DOCTOR OF SCIENCES

Relating inter-agent and intra-agent specifications : the case of life sequence chart

Bontemps, Yves

Award date:
2005

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique

Relating Inter-Agent and Intra-Agent Specifications

The Case of Live Sequence Charts

Yves Bontemps

Thèse présentée en vue de l'obtention du titre de
Docteur en Informatique .

Jury: Prof. Jean-Luc Hainaut, FUNDP, (Président),
Prof. Wolfgang Thomas, RWTH Aachen.
Prof. Sebastian Uchitel, Imperial College London.
Prof. Pierre-Yves Schobbens, FUNDP, (Promoteur).
Prof. Patrick Heymans, FUNDP.

Abstract

The problem of relating inter-agent and intra-agent behavioral specifications is investigated. These two views are complimentary, in that the former is closer to scenario-based user requirements whereas the latter is design-oriented. We use a graphical, user-friendly and very simple language as inter-agent specification language: Live Sequence Charts (LSC). LSC is presented and its properties are investigated: it is highly succinct, but inexpressive. There are essentially two ways to relate inter-agent and intra-agent specifications: (i) by checking that an intra-agent specification is correct with respect to some LSC specification and (ii) by automatically constructing an intra-agent specification from an LSC specification. Several variants of these problems exist: closed/open systems and centralized/distributed systems. We give inefficient but optimal algorithms solving all problems, besides synthesis of open distributed systems, which we show is undecidable. All the problems considered are difficult, even for a very restricted subset of LSCs, without alternatives, interleaving, conditions nor loops. We investigate the cost of extending the language with control flow constructs, conditions, real-time and symbolic instances. An implementation of the algorithms is proposed. The applicability of the language is illustrated on a real-world case study.

Keywords: Live Sequence Charts, Scenarios, Reactive Systems, Synthesis, Realizability, Game Theory, Automata Theory.

Contents

Acknowledgements

This work has been supported by the "Fonds National de la Recherche Scientifique" through a "Research Fellow Grant" and travel funding, mainly via the "Centre Fédéré en Vérification".

I am thankful to all Jury members: Prof. Jean-Luc Hainaut, Prof. Wolfgang Thomas, Prof. Sebastiàn Uchitel, Prof. Patrick Heymans and Prof. Pierre-Yves Schobbens. I especially express my gratitude to Prof. Thomas for his hospitality during my stay in Aachen. Prof. Schobbens has been a great advisor. His insights on the topic of my thesis were very valuable. His intuitions about forthcoming results were *almost always* correct, which avoided us to err in wrong directions but also obliged me to double-check every result we obtained, just in case (and when they were correct, he could just say "didn't I tell you this would be so?").

Since my very first steps in research, I have closely collaborated with Patrick Heymans. This was a grand experience. Patrick is a psycho proofreader: he annotates every paper he can get his hands on, until the paper turns into some kind of red-coloured cubistic painting. As an author, this is a most depressing experience as it transforms an "almost perfect paper" into "almost a paper". But, as a colleague and co-author, Patrick is also a continuous source of motivation and inspiration. Our collaboration was very fruitful and I thank him for this.

When my research was stuck and new results did not want to appear, I have been lucky enough to meet people who showed me the right direction to proceed. In particular, the initial idea for an algorithm solving merciful games is due to Dr Christof Löding. Prof. Salvatore La Torre referred me to the PSPACE-hard problem I had to reduce from in order to show most complexity lower bounds appearing in this thesis. Ironically enough, the proof was in the very book I had been struggling with for months... but I was just not looking at the right chapter.

I would also like to thank my colleagues for the great time I had sharing an office with them: Michaël Petit, a serious-minded, austere and selfish person who can't stand men conversations; Germain Saval, with his unbearable Belgian accent and his total lack of culture, and Jean-Christophe Trigaux, a coffee-addict who has no sense of humour and despises sharing anything about his own research. Thank you also to Gaëtan, Simon, Vincent, Isabelle, Emmanuel, Pascal, Mama, and all other colleagues I forgot to list.

Finally, my girlfriend has to be thanked for her constant love and support. With her by my side, these last four years were a nearly painless experience. In return, I can only dedicate her this very manuscript, the result of all the evenings I spent working on my computer. This is objectively a poor gift, but she will understand how much it represents to me. To Roxane, with all my gratitude and love.

Introduction

Contents

A méchant ouvrier, point de bon outil.

French Proverb

Context

This thesis investigates a novel way to engineer *distributed reactive systems*. A distributed computer system is made of several components, executing concurrently and interacting each other to deal with the computerized task at hand [107]. A reactive system, as opposed to a *transformational system*, is a computer system that "keeps an ongoing relationship with its environment" [76]. While the purpose of a transformational system is essentially to transform some input data to output data, reactive systems interact continuously with their environment. In response to stimuli, occurring in the environment, reactive systems adapt their state, output some action and get ready to react to the next occurrence of stimuli. Since their state changes as a result of this continuous interaction, reactions are history-dependent: the system system may respond differently to the first or the second occurrence of the same stimulus. Typically, transformational systems are stateless: they always output the same data when fed in with the same inputs.

We chose to focus on reactive distributed systems because they are pervasive in our society. They are also getting more and more complex. For example, phones contain a tremendous number of concurrent processes. The automotive industry is embedding ever more software into modern cars. This automotive software ranges from highly-critical, real-time systems to low-priority information systems. In this sector, distribution is a matter of fact, conceptually *and* physically: several processors execute code in parallel and components that are conceptually seen as separate can end up being mapped onto the same processor, thus sharing a common resource. Even further, the same components could be mapped to another physical architecture in another model of a car, thus executing in parallel. Undoubtfully, the reader is aware that correcting flaws in reactive distributed systems is often very expensive, sometimes impossible, in the case of hardware-implemented systems.

Nevertheless, the problem of engineering reactive distributed is essentially a simple task. It amounts to "write some executable code fulfilling user requirements" as shown in Fig. 1. Of course, this represents a huge amount of

work and the engineer is likely to be left with the question: "does my code behave as I intended it to?". Actually, this question will arise as a consequence of the technical nature of the system to be developed. Remember that systems are distributed and meant to be deployed on heterogeneous architectures. Understanding the global emergent behaviour of all those concurrent pieces of software and hardware is a tremendously difficult task.

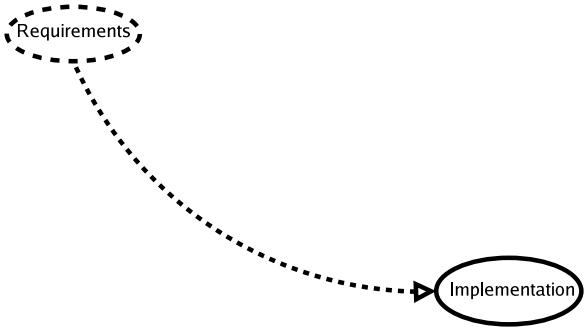


Figure 1: One big leap

When trying to answer this question, one will be confronted with the fact that the artifact to analyze is complex and hard to comprehend. Executable code is, by definition, polluted by implementation details. Those details are not necessary to understand the logics behind it. Furthermore, code is component-centric, because components are distributed: every component needs to have its own executable code which fully describes its behaviour. Different parts of the system may also be written in different languages or even be implemented as hardware. This is also an impediment to comprehension. Finally, the same "logical" system can be mapped onto different physical architectures, as already said above.

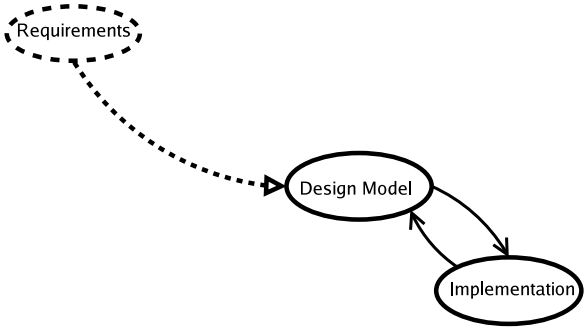


Figure 2: Introducing Design Models

In order to understand reactive distributed systems, instead of using executable code, one can resort to an abstraction of it. This abstraction is a *model*, tailored for human comprehension. This model of the actual system ignores implementation details and only presents the control logic of every component, preferrably in an architecture-independent way. Taking away all the details, the system at hand boils down to a much simpler form that one can read and,

hopefully better understand. Furthermore, tools exist to *automatically* translate these high-level models to executable code. Model-code associations are kept, to ensure that generated code can be traced back to its corresponding design element.

Being an abstraction of the actual code, it is no wonder that a design model inherits most of its properties. Of course, implementation details are ignored and architecture-dependent parts have been erased but a design model is still component-centric and complete. For every component of the system, it has to describe its whole behaviour, how it should act in every possible situation. We will thus say that this model is *intra-agent*, as it considers every component separately. We use here the term agent under its Latin meaning, i.e. "an acting entity". It is thus interchangeable with "component", "sub-system", "process" or "object", in this thesis.

Now, relying on this intra-agent abstract model, can we be sure that the actual system fulfills user requirements? Not really. There is still a huge gap between user requirements and design models. Actually, users do not spontaneously think in terms of intra-agent specifications. They cannot state, even in natural language, what the complete behaviour of the future system should be. Studies have shown that users feel more comfortable with using *scenarios*, i.e. partial concrete stories of the future system [174]. Naturally, users express their requirements first as examples. They give a few samples of execution that they consider are representative. Those executions are not focused on a single agent. When dealing with distributed systems, a story will involve several agents and cross their borders. They are thus said to be *inter-agent* representations. For instance, the following user statement is a scenario: "I engage cruise control by pressing the SET button. A green light flashes on the dashboard for five seconds and remains lit. Then, when I push on the brake pedal, cruise control is disengaged, the car brakes and the green light turns off." This scenario involves several agents (SET button, cruise control, green light, brake pedal, ...) and is partial, as it does not tell about all the other situations that may happen when cruise control is engaged. It also overlaps with other scenarios, eg scenarios talking about ABS or automatic door-locking sytem, although those scenarios also share brake pedal interactions.

Hence, the gap to bridge between requirements and design models is caused by *two* paradigm shifts. First, engineers have to move from a partial and overlapping description of the system to a complete behavioural specification. Second, a transition from inter-agent to intra-agent representations is performed. When these two changes are performed together, mistakes are likely to be introduced: some examples will be wrongly generalized or not correctly distributed among agents.

In order to narrow this gap, we propose to divide this paradigm shift in two parts, see Fig. 3. First, a complete behavioural specification is built, using an inter-agent specification language. Second, this inter-agent specification is transformed to an intra-agent design model. Of course, this does not remove all risks of building the wrong system, for no technique could ever fully remove this risk, but it makes the development more reliable.

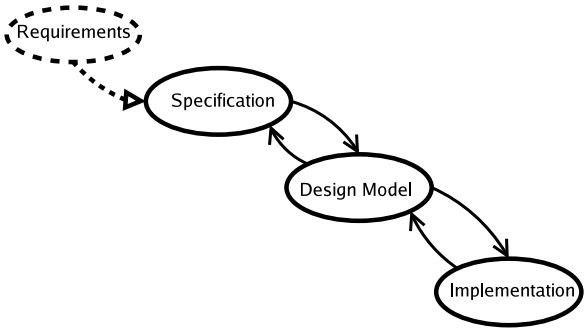


Figure 3: Inter-Agent Specification narrows the gap

This inter-agent specification language is aimed at expressing precisely user's requirements. Hence, it shall be based on good principles of Requirements Engineering, as advocated, for instance, by Jackson [66, 187, 91]. Jackson insists on the fact that, when dealing with reactive systems, *openness* is an essential issue. The system (called "machine" in Jackson) to be built will be interacting with a certain environment (called "world") and that this environment is *given*. It is thus not up to the engineer to modify or control this environment in order for it to behave nicely. However, the environment will not behave in a completely inconsistent way. There are many facts known about it, from common knowledge or additional hypotheses. What is actually required from engineers is to design a machine that will ensure that all user requirements are met, whenever the environment respects all assumptions about its behaviour. Whether those hypothesis make sense or should be ensured by setting up some new technical artifact. In this thesis, we will pay special attention to following this approach, in a transparent way. From the level of scenarios, assumptions on the behaviour of the environment will be made explicit.

If one provides an *automated* translation from inter-agent specifications to intra-agent specifications, risk will be even more decreased. No mistake can be introduced by the transition from specification to design model, provided the translater has been proven correct. This proof can be done once and for all, by experts, and remove the burden of verification from engineers.

There are several links that need to be automated. First, the inter-agent specification should be tested, in order to check that if fulfills user's intent. Second, intra-agent design models should be derived from inter-agent specifications. This implies that inter-agent specifications should be tested for implementability. If the specification is inconsistent, i.e. imposes conflicting requirements on the system, those should be detected and no implementation must be synthesized. Third, both models are likely to evolve on their own. New requirements will be added to the specification and the design model will become more detailed. After some time, it is interesting to check whether the design model still complies with the specification, i.e. that the requirements are still met. This approach has been suggested by Harel [71] and is illustrated by Fig. 4.

The following issues need to be addressed, in order to support this approach:

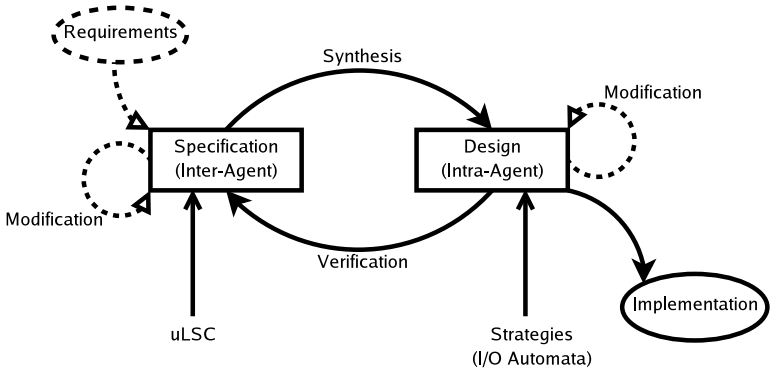


Figure 4: Full development

- A scenario-based inter-agent formal specification language needs to be defined. We propose to use Live Sequence Charts (LSC) [42]. This language has been later substantially extended [75] but we focus on a simple part of the original version, named Constant LSC in [75].

- The various links between models (specification and design) need to be formally defined. In particular, the definition of what it means for an intra-agent to "correctly implement" an inter-agent specification needs to be given.

- The feasibility of these links must be assessed. When feasible, algorithms supporting the links must be developed and implemented.

Contributions

This thesis brings the following contributions:

- The language of LSC is formally defined (sec. 3.2), its expressiveness (sec. 3.3) and its succintness (sec. 3.3.3) are studied. We show that the variant of the language we use is inexpressive, succinct, yet usable for modeling real-world systems. The last part is demonstrated thanks to a case-study.

- Formal models for inter-agent and intra-agent specification are introduced. This model is suited to reactive systems, separating clearly the environment and the system (sec. 4.2). We determine precisely what it means for an intra-agent specification to be a correct implementation of an inter-agent specification, in the spirit of Jackson (def. 4.8).

- All problems pertaining to the development approach of Fig. 4 are formally defined. Their computational complexity is studied (sec. 4.3, 4.4 and 4.5). The main conclusions are:

  – Verification is PSPACE-complete (Th. 4.21),

  - Checking that a specification allows some execution is PSPACE-complete (Th. 4.10),
  - Synthesis is undecidable (Th. 4.36).

  We study variants of these problems, namely verification of closed systems and centralized realizability which are, respectively, coNP-complete and EXPTIME-complete. These two problems are thus simpler to solve on LSC than on LTL, which justifies the claim that the loss in expressiveness pays off. For every problem, we give an optimal algorithm implementing it. Since LSC is less expressive than LTL, but verification is as hard in both languages, the thesis also provides us with sharper complexity results than [149].

- We introduce the notion of mercifulness (sec. 4.5.3). An implementation is merciful if, in addition to being correct, it always *allows* its environment to fulfill liveness hypotheses. We show that, under the perfect information hypothesis, synthesizing merciful strategies is not essentially harder than synthesizing strategies.

- The subset of LSC is extended to conditions (sec. 5.2), real-time (sec. 5.3) and symbolic instances (sec. 5.4). Our previous definitions extend smoothly to this setting which provides us with a very clean formalization of conditions. We believe that this formalization is easier to understand than the one presented in the original LSC paper. The cost of each addition is investigated. Conditions do not increase the complexity of any problems. Real-time makes verification of closed systems PSPACE-complete and does not change the complexity of all other problems. Satisfiability becomes undecidable when symbolic instances enter the stage.

- In order to work around the undecidability of synthesis, an incomplete approach is presented (sec. 4.6.1). It may fail to synthesize a design model for some agent, even though a correct implementation exists.

- An implementation, named REMoRDS, is presented in Chapter 6. It supports centralized synthesis, verification and incomplete distributed synthesis. The two latter problems rely on an additional tool, namely LTSA [113].

Structure

Chapter 1 presents and defines the notations used throughout the rest of the thesis.

Chapter 2 surveys several languages for describing scenarios. Among these languages, we argue that LSC has many interesting properties making it suitable as a formal scenario-based specification language.

Chapter 3 presents formally the language of LSC that we use in the rest of the thesis. This language is made of constant LSCs, i.e. without symbolic variables or symbolic messages, withou loops, conditions and temperature on

locations. It contains operators for expressing choice and parallel composition. Its usefulness is illustrated on NASA's CTAS case-study. The abstract syntax and semantics of the language is formally defined and its expressiveness and succinctness are studied. In particular, translations to automata are provided, which will be used in algorithms later on.

Chapter 4 defines formally what an inter-agent specification and an intra-agent design model are. It defines what it means for a design model to be a correct implementation of an intra-agent model. Then, the links of Fig. 4 are formally defined as decision problems, their computational complexity is analyzed and optimal algorithms are given. This is contrasted with related work on verification and synthesis of reactive systems, with a focus on scenario-based approaches. The notion of mercifulness is introduced and an incomplete approach to distributed synthesis is given.

Chapter 5 introduces three extensions to our subset of LSC: conditions, time and symbolic instances. These extensions are formally defined, the definition of decision problems is adapted and their computational complexity is analyzed. As said above, we get three types of changes: conditions do not change anything, time makes verification of closed system harder but leaves the complexity all other problems unchanged. Symbolic instances make the simplest decision problem, viz. satisfiability, undecidable.

Chapter 6 describes REMoRDS, a program supporting the approach of Fig. 4 through the implementation of several algorithms described in this thesis. These algorithms are

- Synthesis, under the perfect information hypothesis;
- Verification, through the generation of temporal logic formulae;
- Incomplete distributed synthesis.

Relevant implementation details and choices are discussed in this chapter.

Chapter 1

Preliminaries

Contents

Regular pleasures,
That won't disturb the routine

Patricia Barber, "Verse"

This chapter provides a rapid introduction to the field of languages and automata (Sec. 1.1), temporal logics (Sec. 1.2), infinite games (Sec. 1.3) and the theory of computational complexity (Sec. 1.4), including alternation (Sec. 1.5). Well-known results are recalled. The notation and terminology used throughout the thesis are also introduced here. This aims at making this thesis as self-contained as possible. Yet, we will be obliged to make some forward references. Readers not familiar with these topics should consult tutorial introductions and standard textbooks, of which we cite a few: [86, 133, 159, 156, 142] for automata, [51, 133, 117, 38] for temporal logics and model checking, [63, 156, 159, 158] for infinite games, [132, 34] for computational complexity, including alternation. The relation between automata and logics is surveyed in [157]. Chandra, Kozen and Stockmeyer's seminal paper on alternation provides a good introduction to this extension of Turing Machines [36].

1.1 Languages and automata

1.1.1 Regular Languages and Finite Automata

Take some arbitrary set A. Then, for any $i \in \mathbb{N}$, A^i denotes the set

$$\{a_1 \dots a_i | \forall j : 1 \leq j \leq i : a_j \in A\}$$

of all sequences of length i of elements from A. The special symbol ϵ represents the empty sequence, which is the only element contained in A^0. The *Kleene's star* operation is defined as

$$A^* = \bigcup_{i \in \mathbb{N}} A^i.$$

The set A^+ is $A^* \setminus \{\epsilon\}$.

Assume that we are given a finite set, called an *alphabet*, Σ. Any subset L of Σ^* is a *language* over Σ. If L is finite, we will say that it is a finite language.

