsf{com}\}$ with $\mathit{arity}(\mathsf{ack}) = 1$ and $\mathit{arity}(\mathsf{com}) = 0$. In that protocol, a spawn action rather plays the role of joining a host and making a request. The request is either forwarded to another host, or acknowledged (ack), in which case a connection between the user and the latest host is established so that they can henceforth communicate (com).

The first part of the DCA, comprising the states $s_0, \ldots, s_3$, is only executed by the very first, i.e., the requesting process. All other processes start in t_1. Going from s_0 to s_1, the initial process, say with pid 1, creates a new process, say with pid 2. Pid 2 is henceforth stored in register r, but this is actually irrelevant, since it will later be overwritten without being used. The new process, on the other hand, starts its execution in t_1. It creates/joins another process, say with pid 3. While process 3 stores the pid 2 in r', it inherits the contents of r, i.e., its register assignment will be $\{r \mapsto 1, r' \mapsto 2\}$. In this manner, the pid of the initial process is forwarded to any other process. When, at some point, we have $n \geq 3$ processes, the register assignment of process n is $\{r \mapsto 1, r' \mapsto n-1\}$ (for $n = 2$, we have $\{r \mapsto 1\}$). In particular, r holds the pid of the initial process, which had sent the initial request. Now, suppose that process n is capable of satisfying the request. It will then go from t_1 to t_2 and send an acknowledgment to the process whose pid is stored in r, namely the initial process. Along with ack, process n sends its own pid, indicated by self. Though 1 does not know n, it may receive that message, executing the *non-selective* receive associated with the transition from s_1 to s_2. As a result, register r of process 1 takes the value n. Now, both processes know each other (their respective pids are stored in register r) and can exchange messages of type com, using *selective* receives. ♢

5.2.2 Semantics

Let us describe the semantics of DCAs formally. Like for nested-word automata (Chapter 2) and communicating automata (Chapter 3), there are essentially two

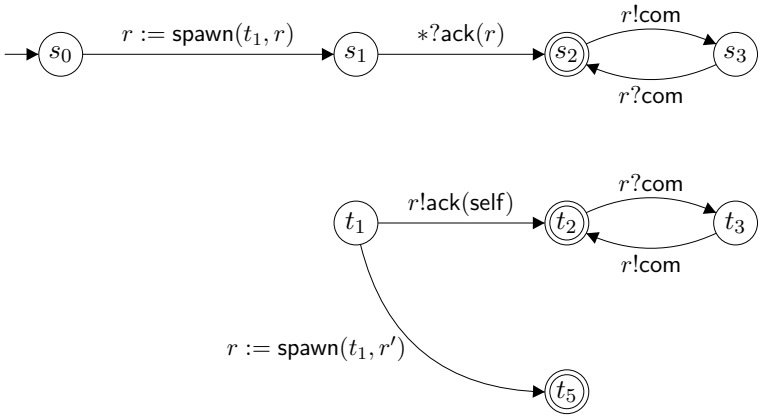


Figure 5.2: The DCA $\mathcal{A}_{5.2}$ modeling a peer-to-peer protocol

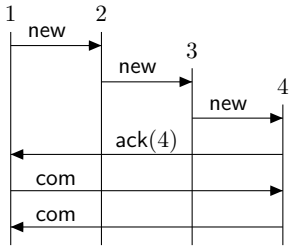


Figure 5.3: An MSC accepted by the DCA $\mathcal{A}_{5.2}$

ways of defining the semantics: one running directly on MSCs, and one accepting their linearizations. Again, we stick to the former.

Let $\tau : \mathcal{R} \rightharpoonup \mathbb{P}$ be a partial mapping, assigning identities to registers. For $p \in \mathbb{P}$ and $m = a(r_1, \ldots, r_n) \in A\langle \mathcal{R} \uplus \{\mathsf{self}\}\rangle$ such that $\{r_1, \ldots, r_n\} \subseteq \mathrm{dom}(\tau) \cup \{\mathsf{self}\}$, we let $m[\tau, p]$ denote the message $a(p_1, \ldots, p_n) \in Msg$ where

$$p_i = \begin{cases} \tau(r_i) & \text{if } r_i \in \mathcal{R} \\ p & \text{if } r_i = \mathsf{self}\,. \end{cases}$$

Intuitively, $m[\tau, p]$ is the message generated by a process with pid p and register contents τ, when executing an action of the form $r!a(r_1, \ldots, r_n)$.

Now, when a process with register contents $\tau : \mathcal{R} \rightharpoonup \mathbb{P}$ executes $r?a(r_1, \ldots, r_n)$ (recall that $a(r_1, \ldots, r_n) \in A\langle \mathcal{R} \uplus \{-\}\rangle$ such that $r_i = r_j \in \mathcal{R}$ implies $i = j$) and receives message $a(p_1, \ldots, p_n) \in A\langle \mathbb{P}\rangle$, then its new register contents will be $\tau[r_1, \ldots, r_n \mapsto p_1, \ldots, p_n]$, which is defined to be the partial mapping $\tau' : \mathcal{R} \rightharpoonup \mathbb{P}$ given by

$$\tau'(r) = \begin{cases} p_i & \text{if } r = r_i \\ \tau(r) & \text{if } r \notin \{r_1, \ldots, r_n\}\,. \end{cases}$$

Let $\mathcal{A} = (S, \mathcal{R}, \iota, F, \Delta)$ be a DCA and $M = (E, \lhd, \pi, \mu) \in \mathsf{dMSC}(A, \mathbb{P})$. To facilitate the definition of a run of $\mathcal{A}$ on M, we assume that E contains some additional

event $\mathbf{e}$, without particular labeling, such that $\mathbf{e} \lhd_{\mathsf{proc}} \mathit{init}(M)$. A *run* of $\mathcal{A}$ will be a pair (σ, τ), where $\sigma : E \to S$ and $\tau : E \to [\mathcal{R} \rightharpoonup \mathbb{P}]$. In the following, we will write σ_e and τ_e for $\sigma(e)$ and $\tau(e)$, respectively. Then, (σ, τ) is a run if it respects the following conditions:

1. $\sigma_{\mathbf{e}} = \iota$ and $\tau_{\mathbf{e}}$ is undefined everywhere.
2. For all $e_1, e_2, f \in E$ with $e_1 \lhd_{\mathsf{proc}} e_2 \lhd_{\mathsf{new}} f$, the relation Δ contains a transition
$$\sigma_{e_1} \xrightarrow{r := \mathsf{spawn}(s, r')} \sigma_{e_2}$$
such that $\sigma_f = s$, $\tau_{e_2} = \tau_{e_1}[r \mapsto \pi(f)]$, and $\tau_f = \tau_{e_1}[r' \mapsto \pi(e_1)]$.
3. For all $e_1, e_2, f \in E$ with $e_1 \lhd_{\mathsf{proc}} e_2 \lhd_{\mathsf{msg}} f$, we have $\tau_{e_1} = \tau_{e_2}$, and there is a transition
$$\sigma_{e_1} \xrightarrow{r!a(r_1,\ldots,r_n)} \sigma_{e_2}$$
such that $\{r, r_1, \ldots, r_n\} \subseteq \mathrm{dom}(\tau_{e_1}) \cup \{\mathsf{self}\}$, $\tau_{e_1}(r) = \pi(f)$, and $\mu(e_2, f) = a(r_1, \ldots, r_n)[\tau_{e_1}, \pi(e_2)]$.
4. For all $e_1, e_2, f \in E$ with $e_1 \lhd_{\mathsf{proc}} e_2$, $f \lhd_{\mathsf{msg}} e_2$, and $\mu(f, e_2) = a(p_1, \ldots, p_n)$, there is a transition
$$\sigma_{e_1} \xrightarrow{r?a(r_1,\ldots,r_n)} \sigma_{e_2}$$
such that $r = *$ or $\tau_{e_1}(r) = \pi(f)$, and $\tau_{e_2} = \tau_{e_1}[r_1, \ldots, r_n \mapsto p_1, \ldots, p_n]$.

The run (σ, τ) is accepting if $\{\sigma(\max_p(M)) \mid p \in \mathit{Pids}(M)\} \subseteq F$. By $L(\mathcal{A})$, we denote the set of MSCs M over A and $\mathbb{P}$ such that there is an accepting run of $\mathcal{A}$ on M. Note that $L(\mathcal{A})$ is closed, i.e., $L(\mathcal{A}) = [L(\mathcal{A})]$.

5.2.3 Realizability vs. Implementability

Like for communicating automata, there is a discrepancy between the languages that are recognized by DCAs and languages that one would actually consider to be implementable. More precisely, there are languages L that are not the language of a DCA, but for which there is a DCA *implementing* them up to some refinement. The refinement allows a DCA to attach more information to a message than the specification provides, for example additional pids. This is formalized as follows. Let A, B be finite ranked alphabets and let $h : B \to A$. We say that the pair (B, h) is a refinement of A if, for all $b \in B$, $\mathit{arity}(h(b)) \leq \mathit{arity}(b)$. We can extend h to a mapping $h : \mathsf{dMSC}(B, \mathbb{P}) \to \mathsf{dMSC}(A, \mathbb{P})$ as follows: for an MSC $M = (E, \lhd, \pi, \mu) \in \mathsf{dMSC}(B, \mathbb{P})$, we let $h(M) = (E, \lhd, \pi, \mu') \in \mathsf{dMSC}(A, \mathbb{P})$ where $\mu'(e, f) = h(b)(p_1, \ldots, p_{\mathit{arity}(h(b))})$ whenever $\mu(e, f) = b(p_1, \ldots, p_n)$. The mapping is then further extended to sets of MSCs as expected.

Definition 5.6 (realizability and implementability). *A set* $L \subseteq \mathsf{dMSC}(A, \mathbb{P})$ *is called*

- realizable *if* $[L] = L(\mathcal{A})$ *for some DCA* $\mathcal{A}$*,*
- implementable *if there are a refinement* (B, h) *of* A *and a DCA* $\mathcal{A}$ *over* B *and* $\mathbb{P}$ *such that* $[L] = h(L(\mathcal{A}))$*.* ♢

For both, realizability and implementability, it is necessary that the sender p of a message knows the receiver q at the time of sending, i.e., q should be stored in some register of p. Note that this aspect does not arise in simple communicating automata (be it with fixed or parameterized topology).

Example 5.7. Consider Figures 5.4 and 5.5. The MSC language $\{M_1\}$ is not implementable, as process 1 does not know 2 when sending message a. However, $\{M_2\}$ is implementable (and even realizable), as 2 may know 1: when spawning 2, process 0 can communicate the pid 1 to 2.

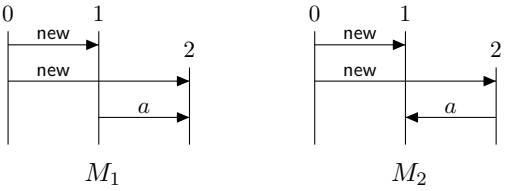


Figure 5.4: Implementability

The language $\{M_3\}$ is not realizable: as process 0 knows neither 2 nor 3 when it receives the messages, it has to use non-selective receives. But then, the DCA also accepts M_4. On the other hand, $\{M_3, M_4\}$ is realizable. However, $\{M_3\}$ and $\{M_4\}$ are both implementable, since we can attach additional information to the message contents a sent from 2 and 3 to 0, such as "1st/2nd message to be received", respectively. Alternatively, the first message may be refined to contain the pid of the sender of the second message so that, for the latter, 0 can use a selective receive. ♢

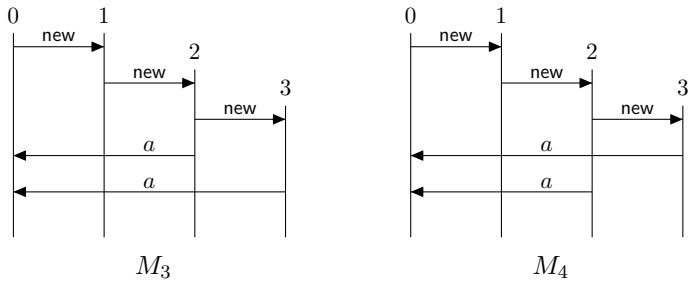


Figure 5.5: Realizability vs. Implementability

5.3 Register MSC Graphs

DCAs serve as a model of an implementation, reflecting low-level primitives—such as spawn, send, and receive—that are provided by concurrent programming languages. As such, DCAs are inherently hard to come up with or to analyze, all the more since basic decision problems are undecidable. In this section, we provide a specification formalism that allows for a high-level view of a system. We define *register message sequence graphs (rMSGs)*, which extend the well-studied message sequence graphs (MSGs) without process creation (see, for example, [GMP03, AEY05, HMK+05, GMSZ06, GKM06]). In a sense, MSGs are to finite automata what rMSGs are to session automata (cf. Chapter 4). Actually, rMSGs combine MSGs with session automata: the transition labels are MSCs whose processes are registers. Free processes are executed by the pids indicated by the corresponding registers, while bound processes obtain a fresh pids, which are henceforth stored in the allotted registers.

Like MSGs, rMSGs compose single behaviors towards bigger ones via concatenation. Let $M = (E, \lhd, \pi, \mu)$ and $M' = (E', \lhd', \pi', \mu')$ be partial MSCs over A and P. The *concatenation* $M \circ M'$ glues identical processes together. Thus, it is defined if $\mathit{Pids}(M) \cap \mathit{Bnd}(M') = \emptyset$. In that case, $M \circ M' \stackrel{\text{def}}{=} (E \uplus E', \hat{\lhd}, \pi \cup \pi', \mu \cup \mu')$ where

- $\hat{\lhd}_{\mathsf{proc}} = \lhd_{\mathsf{proc}} \cup \lhd'_{\mathsf{proc}} \cup \{(\max_p(M), \min_p(M')) \mid p \in \mathit{Pids}(M) \cap \mathit{Pids}(M')\}$,
- $\hat{\lhd}_{\mathsf{msg}} = \lhd_{\mathsf{msg}} \cup \lhd'_{\mathsf{msg}}$, and
- $\hat{\lhd}_{\mathsf{new}} = \lhd_{\mathsf{new}} \cup \lhd'_{\mathsf{new}}$.

Next, we define the new formalism to describe sets of MSCs. As already mentioned, it is analogous to session automata, but the transitions are labelled with partial MSCs.

Definition 5.8 (rMSG). *A* register message sequence graph (rMSG) *over A (and $\mathbb{P}$) is a tuple $\mathcal{H} = (\mathcal{L}, \mathcal{R}, \mathcal{L}_0, \mathcal{L}_{\mathrm{acc}}, r_0, T)$ where*

- *$\mathcal{L}$ is the nonempty finite set of* locations*,*
- *$\mathcal{L}_0 \subseteq \mathcal{L}$ is the set of* initial locations*,*
- *$\mathcal{L}_{\mathrm{acc}} \subseteq \mathcal{L}$ is the set of* accepting locations*,*
- *$\mathcal{R}$ is the nonempty finite set of* registers *with* initial register *$r_0 \in \mathcal{R}$, and*
- *T is the finite set of* transitions*.*

A transition is a triple $(\ell, M, \ell') \in \mathcal{L} \times \mathsf{pMSC}(A, \mathcal{R}) \times \mathcal{L}$, usually written $\ell \xrightarrow{M} \ell'$, such that $\mathit{MsgPar}(M) \cap \mathit{Bnd}(M) = \emptyset$ (which will guarantee an unambiguous interpretation of message parameters). ♢

Like in (fresh-)register automata, we associate MSCs with an rMSG through the notion of runs, which we will define after some preparation. Henceforth, a *register assignment* is an *injective* partial mapping $\tau : \mathcal{R} \rightharpoonup \mathbb{P}$ from the set of registers to the set of pids.

A configuration of $\mathcal{H}$ is a triple $\gamma = (\ell, \tau, U)$ where $\ell \in \mathcal{L}$ is the current location, $\tau : \mathcal{R} \rightharpoonup \mathbb{P}$ is a register assignment, and $U \subseteq \mathbb{P}$ is the set of pids that have been used so far. We say that γ is *initial* if $\ell \in \mathcal{L}_0$, $\tau = \{r_0 \mapsto p\}$, and $U = \{p\}$ for some $p \in \mathbb{P}$. It is *final* if $\ell \in \mathcal{L}_{\text{acc}}$. Via the global transition relation, we can switch from one configuration to another, while reading a partial MSC from the set $\mathsf{pMSC}(A, \mathbb{P})$. More precisely, we let $(\ell, \tau, U) \overset{M'}{\Longrightarrow} (\ell', \tau', U')$ if there are a transition $\ell \xrightarrow{M} \ell'$ and a bijection $\xi : Bnd(M) \to Bnd(M')$ such that the following hold:

1. $Free(M) \cup MsgPar(M) \subseteq \mathrm{dom}(\tau)$,
2. $\tau' = \tau[\xi]$ and $M' = \tau'(M)$, and
3. $U' = U \uplus Bnd(M')$.

Here, $\tau'(M)$ is defined as expected: every register occurrence $r \in \mathcal{R}$ in M is replaced by $\tau'(r) \in \mathbb{P}$. Condition 1. says that free processes can be instantiated. Condition 2. makes sure that registers remain unchanged unless they are overwritten for a new process. Finally, Condition 3. guarantees that bound processes obtain fresh pids.

A run of $\mathcal{H}$ is a sequence $\rho = \gamma_0 \overset{M_1}{\Longrightarrow} \gamma_1 \overset{M_2}{\Longrightarrow} \ldots \overset{M_n}{\Longrightarrow} \gamma_n$ with $n \geq 1$. It generates the MSC $M(\rho) \overset{\text{def}}{=} M_1 \circ \ldots \circ M_n$. Note that $M(\rho)$ is indeed well-defined and has exactly one free pid. We say that ρ is accepting if γ_n is final. Finally, $L(\mathcal{H}) \overset{\text{def}}{=} \{M(\rho) \mid \rho \text{ is an accepting run of } \mathcal{H}\}$ is the *language* of $\mathcal{H}$.

Example 5.9. The rMSG $\mathcal{H}_{5.6}$ from Figure 5.6 below gives a high-level view of the peer-to-peer protocol considered in Example 5.5 on page 83. Recall that $A = \{\mathsf{ack}, \mathsf{com}\}$ (acknowledgment, communication) with $arity(\mathsf{ack}) = 1$ and $arity(\mathsf{com}) = 0$. The initial register is r_0. A request is forwarded to new processes along with the pid p of the initial process. At some point, a process acknowledges the request, sending its own pid q to the initial process. Processes p and q may then communicate and exchange messages. A generated MSC is depicted in Figure 5.3 on page 84. Note that $L(\mathcal{H}_{5.2})$ is realizable, since we have $L(\mathcal{A}_{5.2}) = L(\mathcal{H}_{5.6})$, i.e., $L(\mathcal{H}_{5.6})$ is recognized by a DCA without any message refinement. ♢

5.4 Executability of Register Message Sequence Graphs

Like for classical MSGs, the language of an rMSG is not necessarily realizable, nor implementable. Even worse, undecidability of realizability and implementability carries over to the dynamic setting. More precisely, we consider the following decision problems:

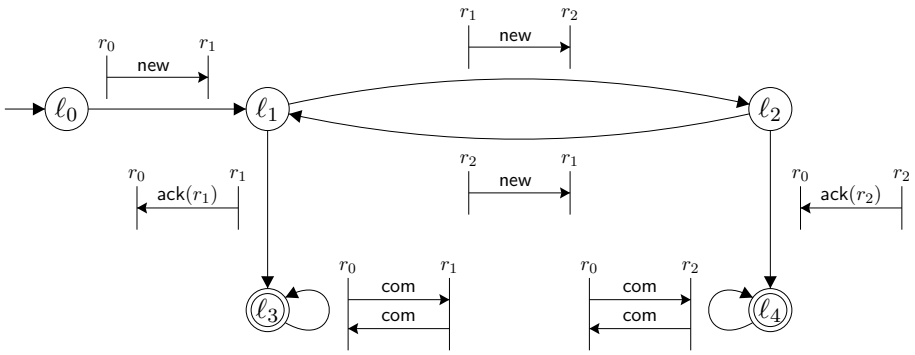


Figure 5.6: The rMSG $\mathcal{H}_{5.6}$ modeling the peer-to-peer protocol

Problem 5.10. rMSG-Realizability:

Input:	A; rMSG $\mathcal{H}$ over A
Problem:	Is $L(\mathcal{H})$ realizable?

Problem 5.11. rMSG-Implementability:

Input:	A; rMSG $\mathcal{H}$ over A
Problem:	Is $L(\mathcal{H})$ implementable?

Unfortunately, both problems are undecidable, which is a consequence of the fact that both problems are undecidable in the case of a fixed number of processes:

Theorem 5.12 ([AEY05, HMK+05]). *Both problems,* rMSG-Realizability *and* rMSG-Implementability*, are undecidable.*

In the following, we focus on the implementability question. To overcome undecidability, we introduce a criterion, called *executability*, that is sufficient for implementability but turns out to be decidable. As a second step, we then identify a fragment of rMSGs for which executability and implementability coincide so that implementability is actually decidable for that class.

Let $M = (E, \lhd, \pi, \mu) \in \mathsf{dMSC}(A, \mathbb{P})$ be an MSC. Intuitively, M is *executable* if a process p may know the pid of a process q at the time of (i) sending a message to q, or (ii) sending the pid q to some other process. Here, process p may know q if is it possible to pass q to p along the message flow. Let us be more precise. Given $q \in \mathit{Pids}(M)$ and an event $e \in E$ of M, we write $q \rightsquigarrow_M e$ if there is a path from the minimal event $\min_q(M)$ of q to e in M. This path might involve the reversal of the spawn edge that started q. That is, $q \rightsquigarrow_M e$ if $(\min_q(M), e) \in (\lhd \cup \lhd_{\mathsf{new}}^{-1})^*$. In a sense, $q \rightsquigarrow_M e$ indicates that the process executing e is aware of process q. Next, we formally define executability of MSCs.

Definition 5.13 (executability). *Let $M = (E, \lhd, \pi, \mu) \in \mathsf{dMSC}(A, \mathbb{P})$. A message $(e, f) \in \lhd_{\mathsf{msg}}$ with contents $\mu(e, f) = a(p_1, \ldots, p_n)$ is* executable *if*

- $\{p_1, \ldots, p_n\} \subseteq \mathit{Pids}(M)$*, and*
- $q \rightsquigarrow_M e$ *for every* $q \in \{\pi(f), p_1, \ldots, p_n\}$*.*

Moreover, M is executable *if each of its messages is executable. Finally, an rMSG $\mathcal{H}$ is* executable *if each MSC from $L(\mathcal{H})$ is executable.* ♢

Let M be an MSC and $\mathcal{H}$ be an rMSG. One can verify that

- M is executable iff $\{M\}$ is implementable, and
- $\mathcal{H}$ is executable if it is implementable (while the converse might fail).

Example 5.14. For illustration, consider Figures 5.4 and 5.5 on page 86. There, M_2, M_3, M_4 are executable, while M_1 is not. Moreover, the rMSG $\mathcal{H}_{5.6}$ is executable (recall that it is actually implementable). ♢

Executability leads us to another decision problem:

Problem 5.15. rMSG-Executability:

Input:	A; rMSG $\mathcal{H}$ over A
Question:	Is $L(\mathcal{H})$ executable?

Unlike implementability, executability of rMSGs is decidable:

Theorem 5.16 ([BCH+13]). rMSG-Executability *is in* Pspace.

Proof (sketch). We will sketch the proof idea. The problem is reduced to a reachability problem in a finite transition system. Essentially, the transition system is a finite unfolding of the given rMSG $\mathcal{H}$. In particular, transitions are labeled with the finitely many partial MSCs that occur in $\mathcal{H}$. The locations of $\mathcal{H}$ are extended to contain information about which processes know each other, or, equivalently, about the $\rightsquigarrow$-paths that exist between processes. We only need to take into account the *active processes*, i.e., those processes that possibly execute further actions. As these active processes can be identified with their respective registers, all the information we need can be maintained within a finite alphabet.

More precisely, a state of the transition system that we are going to define is a pair (ℓ, CS) where ℓ is a location of $\mathcal{H}$ and CS is the current *communication structure*. The latter is a graph whose nodes are some of the (finitely many) registers. There is an edge $r \rightsquigarrow r'$ if, informally speaking, the process associated with r is known by (or, was communicated to) the process associated with r'. Now, there is a simulation relation between the "real", infinitely many configurations of $\mathcal{H}$ and the finitely many states of the transition system. This simulation relation is correct in the following sense. If a path to (ℓ, CS) in the transition system simulates a run ρ of $\mathcal{H}$, say, to configuration (ℓ, τ, U), then, for all distinct $r, r' \in \mathrm{dom}(\tau)$, we have that $\tau(r) \rightsquigarrow_{M(\rho)} \max_{\tau(r')}(M(\rho))$ iff $r \rightsquigarrow r'$ is an edge in the communication structure CS.

Now, rMSG $\mathcal{H}$ is *not* executable iff there is an accepting path in the transition system using a transition $(\ell, CS) \xrightarrow{M} (\ell', CS')$ such that the (symbolic) partial MSC M is *not* executable wrt. CS. Figure 5.7 illustrates a part of the transition

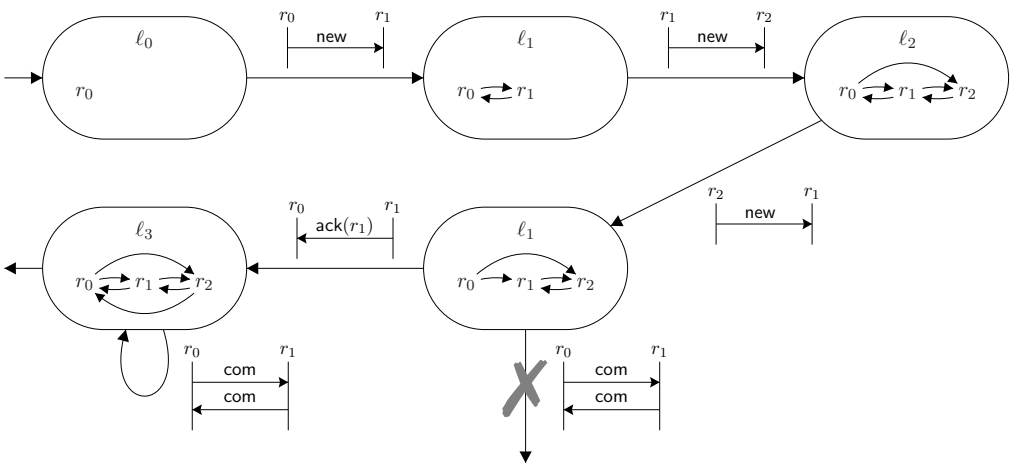


Figure 5.7: Part of the finite symbolic transition system associated with $\mathcal{H}_{5.6}$

system for the executable rMSG $\mathcal{H}_{5.6}$ from Figure 5.6. Note that the crossed transition is actually not in the transition system and just added for illustration: since r_0 does not know r_1 in the current communication structure, it could not send a message to r_1. From the existence of an accepting path using such as transition, we could actually deduce that the underlying rMSG is not executable. ■

There is an alternative proof of decidability of rMSG-Executability, though it does not directly give us a complexity upper bound. Indeed, executability is a property that is definable in MSO logic. In turn, model checking rMSGs against MSO properties is decidable, which can be shown along the same lines as for session automata (cf. Theorem 4.35 on page 77).