As a convention, elements from Σ, *letters* for short, are represented as small latin letters from the beginning of the alphabet $(a, b, \ldots)$ whereas elements from Σ^*, called *words*, are represented by small latin letters from the end of the alphabet, such as $u, v, w, \ldots$. Languages are denoted by capital latin letters.

In addition to finite repetition (Kleene's star), *concatenation*, represented by "$\cdot$" is an operator to build languages. It is simply defined as the appending of a sequence to another

$$(a_1 \ldots a_n) \cdot (b_1 \ldots b_m) = a_1 \ldots a_n b_1 \ldots b_m.$$

When it is clear from the context that we are applying concatenation, we will omit the $\cdot$ operator. Concatenating languages results in the language containing the pairwise concatenation of every word from original languages:

$$L_1 \cdot L_2 = \{u_1 \cdot u_2 | u_i \in L_i,\ i = 1, 2\}$$

It is also possible to apply * to languages, still yielding languages, i.e. subsets of Σ^*:

$$L^* = \bigcup_{i \in \mathbb{N}} \{u_1 \ldots u_i | \forall j : 1 \leq j \leq i : u_j \in L\}.$$

Words from Σ^* can be ordered using the *prefix relation*. We say that a word u is a prefix of a word v, and write it $u \sqsubseteq v$ if there is a word w such that appending w to u yields v

$$\exists w \in \Sigma^* : uw = v.$$

We say that u is a *strict prefix* of v if there is a w such that $uw = v$ and $w \neq \epsilon$. We denote this $u \sqsubset w$. Clearly, $\sqsubseteq$ and $\sqsubset$ are respectively partial orders and strict partial orders on Σ^*.

A word u is *suffix* of a word v if there is some w such that $v = wu$. This is denoted $u \sqsupseteq v$. It is a strict suffix if w is not the empty word $(u \sqsupset v)$.

The class of *regular languages* over some finite alphabet Σ is the least fix point satisfying the following constraints:

1. $\{a\}$ is a regular language, with $a \in \Sigma$,
2. if L_1 and L_2 are regular languages, $L_1 \cup L_2$ is a regular language.
3. if L_1 and L_2 are regular languages, $L_1 \cdot L_2$ is a regular language.
4. if L_1 and L_2 are regular languages, $L_1 \cap L_2$ is a regular language.
5. if L_1 is a regular language, L_1^* is a regular language.

Definition 1.1 (Finite Automaton) A *finite automaton* is a tuple

$$\langle \Sigma, Q, Q_0, \Delta, F \rangle,$$

where

Σ is the set of symbols read by the automaton;

Q is a finite set of *states*;

$Q_0 \subseteq Q$ is a set of *initial states*;

$\Delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*;

$F \subseteq Q$ is a set of *final states*.

■

Automata will be denoted by capital rounded letters, such as $\mathcal{A}, \mathcal{B}, \ldots$. They can be graphically displayed as in fig 1.1. States are circles and arrows represent transitions between nodes. Final states are denoted by a double line while initial states have a dangling incoming arrow. The graph of fig 1.1 represents the automaton

$$\left\langle \{a,b\}, \{q_1,q_2\}, \{q_1\}, \left\{ \begin{array}{c} (q_1,a,q_1),(q_1,b,q_1), \\ (q_1,a,q_2),(q_2,b,q_2) \end{array} \right\}, \{q_2\} \right\rangle$$

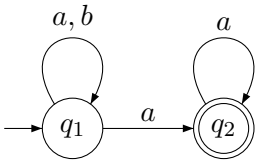


Figure 1.1: Nondeterministic finite automaton

For such an automaton, a sequence of states $q_0 q_2 \ldots q_n \in Q^+$ is called a *run* on a word $a_1 a_2 \ldots a_n \in \Sigma^*$ iff for every $0 < i \leq n$, there is a transition $(q_{i-1}, a_i, q_i) \in \Delta$. It is an *initial run* if $q_0 \in Q_0$ and it is *final* if $q_n \in F$. We say that an automaton *accepts* a word u if there is some run $r \in Q^+$ on u such that

1. r is initial;
2. r is final.

The language of a finite automaton is the set of all words accepted by this automaton:

$$\mathscr{L}(\mathcal{A}) = \{w \in \Sigma^* | \mathcal{A} \text{ accepts } w\}.$$

We alternatively say that $\mathcal{A}$ recognizes $\mathscr{L}(\mathcal{A})$.

A language is regular iff there is some finite automaton recognizing it. Thus, languages recognized by finite automata are closed under boolean operations (union, intersection, complementation), difference, concatenation and Kleene's star. As a convention, we will denote by $\overline{W}$ the complement of W, i.e. $\Sigma^* \setminus W$.

A finite automaton is *deterministic* (DFA) if there is at most one outgoing transition labeled by every letter, from every state:

$$\forall q \in Q : \forall a \in \Sigma : |\{q' | \Delta(q, a, q')\}| \leq 1.$$

It is *complete* if there is at least one outgoing transition labeled by each letter from every state:

$$\forall q \in Q : \forall a \in \Sigma : |\{q' | \Delta(q, a, q')\}| \geq 1.$$

An automaton which is not deterministic is dubbed *non-deterministic* (NFA).

Every automaton can be made complete without altering its language. Every NFA can be *determinized* i.e. transformed into a DFA that accepts the same language. This operation can cause an exponential blow-up in the number of states. Thus, every language $W \subseteq \Sigma^*$ is regular iff it is accepted by some NFA iff it is accepted by some DFA. All these equivalences are effective, i.e. there are algorithms implementing them, which is true of boolean operations, too.

Any DFA can also be minimized. This operation results in a smaller automaton, recognizing the same language and unique, up to isomorphism.

1.1.2 ω-languages and automata

In the previous section, we looked at finite words. In this section, we will look at infinite sequences of letters. We let Σ^ω be the set of all *infinite* sequences, built from letters in A.

As previously, it is possible to append an ω-word $\gamma \in \Sigma^\omega$ to some finite word $w \in \Sigma^*$, yielding a word in Σ^ω. The strict prefix relation can thus be defined on ω-words, too:

$$\forall \gamma \in \Sigma^\omega, \forall w \in \Sigma^* : w \sqsubset \gamma \iff \exists \gamma' \in \Sigma^\omega : w\gamma' = \gamma$$

The prefix relation is not defined between infinite words.

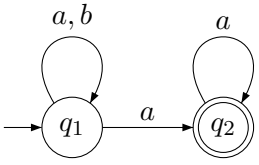


Figure 1.2: Nondeterministic Büchi automaton

Finite automata can be modified to recognize ω-languages. The most basic type of automata recognizing ω-languages are *Büchi automata*. They have the same structure as NFA, i.e. $\langle \Sigma, Q, Q_0, \delta, F \rangle$, except that they run on infinite words $\gamma \in A^\omega$. Since an automaton has only a finite number of states, when it runs over an infinite word γ, the resulting run must contain some states that are repeated infinitely often.

We say that $\mathcal{A}$ accepts γ if $\mathcal{A}$ has an initial run on γ in which an accepting state occurs infinitely often. The language recognized by $\mathcal{A}$ is the set of words it accepts:

$$\mathscr{L}(\mathcal{A}) = \{\gamma \in \Sigma^\omega | \mathcal{A} \text{ has an initial run } \rho \text{ on } \gamma \text{ s.t. } \inf(\rho) \cap F \neq \emptyset\}$$

We denote by $\inf(\rho)$ the set of states that occur infinitely often in ρ. We say that $\mathcal{A}$ *universally accepts* γ if, on *every* initial run ρ of $\mathcal{A}$ on γ, $\inf(\rho) \cap F \neq \emptyset$. Languages recognized by *non-deterministic Büchi automata* (NBA) are called *ω-regular languages*. They are closed under all boolean operations.

However *Deterministic Büchi Automata* (DBA) are strictly less expressive than NBA. For instance, there is no DBA recognizing the ω-language of the NBA in fig. 1.2. This language is the set of words containing only a finite number of b:

$$(\{a\} \cup \{b\})^*\{a\}^\omega.$$

Nevertheless, the complement of this language (an infinite number of b) is recognized by the DBA of fig. 1.3. The set DBA of all languages recognized by deterministic Büchi automata is consequently *not* closed under complementation.

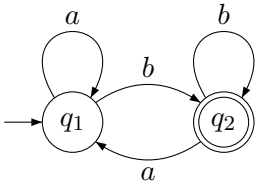


Figure 1.3: Deterministic Büchi automaton

However, there are other *acceptance conditions* for which deterministic and non-deterministic versions of automata are equivalent.

Muller: the set of accepting states F is replaced by a set $\mathcal{F}$ of sets of states. A run ρ is then accepted if

$$\inf(\rho) \in \mathcal{F}.$$

Rabin: the set of accepting states F is replaced by a set

$$\mathcal{F} = \{(E_1, F_1), (E_2, F_2), \ldots, (E_n, F_n)\}$$

of pairs of sets of states. A run ρ is then accepted if there is a pair (E_i, F_i) in $\mathcal{F}$ such that

$$\inf(\rho) \cap E_i = \emptyset \text{ and } \inf(\rho) \cap F_i \neq \emptyset.$$

Streett: this is the dual of the Rabin condition. A run ρ is accepted if, for

every pair $(E_i, F_i) \in \mathcal{F}$

$$\inf(\rho) \cap E_i \neq \emptyset \implies \inf(\rho) \cap F_i \neq \emptyset$$

Parity: the acceptance set F is replaced by a *colouring function*, which assigns to every state some natural number $F : Q \times \mathbb{N}$. Then, ρ is accepted iff

$$\max\{F(q) | q \in \inf(\rho)\} \text{ is even}$$

All these acceptance condition are equivalent: any (Muller,Rabin,Streett,Parity)-automaton can be turned into a (Muller,Rabin,Streett,Parity)-automaton accepting the same language. Furthermore, they can be determinized and they are all equivalent to NBA. When we want to refer to some ω-automaton with a generic acceptance condition, we will write $\langle \Sigma, Q, Q_0, \delta, \Omega \rangle$ and detail what Ω is.

Finally, we introduce the notion of a *weak automaton*. A NBA is weak if its states can be partitioned into $Q_1, \ldots, Q_n$ such that,

1. every partition contains either only accepting states or only non-accepting states:
$$\forall i : 1 \leq i \leq n : Q_i \subseteq F \text{ or } Q_i \cap F_{=}\emptyset,$$
2. whenever a transition is followed in partition Q_i, it can only lead to partitions with higher indices:
$$\forall i, j : 1 \leq i, j \leq n : \forall a \in \Sigma : \forall q \in Q_i : \forall q' \in Q_j : \delta(q, a, q') \implies i \leq j.$$

A weak automaton is linear if state classes are singleton. In a linear automaton, the transition relation induces a partial ordering on states.

1.2 Temporal Logic

Linear time temporal logic (LTL) can be used to describe infinite words [51].

Definition 1.2 (Linear Temporal Logic (LTL)) Given an alphabet Σ, the set of LTL formulae is the smallest fixed point satisfying the following constraints

- a is an LTL formula, with $a \in \Sigma$,
- if ϕ_1 and ϕ_2 are LTL formulae, $\phi_1 \vee \phi_2$ is an LTL formula,
- if ϕ is an LTL formula, $\neg\phi$ is an LTL formula,
- if ϕ_1 and ϕ_2 are LTL formulae, $\phi_1 \,\mathcal{U}\, \phi_2$ is an LTL formula,
- if ϕ_1 is an LTL formula, $\bigcirc\phi_1$ is an LTL formula.

Infinite words ($\gamma \in \Sigma^\omega$) are interpretations of LTL formulae. An infinite word $\gamma = e_0 e_1 \ldots \in \Sigma^\omega$ is a *model* of an LTL formula φ, written $\gamma \models \varphi$ iff

- if $\varphi = a$ then $e_0 = a$;
- if $\varphi = \varphi_1 \vee \varphi_2$ then $\gamma \models \varphi_1$ or $\gamma \models \varphi_2$;
- if $\varphi = \neg\varphi_1$ then $\gamma \not\models \varphi_1$;
- if $\varphi = \bigcirc\varphi_1$ then $e_1 e_2 \ldots \models \varphi_1$;
- if $\varphi = \varphi_1 \mathcal{U} \varphi_2$ then

$$\exists j \geq 0 : (\forall i \in \{0, \ldots, j-1\} : e_i e_{i+1} \ldots \models \varphi_1) \text{ and } e_j e_{j=1} \ldots \models \varphi_2$$

■

As usual, we define the following abbreviations:

- $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$;
- $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$;
- $\top \equiv a \rightarrow a$, for some $a \in \mathbb{P}$;
- $\bot \equiv \neg\top$;
- $\Diamond\varphi_1 \equiv \top \mathcal{U} \varphi_1$;
- $\Box\varphi_1 \equiv \neg\Diamond\neg\varphi_1$.

An LTL formula φ defines the following ω-language

$$\mathscr{L}(\varphi) = \{\gamma \in \Sigma^\omega | \gamma \models \varphi\}.$$

The class LTL contains all languages that are definable by LTL formulae. ω-automata provide us with a means to check the satisfiability of LTL formulae. Indeed, from every LTL formula φ, it is possible to build an NBA $\mathcal{A}_\varphi$ such that [171]

$$\mathscr{L}(\mathcal{A}_\varphi) = \mathscr{L}(\varphi).$$

This approach, called the tableau method, is at the heart of model checking techniques, see [51, 133, 38]. However, the converse is not true: not every NBA can be turned into an LTL formula. Indeed, LTL is equivalent to *counter-free automata* [125, 93]. Thus, LTL is not as expressive as NBA but can be extended to enjoy this property [181].

A *counter-free* automaton is an automaton which cannot count modulo k, for any $k \geq 1$. Formally, an automaton has a counter if

1. it has only reachable and non-dead states;

2. we can find some word u and some sequence of states $q_0 \dots q_k$ such that, from two different states in it, different ω-words can be accepted and reading u from q_i leads to $q_{i+1 \mod k}$.

This definition means that, from q_k, we can read an odd number of u and reach q_0. From q_0, some word w can be recognized that cannot be recognized from q_k. Thus, this automaton can count modulo. An automaton is counter-free if it has no counter.

1.3 Infinite games

In this thesis, we will make use of the theory of infinite games [63]. Suppose that there are two players: player 0 and player 1. These two players are playing over deterministic automata. Player 0 tries to find accepting runs of the automaton, whereas player 1 has the opposite objective. States are partitioned and shared out among players. Players move a pebble from state to state, according to the transition relation. Of course, player 0 may only move the pebble when it lies in one of his states. The game goes on forever, i.e. generates an infinite run of the automaton. Player 0 wins if it is accepted by the automaton, depending on the precise acceptance condition considered.

Formally, a *game graph* is a deterministic automaton

$$G = \langle \Sigma, V, V_0, q_0, \Delta, \Omega \rangle,$$

where $V_0 \subseteq V$ are flagged as player 0's states. When discussing games, we will often refer to states as *vertices*. We denote $V_1 = V \setminus V_0$. We will assume that every state in G has a successor.

A *play* is an infinite run in G. A play is winning in G if it is accepted, wrt Ω.

A *strategy* is a function, assigning to a finite play, a successor: $f : V^+ \to V$ such that

$$\forall w \in V^* : \forall q \in V_0 : \exists a \in \Sigma : (q, a, f(wq)) \in \Delta.$$

A *positional* strategy only uses the *last* state to make its decision: f is positional iff $\forall w \in V^* : \forall q \in V : f(wq) = f(q)$. If we fix a strategy for both players f_0, f_1, their *joint play* from state q yields a run $f_0 \circ_q f_1 = q_0 q_1 \dots$ such that every event is decided according to the strategy of the player which is up to play at position i:

$$q_0 = q \wedge \forall i > 0 : q_{i-1} \in V_j \implies q_i = f_j(q_0 \dots q_{i-1})$$

The *outcome* of a strategy f_j for player j from state q is the set of runs

$$Out(f_j, q) = \{f_J \circ_q g | g \text{ is a strategy for player } 1 - j\}.$$

Remember that a play is winning if it is an accepting run in G, wrt Ω. Following this, a strategy is winning in state q if all its outcomes from state q

are winning. The *winning region* of player j is

$$W_j = \{q | \text{Player } j \text{ has a winning strategy from } q\}.$$

The acceptance condition Ω determines whether a play is winning or losing. Therefore, we call it *winning condition*. If Ω is a Streett condition, the game is said to be a Streett game, etc.

Clearly, W_0 and W_1 are disjoint, as both players may not have a winning strategy from the same state. It is not obvious that winning regions partition the set of vertices $W_0 \cup W_1 = V$. This property is called *determinacy*. It asserts that, from every state, there is always a winning strategy, for one of the two players. In our case, determinacy is guaranteed by Martin's theorem, which states that games whose acceptance conditions correspond to Borel sets are determined [120]. Roughly speaking, Borel sets are the sets that can be constructed from open or closed sets by repeatedly taking countable unions and intersections [175]. Since ω-regular sets are clearly Borel sets, every game that we will consider in this thesis will be determined.

Although they are determined, our games still look very difficult. Indeed, deciding membership of states in winning regions amounts to deciding nontrivial properties about functions. Knowing that there *exists* a winning strategy for one player from every state q does not tell anything about our ability to *construct* it. In particular, this strategy might even be uncomputable. However, for parity games, positional strategies are sufficient.

Theorem 1.3 (Parity games) If G is a parity game, then if there is a winning strategy for player 0 from q, there is also a winning *positional* strategy. [63] ■

Winning regions become computable, because there is only a finite number of positional strategies! One can decide membership to W_0 (and W_1, obviously) in parity games. By the equivalence of the various winning conditions, this is also true of other acceptance conditions; still, there might be some blow-up involved in the reduction from an acceptance condition to another.

Theorem 1.4 (Parity games (complexity)) It is possible to solve parity games in time $\mathcal{O}\left(|\Delta| \cdot \left(\frac{|V|}{d}\right)^d\right)$, where d is the number of colours in the game graph. [92, 189]. ■

1.4 Computational Complexity

1.4.1 Problems and Algorithms

This thesis is about modeling languages and their automated analysis. We will encounter several questions of the type "does a model enjoy a certain property?". We will provide algorithms solving these questions. Nevertheless, most of these algorithms are inefficient. Naturally, one then wonders whether this inefficiency

comes from badly designed solutions, or from the very nature of the problem at hand. We will spend quite some time demonstrating that we are struggling with difficult problems and thus, we will need a framework for assessing problem hardness.

The theory of computational complexity provides us with such a framework [132]. It defines precisely the concepts of "problem", "algorithmic solution", "efficiency", and "hardness". This theory has proven very valuable to effectively separate easy problems from difficult problems. Here, we will follow the presentation given in [34]. It focuses on the intersection between formal languages, as presented in Section 1.1, and complexity theory.

The first concept we will need to define is a *problem*. In order to have a unifying framework, we will restrict ourselves to language decision problems. Formally, a decision problem in our setting will be of the form $P(x) \equiv x \in L$, for some language $L \subseteq \Sigma^*$. Here, Σ is a predefined alphabet, which we assume fixed for the rest of this section. In summary, we are only interested in queries of the form "does some word u belong to L?". Of course, we are only interested in computer-based problem solving. Thus, we want our notion of *solution to a problem* to be algorithmic, that is to provide "a systematic method enabling a computer to solve a problem". There are many ways to describe algorithms and, again, we need some unifying framework.

In line with literature, we define an algorithm as a Turing Machine. We thus adopt Church's thesis, that all computing devices are equivalent in power with Turing Machines. Such a machine is an extremely operational device, having a finite set of states. It uses as a memory a long thin tape, divided into small cells, each of them containing exactly one symbol. The machine has a tape head, which stays on one cell at a time. The machine starts its execution with the tape head on the leftmost cell. At every step, it scans the symbol under the tape head and, depending on the value read and its current control state, performs one of three operations:

1. it replaces the symbol under the tape head by a new symbol of its choice;
2. it moves the tape head to the next cell, either to the left or to the right;
3. it changes the control state to a new state.

Then, it proceeds to a new step. Some control states are special; they are marked as halting states. Reaching one of them results in stopping the machine. Those halting states are partitioned into "yes" and "no" states. The former mean that the machine halts successfully (it accepts its input) while the latter mean that the machine halts in error (it rejects its input).