Definition 5.17. *The set* $\mathrm{MSO}(A, \mathbb{P})$ *of MSO formulas over* A *and* $\mathbb{P}$ *is built from the following grammar:*

$$\varphi \quad ::= \quad x \lhd_{\mathsf{new}} y \;\mid\; x \lhd_{\mathsf{proc}} y \;\mid\; x \lhd_{\mathsf{msg}}^{a(x_1,\ldots,x_n)} y \;\mid\; x = y \;\mid\; x \in X \;\mid$$
$$\neg\varphi \;\mid\; \varphi_1 \vee \varphi_2 \;\mid\; \exists x.\varphi \;\mid\; \exists X.\varphi$$

where $a \in A$ *with* $\mathit{arity}(a) = n$. ♢

The only predicate that needs explanation is $x \lhd_{\mathsf{msg}}^{a(x_1,\ldots,x_n)} y$. It says that events x and y exchange a message of the form $a(p_1, \ldots, p_n)$ such that event x_i is located on p_i, for all $i \in [n]$.[2] With this, the satisfaction relation $M \models \varphi$ for an MSC M over A and $\mathbb{P}$ and a sentence $\varphi \in \mathrm{MSO}(A, \mathbb{P})$ is defined as expected. We are now ready to state that MSO model checking of rMSGs is decidable.

[2] Alternatively, we could choose a logic in the style of Definition 3.19 on page 50, which quantifies over processes. Since a process can be identified with one of its event, this would not change the expressive power.

Theorem 5.18. *The following problem is decidable:*

INSTANCE: A; rMSG $\mathcal{H}$ over A and $\mathbb{P}$; $\varphi \in \mathrm{MSO}(A, \mathbb{P})$

QUESTION: $M \models \varphi$ for all $M \in L(\mathcal{H})$?

The theorem generalizes a similar model-checking result for MSGs (without process creation) [Mad01, MM01].

Let us come back to the alternative proof of decidability of executability of rMSGs. By Theorem 5.18, it is sufficient to come up with an $\mathrm{MSO}(A, \mathbb{P})$-formula that defines executability. We let

$$\varphi_{\text{exec}} = \bigwedge_{\substack{a \in A \\ n = arity(a)}} \forall x, y, x_1, \ldots, x_n . \big(x \lhd_{\mathsf{msg}}^{a(x_1,\ldots,x_n)} y \;\rightarrow\; (y \rightsquigarrow x \wedge \bigwedge_{i \in [n]} x_i \rightsquigarrow x) \big)$$

where $(y \rightsquigarrow x) = \exists y' \, (y' \lhd_{\mathsf{proc}}^{*} y \wedge (y', x) \in (\lhd \cup \lhd_{\mathsf{new}}^{-1})^{*})$. Then, M is executable iff $M \models \varphi_{\text{exec}}$. This also proves decidability of rMSG-EXECUTABILITY (Problem 5.15 and Theorem 5.16).

5.5 Guarded Register MSC Graphs

Now that we have a decidable criterion that is sufficient for implementability, we are interested in a fragment of rMSGs for which the criterion and implementability coincide so that the latter actually becomes decidable. To do so, we adapt definitions that were given for MSGs [GMSZ06] and guarantee their implementability. Guarded rMSGs are based on the notion of a leader process, which determines the next transition to be taken in an rMSG.

Definition 5.19 (guarded rMSG). *An rMSG $\mathcal{H} = (\mathcal{L}, \mathcal{R}, \mathcal{L}_0, \mathcal{L}_{\text{acc}}, r_0, T)$ over A is said to be* guarded *if*

1. *for all partial MSCs $M = (E, \lhd, \pi, \mu) \in \mathsf{pMSC}(A, \mathcal{R})$ that occur in $\mathcal{H}$, $(E, \lhd^{*})$ has a unique minimal element e—we let $first(M) \stackrel{\text{def}}{=} \pi(e)$, and*

2. *there is a mapping $leader : \mathcal{L} \to \mathcal{R}$ such that, for all transitions $\ell \xrightarrow{M} \ell'$,*

   - *$leader(\ell) = first(M)$,*
   - *$leader(\ell') \in Pids(M)$, and,*
   - *for all $r \in Pids(M)$, $\max_r(M) \lhd^{*} \max_{leader(\ell')}(M)$.* ♢

Intuitively, the last condition of 2. says that all processes of M terminate before $leader(\ell')$ terminates.

Example 5.20. The rMSG $\mathcal{H}_{5.6}$ from Example 5.9 on page 88 is guarded. The leaders as required in Definition 5.19 are given by $leader(\ell_i) = r_i$ for $i \in \{0, 1, 2\}$, and $leader(\ell_i) = r_0$ for $i \in \{3, 4\}$. ♢

Indeed, executability is both sufficient and necessary for implementability when we restrict to the class of guarded rMSGs.

Theorem 5.21 ([BCH+13]). *A guarded rMSG is implementable iff it is executable. Moreover, if it is implementable, then an equivalent DCA can be constructed in exponential time.*

Proof (sketch). The proof is by construction of a DCA from an executable rMSG. We give only a rough description of it. Essentially, we start from the rMSG enriched with communication structures, as described in the proof of Theorem 5.16. This enriched structure is then projected onto the processes (i.e., registers). However, the nondeterminism in the rMSG can lead to potentially inconsistent runs if each process is allowed to choose the next transition. The guardedness property helps us to avoid this problem. We let the leader process nondeterministically choose the next transition. This decision can be communicated to all other active processes, thanks to the connectedness of the partial MSCs. During the simulation of each process, we also attach the current communication structure to the messages so that each process is aware of the processes that it knows and that other processes know. This is essential to infer which registers have to be/must not be updated. By guardedness of the underlying rMSG, it will be guaranteed that the communication structure sent/received is always up-to-date. ■

5.6 Perspectives

In [BCH+13], we actually presented a richer version of rMSGs, which is equipped with a branching operator (the formalism is called *branching high-level MSCs*). This operator allows one to divide a current computation into subcomputations, splitting the registers, and to merge the subcomputations once they terminate. Branching high-level MSCs were inspired by *branching automata* [LW00, LW01], which serve as recognizers of *series-parallel pomsets*. In terms of sequentializations, adding branching capabilities may be seen as using an additional stack. Some protocols such as the leader-election protocol, which are perfectly implementable in terms of DCAs, can only be specified *globally* using pushdown stacks. The seemingly close relation with series-parallel pomsets may help us to establish an algebraic approach to dynamic concurrent systems. A related question for future research concerns extensions of Zielonka's theorem to a dynamic setting (cf. also its generalization to a recursive setting in Chapter 2). We may consider rMSGs as analogon of a finite automaton and study its closure under independent events. One may then ask if this guarantees implementability in terms of a DCA.

CHAPTER 6

Static Timed Shared-Memory Systems

In this chapter, we turn towards quantitative systems. More precisely, we consider concurrent systems with a static and fixed communication structure, whose processes obey timing constraints. Timed automata [AD94] are a well-studied formalism to describe sequential systems with timing. Networks of timed automata have then been extensively studied as models of distributed timed systems (see, e.g., [LPY95, BHR06, GL08, CCJ06, BCH12]). In most cases, it was assumed that all processes evolve at the same speed, having knowledge of a global time.

However, it is not aways justified to assume that all processes proceed at the same speed. Clock evolution may depend on temperature, workload, clock precision, etc., and drifting clocks are all the more a realistic assumption, since processes are more and more often executed on physically distributed computers. In this section, we study timed automata whose clocks evolve independently. More precisely, the set of clocks is partitioned into equivalence classes, and equivalent clocks are running at the same speed, while others do not necessarily. We will model timed systems with one global state space, focussing on the new aspects of independently evolving clocks. There are actually several natural *distributed* versions of that model (see [DL07, ABG+14], for example), involving several processes. In particular, one may assume that every clock is assigned to an owner process, and consider that two clocks are equivalent if they belong to the same process. Actually, the *semantics* of the distributed model considered in [ABG+14] is defined in terms of the timed systems defined below.

6.1 Timed Automata with Independently Evolving Clocks

In the following, we let $\mathbb{R}_{\geq 0}$ denote the set of non-negative real numbers and set $\mathbb{R}_{>0} = \mathbb{R}_{\geq 0} \setminus \{0\}$. For an alphabet Σ, we let $\Sigma_\varepsilon \stackrel{\text{def}}{=} \Sigma \uplus \{\varepsilon\}$.

Let us formally define our model of timed automata with independently evolving clocks, as well as their semantics.

Definition 6.1 (icTA). *A* timed automaton with independently evolving clocks (icTA) *over P is a tuple*

$$\mathcal{A} = (S, \Sigma, \mathcal{C}, \Delta, \iota, F, \sim)$$

where $(S, \Sigma, \mathcal{C}, \Delta, \iota, F)$ is a classical timed automaton[1] *[AD94] and $\sim$ is an equivalence relation to determine which clocks run at the same speed. Let us be more precise:*

- *S is the nonempty finite set of* states*,*
- *Σ is the finite alphabet of* actions*,*
- *$\mathcal{C}$ is the nonempty finite set of* clocks*,*
- *$\Delta \subseteq S \times \Sigma_\varepsilon \times \mathrm{Constr}(\mathcal{C}) \times 2^{\mathcal{C}} \times S$ is the finite set of* transitions *(where the set* $\mathrm{Constr}(\mathcal{C})$ *of* clock constraints *is specified below),*
- *$\iota \in S$ is the* initial state*,*
- *$F \subseteq S$ is the set of* final states*, and*
- *$\sim \subseteq \mathcal{C} \times \mathcal{C}$ is an equivalence relation.* ♢

In the above definition, the set $\mathrm{Constr}(\mathcal{C})$ of *clock constraints* over $\mathcal{C}$ is given by the grammar

$$\varphi ::= \mathrm{true} \mid \mathrm{false} \mid x \bowtie c \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2$$

where x ranges over $\mathcal{C}$, $\bowtie \in \{<, \leq, >, \geq, =\}$, and $c \in \mathbb{N}$. A *(clock) valuation* is a mapping $\nu : \mathcal{C} \to \mathbb{R}_{\geq 0}$ assigning to each clock its current value. Satisfaction of a clock constraint φ wrt. a valuation $\nu \in \mathrm{Val}(\mathcal{C})$ is defined as expected. If ν satisfies φ, we write $\nu \models \varphi$. The set of clock valuations over $\mathcal{C}$ is denoted by $\mathrm{Val}(\mathcal{C})$.

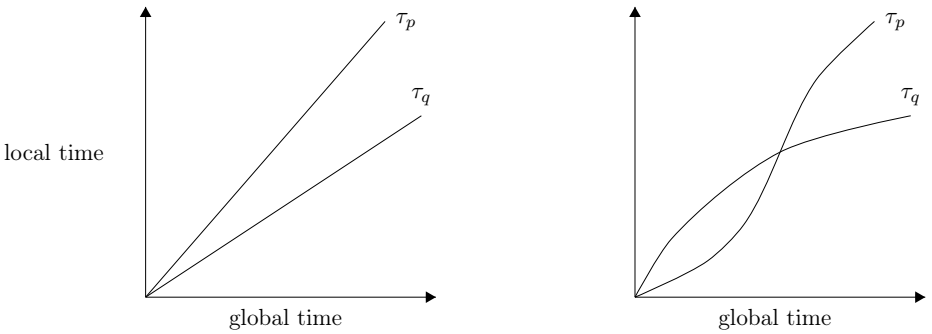
A transition $(s, a, \varphi, R, s') \in \Delta$ shall be read as follows: being in state s, the automaton can move on to state s' and perform $a \in \Sigma_\varepsilon$ provided the current clock values satisfy constraint φ. When the transition is taken, clocks from the set $R \subseteq \mathcal{C}$ are reset.

When $x \sim y$, then the clocks x and y evolve at the same speed, whereas $x \not\sim y$ means that they evolve independently. Thus, if $\sim = \mathcal{C} \times \mathcal{C}$, then we deal with a classical timed automaton. By $[x] \stackrel{\text{def}}{=} \{y \in \mathcal{C} \mid x \sim y\}$, we denote the equivalence

[1] Note that our model from [ABG+14] includes invariants (in terms of clocks constraints) on states, which were necessary to obtain some negative results in the distributed setting. They are, however, not essential for the presentation in this chapter so that we omit them for simplicity.

class of x. Since $[x]$ can be seen as a set of clocks that belong to the same process, we let $\mathcal{P} \stackrel{\text{def}}{=} \{[x] \mid x \in \mathcal{C}\}$ denote the set of equivalence classes induced by $\sim$ (a corresponding index is omitted, as $\sim$ will be clear from the context). Moreover, we let p and q range over $\mathcal{P}$.