Definition 1.5 (Turing Machine) A (nondeterministic) *Turing Machine* (TM) is a tuple

$$M = \langle Q, q_0, \Sigma, \Delta, Y, N \rangle,$$

with

Q is a finite set of *control states* ;

$q_0 \in Q$ is an initial control state;

Σ is a finite alphabet;

$\Delta \subseteq Q \times \Sigma \cup \{\sqcup\} \times \{\mathbf{l}, \mathbf{r}\} \times \Sigma \cup \{\sqcup\} \times Q$ is a transition relation;

$Y \subseteq Q$ is a set of *accepting states*.

$N \subseteq Q$ is a set of *rejecting states*. Accepting and rejecting states must be disjoint: $Y \cap N = \emptyset$.

■

A TM configuration is a triple (q, T, i) made of a control state $q \in Q$, some tape content $T \in (\Sigma \cup \{\sqcup\})^*$ and a tape head cursor $i \in [|T|]$. We say that a TM M makes a step from (q, T, i) to (q', T', i'), and denote it with $(q, T, i) \rightarrow_M (q', T', i')$ iff there is a transition $(q, a, m, b, q') \in \Delta$ such that

1. $q \notin Y \cup N$, the current control state is not a halting state,
2. if $m = \mathbf{l}$, then $i > 0$.
3. if $m = \mathbf{l}$, then $i' = i - 1$, otherwise, $m = \mathbf{r}$ and $i' = i + 1$.
4. $T = uav$, with $|u| = i$.
5. If $v = \epsilon$ and $m = \mathbf{r}$, then $T' = \sqcup$. Otherwise, $T' = ubv$.

Definition 1.6 (Deterministic Turing Machine) A *deterministic TM* (DTM) is a TM, with the additional constraint that Δ must be functional on $Q \times \Sigma$ (its first two arguments). In words, the move of a DTM is entirely determined by the current control state and the symbol scanned. ■

The initial configuration of M on input $x \in \Sigma^*$ is $(q_0, x, 0)$, i.e. the control state is the distinguished initial state (q_0), the tape contains only the input (x) and the tape head is at the leftmost position (0). A computation of M on x is a finite or infinite sequence of configurations $C = \Gamma_0\Gamma_1 \ldots$, starting with the initial configuration on input x and following the step relation: $\Gamma_i \rightarrow_M \Gamma_{i+1}$, for every $0 \leq i \leq |C|$. We let $|C| = \omega$ if C is infinite. A *halting computation* is a *finite* computation ending in a configuration with a halting control state. Formally, the last configuration $\Gamma_{|C|+1} = (q_H, T, i)$, with $q_H \in Y \cup N$. Among halting computations, we will distinguish *accepting computations*, ending with a control state in Y and *rejecting computations*, ending in N. A TM accepts some word u iff it has some accepting computation on u. It rejects u iff *all* its computations on u are rejecting. Remark that some computations are neither accepting nor rejecting, they are simply diverging.

1.4.2 Efficiency: time, space and complexity classes

The time of a computation C is the number of steps involved in C: $T_M(C) = |C| - 1$, if C is finite, and $T_M(C) = \infty$, otherwise. The working space of a computation C, $S_M(C)$, is the difference between the longest tape's length in all configurations of the computation and in the initial configuration, if C is finite. We let $S_M(C) = \infty$, otherwise.

As in Section 1.1, we can define the language accepted by a Turing Machine, as

$$\mathscr{L}(M) = \{u \in \Sigma^* | M \text{ has an accepting computation on } u\}.$$

The class of all languages accepted by a Turing Machine is the class of *recursively enumerable* (r.e.) languages.

$$\mathsf{r.e.} = \{\mathscr{L}(M) | M \in \mathrm{TM}\}$$

This class is not closed under complement. For instance, the language of all TM which do not halt on some input is not in this class, whereas the set of halting TM is in r.e..

There is a subset of this class, in which diverging machines are ruled out, namely the class of recursive languages. A Turing Machine *decides* a language L iff, for every $u \in \Sigma^*$,

- if $u \in L$, then M accepts u;
- if $u \notin L$, then M rejects u.

A language L is *recursive* if it is decided by some Turing Machine. The decision problem associated with some language L ($P(x) \equiv x \in L$?) will be called *decidable* if L is recursive. A function on strings $f : \Sigma^* \to \Sigma^*$ is *computable* if its table, seen as a language, is decidable. That is, $\{u\$f(u) | u \in \Sigma^*\}$ is recursive. We extend the alphabet with a separator, i.e. some fresh simple symbol \$, the role of which is separating u from its image in the TM input.

When assessing the efficiency of a TM, we will accept some rough estimates. Basically, we are only interested in asymptotical behaviour of their resource consumption. Actually, our goal is to compare problems, and see if a problem is much more difficult than another. For this purpose, we introduce the "big-oh" notation. Let f and g be functions from $\mathbb{N}$ to $\mathbb{N}$. Then, $f(n) = \mathcal{O}(g(n))$, if there are positive integers c and n_0 such that, for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$. Remark that saying that $f(n) = \mathcal{O}(g(n))$ means that $f(n)$ and $g(n)$ have the same rate of growth asymptotically, or that they only differ by some linear factor. We write $f(n) = \Omega(g(n))$ if the opposite happens, i.e. $g(n) = \mathcal{O}(f(n))$. If $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$, we write $f(n) = \Theta(g(n))$.

A function $f : \mathbb{N} \to \mathbb{N}$ is time-constructible if there is a DTM M deciding the following language: $\{0^n \cdot 0^{f(n)} | n \in \mathbb{N}\}$ and $T_M(n) = \mathcal{O}(f(n))$. It is space-constructible if there is a DTM M deciding the same language, and $S_M(n) \leq s(n)$. Following usual conventions [132], we will only consider time-constructible and space-constructible functions when defining complexity classes.

A TM M decides a language L in time $f(n)$, where f is a function from nonnegative integers to nonnegative integers if M decides L and furthermore, for every $u \in \Sigma^*$ and every computation C of M on u, $T_M(C) \leq f(|u|)$. The class of languages decided by TM within time f is a complexity class, denoted

$$\begin{aligned} \mathsf{NTIME}(f(n)) &= \{L \mid \exists M \in \mathrm{TM} : M \text{ decides } L \text{ within time } f(n)\} \\ \mathsf{TIME}(f(n)) &= \{L \mid \exists M \in \mathrm{DTM} : M \text{ decides } L \text{ within time } f(n)\} \end{aligned}$$

The same definitions can be applied to space complexity, yielding the following sets of languages:

$$\begin{aligned} \mathsf{NSPACE}(s(n)) &= \{L \mid \exists M \in \mathrm{TM} : M \text{ decides } L \text{ within space } s(n)\} \\ \mathsf{SPACE}(s(n)) &= \{L \mid \exists M \in \mathrm{DTM} : M \text{ decides } L \text{ within space } s(n)\} \end{aligned}$$

It is quite important to note that time and space complexities differing only by some linear constant do not really matter, once we consider complexity classes. This justifies our choice of discarding such differences in analysis and rather resort to the "big-oh" notation described above.

Theorem 1.7 (Linear Speed-Up) Let c be a positive real number.

1. Let $s : \mathbb{N} \to \mathbb{N}$ be a function. Then
$$\mathsf{SPACE}(s(n)) = \mathsf{SPACE}(c \cdot s(n)).$$
2. Let $t : \mathbb{N} \to \mathbb{N}$ be a function such that $\lim_{n\to\infty} \frac{t(n)}{n} = 0$. Then,
$$\mathsf{TIME}(t(n)) = \mathsf{TIME}(c \cdot t(n)).$$

■

We now define several standard complexity classes and the straightforward inclusions, coming from the fact that deterministic machines are also nondeterministic ones (DTM $\subset$ TM).

$$\begin{array}{rclcrcl} \mathsf{P} &=& \bigcup_{k>0} \mathsf{TIME}\left(n^k\right) & \subseteq & \mathsf{NP} &=& \bigcup_{k>0} \mathsf{NTIME}\left(n^k\right) \\ \mathsf{EXPTIME} &=& \bigcup_{k>0} \mathsf{TIME}\left(2^{n^k}\right) & \subseteq & \mathsf{NEXPTIME} &=& \bigcup_{k>0} \mathsf{NTIME}\left(2^{n^k}\right) \\ \mathsf{PSPACE} &=& \bigcup_{k>0} \mathsf{SPACE}\left(n^k\right) & \subseteq & \mathsf{NPSPACE} &=& \bigcup_{k>0} \mathsf{NSPACE}\left(n^k\right) \end{array}$$

Only decision problems in P are considered tractable. Furthermore, it is proven that $\mathsf{P} \subset \mathsf{EXPTIME}$. Therefore, $\mathsf{EXPTIME}$ problems are viewed as really intractable.

We will make use of the "co" version of these classes. A language is in the "co" version of some complexity class C, i.e. coC, if its complement is in C. Thus,

$$\begin{aligned} \mathsf{coNP} &= \{L | \Sigma^* \setminus L \in \mathsf{NP}\} \\ \mathsf{coPSPACE} &= \{L | \Sigma^* \setminus L \in \mathsf{PSPACE}\} \end{aligned}$$

Deterministic classes are closed under complement: coPSPACE = PSPACE, coEXPTIME = EXPTIME and coP = P.

Savitch's theorem states that for all space complexity classes above logarithmic space, non-deterministic machines can be simulated by deterministic machines, with only a quadratic increase in space.

Theorem 1.8 Let $f(n) \geq \log_2 n$ be a space-constructible function, then

$$\mathsf{NSPACE}(s(n)) \subseteq \mathsf{SPACE}(s(n)^2).$$

Therefore, from $\mathsf{SPACE}(s(n)^c) = \mathsf{SPACE}(s(n))$, it follows that

$$\mathsf{SPACE}(s(n)) = \mathsf{NSPACE}(s(n))$$

■

We end up this short walkthrough of complexity classes by relating non-deterministic time classes and space classes:

$$\mathsf{NP} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE} = \mathsf{coPSPACE} \subseteq \mathsf{EXPTIME}$$

So far, we have defined the concept of problem, as a question of the form "does a word belong to some language?", the concept of algorithmic solution, as a Turing Machine deciding this problem, and techniques for measuring the efficiency of this TM, with respect to two resources, namely time and space. We have also introduced complexity classes, as a means for classifying problems. We still need a technique for comparing problems and assessing their hardness. In fact, we do not have any mathematical method for proving higher than quadratic lower bounds on the time of restricted computing models, as we are using here. Thus, we cannot really classify problems because we are unable to obtain tight lower bounds on their complexity [34]. Cook has introduced a method enabling one to prove *relative* lower bounds [41]. The idea is to show that a problem P_1 is at least as hard as another problem P_2. If P_1 turns out to be tractable, P_2 will also be.

Definition 1.9 (Polynomial-time reduction) Let $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ be two languages. We say that L_1 is *polynomial-time reducible* to L_2, denoted $L_1 \leq_p L_2$, if there exists some function $f : \Sigma^* \to \Sigma^*$ such that

1. f is computable in polynomial-time ($f \in \mathsf{P}$) and logarithmic space ($f \in \mathsf{LOGSPACE}$);
2. $u \in L_1 \iff f(u) \in L_2$.

■

Using this notion, we can find problems which are as difficult as all problems of some complexity class. A language L is hard for some class C if, for every $L' \in \mathsf{C}$, $L' \leq_p L$. A language L is complete for C if $L \in \mathsf{C}$, in addition to being hard for C.

1.5 Alternation

Obviously, the reader will have noticed that Turing Machines are a flavour of automata, with an unbounded memory (finite automata have no memory, only control locations) and a tape head reading the input in a versatile way (finite automata must read the input left-to-right and accept/reject as soon as they reach the input end). Thus, we can say that a finite automaton *has a computation on some word* $w \in \Sigma^*$.

The concept of non-determinism has already been presented, several times in this section, in the framework of automata and of Turing Machines. We have also introduced two types of acceptance, depending on two different interpretations of non-determinism. The first is called "existential acceptance" and is the usual acceptation of non-determinism: "w is accepted iff there is some run on w that leads to an accepting configuration". The second is named "universal acceptance": "w is accepted iff all executions on a word w leads to some accepting configuration".

These two views (universal/existential) can be generalized, in order to nest quantifiers along a computation. This leads to *alternating machines*. An alternating TM is a TM, except that its set of control locations are partitioned into *universal* and *existential* locations. A computation starting from a configuration with an existential location is accepting if there is an accepting computation starting from one of its successor configurations. A computation starting from a universal location accepts if *all* computations starting from successor locations accept, too. Thus, a computation of an alternating TM can be seen as a tree. Nodes are TM configurations and a node n' is a child of some node n iff n' is a successor of n. Existential nodes have exactly one child, whereas universal nodes have as many children as they have successor configurations. Such a computation tree is accepted if all branches end with an accepting configuration. This concept was introduced in [36]. The intuitive formulation above is not adequate and harder to formalize, because one has to take diverging computations into account. In the initial paper, this is done by using a three-valued logics for labeling configurations, as well as labeling functions. The correct labeling is then defined as the least fix-point of a transformation, which propagates acceptance and rejection "backwards".

Of course, it is possible to lift the concept of time and space resources consumed by a computation (T_M) to computation trees: "a computation tree C takes time n iff the longest branch in C has length n." Similarly, for space resources, a computation tree C takes space n if no node in C uses more than n tape cells, ignoring input cells. Then, all other concepts presented in section 1.4.2 can be reused. Complexity classes are of course adapted to the alternating case. For instance, APSPACE represents the class of all languages recognized by *alternating* Turing Machine using polynomial space.

Chandra, Kozen and Stockmeyer have proven that the deterministic hierarchy shifts by exactly one level when alternation is introduced [36].

Theorem 1.10

$$\begin{aligned} \mathsf{EXPSPACE} &= \mathsf{AEXPTIME} \\ \mathsf{EXPTIME} &= \mathsf{APSPACE} \\ \mathsf{PSPACE} &= \mathsf{APTIME} \\ \mathsf{P} &= \mathsf{ALOGSPACE} \end{aligned}$$

■

Alternation can be applied to finite automata over finite and infinite words as well. In this thesis, we will use a simpler kind of alternating finite automata, over infinite words, called *Alternating Linear Automata* (ALA). As their names hints to, they are linear automata (see Sec. 1.1.2), extended with alternation. Every LTL formula φ can be translated to some ALA Büchi $\mathcal{A}_\varphi$ recognizing all models of φ and having as many states as φ has syntactic operators.

Chapter 2

Scenario-based Languages: a Walkthrough

Contents

Un bon dessin vaut surtout mieux qu'un mauvais dessin.

Le Chat, P. Geluck

2.1 Introduction

The topic of this thesis is the formal specification of reactive distributed systems using scenarios. In this chapter, we present several scenario-based languages. It is clearly out of the scope of the thesis to survey all of them. A survey of scenario-based languages for telecommunication usage is available in [14].

The purpose of this chapter is to introduce scenario languages and illustrate that Live Sequence Charts, the language on which we will focus in the rest of the thesis, is a state-of-the-art scenario-based specification language. It features most interesting elements that can be found in mainstream languages and is completely formal. At the end of this chapter, the reader should be familiar with the general idea of scenario-based specification and have a good intuition of Live Sequence Charts.

We start with Message Sequence Charts. It is widespread in industry, especially in Telecommunication and has found its way into Object Management Group's Unified Modeling Language (OMG's UML), where it is known as Interaction Diagrams. In previous UML versions, they were called Sequence Diagrams. We discuss some shortcomings of MSCs, namely their weak semantics, the ambiguity of scenario composition, their lack of message abstraction and of scenario modality. We then present an older language, namely Zave's Sequence Diagrams, which is not a Sequence Chart language but features message abstraction and a well-defined scenario composition operator. We also present Use Case Maps and Requirement/Design Behaviour Trees, which form the core of the "genetic design" approach [47]. We conclude with the presentation of Live Sequence Charts.

2.2 Message Sequence Charts

2.2.1 Context

The language of Message Sequence Charts (MSC) stems from industrial practice in Telecommunication. Engineers have used variants of it for years to communicate protocols, as a "back-of-the-envelope" notation [150]. The International Telecommunication Union started working on its standardization in 1992 and issued recommendation Z.120 that defined officially MSCs in 1996 [90], including all syntactical elements that were needed by engineers. A revised version of the standard is published every four years [169]. In 1998, a subset of the language, namely bMSC (basic Message Sequence Charts) without data, was given a formal semantics by Cobben, Engels, Mauw and Reniers [40, 122, 121]. This semantics was given thanks to a mapping to a process algebra, designed especially to take the special features of MSCs into account. It has been later extended to deal with data and conditions [131, 53, 121, 58]. These semantics all consider interleaving as a model of concurrency. Other approaches are based on Petri nets [64, 85], considering directly partial order families [84, 94] or event structures [81]. The reader is referred to several surveys on MSCs to obtain more information and pointers on this language [123, 80, 128].

MSC is the most widespread scenario language. It forms the root of a whole family tree of languages, which we shall simply refer to as *sequence charts languages*. For instance, they have found their way into OMG's UML, where they mutated into *Interaction Diagrams* [130]. They can also be found in FIPA's Agent Modeling Language [89].

MSCs have also been extended by researchers to provide new features. Among these extensions, let us cite Triggered Message Sequence Charts [148], Compositional Message Sequence Charts [67], Message Sequence Charts with variation points, which are used in Software Product-Line Engineering [188], Concurrent Transaction Processes [141], Shared Variable Interaction Diagrams [8], Hybrid Sequence Charts [65], Template Message Sequence Charts [59] and Live Sequence Charts [42], which are presented in Section 2.6.

2.2.2 Basic Message Sequence Charts (bMSC)

Message Sequence Charts are aimed at describing process interactions. Since they stemmed from the telecommunication world, processes are assumed to communicate asynchronously, through message passing. This corresponds to the actual communication situation in which physically distant devices send messages to each other via buffered order-preserving channels, namely FIFO queues. MSC provides a means to represent such processes, as *instance axes*. There are three instance axes in the bMSC of Fig. 2.1 corresponding to three processes (or instances) named P_1, P_2 and P_3. Messages are represented as downward-sloping or horizontal arrows. They are made of two events: a sending and a receiving event. Communication is asynchronous: sending and receiving events do not take place at the same moment. In Fig. 2.1, P_1 sends message a to P_2, for instance. Messages can be sent to or received from "the environment".

These are shown as arrows going to or coming from the border of the chart, see c in Fig. 2.1. Lost, i.e. sent but never received, and found, i.e. appearing out of nowhere, messages can also be represented. For example, in Fig. 2.1, d and e are lost and found messages, respectively. Internal atomic actions can be represented as boxes containing an action name, see 'act' in Fig. 2.1. Action labels can be arbitrary natural language sentences or valid expressions in a chosen formal "data language".

The semantics of bMSCs, as formally specified in [40] and informally described in [169, 166, 90], imposes only two constraints on the ordering of events (i.e. sending and receiving messages):

1. events from the same instance axis are ordered from top to bottom,
2. a message must be sent before it is received.

Hence, the event "receive c" must occur before "send b". "Send a" shall happen before "receive a", but "send a" and "send b" are not ordered with the rules above and can thus occur in any order.

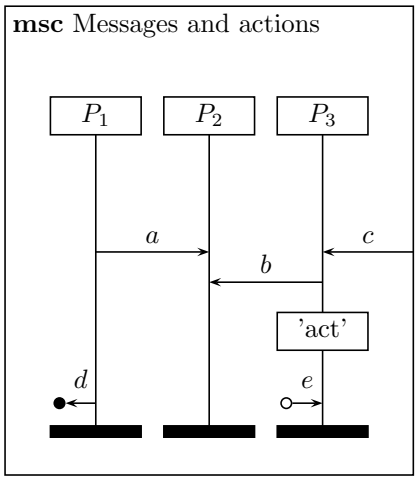


Figure 2.1: Basic Message Sequence Chart (bMSC)

One can use *guarding* and *setting* conditions, see Fig. 2.2. The former is used to restrict the possible behaviours: the guard must be true in order to proceed to the considered section. The latter is used to illustrate in which state the system is, when it reaches that point. They can be used to state how scenarios are glued together; a scenario starting with a guarding condition X can be appended to a scenario ending with a setting condition X [58]. Guarding conditions start with the keyword `when`.

MSC-2000 introduced the ability to model *method calls*, as presented in Fig. 2.3 [166]. The essential difference between message passing and method calls is that the latter conveys the intent that the message's receiver has to perform some computation, on behalf of the sender. The sender is also assumed to be suspended until the method returns, unless it is re-activated by another

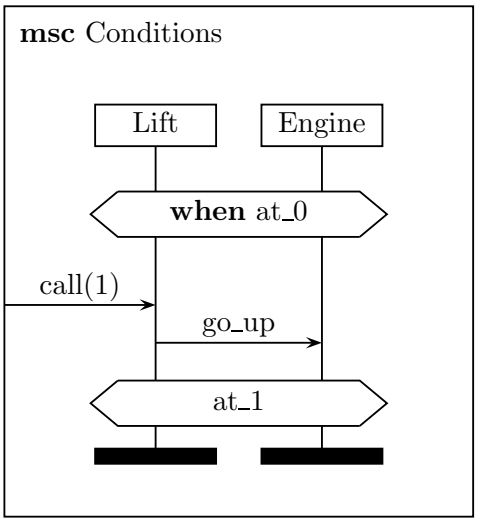


Figure 2.2: bMSC with conditions

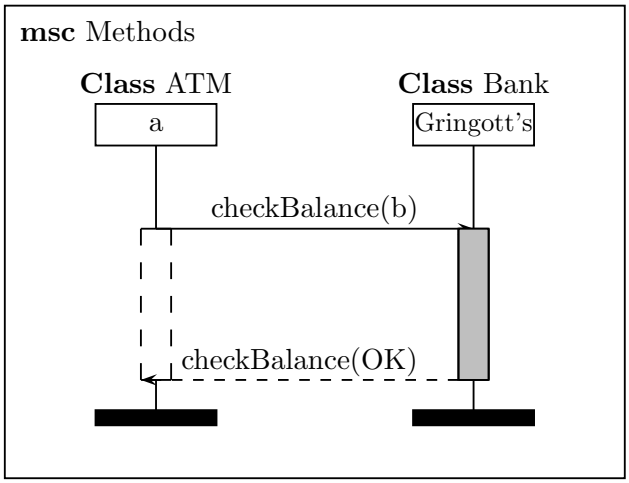


Figure 2.3: bMSC with methods

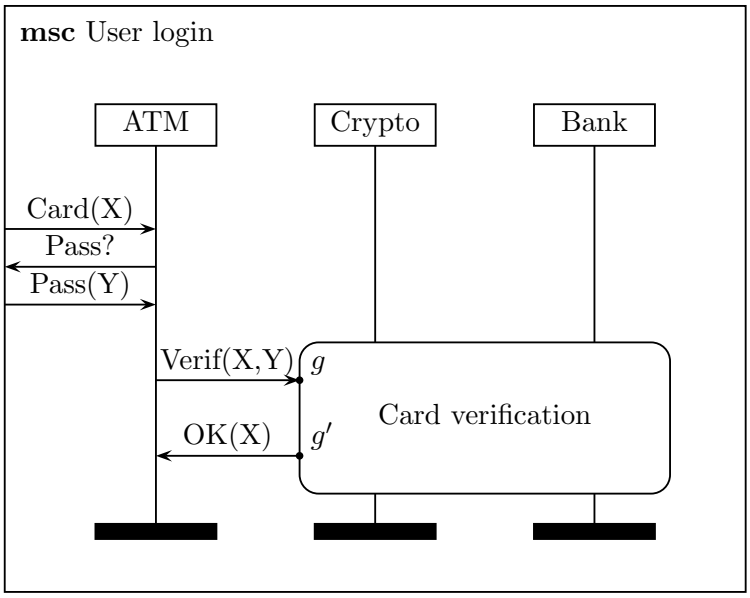


Figure 2.4: bMSC with references and gates

re-entering method call, or the method call is asynchronous. In this case, no suspension region is indicated along the sender's instance axis. MSC-2000 adds a syntactical rule stating that no event can be drawn within a suspension region, therefore ensuring that the process remains idle until its method call returns.

In MSC-2000, MSCs can be refined, by referring to other MSCs. Interfaces to these referred MSCs are defined through named *gates*, see Fig. 2.4, in which there are two gates: g and g'. The corresponding gates in the detailed sub-scenario will hold the same names and be depicted as messages sent by/to the environment. The scenario named "Card verification" is detailed in Fig. 2.5.

When describing complex behaviour, it is interesting to decompose the problem at hand, in order to structure reasoning and better cope with its complexity. Structuring constructs allow one to do so. Basic control flow constructs, such as alternative, optional parts, parallelism (as interleaving) and loops, are thus available in MSCs, either as "inline expressions", i.e. boxes put in the MSC itself like in Fig. 2.5, or as hierarchical graphs (i.e. nodes can contain graphs or be labeled with simple bMSCs), called High-Level MSCs (hMSC). They will be studied in the next section.

2.2.3 High-Level Message Sequence Charts (hMSC)

High-Level Message Sequence Charts (hMSCs) form an alternative to inline expressions for building structured specifications from bMSCs. A hMSC is a graph as illustrated by Fig. 2.6. It contains a "start symbol", a reversed triangle, and an "end symbol", which is a triangle standing on its base. Its basic elements are conditions and nodes, that can refer to bMSCs or other hMSCs. Nevertheless, one can consider only two-level hMSCs: hMSCs at the top and

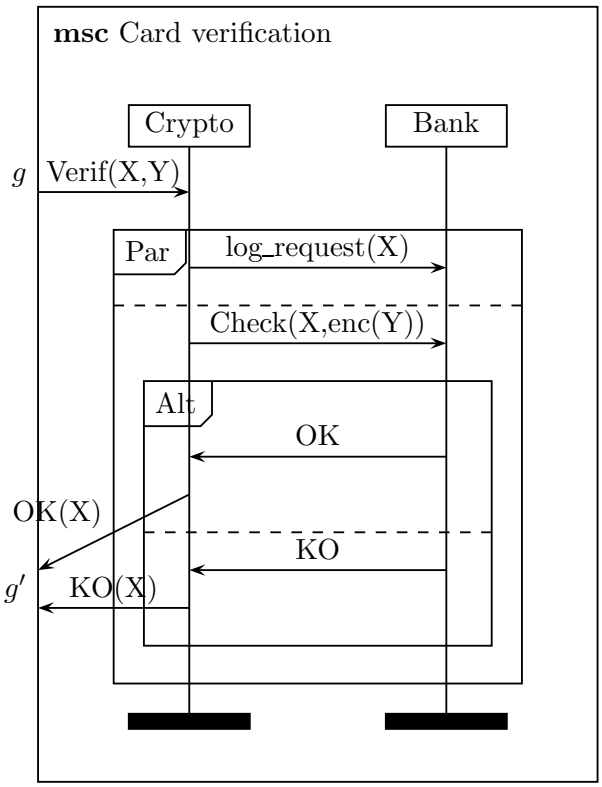


Figure 2.5: bMSC with structuring constructs

nodes containing bMSCs only, without loss of generality. An "execution" of a hMSC is a path from its start symbol to one of its end symbols. Such a finite path sequentially traverses nodes, defining a sequence, say $N_1 \dots N_m$. This execution defines the concatenation of all these bMSCs. The semantics of a hMSC is the set of all concatenations of bMSCs that are defined by some path in the hMSC.

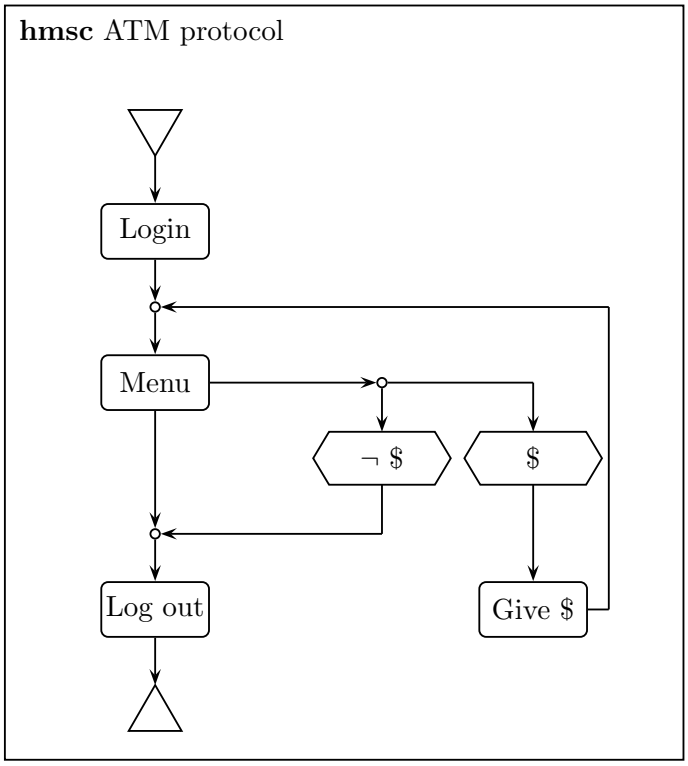


Figure 2.6: High-Level Message Sequence Chart

The question arises as to how bMSCs should be sequentially composed. There are basically two ways to proceed:

strong sequence: the first scenario must complete before the second one starts. Note that in general an instance has *not* enough knowledge to decide whether the scenario has completed. Synchronization messages are thus needed to implement (in a distributed fashion) this synchronization;

weak sequence: instances can proceed to the second scenario whenever they have *locally* completed the first one. This is illustrated in Fig. 2.8. It corresponds to the local concatenation of partial orders defined by Pratt [136].

The MSC standard prescribes the latter interpretation [40, 90]. To illustrate the difference between the two interpretations, remark that weakly sequencing seq_1 and seq_2 from Fig. 2.7 yields the bMSC of Fig. 2.8. In this bMSC, P_1 may proceed to seq_2 and send c to P_2 before P_2 has received b.

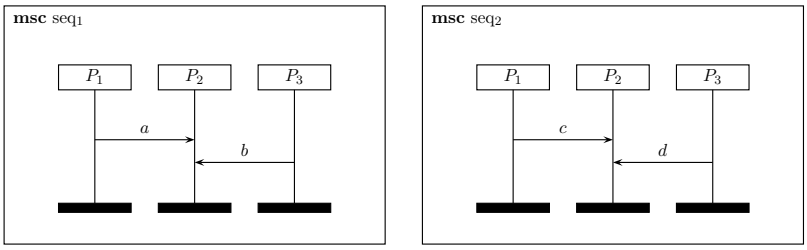


Figure 2.7: Sequence of bMSCs

There are two motivations for choosing weak sequence. The first is that the concatenation of two bMSCs under weak interpretation still forms a bMSC; bMSCs are thus *closed* under concatenation. On the contrary, the strong concatenation of two bMSCs might yield an object which is not a bMSC. Second, it seems to make the derivation of a distributed program easier, as instances do not need to synchronize on the end of bMSCs. At first sight, it seems that projecting the (h)MSC onto its components and extracting a program following the resulting sequence of events is sufficient [185]. It is not the case because processes might locally lack some globally-available information. Actually, this discrepancy between the bird's eye view provided by sequence charts notations and the local specification of processes leads to a whole range of problems.

The first problem is called *non-local choice*. It was identified by Ben-Abdallah and Leue who provided syntactic criteria to detect it [18]. A choice between two scenarios is non-local if there is more than one instance that can choose which branch of the choice will be taken. A choice is thus local if only one instance is responsible for making it. The criterion for detecting non-local choice is essentially to ensure that all minimal events following a branch in an MSC belong to the same instance.

Even with this condition, strange behaviours can occur because although one instance chooses the branch to be taken, other instances have not enough information to know that this branch was picked. This problem gives rise to *implied scenarios*. Actually, the problem of implied scenarios encompasses the problem of non-local choice. An implied scenario is simply an execution that takes place when all processes act according to their local knowledge but this execution is not a legal execution of the hMSC. Implied scenarios were first discovered by Alur, Etessami and Yannakakis [6], who found syntactic rules to detect implied scenarios in a collection of MSCs. Uchitel investigated the problem of detecting implied scenarios and using them for behavioural requirements elaboration and synthesis, in the framework of hMSCs, with instantaneous communication [161, 164, 162, 163, 165]. Muccini provided an algorithm for detecting implied scenarios in hMSCs as well, based on the idea that implied scenarios were a generalization of non-local choice [126].

The problem of determining whether an MSC could be distributely implemented without generating new scenarios motivated researchers to find syntactic rules determining a subset of hMSCs for which this property would hold. Finkbeiner and Krüger came up with a sub-class named *causal MSCs*[55].

Hélouët identified conditions for which the synthesis of state machines is also feasible [79].

hMSCs describe languages that are not regular and their regularity is not decidable, either [82]. Hence, it is impossible to determine whether a set of finite-state automata can implement a given hMSC.

Finally, we mention another problem that may arise, as a result of the discrepancy between local and global information. In MSCs, it is assumed that processes use a single FIFO buffer to communicate with other processes. In this configuration, the order in which processes write in this queue is very important. For instance, the MSC of Fig. 2.1 is not distributable on this architecture. This is due to the fact that P_3 may only append b to P_2's buffer if P_1 has already posted a. However, P_3 cannot know this. Alur, Holzmann and Peled have defined another order, named *visual order* which is weaker than the causal order specified in ITU's recommendation [10]. The visual order states that an event e is smaller than an event e' if

1. e is a receiving event and e' is a sending event, with e' following directly, i.e. is drawn immediately beneath, e on the same instance axis,
2. e is a sending event and e' is a receiving event of the same message (arrow).
3. e and e' are sending events and e' follows directly e on the same instance axis.

The visual order is the smallest partial order containing these three rules. Alur, Holzmann and Peled exhibited a quadratic-time algorithm for verifying that the visual order is compliant with the causal order [10], thus, that an MSC is free of race conditions.

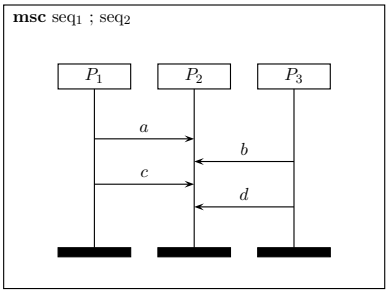


Figure 2.8: Weak sequence

2.2.4 Discussion

In this section, we discuss some shortcomings of MSCs. We insist on four particular shortcomings.

- The semantics of MSC is weak and aims at describing execution samples only. This is very different from *specifying* the expected behaviour of the future system.

- Composition of MSCs requires some very hard work, as the semantics is silent about the meaning of a set of scenarios, i.e. how the requirements they impose on the future system shall be composed.
- MSC lacks syntactic construct for expressing the *scope* of a scenario, i.e. whether events not appearing in this scenario can occur at will or are forbidden by their mere absence.
- MSC does not distinguish between events that *trigger* the scenario and events that occur *in response to* this activation, even though this is a usual informal distinction in scenarios.
- MSC has no syntactic means to distinguish between universal rules and examples.

Let us go through these points and detail them. A first precision we would like to make is that, by itself, the language is not bad. No language is. But, for our particular usage, namely specifying the behaviour of distributed reactive systems, they would not be the best choice. ITU's recommendation states explicitly that MSC describes "partial behaviour". MSC can therefore specify sets of examples only.

At least, specifying the behaviour of a reactive system does not amount to listing a set of examples. A specification should draw the border-line between admissible and inadmissible behaviour. Legal and illegal behaviour. Reasonable systems embed an infinity of virtual executions. How can one reasonably expect to describe this infinite set of executions as a finite set of examples?

Of course, one could resort to hMSCs or inline loops. Thus, analysts would have to build a single integrated behavioural model. Note that it is merely impossible to use *several* hMSCs to model this behaviour, because the language definition is completely silent about the meaning of a *set of hMSCs*. Nothing is said about how several scenarios shall be composed. A likely attempt is through union (of execution sets), but this is not different from a single hMSC with a top-most choice node. Building an integrated model requires finding out all relationships between individual scenarios, factoring and merging them. In this case, how could hMSCs support fully scenario-based approaches to specification? The essential appeal of the scenario-based paradigm is to proceed from unrelated, small and concrete requirements chunks, i.e. scenarios, and get the most out of them through composition. This allows one to deal with requirements creep; when customers come up with a fresh requirement, this new scenario is simply thrown in the "big bag of requirements", extending the specification to cope with these new wishes. It is of course possible to refine by hand the big hMSC model but it requires considering the whole "specification" again.

Therefore, one is left with two alternatives: either use sets of hMSCs or bMSCs to describe behaviour samples, benefit from a true scenario-based approach but losing the ability to *specify* the future system behaviour, or use a global integrated behavioural model, in the form of a hMSC, and lose (nearly) all benefits of scenarios.

Actually, the problem is even worse if one refers to the *intent* of analysts. Sometimes, they want to write down a possible behaviour of the future system, for the sake of explanation. Sometimes, they are more assertive and want to describe a *reaction* of the system to a certain stimulus. For example, the scenario of Fig. 2.4 was drawn with the intent of capturing a "sunny day" scenario: the customer inserts his card X, types in his code Y and the code is correct. Of course, everyone knows that there is also a "rainy day" scenario, in which the customer types a wrong code. In fig. 2.5, the intent was rather to state that *whenever* a customer card number X and a code Y were sent to the crypto module, the module had to write the request to some log, encode the password, ask the remote bank for verification and propagate the result backwards. There are no syntactical constructs in MSCs to make the distinction between these two different intents: examples and universal rules. In the latter case, there are no syntactic constructs to single out the *activation part*, neither.

We close this discussion section by highlighting that MSCs are not syntactically equipped to express their *scope*. The scope of a scenario is the set of events or phenomena that are concerned with this scenario. For instance, consider the ATM situation. Assume that we want to add a scenario taking into account the fact that the machine can send an alarm to the bank employee in charge when the level of bank notes is going below some threshold. This scenario shares absolutely no phenomenon with the scenario for withdrawing money from the machine. Now, consider the scenario in which the employee reloads the ATM with bank notes. Then, we do not want customers to take money at this moment. Thus, the event "withdraw money" is shared between these two scenarios, even if it does not appear explicitly in the second. It is impossible to state this with MSCs. Thus, one is unable to state if events not appearing explicitly in a scenario may occur while the scenario is executing or if they are forbidden by their mere absence.

In summary, MSC is nowadays the main language used for representing scenarios and has proven to be very intuitive. It is at the root of a whole family tree of languages: sequence chart languages. However, we argued that MSC is not suitable for the specification of reactive distributed systems, because of the following reasons:

- Its semantics is weak;
- Composition of scenarios is defective;
- It misses syntactic constructs to express the difference between examples and universal rules. It is impossible to distinguish events activating the scenario from events occurring in response to this activation;
- It is impossible to specify the scope of a scenario.

In an attempt to tackle these problems, the OMG has substantially extended the language of MSC in the UML 2.0 [130], where they are called Interaction Diagrams. The semantic domain of Interaction Diagram is a couple of sets of executions. When considering one of these couples, eg (G, B), G is the set of

"good executions", i.e. positive examples, and B is the set of "bad executions", i.e. counter-examples. G and B must be disjoint but not necessarily cover the universe of executions. Therefore, there are some executions about which it is not known if there are good or bad. They are called *inconclusive*. Interaction Diagrams introduce new operators: restrict, ignore, assert and negation. The first two operators specify the scope of a scenario, by adding/removing events from its scope. The "assert" operator means that only the traces described in the diagram are good; all other traces are bad. The "neg" operator means that the traces described are counter-examples. However, the UML standard is ambiguous about the meaning of these operators. The semantics of these operators is only defined with respect to good scenarios. The interpretation of negation is ambiguous as demonstrated in [35]. Furthermore, we showed that, under some reasonable assumptions, it is undecidable whether an Interaction Diagram is consistent, i.e. good and bad executions do not overlap [25].

2.3 Zave's Sequence Diagrams

2.3.1 Presentation

In 1985, Zave introduced a language for modeling the behaviour of reactive systems, called *Sequence Diagrams* (SD) [186]. This language is different from UML 1.4's sequence diagram language [129], although both languages belong to the scenario languages family.

Zave's SD provides several features that help analysts specify the behaviour of reactive systems. First, it is based on the idea of *decomposition*: the global behaviour is decomposed into sub-behaviours. These sub-behaviours can be sequentially composed, iterated or composed by choice. They can also be reused at other places in the specification, thus enabling analysts to factor out common sequences. Second, the distinction between input and output actions is made. The language specifies valid sequences of input events as well as appropriate system responses. Third, every scenario is given a scope: input events outside its scope are simply discarded. Fourth, the language is partly graphical. Fifth, different aspects of the system can be described. The language then provides a built-in meaning of composition. Zave advocates that, whith these features, the specification of distributed reactive systems is greatly simplified.