Equivalent clocks evolve according to some local time. In turn, local time is modeled relative to some *global* time. More precisely, for $p \in \mathcal{P}$, a *local time function* is a mapping $\tau_p : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ (from global time to local time), with $\tau_p(0) = 0$, that is strictly increasing, diverging, and continuous. We set *Rates* to be the set of tuples $(\tau_p)_{p \in \mathcal{P}}$ where each τ_p is a local time function. Note that $\tau \in \mathit{Rates}$ can also be seen as a function $\mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}^{\mathcal{P}}$. Each of the two diagrams below illustrates an element from *Rates*, where we assume that there are two equivalence classes, p and q.



For a valuation $\nu : \mathcal{C} \to \mathbb{R}_{\geq 0}$ and $t = (t_p)_{p \in \mathcal{P}} \in \mathbb{R}_{\geq 0}^{\mathcal{P}}$, let $\nu + t$ be the valuation defined by $(\nu + t)(x) = \nu(x) + t_{[x]}$. Intuitively, when time passes, we add, to every clock x, the time elapse that corresponds to the local time of $[x]$.

Usually, the semantics of a timed automaton is defined in terms of an infinite automaton. A configuration of the infinite automaton keeps track of the current state and the current clock valuation. In our setting, since the semantics will be defined wrt. local times, we have to keep track of the current reference point in the global-time scale. Thus, for $\tau \in \mathit{Rates}$, we define an infinite automaton $\mathcal{A}_\tau = (\mathcal{S}, \Sigma, \to, \bar{\iota}, \mathcal{F})$ where

- $\mathcal{S} = S \times \mathrm{Val}(\mathcal{C}) \times \mathbb{R}_{\geq 0}$ is the set of configurations,
- $\bar{\iota} = (\iota, \{x \mapsto 0 \mid x \in \mathcal{C}\}, 0)$ is the initial configuration, and
- $\mathcal{F} = F \times \mathrm{Val}(\mathcal{C}) \times \mathbb{R}_{\geq 0}$ is the set of final configurations.

Moreover, $((s, \nu, t), a, (s', \nu', t')) \in \mathcal{S} \times \Sigma_\varepsilon \times \mathcal{S}$ is contained in the transition relation $\to$ (and we write $(s, \nu, t) \xrightarrow{a} (s', \nu', t')$) if $t \leq t'$, there are $\varphi \in \mathrm{Constr}(\mathcal{C})$ and $R \subseteq \mathcal{C}$ such that

- $(s, a, \varphi, R, s') \in \Delta$,
- $\nu + \tau(t') - \tau(t) \models \varphi$, and

- $\nu' = (\nu + \tau(t') - \tau(t))[R]$ (which behaves like $\nu + \tau(t') - \tau(t)$ on all clocks from $\mathcal{C} \setminus R$, and maps all clocks from R to 0).

The language $L(\mathcal{A}_\tau)$ of $\mathcal{A}_\tau$ is then defined as expected as a subset of Σ^*. The *τ-semantics* of $\mathcal{A}$ is $L(\mathcal{A}, \tau) \stackrel{\text{def}}{=} L(\mathcal{A}_\tau)$.

However, the precise local time rate τ may be difficult to predict or simply be unknown. This is why we introduce two more semantics (actually three as we will see later) that do not depend on concrete time rates. One is an overapproximation of the τ-semantics, while the other is an underapproximation. The choice of the precise semantics depends on the system property that one may wish to verify.

Definition 6.2 (semantics). *Let $\mathcal{A} = (S, \Sigma, \mathcal{C}, T, s_0, F, \sim)$ be an icTA. We let*

$$L_\exists(\mathcal{A}) \stackrel{\text{def}}{=} \bigcup_{\tau \in Rates} L(\mathcal{A}, \tau)$$

$$L_\forall(\mathcal{A}) \stackrel{\text{def}}{=} \bigcap_{\tau \in Rates} L(\mathcal{A}, \tau)$$

be the existential *and, respectively,* universal semantics *of $\mathcal{A}$.* ♢

Example 6.3. Consider the icTA $\mathcal{A}$ from Figure 6.1, with clocks x and y such that $x \not\sim y$. Suppose first that both clocks evolve at the same speed as global time, i.e., we assume $\mathrm{id} = (\tau_{[x]}, \tau_{[y]}) \in Rates$ where $\tau_{[x]}$ and $\tau_{[y]}$ are both the identity function on $\mathbb{R}_{\geq 0}$. Then, we actually deal with an ordinary timed automaton, and it is easily seen that $\mathcal{A}$ accepts a, ab, and b. By definition, all these words are also in the existential semantics. On the other hand, $L(\mathcal{A}, \mathrm{id})$ does not contain c: since there are no resets in $\mathcal{A}$, accepting c would require y to run faster than x at some point, which contradicts the assumption of synchronous clocks. We deduce that c is not contained in the universal semantics of $\mathcal{A}$. However, it is in the existential one: we can choose time rates such that $x < 1 < y$ is eventually satisfied, activating the transition from s_0 to s_5. The same rates witness the fact that b is not contained in the universal semantics. We still have to check containment of a and ab in $L_\forall(\mathcal{A})$. Clearly, $a \in L_\forall(\mathcal{A})$: since local time rates are strictly increasing, $(0 < x < 1) \wedge (0 < y < 1)$ is eventually satisfied so that the transition from s_0 to s_1 can always be taken. Finally, $ab \in L_\forall(\mathcal{A})$ also holds, though by a more subtle argument. First, note that guard $y \leq 1 \leq x$ on the transition from s_1 to s_3 and guard $x < 1 = y$ on the transition from s_2 to s_3 are complementary: one of them will eventually be satisfied, but not both of them. As local time rates are given in advance, there is always a way to reach the final state s_3 while reading ab. Summarizing, we have

- $L(\mathcal{A}, \mathrm{id}) = \{a, ab, b\}$,
- $L_\exists(\mathcal{A}) = \{a, ab, b, c\}$, and
- $L_\forall(\mathcal{A}) = \{a, ab\}$. ♢

As suggested above, we may now use $L_\exists(\mathcal{A})$ to check safety properties: given a regular set $Bad \subseteq \Sigma^*$ of undesired behaviors, the corresponding model-checking

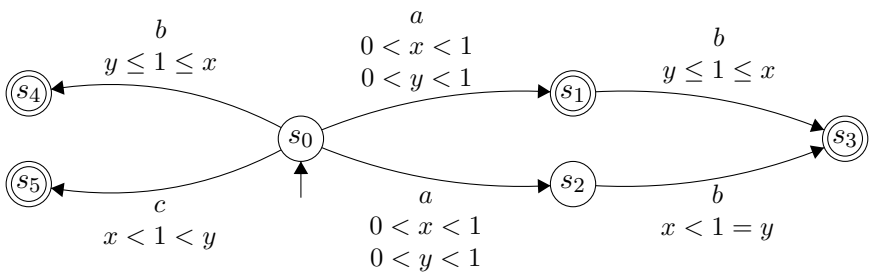


Figure 6.1: The icTA $\mathcal{A}$ with independent clocks x and y

problem asks whether $L_\exists(\mathcal{A}) \cap \mathit{Bad} = \emptyset$. Dually, we use $L_\forall(\mathcal{A})$ to check liveness properties: given a regular set *Good* of behaviors that the system should exhibit, no matter what the local time functions are, we would like to have $\mathit{Good} \subseteq L_\forall(\mathcal{A})$. So, the next question to ask is if the above problems are solvable, say, when *Bad* and *Good* are given as regular languages. While the answer is positive in the case of the existential semantics, we will show that nonemptiness (and universality) of the universal semantics are undecidable.

Note that very similar models with clock drifts have already been studied [Pur00, DDMR04, DL07, SFK08]. However, the semantics considered there rather correspond to the existential semantics and the questions studied were actually quite different.

6.2 The Existential Semantics

In this section, we argue that the existential semantics is always regular and can be effectively computed. Therefore, it can be used to check safety properties of a given icTA.

Theorem 6.4 ([ABG+14]). *Let $\mathcal{A}$ be an icTA. Then, $L_\exists(\mathcal{A})$ is a regular language that can be effectively represented as a finite automaton.*

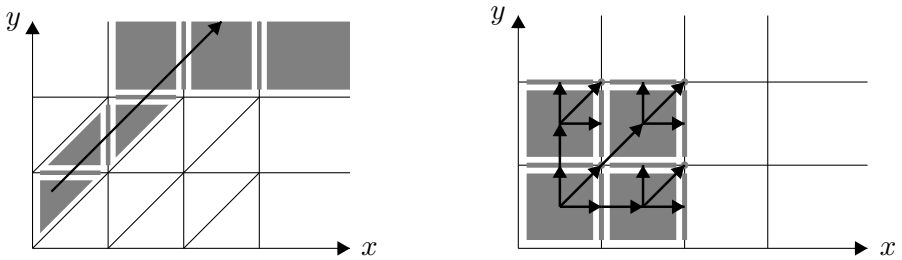
In the rest of this section, we sketch the proof of Theorem 6.4.

Note that the above result was known for timed automata with perfectly synchronous clocks. Actually, the proof of Theorem 6.4 is done via a conservative extension of the region automaton associated with an ordinary timed automaton. The main idea of the region automaton is to discretize clock valuations, i.e., to classify them into finitely many equivalence classes, called (clock) regions. The regions of a classical timed automaton with two clocks are depicted on the left-hand side of Figure 6.2. The clue is that

- regions are coarse enough to give rise to finitely many equivalence classes,

- regions are fine enough to be faithfully evaluated over the guards that occur in the underlying finite automaton, and
- equivalent regions are crossing the same regions when letting time elapse and resetting clocks from time to time. In particular, the time-elapse successors for a given region are uniquely determined.

Note that the region partitioning actually depends on the largest constant that a clock is compared with in the automaton (in the example, 2 for clock y and 3 for clock x). Note that the left-hand side of Figure 6.2 indeed assumes that clocks x and y are synchronous. Speaking in terms of icTAs, they are equivalent, i.e., $x \sim y$. When $x \not\sim y$, the region equivalence is coarser. As illustrated on the right-hand side of Figure 6.2, the diagonal regions are abandoned. The reason is that x and y may evolve at different speeds. In particular, from the initial region, induced by $\{x \mapsto 0, y \mapsto 0\}$, *any* other region is "reachable". Actually, we *must not* include diagonal regions, as the induced region-successor relation would not be a partial order anymore and spoil forthcoming constructions.



Regions when $x \sim y$ Regions when $x \not\sim y$

Figure 6.2: Region successors

Let us be (slightly) more formal, and let $\mathcal{A} = (S, \Sigma, \mathcal{C}, \Delta, \iota, F, \sim)$ be an icTA. Let $p \in \mathcal{P}$. Given a clock valuation $\nu \in \mathrm{Val}(\mathcal{C})$, we define its p-restriction $\nu_p : p \to \mathbb{R}_{\geq 0}$ by $\nu_p(x) = \nu(x)$ for all $x \in p$. We say that two clock valuations ν and ν' over $\mathcal{C}$ are *equivalent* if, for all $p \in \mathcal{P}$, the restrictions ν_p and ν'_p are equivalent in the classical sense wrt. the clocks in p.[2] Recall that this equivalence depends on the largest constants the clocks are compared with. We write $[\nu]$ for the equivalence class of ν. By $\mathit{Regions}(\mathcal{A}) \stackrel{\text{def}}{=} \{[\nu] \mid \nu \in \mathrm{Val}(\mathcal{C})\}$, we denote the set of *(clock) regions* induced by $\mathcal{A}$. Note that $\mathit{Regions}(\mathcal{A})$ is finite.

Let $\gamma, \gamma' \in \mathit{Regions}(\mathcal{A})$ be two clock regions. We say that γ' is *accessible* from γ, written $\gamma \sqsubseteq \gamma'$, if either $\gamma = \gamma'$ or there are $\nu \in \gamma$, $\nu' \in \gamma'$, and $t \in \mathbb{R}^{\mathcal{P}}_{>0}$ such that $\nu' = \nu + t$. Note that $\sqsubseteq$ is a partial-order relation on $\mathit{Regions}(\mathcal{A})$. The *direct-successor* relation, written $\gamma \boxdot \gamma'$, is, as usual, defined by $\gamma \sqsubseteq \gamma'$, $\gamma \neq \gamma'$, and $\gamma'' = \gamma$ or $\gamma'' = \gamma'$ for all clock regions γ'' with $\gamma \sqsubseteq \gamma'' \sqsubseteq \gamma'$. The relation $\boxdot$ is represented on the right-hand side of Figure 6.2 by arrows pointing from one to another region.

[2] We assume the reader is familiar with the region equivalence for ordinary timed automata and simply refer to [AD94] or to the left-hand side of Figure 6.2.