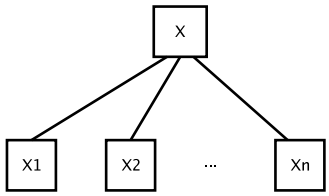


Figure 2.9: `X` is `X1` followed by `X2` ... followed by `Xn`

A Sequence Diagram is a tree. Leaves are labeled by boxes, with two zones: the upper zone contains an alias name and the lower zone an input events. The use of aliases is explained on an example below. Intermediate nodes represent

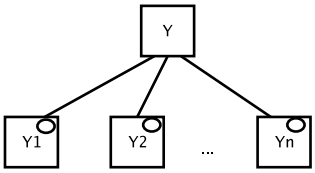


Figure 2.10: `Y` is exactly one of `Y1` or `Y2` ... or `Yn`

Figure 2.11: `Z` is zero or more repetitions of `Z1`

sub-behaviours that are refined in their children nodes. There are three types of intermediate nodes, as illustrated by Fig. 2.9, 2.10 and 2.11.

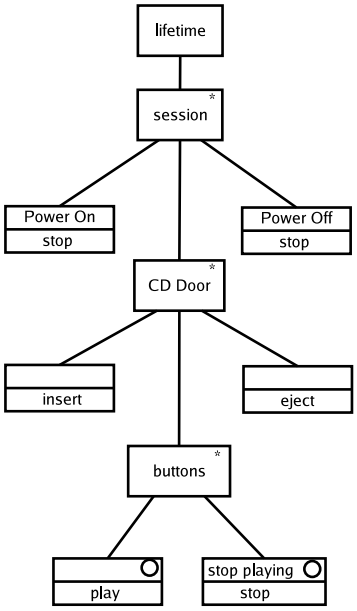


Figure 2.12: Sequence Diagram: "Session" view

For example, the various Sequence Diagrams of this session present the behaviour of a very basic compact Disc player. The player has five buttons: stop, play, forward, reverse, eject and a slot in which CDs can be inserted. Fig. 2.12 shows the high level usage of this player: a playing session starts with turning the player on. This is achieved by pressing the "stop" button. However, for the sake of readability, this Sequence Diagram does not directly consider the event but rather an alias of it. This alias improves the readability of the diagram, because it makes it possible for analysts to assign different context-dependent meaning to a single event. For instance, "stop" can be used to turn

the player off, as well. The diagram of Fig. 2.12 imposes that a session is divided into three phases. First, the system is turned on. Then, any number of CD's can be inserted, however, a CD must always be ejected before another CD is inserted. Third and finally, the player is turned off, by pressing "stop" again.

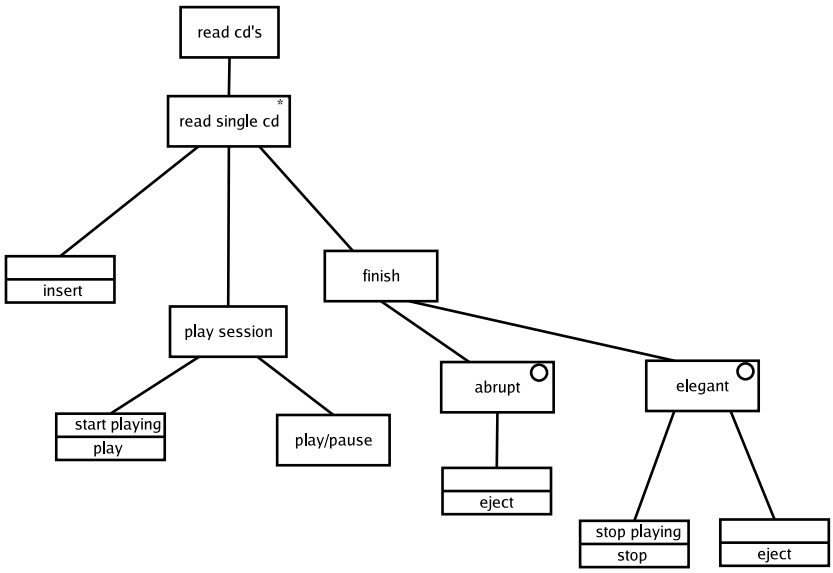


Figure 2.13: Sequence Diagram: "Reading CD" view

Another view on the behaviour of the CD player is given in "reading session" (Fig. 2.13). During the player's lifetime, many reading sessions occur. A reading session starts by inserting a CD and playing it. The session can be paused several times and is eventually terminated. This termination is either abrupt, i.e. the CD is simply ejected, or elegant, i.e. the CD is stopped before being ejected.

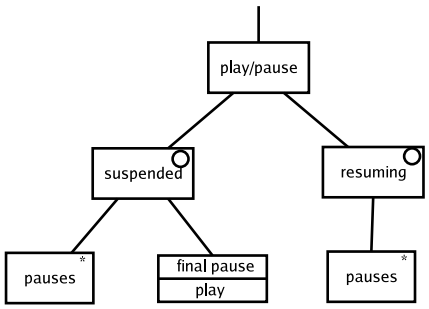


Figure 2.14: Sequence Diagram: "Play/Pause" view

Yet another view is provided by Fig. 2.16. A CD can be read but it can also be browsed *while playing it*.

Sequence Diagrams show valid input sequences only. In order to determine how the system reacts to these inputs, output actions are associated to *aliases*. Hence, output actions are context-dependent as well. For instance, the reaction

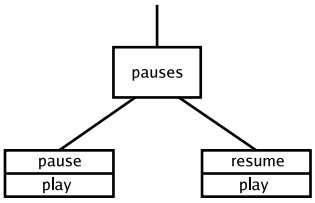


Figure 2.15: Sequence Diagram: “Pauses” view

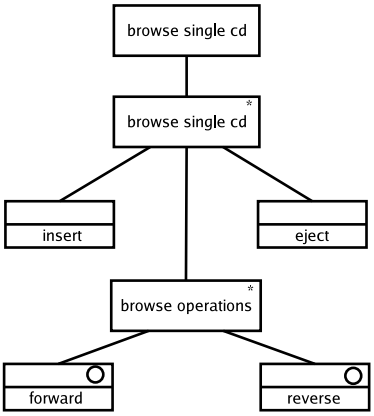


Figure 2.16: Sequence Diagram: “Browse CD’s” view

to “stop” is not the same in all sequence diagrams. It can turn the player off, turn it on or stop CD playing.

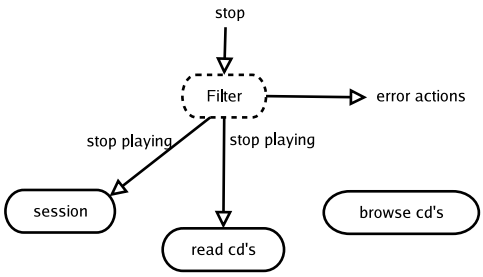


Figure 2.17: Filter mechanism

Views declare which aliases are relevant to them. For instance, “read CD’s” does not consider the “Power On” and “Power Off” aliases. However, “read CD’s” and “Session” share “stop playing” as a common alias.

This specification language was designed for being executable. It uses a “filter” mechanism to do so. When an input event comes in the system, the filter tries to find some alias for this input event that is allowed to happen in the current state, in all views for which it is relevant. If the event is relevant to no view, it is simply discarded. If there is some mismatch between rules, i.e. no alias can be found, some error action should be taken. Those can be user-defined, in order to tune system behaviour. When a matching alias is found, its associated set of output actions is performed, thus producing outputs and

modifying the value of local variables.

This execution scheme defines a meaning of composition of overlapping views. Therefore, requirements can be specified through small overlapping bits and automatically composed to form the future system complete behaviour.

2.3.2 Discussion

Sequence Diagrams are most appropriate to the specification of centralized reactive systems. They are not part of the "sequence charts" family, which can frighten some practitioners. We find some very good ideas in this language: views are given a scope, here called "relevance", and composition has a built-in meaning. Thus, the global behaviour can be constructed from the automated integration of small overlapping scenarios.

They are not completely graphical and not completely formal, either. In particular, the meaning of actions is informal in the language, while output actions are not graphically displayed. The scope (relevance) of a scenario can also be restricted or widened, but only using the textual syntax.

Finally, this language is not really inter-agent. If one wants to specify the behaviour of a distributed system, multiple views must be used. A view is associated to every agent, presenting the local view that this agent has of the whole system, as presented in [186].

2.4 Use Case Maps

2.4.1 Presentation

Use Case Maps (UCM) have been introduced by Buhr as a means to visualize scenarios on architecture [30, 31]. The essential idea behind use case maps is that they form a high-level, user-oriented notation. They are being introduced in ITU's set of notations, together with GRL (Goal Requirement Language) , where they are called the User Requirements Notation (URN) [168, 167].

The purpose of UCM is to describe how the organizational structure of a complex system and the emergent behaviour are intertwined. Buhr's paper says explicitly that UCM is not a behaviour specification language [30]. It is a notation for helping people visualize, think about and explain the "big picture" of a system behaviour.

A UCM is made of two elements: a map displaying the various components as boxes and scenario paths crossing among these components, see Fig. 2.18. The purpose of a scenario path is to illustrate a sequence of "actions" made by components. A scenario path is a wiggly line. It starts at its dot-end and finishes at its line-end. Scenario paths cross components at positions known as *responsibilities*. Paths illustrate causal chains of responsibilities. Components cooperate along a path to achieve some desired result, or make some behaviour emerge. When two components "perform" actions in a row, they implicitly communicate.

Use Case Maps provide constructs to visualize scenarios relationships. For

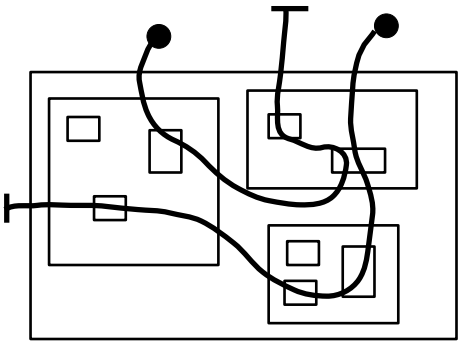


Figure 2.18: Use Case Map

instance, one can use OR-fork and OR-joins, to express that a path splits into two possible continuations or that two paths share a common ending. In Fig. 2.19, the black path and the grey path share a common prefix, intuitively password verification. They split at a OR-fork place. AND-forks and AND-joins are also available to express the fact that two scenarios execute concurrently. Of course, a path can be iterated.

About everything else in UCMs is left implicit: whether several executions of a scenario can occur simultaneously, how communication takes place, what model of concurrency is used, etc. This information is not defined in the language, because it is thought that analysts will grasp these details from the context and their knowledge of the system being visualized.

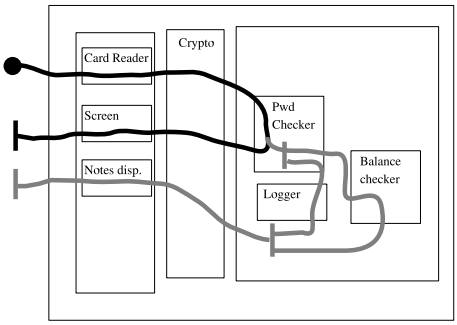


Figure 2.19: Use Case Map with forks and joins

Additional syntactical constructs are available. We cite a few of them but we do not detail them. Timers, with associated timeout paths. Stubs, that are placeholders for paths that can be dynamically branched. Slots, that are placeholders for dynamic components, that are plugged in the system at runtime. Failure points, that indicate where a path can be abruptly terminated, eg because of network communication problems. Dynamic components pools, with operations to create, delete, add and remove components from pools.

2.4.2 Discussion

UCM is not a good candidate for the scenario-based specification of distributed reactive systems. Its stated purpose is not being a specification language but rather a language for sketching system behaviour. Therefore, like MSCs, UCM provides only examples of behaviour of the system. The language has no well-defined semantics, but this is presented as a design choice. The authors of UCM argue that this looseness is a strength of their language rather than a weakness, for leaving many elements implicit allows one to avoid considering architectural details and discuss the big picture of system behaviour.

There are however some points that are found in UCM and are very common to other state-of-the-art scenario-based languages. First, the notation shows clearly the various components. This was not present in Zave's Sequence Diagrams but could be found in MSCs. This seems to be an important element of a notation aimed at describing distributed systems; showing explicitly the agents involved in the scenario seems to be intuitive. Second, UCM insists on causality and responsibility. Agents perform actions because of stimuli, coming from their environment, be it the system's environment or other system's agent. Every agent is then responsible to perform some action and cooperatively with other agents, achieve a certain global behaviour. This idea was also present in MSCs. In Zave's SDs, the distinction was made between input events and output events: the environment was responsible for ensuring that input events came as specified whereas the system was responsible for responding appropriately.

2.5 Genetic Design

2.5.1 Presentation

Genetic Design considers software construction as the process of building software systems *out of* their requirements, instead of building systems *fulfilling* their requirements [48, 46, 47]. The essential problem that Genetic Design tries to address is the inability of human beings to cope with large and complex systems, because of the deficiencies of their short-term memory.

Genetic Design is decomposed into four phases:

1. Translate a functional requirements document, written in natural language, in a formal language, called *Requirements Behaviour Tree* (RBT) on a sentence-per-sentence basis.

2. Integrate RBT one-at-a-time into an integrated *Design Behaviour Tree* (DBT). Detect defects disabling integration.

3. Extract and transform an architecture model.

4. Project the DBT on components to obtain components blueprints.

A RBT is a tree whose nodes show components. These components can realize states (written `[STATE]`), check conditions (`?Condition?`), receive events

(`??Event??`), sending data (`<DataOut>`), receive data (`>DataIn<`), assign values to their attributes (`att := value`) and revert to an equivalent component-state higher up in the tree (`^`).

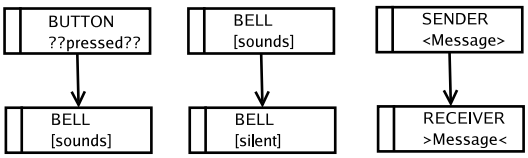


Figure 2.20: Component Interaction

Behaviour is a sequence of component states. For instance, in Fig. 2.20, it is shown that when the `BUTTON` is pressed, the `BELL` sounds. When the `BELL` is sounding, it will get silent. Every sentence is translated in an RBT. The translation process should preserve the intent. Therefore, it should add nor remove information from the initial requirements. This translation is recorded in a *Translation Traceability Table* (TTT).

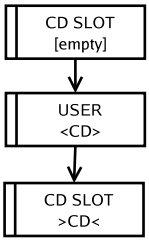


Figure 2.21: "Insert CD" RBT

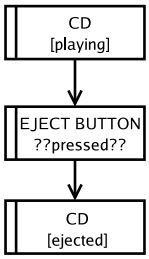


Figure 2.22: "Eject playing CD" RBT

As a translation example, consider the following requirements about a CD player. They are translated as RBT.

- When the CD slot is empty, the user can insert a CD in the slot (Fig. 2.21).
- When the CD slot is full and the play button is pressed, if the CD is a valid audio CD it starts playing, otherwise nothing changes (Fig. 2.26).
- When the CD is playing, pressing play pauses the CD (Fig. 2.24).
- When the CD is paused, pressing play resumes CD playing (Fig. 2.25).

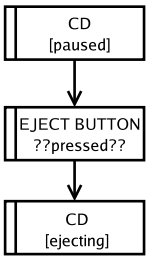


Figure 2.23: "Eject paused CD" RBT

- If the CD is playing or paused, pressing "eject" ejects the CD from the slot (Fig. 2.22 and 2.23).

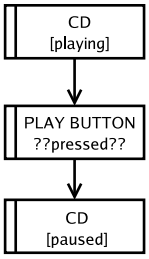


Figure 2.24: "Pause CD" RBT

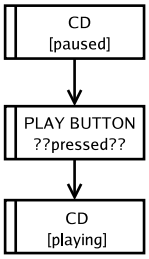


Figure 2.25: "Resume CD" RBT

RBTs need to be integrated afterwards. What makes integration feasible? Genetic Design postulates two axioms:

Precondition Axiom. Every constructive, implementable individual functional requirement of a system, expressed as a behaviour tree, has associated with it a precondition that needs to be satisfied in order for the behaviour encapsulated in the functional requirement to be exhibited.

Interaction Axiom. For each individual functional requirement of a system, expressed as a behaviour tree, the precondition it needs to have satisfied in order to exhibit its encapsulated behaviour, must be established by the behaviour tree of at least one other functional requirement that belongs to the set of functional requirements of the system. The behaviour tree that forms the root of the integrated tree is excused from this requirement.

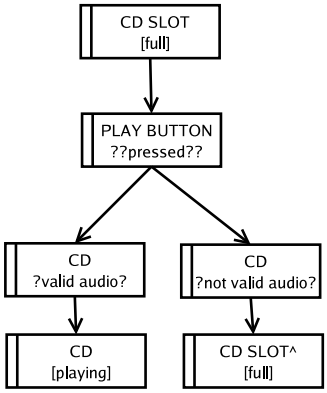


Figure 2.26: "Play CD" RBT

The first axiom states that "behaviour does not just happen". There is always a precondition that must be satisfied in order for the behaviour encapsulated in a functional requirement to be accessible or applicable or executable. The second axiom states that, when a precondition is not met, it will not become true magically; it must be set by another functional requirement.

When these two conditions are met, it is possible to integrate all requirements. This integration is performed RBT by RBT, one-at-a-time. When an RBT cannot be integrated, because a precondition is missing, for instance, it shows a requirement defect. This defect can be corrected, by adding a precondition. When an RBT cannot be integrated, because its precondition is never true, the situation should be investigated to determine whether the functional requirement is useless, i.e. there is noise in the specification, or a piece of specification is missing, i.e. functional requirements are incomplete.

In the CD player example, some RBTs do not end in component state nodes, namely those of Fig. 2.21, 2.22 and 2.23. Thus, it is impossible to attach the RBT of Fig. 2.26 to the DBT, because its precondition `CD SLOT[full]` is never established by any other RBT. To cope with this, we add two node states stating that, when the CD is ejected, the slot is empty and, when the CD has been inserted, the CD slot is full. They are displayed in red in Fig. 2.27 to illustrate that they correspond to requirements defect.

When all RBTs have been integrated into a DBT, one can turn to the solution domain and consider architectural problem. First, from the DBT, it is possible to derive a *Component Interaction Network* (CIN). A CIN is a graph in which nodes represent components. Every component appears thus only once in a CIN. There is an edge between two nodes N_1 and N_2 if there is a communication between the component represented N_1 and the component represented by N_2. Arrow heads carry more information than this: there is a double arrow head on an edge (N_1, N_2) if N_1 always initiates the communication with N_2. The CIN automatically obtained can be transformed, i.e. events and states can be renamed or removed. Since traceability links are kept by the automated process deriving the CIN, the analyst knows where the DBT has to be modified.

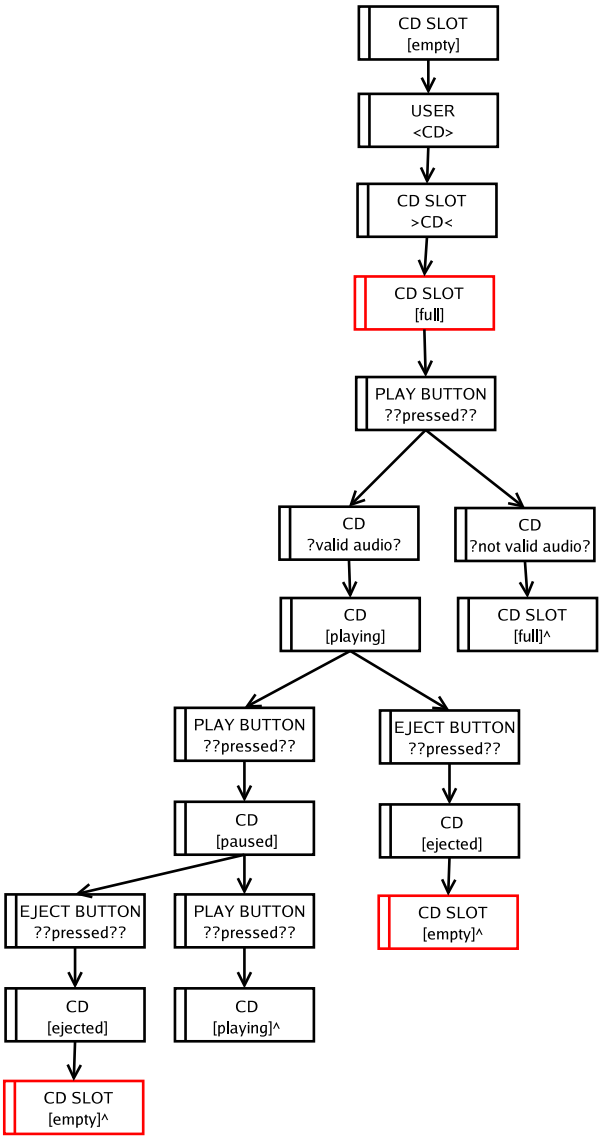


Figure 2.27: DBT for the CD player

At this stage, the DBT describes the integrated overall behaviour of the future system, built out of the functional requirements. Since the translation process preserved the intent, defects have been found and corrected early on, and requirements integration led to a system that is correct by design. Now, in order to implement it, we need a specification of the behaviour of every component. This is achieved by projecting the DBT on every component, removing information that is not local to the component being synthesized. The resulting DBTs can be analyzed to ensure that they are actually implementable. In particular, when a choice occurs in such a DBT or an action is to be carried out by a component, enough information should be locally available to ensure that this action/choice is carried out appropriately.

2.5.2 Discussion

The languages found in the Genetic Design approach feature many elements that are central to scenario-based approach.

First and foremost, translating functional requirements sentences to RBTs is presented as a rather straightforward activity. In [46], Dromey gives some methodological insight to preserve *repeatability*: requirements shall be translated by two independent analysts and the results matched. Analysts should be properly trained to the use of the language. Hence, the language of RBT should match perfectly the user intent, as declared in the functional requirements document. We are pretty certain to have a formal representation of the requirements, that correspond to initial requirements.

Second, individual requirements are integrated, in a very systematic way, to form the system's global behaviour. Genetic Design gives condition to ensure this integration and, as such, allows one to deal with small requirements chunks that will be later aggregated. This is pretty much what we found in Zave's SD as well.

Third, the tree presents information about what components do. In particular, causality and responsibility are two aspects that are clearly visible in the diagram. To be convinced of this, remark the Genetic Design makes it possible to automatically derive a CIN, showing causality relations and communications between components. This is akin to instance axes in MSC or components in UCM.

Fourth, the projection of a global DBT onto components leads to a loss of information. Some components might therefore lack information that is essential to guarantee that the emerging global behaviour matches the one specified by the DBT.

However, some aspects are rather awkward with RBT. In particular, orthogonal or partly overlapping aspects of reactive systems are very difficult to specify. This is possible but is likely to lead to combinatorial explosion. For instance, saying that a CD can be ejected after having been inserted, no matter if it is playing, paused, stopped or skipping tracks, is very difficult. This is exactly the same good old problem that led Harel to introduce orthogonality in Statecharts [70].

2.6 Live Sequence Charts

2.6.1 Presentation

Live Sequence Charts (LSC) have been developed by Damm and Harel [42] to cope with the problems of MSC presented in Section 2.2.4. LSC incorporate new syntactic constructs and elaborate on the semantics of MSC to enable analysts to distinguish between things that *may* happen and things that *must* happen. The syntactic extension is a binary modality, named *temperature*, which can be applied to almost all objects appearing in a chart. Thus, objects can be cold or hot. A hot object means "mandatory" while cold objects mean "provisional". Graphically, cold elements are dashed while hot elements are drawn with solid lines.

Charts themselves can be hot or cold. A cold chart is a chart that *happens* at some point, in some system execution. It is thus a mere example, just like a bMSC. A cold chart is also named *existential*. A hot chart is called *universal*, because it imposes a universal rule, that applies to *every* system execution. As highlighted in Sec. 2.2.4, messages appearing in a chart have different intuitive statuses: some are activation messages while other are answers. LSC makes this distinction in universal charts: one can single out an activation part, and surround it in a dashed-line hexagon. This part of the chart is named *prechart*; the lower part is named *main chart*. Existential charts are surrounded with a dashed line rectangular box.

Temperature applies to almost all objects: messages, conditions and locations, that are points along instance axes at which messages are sent or received. When an instance reaches a hot location, it must reach the next location. This expresses liveness, i.e. the obligation of something "good" to eventually occur. Cold messages need not be received; hot messages must. Remark that not all combinations of location and message temperature make sense. The interplay between message temperature and location temperature is presented in [75]. Hot conditions must be evaluated to true; cold conditions can be false, in which case the chart execution is prematurely but successfully aborted.

Charts can be structured thanks to *sub-charts*. A sub-chart is a distinguished zone of a chart, akin to MSC inline expressions. The beginning and end of a sub-chart act like synchronization points for participating instances. A sub-chart is therefore only entered after all participating instances have arrived at the location associated with the sub-chart. It can only be left once all instances have finished executing it. The definition of a cold condition is generalized to take sub-charts into account: when a cold condition is violated, the *closest surrounding sub-chart* is exited. If-then-else statements can be used. They are subcharts with two operands, guarded by a cold condition. Loops are also available: they make it possible to iterate a subchart, either a certain number of times or an unbounded number of times. Loops can be guarded by cold conditions, providing a halting condition.

LSC supports very well the multi-view aspect of scenario-based approaches: an LSC represents a certain requirements, expressed in a scenario-based fashion,

and has a specified scope. Composing two LSCs boils down to conjuncting the constraints they impose on the future system's behaviour. This corresponds nicely to the idea of "throwing new requirements chunks into the big bag of requirements". Adding a new requirement on the system does not ask any integration effort from the analyst. However, *extracting* a complete intra-object specification requires substantial effort, because all these requirements must be taken into account. Yet, our dream is to automate this step, hence taking this burden away from engineers.

The main appeal of LSC is the play-in/play-out engine that has been developed by Marelly and Harel [75, 73]. This tool allows end users to capture LSC by playing with a mock-up of the final system user interface. The user clicks on the button, fills in text fields and simulates the system reaction. The user is said to play-in the scenarios. Afterwards, scenarios that have been played-in can be played-out, i.e. interpreted by the play-out engine. In order to do so, the engine monitors input events, such as buttons being clicked, text fields being filled in, and so on, and tracks these events in all LSCs of the specification. When a prechart of some universal LSC is matched, the play-out engine starts executing the main chart. This approach actually simulates a minimal implementation of the future system; the system performs events as they become required by the various scenarios. It is actually the simplest way of satisfying the LSC specification: whenever the prechart is matched, perform main chart events until the main chart is fulfilled as well. This simple observation is central to play-out but is also at the heart of our work, as demonstrated by Theorem 3.22.

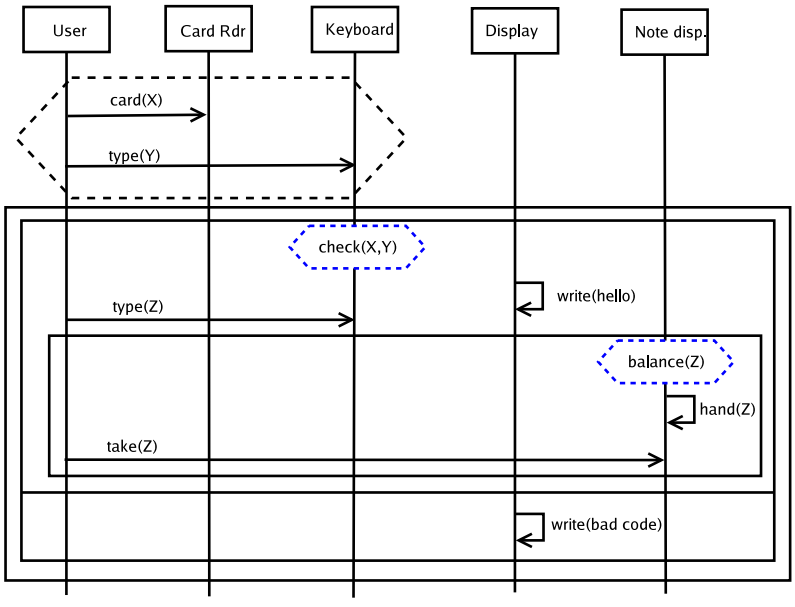


Figure 2.28: Universal LSC with subcharts, conditions, if-then-else and symbolic messages

However, the form of LSC presented so far in this section is rather restrictive and does not allow analysts to capture and play-out very interesting reactive

behaviour. This form of LSC is called *constant* in [75], because only constants are used in scenarios: instances, atomic propositions, constant messages (with fixed parameters), ... A first extension is to allow users to write messages with symbolic parameters. Instead of having to create one scenario per possible parameter value, say `msg(1)`,...,`msg(10)`, one can simply write `msg(X)` and reuse the value of `X` afterwards. Variables can even be assigned new values or be modified using built-in/user provided functions. This extension can lead to some problems because, in two occurrences of the same messsage, say `setName(X)` and `hello(X)`, that are not ordered, the former represents the *initialization* of `X`, while the latter is intended to be its use. It has been decided that the relative vertical position of messages was meaningful: between two messages sharing a symbolic parameter, if they are unordered, the message drawn higher up precedes the other. Fig. 2.28 shows an example of a universal LSC. This LSC uses symbolic messages in the prechart: the user inserts a card with some identifier `X` and types in a code `Y`. The code must be checked; if it is false, the ATM displays an error message, otherwise the user can ask to receive a certain amount of money.

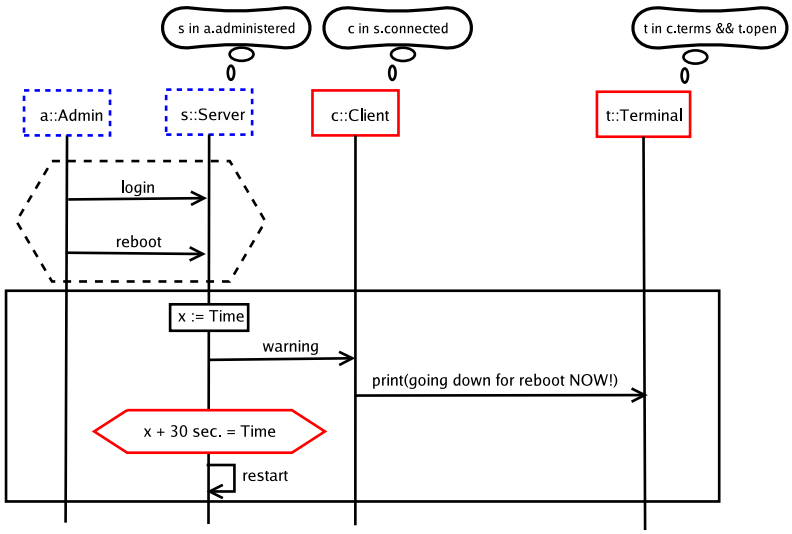


Figure 2.29: LSC with symbolic instances

One can also use *symbolic instances*. Instead of describing scenarios for constant instances, variables can be used instead, making the behaviour applicable to *all* object instances of a class. These variables are bound when the scenario is being monitored. However, this raises a problem: what should the system do when a symbolic instance has not been bound but a message must be sent to it? There are two solutions here: either arbitrarily take some instance and bind it, or bind *every* possible instance. In the latter case, a new copy of the chart being monitored is created for every bindable instance. Constraints can be put on symbolic instances, for restricting the set of possible bindings, for instance. Usual constraints refer to links[1] between objects, as shown in Fig. 2.29.

[1] A link between two objects is an instance of an association between the classes of these

An example of LSC with symbolic instances is given in Fig. 2.29. It expresses the following requirement: "When an administrator logs onto one of its administered server and asks the system to restart, the server sends a warning to all connected clients, that display some user warning on all open terminals and, thirty seconds after the restart order has been received, the system reboots."

This example illustrates another extension of LSCs, namely time. A global clock is assumed. Its value can be assigned to chart variables and conditions can test the value of the global clock.

2.6.2 Discussion

LSC has been designed to extend MSC in order to address the problems presented in Sec. 2.2.4, namely

- MSC only express examples;
- Composition is defective;
- The scope of a scenario cannot be expressed;
- It is syntactically impossible to distinguish triggering events from response events;
- There is no distinction between universal rules and examples.

LSC is a simple syntactic extension of MSC dealing with all these deficiencies. LSC has a richer meaning, being able to distinguish between examples of behaviour and universal rules. The definition of LSC includes a definition of composition of scenarios; an LSC specification is a set of LSCs and their composition is given through conjunction. Adding a (universal) LSC to a specification makes the specification more precise, i.e. constrains more the future system behaviour. Every LSC is given a default scope that contains all events appearing in it. This scope can be modified by including more events. In [75], a finer technique for altering the scope of an LSC through "forbidden events" is presented. Finally, LSC is formally defined.

Therefore, LSC is a true scenario-based language. It supports a scenario-based approach to specification. It belongs to the Sequence Chart family, making it easy to learn for practitioners used to MSC or UML Interaction Diagram. LSC is a true inter-agent language: an LSC clearly displays the various agents involved in the scenario that it describes and shows clearly their interactions.

2.7 Conclusion

We have presented several popular scenario-based languages: Message Sequence Charts (Sec. 2.2), Zave's Sequence Diagrams (Sec. 2.3), Use Case Maps (Sec. 2.4), Genetic Design Requirements/Design Behaviour Trees (Sec. 2.5) and Live Sequence Charts (Sec. 2.6). We will focus on Live Sequence Charts in the remainder of this thesis. This chapter justifies the choice of this language as

objects, as in UML.

a scenario-based specification language. We emphasized that Live Sequence Charts feature many elements that are found in other popular languages and overcome weaknesses of some of them.

We have skipped some scenario-based languages. Most notably, we did not present scenario notations based on automata, because they make the idenfication of agent interactions difficult. Glinz uses Statecharts for representing scenarios. He advocates that inter-relationships between scenarios can easily be modeled using Statecharts [61]. Ryser and Glinz have also developed a notation, named Dependency Charts, for representing inter-relationships between scenarios and derive test-cases from them [144, 143]. Zündorf uses a kind of activity diagrams, with nodes containing object diagrams, to represent scenarios. This notation is called StoryChart [45] and is supported by the FUJABA tool [56]. Hsia and colleagues use grammars to represent scenarios and derive finite state automata from them [87].

In the next chapter, we will formally define a subset of LSC and illustrate it on a case-study. Language-theoretic properties of this language will be studied, viz. expressiveness and succinctness. The reader uninterested in technical developments is invited to read the first sections (Sec. 3.1, 3.2.1 and 3.2.2) of the next chapter and proceed directly to Chapter 4.

Chapter 3

Live Sequence Charts

Contents

It is a curious feature of our existence that we come from a planet that is very good at promoting life but even better at extinguishing it.

Bill Bryson, "A Short History of Nearly Everything"

3.1 Introduction

In this chapter, we define the formalism of "Universal Live Sequence Charts". We present its abstract syntax and its semantics. We formalize a *subset* of the full language of LSCs. Our subset does not include all control-flow constructs, nor conditions, nor message and location temperature. The differences between our subset of LSC and the full-fledged LSC presented in [75] are described by Tab. 3.1. It focuses on two important aspects of LSCs: message abstraction and scenario modality. Although we are only dealing with a subset of the language, it is already sufficiently large to model some real-world systems, such as the interactions in the Center TRACON Automation System (CTAS), from NASA [178, 26]. We will study our subset of LSC, in terms of expressiveness and succinctness. These two measures are good indicators of the naturalness of the language: we will show that LSCs have a relatively low expressiveness but are highly succinct.

There are two reasons for limiting ourselves to a (very) restricted subset of LSC. The main reason is that we want to bring automated tool support to engineers, which quickly turns out to be impossible if we consider a too complex language. The second reason is that we will then analyze the properties of this language and the complexity of some analysis problems related to it. Since our results will mostly be negative, we are interested in their sharpness, by identifying a small subset of the language which is practically usable but the analysis of which is already intractable.

3.2 Language Definition

We first introduce an air traffic control system, named CTAS, which supports the presentation of LSC. We do not model the whole system. Some details are

Construct	**[75]**	Our version
LSC status	Universal/Existential	Universal
Prechart	Yes	Yes
Conditions	Hot/Cold	No
Messages	Asynchronous/Synchronous	Instantaneous
Message Temperature	Hot/Cold	Hot
Location Temperature	Hot/Cold	Hot
Sub-charts	Hot/Cold	No
Loops	Bounded/Unbounded	no
Choice	Yes	Yes
Symbolic Messages	Yes	No
Variables	Yes	No
Symbolic Instances	Hot/Cold	No

Table 3.1: Comparison of our subset of LSC and LSC from [75]

abstracted and we present this abstraction. Then, we show how the real software requirements document for CTAS can be translated to an LSC specification. This translation serves as a tutorial for LSC concrete syntax. Therefore, we do not give a full and formal definition of the concrete syntax of the language but simply present it by means of an example.

3.2.1 Case Study: an Air Traffic Control System

NASA's Center TRACON Automation System (CTAS) is a set of tools designed to help air traffic controllers manage the increasingly complex air traffic flows at large airports. The project began in 1991 and prototypes are now deployed at Denver and Dallas/Fort Worth airports. Extensions to the core CTAS system are constantly being integrated and incorporate the latest developments from research into air traffic control systems. More information is available on CTAS official web site [12].

CTAS is made of several cooperating processes, as sketched by Fig. 3.1. These processes cooperate via TCP/IP and are combined to provide certain facilities. A coherent set of such facilities is a tool. There are several tools in the CTAS. Hence, the same process can be used in different tools, with different run-time options and in combinations with different processes. Tools have been built to be used altogether in order to assist air traffic controllers. We cite three of these tools: TMA (Traffic Management Advisor), FAST (Final Approach Spacing Tool), EDA (En-Route Descent Advisor). The central process of the software system is CM (Communication Manager). Every inter-process communication goes through CM that records this information in its aircraft information database and dispatches the relevant parts to interested processes.

There are three kinds of processes

Communication processes manage the data into and out of CTAS and route messages among processes. For instance, CM is a communication process,

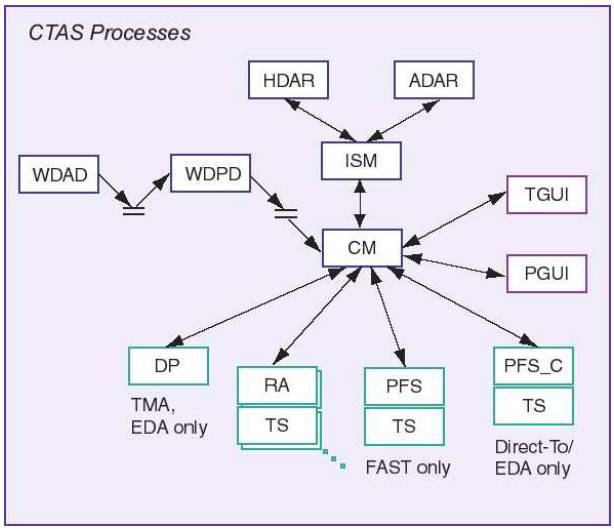


Figure 3.1: CTAS processes (reproduced from [12])

as well as WDAD (Weather Data Acquisition Daemon).

Algorithmic Processes perform the prediction and automation functions. RA (Route Analysis), TS (Trajectory Synthesizer) or WDPD (Weather-Data Processing Daemon) are algorithmic processes.

User Interfaces provide the interface for users and developers of the CTAS tools. PGUI (Planview Graphical User Interface) and TGUI (Timeline Graphical User Interface) are user interfaces.

These processes need accurate geographic, weather and air traffic data. Geographic data are relative to runways, aircraft routes, etc. Weather data include wind components speeds, temperature and pressure as a function of latitude-longitude. Air traffic data include aircraft positions and routes. Regarding weather data, WDPD checks whether some newer report is available by polling periodically a rendez-vous file written by WDAD. WDPD converts the forecast file to a binary file usable by CTAS. In particular, coordinates are translated into the native x-y coordinate system used by CTAS. Then, WDPD writes to CM that a new forecast is available. CM sends the name of this new file to all clients, managing the weather update.

We focus on clients coordination for weather update. The underlying problem is to ensure that all processes use the same data at all time. Algorithmic processes depend crucially on accurate weather information. This is especially true if we recall that CTAS tools are *combined* CTAS processes, running in a certain mode; all computations performed by these processes shall obviously be based on consistent data. In order to ensure synchronization of "weather-aware" clients on weather data, a three-phase protocol is followed. First, clients are prepared for update. Second, all clients are asked to get the new data. Third, they install this new data and use it in future computations. If any phase fails, the system tries to revert to the previous state. In the very unlikely case that

this roll-back fails as well, all clients are disconnected. They are expected to reconnect and reinitialize later on.