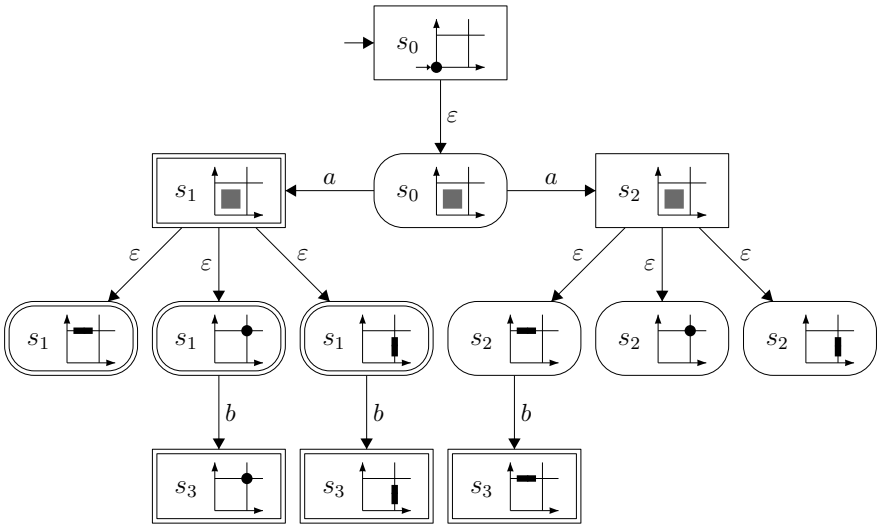


Figure 6.3: Part of the region automaton $\mathcal{A}_\exists$ of the icTA $\mathcal{A}$ from Figure 6.1

We will now associate with an icTA $\mathcal{A} = (S, \Sigma, \mathcal{C}, \Delta, \iota, F, \sim)$ a finite automaton recognizing its existential semantics (the correctness proof is omitted, though). A modified, game-based version of that finite automaton will later be used to define a semantics that underapproximates the universal semantics. We define the *finite* automaton $\mathcal{A}_\exists = (\mathcal{S}, \Sigma, \delta, \bar{\iota}, \mathcal{F})$ over Σ by

- $\mathcal{S} = S \times \mathit{Regions}(\mathcal{A})$,
- $\bar{\iota} = (\iota, [\{x \mapsto 0 \mid x \in \mathcal{C}\}])$, and
- $\mathcal{F} = F \times \mathit{Regions}(\mathcal{A})$.

Moreover, the transition relation $\delta \subseteq \mathcal{S} \times \Sigma_\varepsilon \times \mathcal{S}$ is partitioned into a set δ_0 of *discrete* transitions and a set δ_1 of *timed* transitions:

- The set δ_0 contains $(s, \gamma) \xrightarrow{a} (s', \gamma')$ if there are $\nu \in \gamma$ and $(s, a, \varphi, R, s') \in \Delta$ such that $\nu \models \varphi$ and $\nu[R] \in \gamma'$.
- The set δ_1 contains $(s, \gamma) \xrightarrow{a} (s', \gamma')$ if $a = \varepsilon$, $s = s'$, and $\gamma \boxdot \gamma'$.

Example 6.5. Part of the region automaton $\mathcal{A}_\exists$ belonging to the icTA $\mathcal{A}$ from Figure 6.1 is depicted in Figure 6.3. Discrete transitions origin in rounded boxes, time-elapse transitions in rectangular boxes. The extract shows that a and ab are contained in the existential semantics of $\mathcal{A}$ and that $\mathcal{A}_\exists$ admits even two accepting runs on ab. The latter observation corresponds to the fact that ab is accepted for local time rates that are crossing different regions. ♢

Now, Theorem 6.4 follows by the following lemma:

Lemma 6.6. We have $L(\mathcal{A}_\exists) = L_\exists(\mathcal{A})$.

6.3 The Universal Semantics

Next, we show that, unlike the existential semantics, the universal semantics comes with an undecidable nonemptiness problem.

Theorem 6.7 ([ABG+14]). *The following problem is undecidable:*

INSTANCE: icTA $\mathcal{A}$

QUESTION: $L_\forall(\mathcal{A}) \neq \emptyset$?

In the rest of this section, we prove Theorem 6.7. The proof is by reduction from Post's correspondence problem (PCP).

Problem 6.8. Post's correspondence problem (PCP):

INPUT: Finite alphabet Σ and morphisms $f, g : \Sigma^* \to \{0,1\}^*$

QUESTION: Is there $w \in \Sigma^+$ such that $f(w) = g(w)$?

Let $\Sigma = \{a_1, \ldots, a_k\}$, where $k \geq 1$, and f, g be an instance of the PCP. We will determine an icTA $\mathcal{A}$ with set of clocks $\{x, y\}$ and $x \not\sim y$ such that $L_\forall(\mathcal{A}) = \{w \in \Sigma^+ \mid f(w) = g(w)\}$, i.e., the universal semantics of $\mathcal{A}$ contains precisely the solutions to the PCP instance. One ingredient of the construction is an encoding of sequences over $\{0,1\}$ in terms of local time functions. Let $p = \{x\}$ and $q = \{y\}$ be the equivalence classes induced by $\sim$, and suppose $\tau = (\tau_p, \tau_q) \in \mathit{Rates}$. Actually, τ will encode a word in $\{0,1,2\}^\omega$. We do this using (1×1)-square regions. If the rate function leaves this region by the upper boundary or by the right boundary, then we write 0 or 1, respectively. If it leaves the square by the end-point $(1,1)$, then we write 2. A new square region is started at the point where the rate function left the old one. Thus, the direction sequences partition the space of time rates. This is illustrated in Figure 6.4, where we obtain $\mathit{dir}(\tau) = 101\ldots$

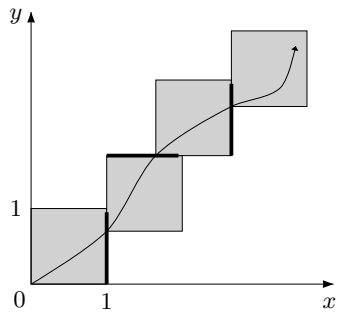


Figure 6.4: Binary sequence associated with local times

Let us formalize the encoding. With τ, we associate a sequence $t\text{-}\mathit{dir}(\tau) = t_1 t_2 \ldots \in (\mathbb{R}_{\geq 0})^\omega$ of time instances and a sequence $\mathit{dir}(\tau) = d_1 d_2 \ldots \in \{0,1,2\}^\omega$ of *directions*

as follows: for $i \geq 1$, we let first (assuming $t_0 = 0$) $t_i = \min\{t > t_{i-1} \mid \tau_r(t) - \tau_r(t_{i-1}) = 1 \text{ for some } r \in \{p, q\}\}$. With this, we set

$$d_i = \begin{cases} 0 & \text{if } \tau_p(t_i) - \tau_p(t_{i-1}) < 1 \text{ and } \tau_q(t_i) - \tau_q(t_{i-1}) = 1 \\ 1 & \text{if } \tau_q(t_i) - \tau_q(t_{i-1}) < 1 \text{ and } \tau_p(t_i) - \tau_p(t_{i-1}) = 1 \\ 2 & \text{otherwise} \end{cases}$$

We will now construct the icTA $\mathcal{A} = (S, \Sigma, \mathcal{C}, T, \iota, F, \sim)$, with $\Sigma = \{a_1, \ldots, a_k\}$, $\mathcal{C} = \{x, y\}$, and $x \not\sim y$, such that $L_\forall(\mathcal{A}) = \{w \in \Sigma^+ \mid f(w) = g(w)\}$. We proceed in two steps:

Step 1: We construct icTAs

$$\mathcal{A}_f = (S, \Sigma, \mathcal{C}, \Delta, \iota, F, \sim)$$
$$\mathcal{A}_g = (S', \Sigma, \mathcal{C}, \Delta', \iota', F', \sim)$$

(with the same Σ, $\mathcal{C}$, and $\sim$) such that, for all $\tau \in \mathit{Rates}$:

$$L(\mathcal{A}_f, \tau) = \{\, w \in \Sigma^+ \mid f(w).2 \not\leq \mathit{dir}(\tau) \,\}$$
$$L(\mathcal{A}_g, \tau) = \{\, w \in \Sigma^+ \mid g(w).2 \leq \mathit{dir}(\tau) \,\}$$

Here, $\leq$ denotes the prefix relation on words and $u.v$ denotes the concatenation of u and v.

Step 2: We build an icTA $\mathcal{A} = \mathcal{A}_f \vee \mathcal{A}_g$, which simply branches nondeterministically into $\mathcal{A}_f$ or $\mathcal{A}_g$.

With this, one can easily show the following:

Claim 6.9. We have

$$L_\forall(\mathcal{A}) = \{w \in \Sigma^+ \mid f(w) = g(w)\}\,.$$

Proof. For the inclusion from left to right, let $w \in L_\forall(\mathcal{A})$. Then, $w \in \Sigma^+$ and, for all $\tau \in \mathit{Rates}$, $w \in L(\mathcal{A}_f, \tau)$ or $w \in L(\mathcal{A}_g, \tau)$, i.e., $f(w).2 \not\leq \mathit{dir}(\tau)$ or $g(w).2 \leq \mathit{dir}(\tau)$. In particular, there is $\tau \in \mathit{Rates}$ with $\mathit{dir}(\tau) \in f(w).2.\{0,1,2\}^\omega$ and such that $f(w).2 \not\leq \mathit{dir}(\tau)$ or $g(w).2 \leq \mathit{dir}(\tau)$. As $f(w), g(w) \in \{0,1\}^*$, we have $f(w) = g(w)$.

For the inclusion from right to left, let $w \in \Sigma^+$ such that $f(w) = g(w)$. Let $\tau \in \mathit{Rates}$. Trivially, we have $f(w).2 \not\leq \mathit{dir}(\tau)$ or $f(w).2 \leq \mathit{dir}(\tau)$. As $f(w) = g(w)$, the latter implies $g(w).2 \leq \mathit{dir}(\tau)$. Therefore, $w \in L(\mathcal{A}, \tau)$. We conclude that $w \in L_\forall(\mathcal{A})$. ∎

The construction of $\mathcal{A}_f$ and $\mathcal{A}_g$ is not difficult and can be found in [ABG+14]. Let us just mention that, to "detect" this encoding in automata, we use the guards

$$x < 1 \ \wedge \ y = 1 \quad \text{(for 0)},$$

$$y < 1 \ \wedge \ x = 1 \quad \text{(for 1)},$$
$$x = 1 \ \wedge \ y = 1 \quad \text{(for 2)},$$

as well as their negations. Then, applying a new square, as described above, corresponds to resetting both clocks simultaneously.

This concludes the proof of Theorem 6.7. A very similar construction yields undecidability of the universality problem:

Theorem 6.10 ([ABG+14]). *The following problem is undecidable:*
INSTANCE: icTA $\mathcal{A} = (S, \Sigma, \mathcal{C}, \Delta, \iota, F, \sim)$
QUESTION: $L_\forall(\mathcal{A}) = \Sigma^*$?

Remark 6.11. The essence of the proof technique presented above is universal in the sense that it can be easily carried over to other settings such as word transducers. A word transducer over two finite alphabets Σ and Γ is a finite automaton $\mathcal{A}$ whose (finitely many) transitions have labels from $\Sigma^* \times \Gamma^*$. In the expected manner, $\mathcal{A}$ determines a (rational) relation $\mathcal{R}(\mathcal{A}) \subseteq \Sigma^* \times \Gamma^*$. A very well known decision problem is *universality*, which asks whether $\mathcal{R}(\mathcal{A}) = \Sigma^* \times \Gamma^*$ holds. The problem can easily be shown undecidable by a reduction from the PCP. Slightly adapting our proof technique, one can now show that the *existential* version of the problem is undecidable, too: is there a word $u \in \Sigma^*$ such that, for all $v \in \Gamma^*$, we have $(u, v) \in \mathcal{R}(\mathcal{A})$? The problem seems natural: just like in the timed setting, the alphabet Γ may model a set of signals, emitted from an environment, that trigger behaviors from Σ. When, now, the environment is not under control of the system, checking liveness specifications indeed amounts to asking the existential version of the better known problem. However, we are not aware of that undecidability result in the literature, where only universality seems to be considered.

6.4 The Reactive Semantics

The undecidability results around the universal semantics are, of course, unsatisfactory, since they imply that liveness properties cannot be checked for icTAs. In this section, we will present the *reactive semantics*. The reactive semantics is an underapproximation of the universal semantics. Like the existential semantics, it is always a regular set that can be effectively represented as a finite automaton. Thus, it will allow us to verify icTAs against (certain) liveness properties.

To motivate the reactive semantics, it is important to understand the reason for undecidability of the universal semantics and decidability in the existential case. To do so, it will be useful to consider both, the existential and the universal semantics, as a game. We may think of a two-player game, played by Player 0 and Player 1, where Player 0 controls the transitions of the icTA, while Player 1 controls the

time evolution. In the existential semantics, both players cooperate, and we may actually think of one single player who chooses, given a word $w \in \Sigma^*$, both, time rates and a run through the automaton accepting w. In the universal semantics, Player 0 aims at "accepting" the word, while Player 1 wants to "reject" the word. In other words, Player 1 tries to choose time rates in a way that does not allow Player 0 to navigate the icTA into a final state while reading w. Note that the game is not turned-based since Player 1 has to reveal its time rates entirely in advance. Thus, we actually deal with an imperfect-information game: Player 1 has no means to react upon the moves played by Player 0, so that it is, in a sense, blind. Now, imperfect information is actually often the reason for undecidability (see, e.g., [PR79, PRA01]). So, we will try to modify the game for the universal semantics to make it a turn-based, perfect-information game. In other words, we will equip Player 1 with more power.

In the following, we will first define a kind of reachability game, and then associate a game with an icTA and an input word. The reactive semantics will be defined as the set of words for which Player 0 has a winning strategy. This set can be represented as the language of an alternating two-way automaton, which shows its regularity.[3]

Definition 6.12 (game). *A* game *is a tuple* $\mathcal{G} = (\mathcal{S}_0, \mathcal{S}_1, \Sigma, \bar{\Delta}, \bar{\iota}, \mathcal{F})$ *where*

- $\mathcal{S}_0$ *and* $\mathcal{S}_1$ *are disjoint sets of states, which belong to Player 0 and Player 1, respectively, and we let* $\mathcal{S} = \mathcal{S}_0 \cup \mathcal{S}_1$,
- Σ *is a finite alphabet,*
- $\bar{\iota} \in \mathcal{S}$ *is the* initial state,
- $\mathcal{F} \subseteq \mathcal{S}$ *is the set of* final states, *which are winning for Player 0, and*
- $\bar{\Delta} \subseteq \mathcal{S} \times \Sigma_\varepsilon \times \mathcal{S}$ *is the transition relation.*

We require that, for all states $s \in \mathcal{S}$, *there are* $a \in \Sigma_\varepsilon$ *and* $s' \in \mathcal{S}$ *such that* $(s, a, s') \in \bar{\Delta}$. ♢

A (positional) *strategy* for Player i, with $i \in \{0, 1\}$, is a mapping $\sigma : \mathcal{S}_i \to \Sigma_\varepsilon \times \mathcal{S}$ such that $\sigma(s) = (a, s')$ implies $(s, a, s') \in \bar{\Delta}$, for all $(s, a, s') \in \mathcal{S}_i \times \Sigma_\varepsilon \times \mathcal{S}$. Let σ_0 and σ_1 be strategies for Player 0 and Player 1, respectively. The *play* of $\mathcal{G}$ induced by σ_0 and σ_1 is the unique infinite sequence $(s_0, a_1, s_1, a_2, s_2, \ldots)$, with $s_i \in \mathcal{S}$ and $a_i \in \Sigma_\varepsilon$, such that $s_0 = \bar{\iota}$ and, for all $i \geq 0$, $s_i \in \mathcal{S}_0$ implies $\sigma_0(s_i) = (a_{i+1}, s_{i+1})$, and $s_i \in \mathcal{S}_1$ implies $\sigma_1(s_i) = (a_{i+1}, s_{i+1})$. We say that σ_0 is *winning* for $w \in \Sigma^*$ if, for all strategies σ_1 for Player 1, the play $(s_0, a_1, s_1, a_2, s_2, \ldots)$ induced by σ_0 and σ_1 contains a position $n \geq 0$ such that both $s_n \in \mathcal{F}$ and $w = a_1 \cdot \ldots \cdot a_n$ (i.e., w is the *concatenation* of $a_1, \ldots, a_n$).

Now, we will associate, with a given icTA $\mathcal{A} = (S, \Sigma, \mathcal{C}, \Delta, \iota, F, \sim)$, the game $\mathcal{G}(\mathcal{A}) = (\mathcal{S}_0, \mathcal{S}_1, \Sigma, \bar{\Delta}, \bar{\iota}, \mathcal{F})$, which is defined as follows:

[3] In the original article [ABG+14], the semantics was presented directly in terms of an alternating two-way automaton.

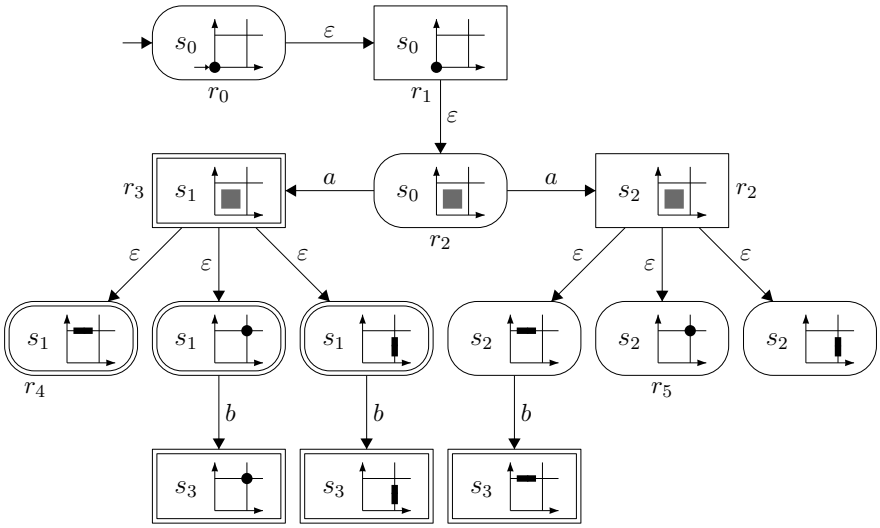


Figure 6.5: Part of the game $\mathcal{G}(\mathcal{A})$ of the icTA $\mathcal{A}$ from Figure 6.1

- $\mathcal{S}_0 = (S \times \mathit{Regions}(\mathcal{A}) \times \{0\}) \cup \{\dagger_0\}$,
- $\mathcal{S}_1 = (S \times \mathit{Regions}(\mathcal{A}) \times \{1\}) \cup \{\dagger_1\}$,
- $\bar{\iota} = (\iota, [\{x \mapsto 0 \mid x \in \mathcal{C}\}], 0)$, and
- $\mathcal{F} = F \times \mathit{Regions}(\mathcal{A}) \times \{0, 1\}$.

Assume that δ_0 and δ_1 are the sets of discrete and, respectively, time-elapse transitions of the finite automaton $\mathcal{A}_\exists$ (cf. Section 6.2). Then, $(s, \gamma, pl) \xrightarrow{a} (s', \gamma', pl') \in \mathcal{S} \times \Sigma_\varepsilon \times \mathcal{S}$ is contained in $\bar{\Delta}$ if one of the following holds:

- $pl = 0$ and $pl' = 1$ and $(s, \gamma) = (s', \gamma')$, or
- $pl = 0$ and $pl' = 0$ and $((s, \gamma), a, (s', \gamma')) \in \delta_0$, or
- $pl = 1$ and $pl' = 0$ and $((s, \gamma), a, (s', \gamma')) \in \delta_1$.

Moreover, $\bar{\Delta}$ contains transitions

- $(\dagger_0, \varepsilon, \dagger_0)$ and $(\bar{s}, \varepsilon, \dagger_0)$ for all $\bar{s} \in \mathcal{S}_0$, and
- $(\dagger_1, \varepsilon, \dagger_1)$ and $((s, \gamma, 1), \varepsilon, \dagger_1)$ for all $s \in S$ and $\boxdot$-maximal γ.

Definition 6.13. *Let $\mathcal{A} = (S, \Sigma, \mathcal{C}, \Delta, \iota, F, \sim)$ be an icTA. The* reactive semantics *of $\mathcal{A}$ is defined as the set*

$$L_{\mathrm{react}}(\mathcal{A}) \stackrel{\mathrm{def}}{=} \{w \in \Sigma^* \mid \textit{Player 0 has a winning strategy for } w \textit{ in } \mathcal{G}(\mathcal{A})\}\,. \qquad \diamondsuit$$

Example 6.14. Part of the game $\mathcal{G}(\mathcal{A})$ of the icTA $\mathcal{A}$ from Figure 6.1 on page 99 is depicted in Figure 6.5. Here, rounded states belong to Player 0, and rectangular states belong to Player 1. We will argue that a is the only word for which Player 0 has a winning strategy in $\mathcal{G}(\mathcal{A})$. To accept a, Player 0 will go from r_0 to r_1 (which is the only possible choice). Player 1 will have to respond by moving to r_2 whereupon Player 0 can conclude by going to the accepting configuration r_3, while reading a. However, Player 0 does not have a winning strategy for ab: whatever a-transition Player 0 will choose in r_2, Player 1 may go to r_4 or, respectively, r_5, so that reading b is no longer possible. We deduce that $L_{\text{react}}(\mathcal{A}) = \{a\}$. ♢

Theorem 6.15 ([ABG+14]). *Let $\mathcal{A}$ be an icTA. Then, $L_{\text{react}}(\mathcal{A})$ is a regular language that can be effectively represented as a finite automaton.*

Proof (sketch). The reactive semantics $L_{\text{react}}(\mathcal{A})$ is the language of some alternating two-way automaton (over words). The automaton has existential transitions (those taken by Player 0) and universal transitions (those taken by Player 1). The fact that we need two-way transitions comes from the ε-moves. More precisely, the alternating two-way automaton does never go backward, but may stay at the same position when taking a transition. ■

6.5 Perspectives

There is a an obvious difference between the model presented in this chapter and those from Chapters 2, 3, and 5: The semantics of a system is defined in terms of a word language, though we deal with a concurrent system. The main reason is that a pure graph and the associated partial order do not necessarily reflect causal dependencies when timing constraints come into play. There are some approaches to "marrying" the partial-order semantics and timed words (see, e.g., [BJLY98, LNZ05, BCH12]). Unfortunately, they are not directly applicable to our setting, since time stamps are in a sense meaningless in icTAs, as explained at the beginning of this chapter. However, since the partial-order approach is important and convenient for specifications, this aspect should be investigated more closely.

It will also be worthwhile to study the *realizability problem*, which we did not tackle for this timed case. In the distributed setting with several timed automata, the problem may read as follows: given a regular language L and a process topology, is there a *distributed timed automaton* $\mathcal{A}$ such that $L_\exists(\mathcal{A}) = L_\forall(\mathcal{A}) = L$? If the answer is affirmative, then $\mathcal{A}$ may be seen as a *robust* implementation of L. Here, the process topology will fix the set of processes and their reading capabilities such as "clocks of p can be read by q" (in terms of guards). This framework is then similar to [Gen05] where untimed message-passing systems have been studied.

CHAPTER 7

Conclusion

In this thesis, we conducted a study of automata models of concurrent programs, covering systems of various kinds. Those may involve recursive procedure calls, feature message-passing or shared-variable communication, and either come with a static and fixed topology or involve an unbounded number of processes, be it in a dynamic or parameterized setting. Our study was driven by applicability in formal design and verification, which stands or falls with the robustness of the formalisms considered. Whether an automata model is robust may be evaluated on the basis of the following criteria: decidability of the emptiness problem, closure under boolean operations, or existence of equivalent logical and algebraic characterizations. Not only are those properties interesting from an algorithmic point of view, but they also provide us with a deeper understanding of an automata class.

Unfortunately, even simple systems such as message-passing programs are inherently not robust: they have an undecidable emptiness problem and are not closed under complementation. However, limiting the behavior in a nontrivial meaningful way allows us to recover those properties. In this thesis, we introduced some automata models (and reviewed known ones in unifying frameworks) that turned out to be robust under some natural behavioral restrictions:

- *nested-trace automata* as a model of concurrent recursive programs,
- *parameterized communicating automata* as a model of message-passing systems with static but unknown process topology, and
- *session automata*, which are models of sequential dynamic systems.

We showed that all these models have a decidable emptiness problem and are expressively equivalent to monadic second-order logic when their behaviors are re-

stricted suitably to a bounded number of contexts or sessions, respectively. Moreover, we provided several decidability results for

- *dynamic communicating automata*, a model of message-passing systems with dynamically evolving communication topology, and
- *distributed timed automata*, which reflect concurrent systems whose processes communicate via independently evolving clocks.

Those results relied on different ideas. In the former case, we introduced a sufficient criterion for implementability to get some decidability, whereas the latter model came with natural under- and overapproximative semantics, though those were quite different from the idea of bounding the number of contexts.

In the future, since more and more systems are designed for an open world, parameterized and dynamic systems are particularly worth being studied further. Though they have indeed received increasing interest in recent years, there is, by now, no canonical approach to modeling and verifying such systems. A long-term goal should, therefore, be a *unifying* theory that lays algebraic, logical, and automata-theoretic foundations to support and facilitate the study of parameterized and dynamic concurrent systems. As we have seen, such theories indeed exist in non-parameterized settings where the number of processes and the way they are connected are fixed in advance. However, parameterized and dynamic systems lack such foundations and often restrict to very particular models with specialized verification techniques. This is especially true as far as algebraic approaches are concerned that may provide high-level expressions for parameterized systems. Algebraic studies of languages over infinite alphabets can be found in [BPT03, FK03, Boj13]. Algebraic approaches to nested-word languages have been proposed in [Cyr10, BS12]. Though it is not clear how all those techniques fit into the synthesis and verification of parameterized and dynamic systems, it will be worthwhile to explore possible connections.

In addition, several extensions/variations of the notions from this thesis are worth being studied:

- While we considered finite, terminating behaviors, the study of *infinite* structures would allow us to model *reactive systems*. There has been a lot of work on such extensions for concurrent systems (e.g., [DM94, Mus94, DGP95, Kus03, BK08]), and we believe that many of the results in this thesis go through in an infinite setting with minor adjustments.
- A substantial change in direction would be to consider branching-time behavior instead of a linear-time semantics. Recall that we defined the semantics of a system in terms of all (accepted) behaviors. As a consequence, a specification can only talk about one *fixed* behavior, but not about the *branching* structure of a system. While the branching behavior of a sequential system can be seen as a tree, that of a concurrent system is better reflected by an event structure [WN95]. Though that setting is substantially more complicated, there have been positive model-checking results [Pen97, Mad03]. An

unresolved question is whether those results can be transferred to recursive, dynamic, or parameterized systems. A starting point may be dynamic sequential systems (cf. Chapter 4) where a branching-time behavior is naturally represented by a *data tree* [BMSS09]. Recursive systems, in turn, would give rise to nested trees [ACM06].