From the requirements document, it also appears that there are several possible forecast sources. An update can sometimes be caused by the manual intervention of a human operator. This operator interacts with CM through a special user interface, called "Weather Control Panel" (WCP).

We now present our abstraction of this system. Its structure is displayed in Fig. 3.2. We hope that this diagram is self-explanatory; rectangles represent components[1], an arrow from a component C_1 to another one (C_2) going through an ellipse labeled with I means that C_1 can communicate with C_2 by sending events in I. Interfaces are detailed in Tab. 3.2. We ignore many details about the processes and actual means of communication (TCP connections, rendez-vous files, polling, ...). Our abstract view of the system is presented in Fig. 3.2. We also assume that there are only two client processes, which we call `client[0]` and `client[1]`. Agents `cm` and `wcp` correspond to processes CM and WCP in the real CTAS.

We added the following agents, that correspond to no actual process in the real CTAS:

- `database` roughly corresponds to the file shared between WDPD and CM. We assume here that there is a database containing weather reports. This agent decides when new weather reports are available and notifies this by a "`new_weather`". Clients connect to it in order to download these forecasts. The database can decide whether this download fails or succeeds. Thus, this agent hides and abstracts real-world details such as conditions determinining when a new forecast is available, network connections unreliability, or file names.

- `term[0]` and `term[1]` are terminals, connected to clients. In the real-world, after an update, algorithmic processes may perform complex operations to make new forecast available to their computing component. Here, we abstract this computation part by assuming that clients only display forecasts on terminals. Every client has its own terminal.

- `user` is a human operator. She can manually trigger updates, through the `wcp`, or query terminals about the current weather report in use.

Agents tagged with a small "monitor" icon belong to the system. We assume that we are in charge of implementing them and that they will be deployed later in an environment consisting of `user`, `database`, `term[0]` and `term[1]`. Agents communicate via *interfaces*. An interface is a set of event names. For instance, `database` can send any event in `If_db2client` to `client[0]` or `client[1]`.

Table 3.2 defines the interfaces referred to in Fig. 3.2. Every interface name corresponds to a set of events.

[1] Components, agents, objects, processes: elements of the system that conceptually are able and responsible to perform actions and are thought of as atomic in the context of a model.

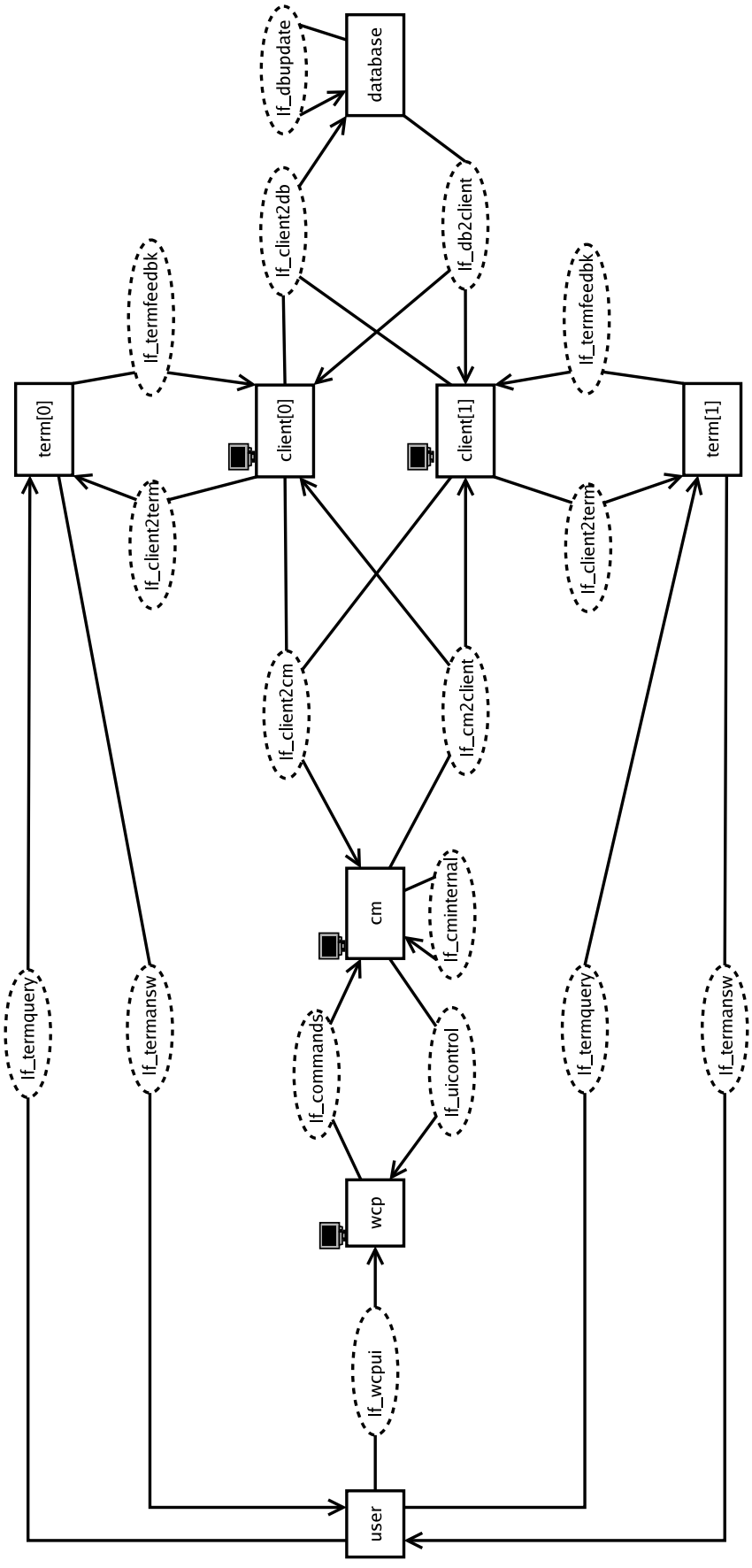


Figure 3.2: Structure specification of CTAS

Interface Name	Events
If_termquery	ask_weather
If_termansw	weather_is[0..1]
If_client2term	use_weather[0..1]
If_term2client	yes,no
If_wcpui	click
If_commands	update
If_uicontrol	enable,disable
If_cminternal	set_status_done, set_status_preupd, set_status_upd, set_status_postupd, set_status_prerevert, set_status_revert, set_status_postrevert, set_status_preinit, set_status_init, set_status_postinit
If_client2cm	yes,no,reinit,connect
If_cm2client	disconnect, ack, nack, set_status_done, set_status_preupd, set_status_upd, set_status_postupd, set_status_prerevert, set_status_revert, set_status_postrevert, get_new_weather, get_old_weather, use_weather, set_status_preinit, set_status_init, set_status_postinit
If_client2db	get_new_data, get_old_data
If_dbupdate	new_weather
If_db2client	data, fail

Table 3.2: CTAS Interfaces

3.2.2 Concrete Syntax

The software requirements document for CTAS weather update has been made publicly available in 2003, in order to serve as a common case study for the participants to the SCESM (Scenarios and State Machines: Models, Algorithms and Tools) workshop series. It served as a support for synthesis and modeling, most notably by Whittle and Schumann who reported on the successful use of their synthesis algorithm on scenarios extracted from this document [178].

The document is structured in paragraphs defining the reactions of the system in the different phases of the protocol. These chunks are thus scenarios. They are identified by numbers.

> " 2.8.10 The CM should perform the following actions when the Weather Cycle status is "pre updating" (i.e., Weather_cycle.status == WTHR_STATUS_PREUPDATING):
>
> - a) it should set the Weather Cycle status to "updating" (i.e., Weather_cycle.status = WTHR_STATUS_UPDATING)
> - b) it should set the weather status of all connected weather aware clients to "updating" (i.e., Socket.wthr_status = WTHR_CLIENT_STATUS_UPDATING)
> - c) it should send CTAS_GET_NEW_WTHR messages to all connected weather aware clients.
>
> These messages will contain the clients new weather state."

This excerpt corresponds to the LSCs of Fig. 3.4 and 3.5.

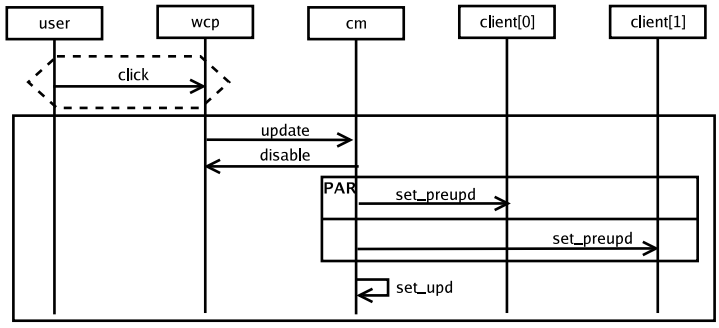


Figure 3.3: Manually triggering a weather update

We provide another example, which has been translated to the LSC of Fig. 3.7.

> " 2.8.12 The CM should perform the following actions when the Weather Cycle status is "updating" (i.e., Weather_cycle.status == WTHR_STATUS_UPDATING) and all connected weather aware clients have responded yes to the CTAS_GET_NEW_WTHR messages (i.e., Socket.wthr_status == WTHR_CLIENT_STATUS_SUCCEEDED_GET):

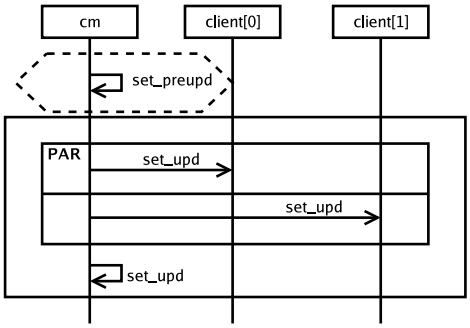


Figure 3.4: Preupdate scenario

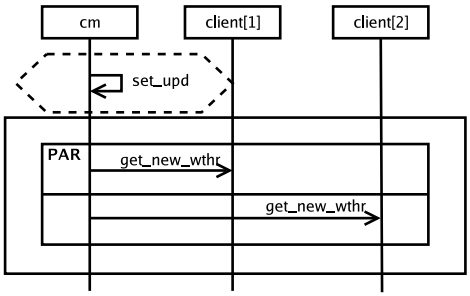


Figure 3.5: Update scenario (1)

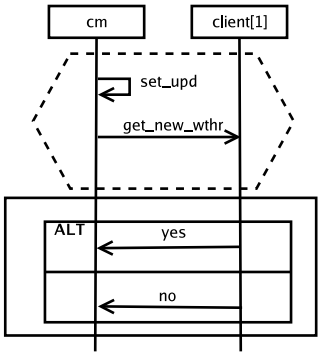


Figure 3.6: Update scenario (2)

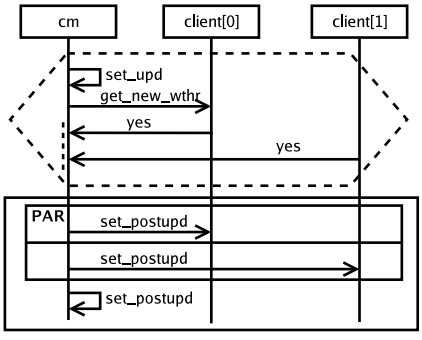


Figure 3.7: Update success scenario

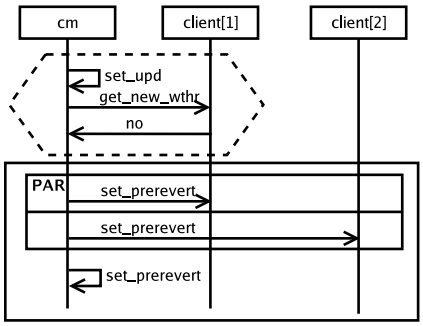


Figure 3.8: Update failure scenario

- a) it should set the Weather Cycle status to "post updating" (i.e., Weather_cycle.status = WTHR_STATUS_POSTUPDATING)
- b) it should set the weather status of all connected weather aware clients to "post updating" (i.e., Socket.wthr_status = WTHR_CLIENT_STATUS_POSTUPDATING)
- c) it should send CTAS_USE_NEW_WTHR messages to all connected weather aware clients.

These messages will contain the clients next weather state. "

In our subset of LSCs, we allow instances, which are represented as vertical lines, on top of which a box is drawn, including the instance name. This is called a lifeline. For instance, the LSC of Fig. 3.4 contains 3 instances: `cm`, `client[0]` and `client[1]`. Second, instances perform events, which are displayed as arrows, linking the "sender" and the "receiver". The most salient feature of LSCs is the decomposition of charts into two regions: an upper one, called *prechart* and surrounded by a dashed-line hexagon, and a lower one, named *main chart*, which is surrounded by a solid-line rectangle. The semantics of an LSC can be informally described as "when the behaviour described by the prechart occurs, the behaviour described by the main chart shall follow".

Sub-charts can be used, to describe alternative behaviors or scenarios executing in parallel. Sub-charts are surrounded by a box, tagged with their kind (ALT for alternatives and PAR for parallelism), and containing two scenarios. Fig. 3.4 makes use of a PAR sub-chart, stating that `cm` can send `set_upd` to both clients in any order. Fig. 3.6 uses an alternative to state that `client[1]` can answer positively or negatively to the order `get_new_weather`. It is also possible to use co-regions to state that several consecutive events on the same lifeline are not temporally ordered. A coregion is drawn as a small dashed line running along the locations belonging to it. For instance, in Fig. 3.7, a coregion is used in the prechart, to state that the order in which `client[0]` and `client[1]` answer positively does not matter.

Two LSCs have been added to deal with database and terminal access. Of course, they do not correspond to any natural language requirements, since these actors are not found in the real system. They are shown in Fig. 3.21

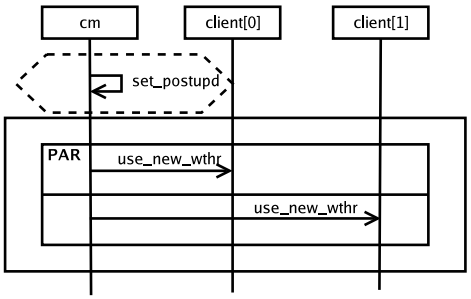


Figure 3.9: Postupdate scenario (1)

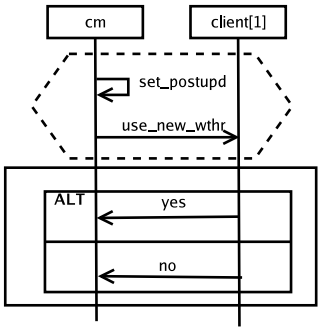


Figure 3.10: Postupdate scenario (2)

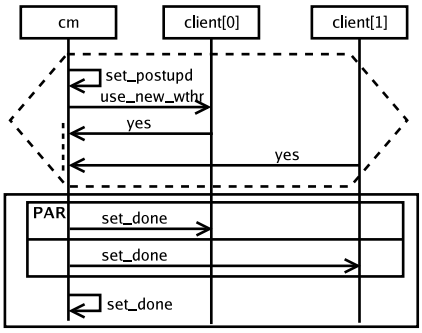


Figure 3.11: Postupdate success scenario

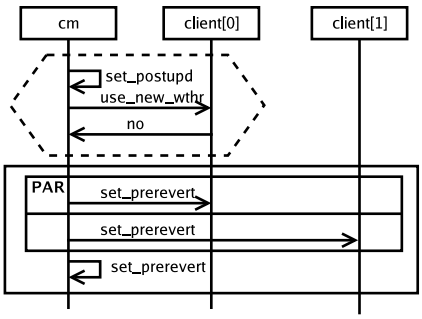


Figure 3.12: Postupdate failure scenario

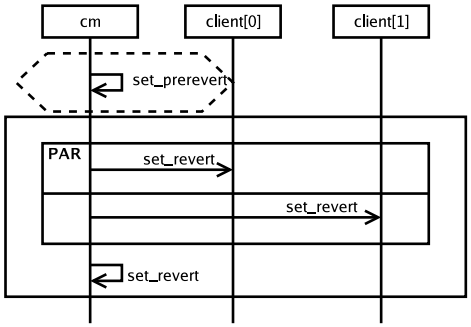


Figure 3.13: Prerevert scenario

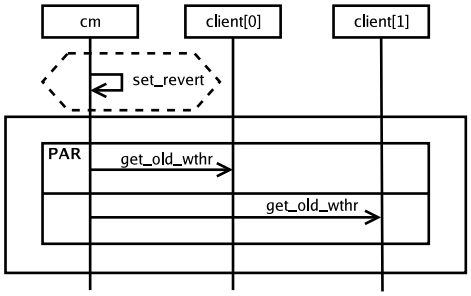


Figure 3.14: Revert scenario (1)

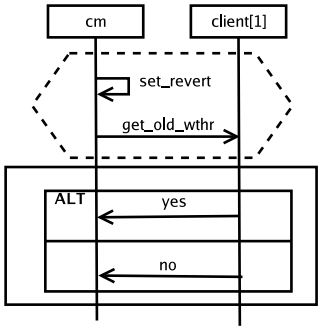


Figure 3.15: Revert scenario (2)

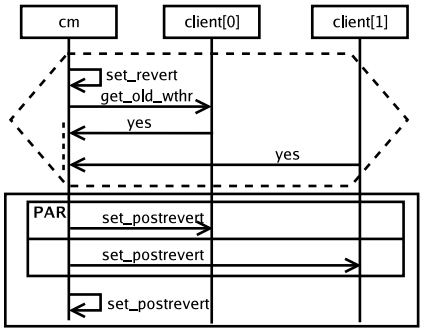


Figure 3.16: Revert success scenario

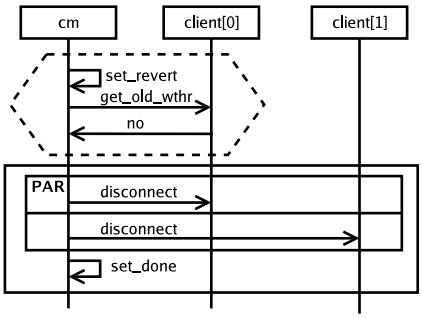


Figure 3.17: Revert failure scenario

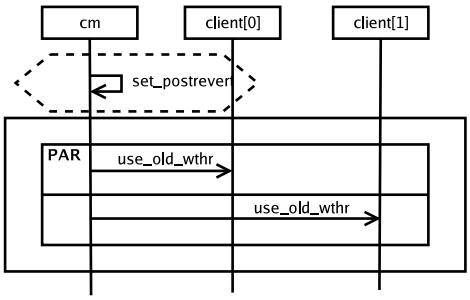


Figure 3.18: Postrevert scenario (1)

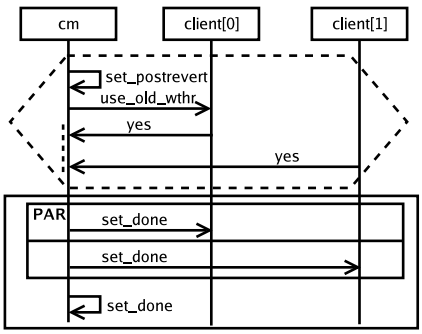


Figure 3.19: Postrevert success scenario

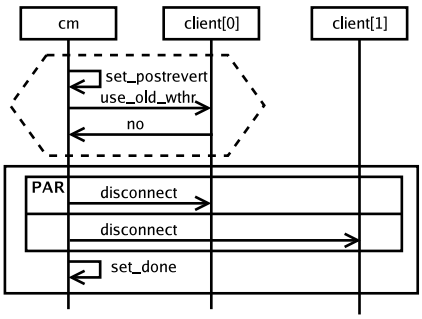


Figure 3.20: Postrevert failure scenario

and 3.22. Another scenario (Fig. 3.23) has been added to state explicitly that, once the user interface has been disabled, it will be eventually re-enabled but meanwhile, the user cannot click on the control panel. This scenario uses a "restricts" clause. This means that, although the event `click`, sent by `user` to `wcp` does not appear in the prechart or the main chart, it belongs to the scope of this LSC. This has the consequence that any occurrence of `click` after `disable` will be considered a violation.