- Models of concurrent systems that communicate with an (uncontrollable) environment are usually dealt with in the context of distributed games and distributed control/synthesis (cf. [SF07, GS13, GGMW13, MW14] for recent advances). Note that there are also some intimate connections with the above-mentioned branching-time setting [MTY05]. Parameterized synthesis has been studied in [JB14], and it would be worthwhile to explore possible extensions to a recursive or dynamic setting.

Bibliography

[AAB+08] R. Alur, M. Arenas, P. Barceló, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. *Logical Methods in Computer Science*, 4(11):1–44, 2008.

[ABG13] S. Akshay, B. Bollig, and P. Gastin. Event-clock message passing automata: A logical characterization and an emptiness checking algorithm. *Formal Methods in System Design*, 42(3):262–300, 2013.

[ABG+14] S. Akshay, B. Bollig, P. Gastin, M. Mukund, and K. Narayan Kumar. Distributed timed automata with independently evolving clocks. *Fundamenta Informaticae*, 130(4):377–407, 2014.

[ABH08] M. F. Atig, B. Bollig, and P. Habermehl. Emptiness of multi-pushdown automata is 2ETIME-complete. In *DLT'08*, volume 5257 of *LNCS*, pages 121–133. Springer, 2008.

[ABKS12] M. F. Atig, A. Bouajjani, K. Narayan Kumar, and P. Saivasan. Linear-time model-checking for multithreaded programs under scope-bounding. In *ATVA'12*, volume 7561 of *LNCS*, pages 152–166. Springer, 2012.

[ABQ11] M. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. *Logical Methods in Computer Science*, 7(4), 2011.

[ACM06] R. Alur, S. Chaudhuri, and P. Madhusudan. Languages of nested trees. In *CAV'06*, volume 4144 of *LNCS*, pages 329–342. Springer, 2006.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[ADGS13] S. Akshay, I. Dinca, B. Genest, and A. Stefanescu. Implementing realistic asynchronous automata. In *FSTTCS'13*, volume 24 of *Leibniz International Proceedings in Informatics*, pages 213–224. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS'04*, volume 2988 of *LNCS*, pages 467–481. Springer, 2004.

[AEY05] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. *Theoretical Computer Science*, 331(1):97–114, 2005.

[AG14] C. Aiswarya and P. Gastin. Reasoning about distributed systems: WYSIWYG. In *FSTTCS'14*, volume 29 of *Leibniz International Proceedings in Informatics*, pages 11–30. Leibniz-Zentrum für Informatik, 2014.

[AGNK14a] C. Aiswarya, P. Gastin, and K. Narayan Kumar. Controllers for the verification of communicating multi-pushdown systems. In *CONCUR'14*, volume 8704 of *LNCS*, pages 297–311. Springer, 2014.

[AGNK14b] C. Aiswarya, P. Gastin, and K. Narayan Kumar. Verifying communicating multi-pushdown systems via split-width. In *ATVA'14*, volume 8837 of *LNCS*, pages 1–17. Springer, 2014.

[AHH13] P. A. Abdulla, F. Haziza, and L. Holík. All for the price of few. In *VMCAI'13*, volume 7737 of *LNCS*, pages 476–495. Springer, 2013.

[AJ93] P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *LICS'03*, pages 160–170. IEEE Computer Society Press, 1993.

[AJKR14] B. Aminof, S. Jacobs, A. Khalimov, and S. Rubin. Parameterized model checking of token-passing systems. In *VMCAI'14*, volume 8318 of *LNCS*, pages 262–281. Springer, 2014.

[AKR+14] B. Aminof, T. Kotek, S. Rubin, F. Spegni, and H. Veith. Parameterized model checking of rendezvous systems. In *CONCUR'14*, volume 8704 of *LNCS*, pages 109–124. Springer, 2014.

[AM09a] R. Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3):1–43, 2009.

[AM09b] R. Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56:1–43, 2009.

[Arm07] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[Ati10] M. F. Atig. Global Model Checking of Ordered Multi-Pushdown Systems. In *FSTTCS'10*, volume 8 of *Leibniz International Proceedings in Informatics*, pages 216–227. Leibniz-Zentrum für Informatik, 2010.

[BCCC96] L. Breveglieri, A. Cherubini, C. Citrini, and S. Crespi Reghizzi. Multi-push-down languages and grammars. *International Journal of Foundations of Computer Science*, 7(3):253–292, 1996.

[BCGNK12] B. Bollig, A. Cyriac, P. Gastin, and K. Narayan Kumar. Model checking languages of data words. In *FoSSaCS'12*, volume 7213 of *LNCS*, pages 391–405. Springer, 2012.

[BCGZ14] B. Bollig, A. Cyriac, P. Gastin, and M. Zeitoun. Temporal logics for concurrent recursive programs: Satisfiability and model checking. *Journal of Applied Logic*, 2014. To appear.

[BCH12] S. Balaguer, T. Chatain, and S. Haar. A concurrency-preserving translation from time petri nets to networks of timed automata. *Formal Methods in System Design*, 40(3):330–355, 2012.

[BCH+13] B. Bollig, A. Cyriac, L. Hélouët, A. Kara, and T. Schwentick. Dynamic communicating automata and branching high-level MSCs. In *LATA'13*, volume 7810 of *LNCS*, pages 177–189. Springer, 2013.

[BD13] K. Bansal and S. Demri. Model-checking bounded multi-pushdown systems. In *CSR'13*, volume 7913 of *LNCS*, pages 405–417. Springer, 2013.

[BDM+11] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27, 2011.

[BET03] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *International Journal on Foundations of Computer Science*, 14(4):551–582, 2003.

[BGH09] B. Bollig, M.-L. Grindei, and P. Habermehl. Realizability of concurrent recursive programs. In *FoSSaCS'09*, volume 5504 of *LNCS*, pages 410–424. Springer, 2009.

[BGK14] B. Bollig, P. Gastin, and A. Kumar. Parameterized communicating automata: Complementation and model checking. In *FSTTCS'14*, volume 29 of *Leibniz International Proceedings in Informatics*, pages 625–637. Leibniz-Zentrum für Informatik, 2014.

[BGP08] J. Borgström, A. Gordon, and A. Phillips. A chart semantics for the Pi-calculus. *Electronic Notes in Theoretical Computer Science*, 194(2):3–29, 2008.

[BGS14] B. Bollig, P. Gastin, and J. Schubert. Parameterized verification of communicating automata under context bounds. In *RP'14*, volume 8762 of *LNCS*, pages 45–57. Springer, 2014.

[BHLM14] B. Bollig, P. Habermehl, M. Leucker, and B. Monmege. A robust class of data languages and an application to learning. *Logical Methods in Computer Science*, 2014.

[BHR06] P. Bouyer, S. Haddad, and P.-A. Reynier. Timed unfoldings for networks of timed automata. In *ATVA'06*, volume 4218 of *LNCS*, pages 292–306. Springer, 2006.

[BJLY98] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial order reductions for timed systems. In *CONCUR'98*, volume 1466 of *LNCS*, pages 485–500. Springer, 1998.

[BK08] B. Bollig and D. Kuske. Muller message-passing automata and logics. *Information and Computation*, 206(9-10):1084–1094, 2008.

[BK12] B. Bollig and D. Kuske. An optimal construction of Hanf sentences. *Journal of Applied Logic*, 10(2):179–186, 2012.

[BKM10] B. Bollig, D. Kuske, and I. Meinecke. Propositional dynamic logic for message-passing systems. *Logical Methods in Computer Science*, 6(3:16), 2010.

[BKM13] B. Bollig, D. Kuske, and R. Mennicke. The complexity of model checking multi-stack systems. In *LICS'13*, pages 163–170. IEEE Computer Society Press, 2013.

[BL06] B. Bollig and M. Leucker. Message-passing automata are expressively equivalent to EMSO logic. *Theoretical Computer Science*, 358(2):150–172, 2006.

[BL10] M. Bojańczyk and S. Lasota. An extension of data automata that captures XPath. In *LICS'10*, pages 243–252. IEEE Computer Society Press, 2010.

[BMSS09] M. Bojańczyk, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data trees and applications to XML reasoning. *Journal of the ACM*, 56(3), 2009.

[Boj13] M. Bojanczyk. Nominal monoids. *Theory of Computing Systems*, 53(2):194–222, 2013.

[Bol08] B. Bollig. On the expressive power of 2-stack visibly pushdown automata. *Logical Methods in Computer Science*, 4(4:16), 2008.

[Bol11] B. Bollig. An automaton over data words that captures EMSO logic. In *CONCUR'11*, volume 6901 of *LNCS*, pages 171–186. Springer, 2011.

[Bol14] B. Bollig. Logic for communicating automata with parameterized topology. In *CSL-LICS'14*. ACM Press, 2014.

[BPT03] P. Bouyer, A. Petit, and D. Thérien. An algebraic approach to data languages and timed languages. *Information and Computation*, 182(2):137–162, 2003.

[BS10] H. Björklund and T. Schwentick. On notions of regularity for data languages. *Theoretical Computer Science*, 411(4-5):702–715, 2010.

[BS12] L. Bozzelli and C. Sánchez. Visibly Rational Expressions. In *FSTTCS'12*, volume 18 of *Leibniz International Proceedings in Informatics*, pages 211–223. Leibniz-Zentrum für Informatik, 2012.

[BS14] L. Bozzelli and C. Sánchez. Visibly linear temporal logic. In *IJCAR'14*, volume 8562 of *LNCS*, pages 418–433. Springer, 2014.

[Büc60] J. Büchi. Weak second order logic and finite automata. *Z. Math. Logik, Grundlag. Math.*, 5:66–62, 1960.

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2), 1983.

[CCJ06] F. Cassez, T. Chatain, and C. Jard. Symbolic unfoldings for networks of timed automata. In *ATVA'06*, volume 4218 of *LNCS*, pages 307–321. Springer, 2006.

[CDK10] J. Chalopin, S. Das, and A. Kosowski. Constructing a map of an anonymous graph: Applications of universal sequences. In *OPODIS'10*, volume 6490 of *LNCS*, pages 119–134. Springer, 2010.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[CGNK12] A. Cyriac, P. Gastin, and K. Narayan Kumar. MSO decidability of multi-pushdown systems via split-width. In *CONCUR'12*, volume 7454 of *LNCS*, pages 547–561. Springer, 2012.

[CHJ+11] S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. In *ATVA'11*, volume 6996 of *LNCS*, pages 366–380. Springer, 2011.

[CLP11] T. Colcombet, C. Ley, and G. Puppis. On the use of guards for logics with data. In *MFCS'11*, volume 6907 of *LNCS*, pages 243–255. Springer, 2011.

[Cyr10] A. Cyriac. Temporal logics for concurrent recursive programs. Rapport de Master, Master Parisien de Recherche en Informatique, Paris, France, 2010.

[Cyr14] A. Cyriac. *Verification of Communicating Recursive Programs via Split-width*. PhD thesis, Laboratoire Spécification et Vérification, ENS Cachan, 2014.

[DDMR04] M. De Wulf, L. Doyen, N. Markey, and J.-F. Raskin. Robustness and implementability of timed automata. In *FORMATS and FTRTFT*, volume 3253 of *LNCS*, pages 118–133. Springer, 2004.

[DG02] V. Diekert and P. Gastin. LTL is expressively complete for Mazurkiewicz traces. *Journal of Computer and System Sciences*, 64(2):396–418, 2002.

[DG06] V. Diekert and P. Gastin. Pure future local temporal logics are expressively complete for Mazurkiewicz traces. *Information and Computation*, 204(11):1597–1619, 2006.

[DGP95] V. Diekert, P. Gastin, and A. Petit. Rational and recognizable complex trace languages. *Information and Computation*, 116(1):134–153, 1995.

[DL07] C. Dima and R. Lanotte. Distributed time-asynchronous automata. In *ICTAC'07*, volume 4711 of *LNCS*, pages 185–200. Springer, 2007.

[DL09] S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic*, 10(3), 2009.

[DM94] V. Diekert and A. Muscholl. Deterministic asynchronous automata for infinite traces. *Acta Informatica*, 31(4):379–397, 1994.

[DR95] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, Singapore, 1995.

[DST13] G. Delzanno, A. Sangnier, and R. Traverso. Parameterized verification of broadcast networks of register automata. In *RP'13*, volume 8169 of *LNCS*, pages 109–121. Springer, 2013.

[DSZ10] G. Delzanno, A. Sangnier, and G. Zavattaro. Parameterized verification of ad hoc networks. In *CONCUR'10*, volume 6269 of *LNCS*. Springer, 2010.

[DSZ11] G. Delzanno, A. Sangnier, and G. Zavattaro. On the power of cliques in the parameterized verification of ad hoc networks. In *FoSSaCS'11*, volume 6604 of *LNCS*, pages 441–455. Springer, 2011.

[EGM13] J. Esparza, P. Ganty, and R. Majumdar. Parameterized verification of asynchronous shared-memory systems. In *CAV'13*, volume 8044 of *LNCS*, pages 124–140. Springer, 2013.

[EK04] E. A. Emerson and V. Kahlon. Parameterized model checking of ring-based message passing systems. In *CSL'04*, volume 3210 of *LNCS*, pages 325–339. Springer, 2004.

[Elg61] C. C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98:21–52, 1961.

[EN03] E. A. Emerson and K. S. Namjoshi. On reasoning about rings. *International Journal of Foundations of Computer Science*, 14(4):527–550, 2003.

[Esp14] J. Esparza. Keeping a crowd safe: On the complexity of parameterized verification. In *STACS'14*, volume 25 of *Leibniz International Proceedings in Informatics*, pages 1–10. Leibniz-Zentrum für Informatik, 2014.

[Fig10] D. Figueira. *Reasoning on Words and Trees with Data*. PhD thesis, Laboratoire Spécification et Vérification, ENS Cachan, 2010.

[FK03] N. Francez and M. Kaminski. An algebraic characterization of deterministic regular languages over infinite alphabets. *Theoretical Computer Science*, 306(1-3):155–175, 2003.

[Gen05] B. Genest. On implementation of global concurrent systems with local asynchronous controllers. In *CONCUR'05*, volume 3653 of *LNCS*, pages 443–457. Springer, 2005.

[GGMW10] B. Genest, H. Gimbert, A. Muscholl, and I. Walukiewicz. Optimal zielonka-type construction of deterministic asynchronous automata. In *ICALP'10*, volume 6199 of *LNCS*, pages 52–63. Springer, 2010.

[GGMW13] B. Genest, H. Gimbert, A. Muscholl, and I. Walukiewicz. Asynchronous games over tree architectures. In *ICALP'13*, volume 7966 of *LNCS*, pages 275–286. Springer, 2013.

[GHR94] D. M. Gabbay, I. Hodkinson, and M. A. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects, vol. 1*. Oxford University Press, 1994.

[GK07] P. Gastin and D. Kuske. Uniform satisfiability in PSPACE for local temporal logics over Mazurkiewicz traces. *Fundamenta Informaticae*, 80(1-3):169–197, 2007.

[GK10] P. Gastin and D. Kuske. Uniform satisfiability problem for local temporal logics over Mazurkiewicz traces. *Information and Computation*, 208(7):797–816, 2010.

[GKM06] B. Genest, D. Kuske, and A. Muscholl. A Kleene theorem and model checking algorithms for existentially bounded communicating automata. *Information and Computation*, 204(6):920–956, 2006.

[GKM07] B. Genest, D. Kuske, and A. Muscholl. On communicating automata with bounded channels. *Fundamenta Informaticae*, 80(1-3):147–167, 2007.

[GL08] O. Grinchtein and M. Leucker. Network invariants for real-time systems. *Formal Aspects of Computing*, 20(6):619–635, 2008.

[GM06] B. Genest and A. Muscholl. Constructing exponential-size deterministic Zielonka automata. In *ICALP'06*, volume 4052 of *LNCS*, pages 565–576. Springer, 2006.

[GMP03] E. Gunter, A. Muscholl, and D. Peled. Compositional message sequence charts. *International Journal on Software Tools and Technology Transfer (STTT)*, 5(1):78–89, 2003.

[GMSZ06] B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-state high-level MSCs: Model-checking and realizability. *Journal of Computer and System Sciences*, 72(4):617–647, 2006.

[GS13] P. Gastin and N. Sznajder. Fair synthesis for asynchronous distributed systems. *ACM Transactions on Computational Logic*, 14(2:9), 2013.

[GW10] S. Grumbach and Z. Wu. Logical locality entails frugal distributed computation over graphs (extended abstract). In *WG'09*, volume 5911 of *LNCS*, pages 154–165. Springer, 2010.

[Han65] W. Hanf. Model-theoretic methods in the study of elementary logic. In J. W. Addison, L. Henkin, and A. Tarski, editors, *The Theory of Models*. North-Holland, Amsterdam, 1965.

[HJJ+95] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS'95*, volume 1019 of *LNCS*, pages 89–110. Springer, 1995.

[HLMS12] A. Heußner, J. Leroux, A. Muscholl, and G. Sutre. Reachability analysis of communicating pushdown systems. *Logical Methods in Computer Science*, 8(3:23):1–20, 2012.

[HMK+05] J. G. Henriksen, M. Mukund, K. Narayan Kumar, M. Sohoni, and P. S. Thiagarajan. A theory of regular MSC languages. *Information and Computation*, 202(1):1–38, 2005.

[Hol03] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[JB14] S. Jacobs and R. Bloem. Parameterized synthesis. *Logical Methods in Computer Science*, 10(1), 2014.

[Kam68] H. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, 1968.

[KF94] M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[KST12] A. Kara, T. Schwentick, and T. Tan. Feasible automata for two-variable logic with successor on data words. In *LATA'12*, volume 7183 of *LNCS*, pages 351–362. Springer, 2012.

[Kus03] D. Kuske. Regular sets of infinite message sequence charts. *Information and Computation*, 187:80–109, 2003.

[KZ10] M. Kaminski and D. Zeitlin. Finite-memory automata with nondeterministic reassignment. *International Journal of Foundations of Computer Science*, 21(5):741–760, 2010.

[Lib04] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[LM04] M. Lohrey and A. Muscholl. Bounded MSC Communication. *Information and Computation*, 189(2):160–181, 2004.

[LMM02] M. Leucker, P. Madhusudan, and S. Mukhopadhyay. Dynamic message sequence charts. In *FSTTCS 2002*, volume 2556 of *LNCS*, pages 253–264. Springer, 2002.

[LMP07] S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS'07*, pages 161–170. IEEE Computer Society Press, 2007.

[LMP08a] S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In *Proceedings of TACAS'08*, volume 4963 of *LNCS*, pages 299–314. Springer, 2008.

[LMP08b] S. La Torre, P. Madhusudan, and G. Parlato. An infinite automaton characterization of double exponential time. In *CSL'08*, volume 5213 of *LNCS*, pages 33–48. Springer, 2008.

[LMP10a] S. La Torre, P. Madhusudan, and G. Parlato. The language theory of bounded context-switching. In *LATIN'08*, volume 6034 of *LNCS*, pages 96–107. Springer, 2010.

[LMP10b] S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV'10*, volume 6174 of *LNCS*, pages 629–644. Springer, 2010.

[LN11] S. La Torre and M. Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In *CONCUR'11*, volume 6901 of *LNCS*, pages 203–218. Springer, 2011.

[LNP14a] S. La Torre, M. Napoli, and G. Parlato. Scope-bounded pushdown languages. In *DLT'14*, volume 8633 of *LNCS*, pages 116–128. Springer, 2014.

[LNP14b] S. La Torre, M. Napoli, and G. Parlato. A unifying approach for multistack pushdown automata. In *MFCS'14*, volume 8634 of *LNCS*, pages 377–389. Springer, 2014.

[LNZ05] D. Lugiez, P. Niebert, and S. Zennou. A partial order semantics approach to the clock explosion problem of timed automata. *Theoretical Computer Science*, 345(1):27–59, 2005.

[LP12] S. La Torre and G. Parlato. Scope-bounded Multistack Pushdown Systems: Fixed-Point, Sequentialization, and Tree-Width. In *FSTTCS'12*, volume 18 of *Leibniz International Proceedings in Informatics*, pages 173–184. Leibniz-Zentrum für Informatik, 2012.

[LPY95] K. G. Larsen, P. Pettersson, and W. Yi. Compositional and symbolic model-checking of real-time systems. In *RTSS'95*, pages 76–89. IEEE Computer Society Press, 1995.

[LST95] C. Lautemann, T. Schwentick, and D. Thérien. Logics for context-free languages. In *CSL'94*, volume 933 of *LNCS*, pages 205–216. Springer, 1995.

[LTKR08] A. Lal, T. Touili, N. Kidd, and T. W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS'08*, volume 4963 of *LNCS*, pages 282–298. Springer, 2008.

[LTN12] S. La Torre and M. Napoli. A temporal logic for multi-threaded programs. In *IFIP-TCS'12*, volume 7604 of *LNCS*, pages 225–239. Springer, 2012.

[LW00] K. Lodaya and P. Weil. Series-parallel languages and the bounded-width property. *Theoretical Computer Science*, 237(1-2):347 – 380, 2000.

[LW01] K. Lodaya and P. Weil. Rationality in algebras with a series operation. *Information and Computation*, 171(2):269 – 293, 2001.

[Mad01] P. Madhusudan. Reasoning about sequential and branching behaviours of message sequence graphs. In *ICALP'01*, volume 2076 of *LNCS*, pages 809–820. Springer, 2001.

[Mad03] P. Madhusudan. Model-checking trace event structures. In *LICS'03*, pages 371–380. IEEE Computer Society Press, 2003.

[Men14] R. Mennicke. Model checking concurrent recursive programs using temporal logics. In *MFCS'14*, volume 8634 of *LNCS*, pages 438–450. Springer, 2014.

[Mey08] R. Meyer. On boundedness in depth in the π-calculus. In *IFIP TCS'08*, volume 273 of *IFIP*, pages 477–489. Springer, 2008.

[MM01] P. Madhusudan and B. Meenakshi. Beyond message sequence graphs. In *FSTTCS 2001*, volume 2245 of *LNCS*, pages 256–267. Springer, 2001.

[MMP13] A. Manuel, A. Muscholl, and G. Puppis. Walking on data words. In *CSR'13*, volume 7913 of *LNCS*, pages 64–75. Springer, 2013.

[MP11] P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In *POPL'11*, pages 283–294. ACM, 2011.

[MST02] O. Matz, N. Schweikardt, and W. Thomas. The monadic quantifier alternation hierarchy over grids and graphs. *Information and Computation*, 179(2):356–383, 2002.

[MTY05] P. Madhusudan, P. S. Thiagarajan, and S. Yang. The MSO theory of connectedly communicating processes. In *FSTTCS'05*, volume 3821 of *LNCS*, pages 201–212. Springer, 2005.

[Mus94] A. Muscholl. *Über die Erkennbarkeit unendlicher Spuren*. PhD thesis, Institut für Informatik, Universität Stuttgart, 1994.

[MW14] A. Muscholl and I. Walukiewicz. Distributed Synthesis for Acyclic Architectures. In *FSTTCS'14*, volume 29 of *Leibniz International Proceedings in Informatics*, pages 639–651. Leibniz-Zentrum für Informatik, 2014.

[NSV04] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic*, 5(3):403–435, 2004.

[Och95] E. Ochmański. Recognizable Trace Languages. In Diekert and Rozenberg [DR95], chapter 6, pages 167–204.

[Pen97] W. Penczek. Model-checking for a subclass of event structures. In *TACAS'97*, volume 1217 of *LNCS*, pages 145–164. Springer, 1997.

[PR79] G. Peterson and J. Reif. Multiple-person alternation. In *FOCS'79*, pages 348–363. IEEE Computer Society Press, 1979.

[PRA01] G. Peterson, J. Reif, and S. Azhar. Lower bounds for multiplayer noncooperative games of incomplete information. *Computers & Mathematics with Applications*, 41:957–992, 2001.

[Pur00] A. Puri. Dynamical properties of timed automata. *Discrete Event Dynamic Systems*, 10(1-2):87–113, 2000.

[PWW98] D. Peled, Th. Wilke, and P. Wolper. An algorithmic approach for checking closure properties of temporal logic specifications and omega-regular languages. *Theoretical Computer Science*, 195(2):183–203, 1998.

[QR05] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS'05*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000.

[SB99] T. Schwentick and K. Barthelmann. Local normal forms for first-order logic with applications to games and automata. *Discrete Mathematics & Theoretical Computer Science*, 3(3):109–124, 1999.

[Seg06] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL'06*, volume 4207 of *LNCS*, pages 41–57. Springer, 2006.

[SEM03] A. Stefanescu, J. Esparza, and A. Muscholl. Synthesis of distributed algorithms using asynchronous automata. In *CONCUR'03*, volume 2761 of *LNCS*, pages 27–41. Springer, 2003.

[SF07] S. Schewe and B. Finkbeiner. Semi-automatic distributed synthesis. *International Journal of Foundations of Computer Science*, 18(1):113–138, 2007.

[SFK08] M. Swaminathan, M. Fränzle, and J.-P. Katoen. The surprising robustness of (closed) timed automata against clock-drift. In *IFIP-TCS'08*, volume 273 of *IFIP*, pages 537–553. Springer, 2008.

[Tho90] W. Thomas. On logical definability of trace languages. In *Proceedings of Algebraic and Syntactic Methods in Computer Science (ASMICS)*, Report TUM-I9002, Technical University of Munich, pages 172–182, 1990.

[Tho96] W. Thomas. Elements of an automata theory over partial orders. In *POMIV'96*, volume 29 of *DIMACS*. AMS, 1996.

[Tho97] W. Thomas. Languages, automata and logic. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, volume 3, pages 389–455. Springer, 1997.

[Tra62] B. A. Trakhtenbrot. Finite automata and monadic second order logic. *Siberian Math. J*, 3:103–131, 1962. In Russian; English translation in *Amer. Math. Soc. Transl.* 59, 1966, 23–55.

[TW97] P. S. Thiagarajan and I. Walukiewicz. An expressively complete linear time temporal logic for Mazurkiewicz traces. In *LICS'97*, pages 183–194. IEEE Computer Society Press, 1997.

[Tze11] N. Tzevelekos. Fresh-register automata. In *POPL'11*, pages 295–306. ACM, 2011.

[WN95] G. Winskel and M. Nielsen. Models for concurrency. In *Handbook of Logic in Computer Science (Vol. 4)*, pages 1–148. Oxford University Press, 1995.

[Zie87] W. Zielonka. Notes on finite asynchronous automata. *R.A.I.R.O. — Informatique Théorique et Applications*, 21:99–135, 1987.

Index