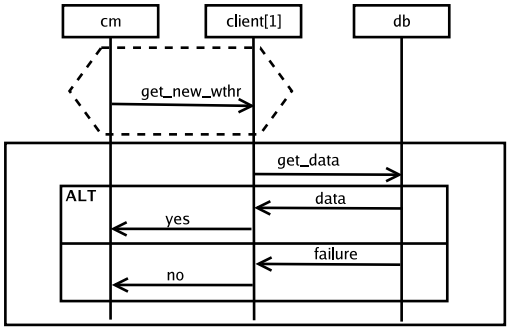


Figure 3.21: Database access

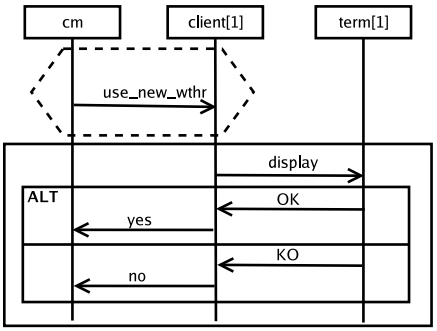


Figure 3.22: Term access

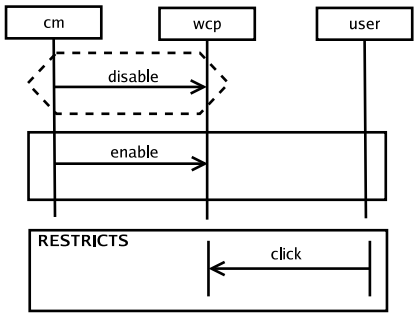


Figure 3.23: UI disabling

3.2.3 Abstract Syntax and Semantics

We represent abstractly a scenario as a labeled partial order. This is in line with the standard semantics of Message Sequence Charts [166, 40] or UML 2.0 Interaction Diagrams [130].

Definition 3.1 (Labeled Partial Order) A (finite) *A-labeled partial order* (LPO) is a tuple $\langle L, \leq, \lambda, A \rangle$, where

- L is a (finite) set of *locations*;
- $\leq \subseteq L \times L$ is a partial order relation on L. It is thus a reflexive ($l \leq l$), anti-symmetric ($l \leq l'$ and $l' \leq l$ implies $l = l'$) and transitive ($l \leq l'$ and $l' \leq l''$ implies $l \leq l''$) relation.
- $\lambda : L \to A$ is a labeling function.

■

A system's behaviour is given as the set of all its executions, that are sequences of events. Since we use LSCs as a specification language, we must relate partial orders with sequences of events. This is achieved through the concept of linearization. Intuitively, two events are ordered in one of our LPOs if they are temporally or causaly related. This temporal/causal relationship shall be reflected in the executions of the LPO. In particular, if some LPO imposes that $l \leq l'$, then l shall always occur before l' in every execution.

Definition 3.2 (Linearization) A *linearization* of a partial order $\mathcal{L} = \langle L, \leq, \lambda \rangle$ is a word $u = e_0 \dots e_n$ such that the LPO $\langle L_u, \leq_u, \lambda_u \rangle$, is isomorphic to $\langle L, \leq', \lambda \rangle$ with $\leq \subseteq \leq'$, where

- the locations are the indices in u, $L_u = \{0, \dots, n\}$,
- the total ordering is the ordering of natural numbers: $i \leq_u j \iff i \leq j$ for $0 \leq i, j \leq n$.
- the labeling function maps each index to the symbol in u at that position, $\lambda_u(i) = e_i$.

■

To say things differently, a word u is a linearization of $\mathcal{L}$ if there is a sequence of locations $l_0 \dots l_n$ such that (i) locations labelings match u ($\lambda(l_0) \dots \lambda(l_n) = u$) and (ii) the partial order described by $\mathcal{L}$ is respected in the sequence; if the LPO dictates that l_i is smaller than l_j ($l_i \leq l_j$), then l_i actually occurs before l_j in the sequence ($i \leq j$).

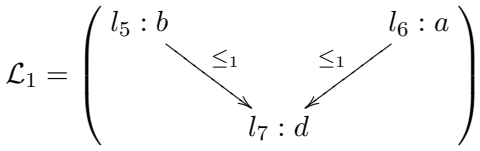


Figure 3.24: An $\{a, b\}$-LPO named $\mathcal{L}_1$

Example 3.3 In fig. 3.24 and 3.25, two LPOs are displayed as variants of Hasse Diagrams [176]: locations are named l, with subscripts. Their labels follow, after a colon, eg. $l : a$, means that $\lambda(l) = a$. Two locations l_1 and l_2 are respectively source and target of an edge if $l_1 < l_2$ and furthermore, there is no other location l_3 with $l_1 < l_3 < l_2$. The linearizations of $\mathcal{L}_1$ are $\{bad, abd\}$ and the linearization of $\mathcal{L}_2$ is $\{be\}$. ■

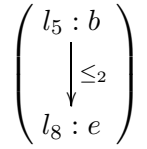


Figure 3.25: A $\{b, e\}$-LPO named $\mathcal{L}_2$

Definition 3.4 ($\models \subset \Sigma^\infty \times$ LPO) A finite or infinite word $\gamma \in \Sigma^\infty$ satisfies an LPO $\mathcal{L} = \langle L, \leq, \lambda, A \rangle$, denote by $\gamma \models \mathcal{L}$ iff

- $\gamma \in \Sigma^*$ and $\gamma|_A$ linearizes $\mathcal{L}$.
- $\gamma \in \Sigma^\omega$ and $\exists w \in \Sigma^* : w \sqsubset \gamma$ and $w \models \mathcal{L}$.

■

As we already stated, the behaviour of an LPO, i.e. its language, is its set of linearizations.

Definition 3.5 ($\mathscr{L}(\mathcal{L})$)

$$\mathscr{L}(\mathcal{L}) = \{w \in \Sigma^\infty | w \models \mathcal{L}\}$$

■

There is an operational machinery for recognizing the linearizations of an LPO. We will make a heavy use of this "automaton", in many different flavours, in the rest of this thesis. It is called the cut transition system.

Definition 3.6 (Cut) A *cut* in an LPO $\mathcal{L} = \langle L, \leq, \lambda \rangle$ is a downward-closed subset of L. Formally, c is a cut in $(\mathcal{L})$ iff $c \subseteq L$ and $\forall l, l' \in L : l \in c \wedge l' \leq l \implies l' \in c$. A cut c' is an e-successor of a cut c if there is a location l such that

1. $\lambda(l) = e$,
2. $l \notin c$,
3. $c' = c \cup \{l\}$.

We write $c \xrightarrow{e} c'$ if c' is an e-successor of c. This relation is extended to words in the obvious way: (i) $c \xrightarrow{\epsilon} c$ and (ii) if $c \xrightarrow{u} c'$ and $c' \xrightarrow{v} c''$, then $c \xrightarrow{uv} c''$. A cut in $\mathcal{L}$ is *full* if it contains all locations in $\mathcal{L}$, i.e. $c = L$. The *operational semantics* of an LPO is the set of words labeling paths from the empty cut to the full cut:

$$\mathcal{O}(\mathcal{L}) = \{w | \emptyset \xrightarrow{w} c \textit{ with } c \textit{ full}\}.$$

■

Example 3.7 The cuts of $\mathcal{L}_1$ (from Fig. 3.24) are

$$\emptyset, \{l_5\}, \{l_6\}, \{l_5, l_6\}, \{l_5, l_6, l_7\}.$$

The transition system is shown in Fig. 3.26. ■

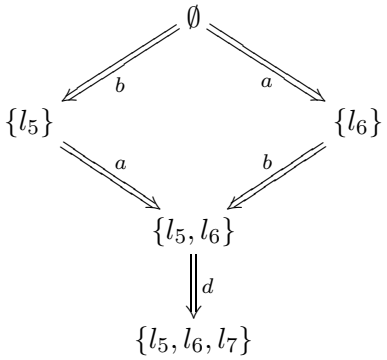


Figure 3.26: Cut transition system of $\mathcal{L}_1$ from fig. 3.24

By abuse of language, we will talk about "the linearizations of a cut c", instead of "the linearizations of the LPO resulting from the projection of an LPO $\mathcal{L}$ onto a cut c".

Proposition 3.8 For every $w \in \Sigma^*$, w is a linearization of a cut c in $\mathcal{L}$ iff $\emptyset \xrightarrow{w} c$. ■

Proof 3.8

First, remark that every linearization of a cut c is of length $|c|$, by definition 3.2.

By induction on the size of c. If $c = \emptyset$, its only linearization is ϵ, which is indeed the only sequence of transitions leading from $\emptyset$ to $\emptyset$.

If $|c| > 0$, suppose that for every c' such that $|c'| < |c|$, u is a linearization of c' iff $\emptyset \xrightarrow{u} c'$. Because $|c| > 0$, $\emptyset \xrightarrow{w} c$ iff, for some c', $\emptyset \xrightarrow{u} c' \xrightarrow{a} c$, letting $w = ua$.

Otherwise, c would have to be empty. By induction hypothesis, u linearizes c' and it is easy to check that ua linearizes c, too. In the other direction, if w linearizes c, then $w = ua$, with a labeling a maximal location, say l, in c. Removing l from c, we obtain some location c', with $c' \xrightarrow{a} c$. By construction of c', u linearizes c', hence, applying our induction hypothesis, we have that $\emptyset \xrightarrow{u} c' \xrightarrow{a} c$. □

This proposition implies that the operational semantics of LPOs coincides with its language-based semantics:

Theorem 3.9

$$\mathcal{O}(\mathcal{L}) = \mathscr{L}(L).$$

■

Proof 3.9

This is implied by Lemma. 3.8 □

We introduced LPOs to have simple objects describing the information contained in an LSC. The semantics of LSCs can be given solely in terms of these concepts. This is the *abstract syntax*. However, we have to explain how we turn a concrete basic chart into an LPO. This is reminiscent of the approach taken in MSCs, see [166, 40, 90]. We assume that communication is instantaneous and we shall be careful about the implications of this decision.

First, a basic chart is made of vertical lines, called *life lines*. They represent the actions that an agent performs during the scenario. Those actions may be either events, which are denoted by an arrow, pointing to their receiver, or local actions, which are assumed to change the agent's state. The points on the diagram at which actions occur will be simply called "points" in this section. An LPO is obtained from a basic chart by following these rules. A location in this LPO is a set of points of the form $\{p\}$, where a local action is performed at p or $\{p_1, p_2\}$, if there is an arrow from p_1 to p_2. The labeling of a location is simply the name of the action performed at the points of this location. By definition, two points belonging to the same location agree on their label. It remains to define how these locations shall be ordered. Two locations are directly ordered ($l_1 <_d l_2$) if, and only if, some point $p_1 \in l_1$ and some point $p_2 \in l_2$ belong to the same instance line and

- p_1 is drawn higher up than p_2, and
- p_1 and p_2 do not belong to the same coregion.

The relation $\leq$ is simply the reflexive-transitive closure of $<_d$.

Example 3.10 In fig. 3.27, a basic chart is presented, which specifies some interaction between 4 agents, P_1 to P_4. There are 12 points in this diagrams, as there are 6 arrows. Its corresponding LPO, displayed in Fig. 3.28, contains thus 6 locations, l_1 to l_6, each of them being labeled with the name of the event.

Remark how event names are extended with their sender and receiver. This will make sure that two arrows with the same labeling, but different source and/or target, denote distinct events.

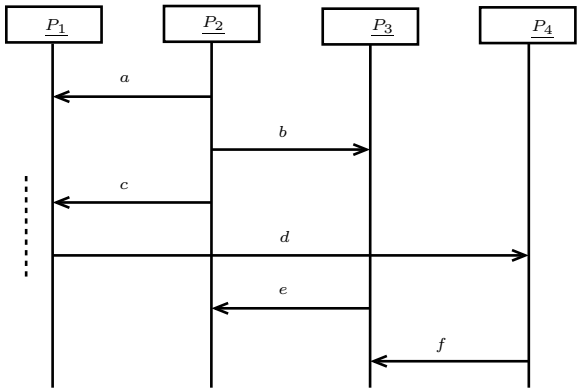


Figure 3.27: A Sample Basic Chart

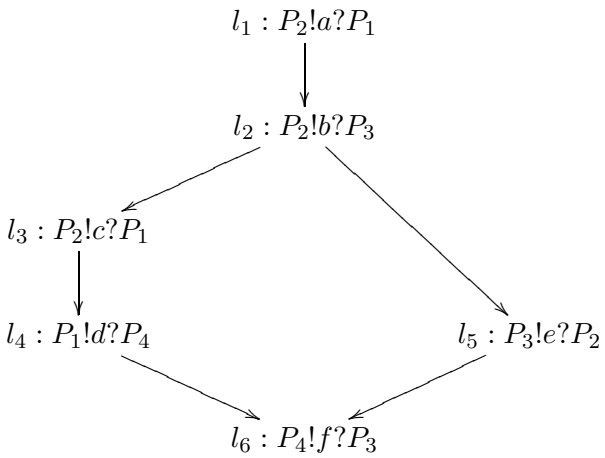


Figure 3.28: LPO corresponding to the basic chart of Fig. 3.27

■

Remember that, in basic charts, one can use "choice" constructs, to express choice between two sub-charts. We still lack features to represent choice in LPOs. We will now extend LPOs to do so, and show how these new LPOs, which we call *Choice LPO* (CLPO), can be mapped back to LPOs. In these extended objects, a choice operand, over some alphabet A, is an element of the form $L_1 + L_2$, where L_1 and L_2 are A-LPOs.

Definition 3.11 (CLPO) A CLPO over some alphabet A is (recursively) defined as an LPO over A and couples of A-CLPO. To put things more clearly: in an A-CLPO, locations may be either labeled by simple symbols, taken from A, or by elements of the form $C_1 + C_2$, representing a choice between two A-CLPO.

■

Thus CLPO are recursively defined and A-CLPO is actually the smallest fix point which satisfies this definition.

Example 3.12 (CLPO) A CLPO is shown in Fig. 3.29. Its location l_4 is labeled by a choice between two other CLPOs, namely $\mathcal{L}_1$ and $\mathcal{L}_2$, from Fig. 3.24 and 3.25, respectively. Its set of expansions, $\mathit{lpo\_expand}(\mathcal{L}) = \{\mathcal{C}_1, \mathcal{C}_2\}$, where $\mathcal{C}_1$ and $\mathcal{C}_2$ are displayed in Fig. 3.30 and 3.31. ■

We will not directly deal with CLPOs. We rather consider them as a useful intermediate formalism and expand them to a set of LPOs. The expansion function, $\mathit{lpo\_expand} : A\text{-}CLPO \rightarrow 2^{A\text{-}LPO}$, is also recursively defined[2], as CLPO is a recursive domain.

Definition 3.13 (*lpo_expand*) For every A-LPO $\mathcal{L}'$, $\mathcal{L}' \in \mathit{lpo\_expand}(\mathcal{L})$ if there is a sequence of one-step substitutions, starting from $\mathcal{L}$ and leading to $\mathcal{L}'$.

$\mathcal{L}'$ is a one-step substitution of $\mathcal{L} = \langle L, \leq, \lambda \rangle$ if there is some $l \in L$, such that $\lambda(l) = C_1 + C_2$ and, for some A-LPO $C = \langle L_C, \leq_C, \lambda_C \rangle \in (\mathit{lpo\_expand}(C_1) \cup \mathit{lpo\_expand}(C_2))$,

$$\mathcal{L}' = \left\langle (L \setminus \{l\}) \cup L_C, \leq', \lambda \setminus (\{(l, C_1 + C_2)\}) \cup \lambda_C \right\rangle,$$

and for every pair of locations l_1, l_2, $l_1 \leq' l_2$ if one of the following holds:

- $l_1, l_2 \in L$, $l_1 \neq l \neq l_2$ and $l_1 \leq l_2$,
- $l_1, l_2 \in L_C$ and $l_1 \leq_C l_2$,
- $l_1 \in L$, $l_2 \in L_C$ and $l_1 < l$,
- $l_1 \in L_C$, $l_2 \in L$ and $l < l_1$.

■

Informally, in a one step substitution, we simply select some location l to be expanded, because it is labeled by a choice between two CLPOs $(C_1 + C_2)$. We take one of the branches of this choice and expand it in full. This gives a set of LPOs, among which we choose one (C). We then replace l by C. This requires to update the partial order and labeling function accordingly. For the former, we keep all previous relations, in $\mathcal{L}$ and $\mathcal{C}$, and we add new relations, to ensure that all locations smaller than l are now smaller than all the locations replacing l. The same is done for larger locations. For the latter, l is removed and the labeling function of C is added.

The linearizations of a CLPO $\mathcal{C}$ are all the linearizations of its expanded LPOs. The language of a CLPO is its set of linearizations.

[2]To be fully rigorous, *lpo_expand* is the smallest fix point which is a solution of the equation described. We should prove that this fix point exists and is unique, but we skip this tedious and mildly interesting proof.

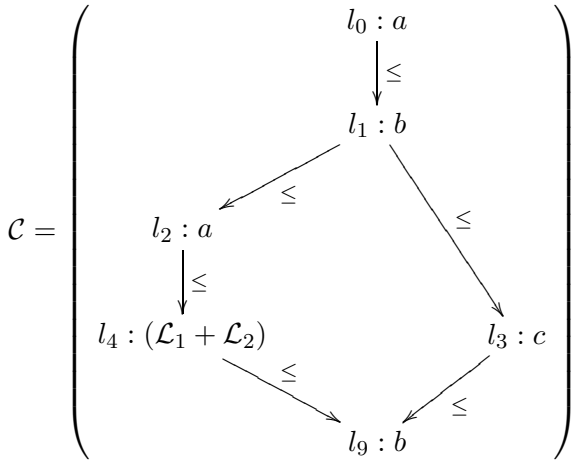


Figure 3.29: An $\{a, b, c, d, e\}$-CLPO named $\mathcal{L}$

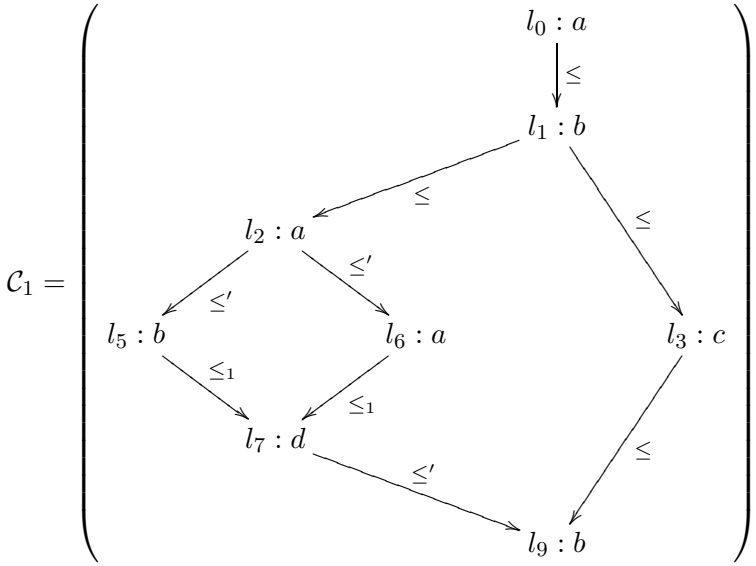


Figure 3.30: $\mathcal{C}_1 \in \mathit{lpo\_expand}(\mathcal{L})$

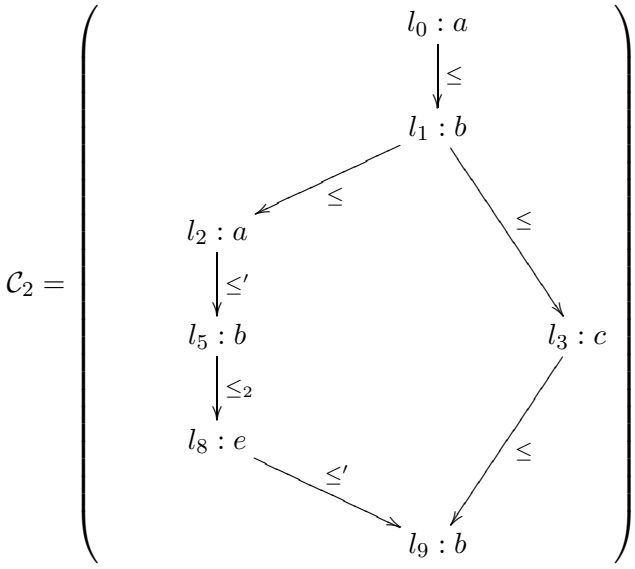


Figure 3.31: $\mathcal{C}_2 \in$ *lpo_expand*$(\mathcal{L})$

Definition 3.14 ($\models \subset \Sigma^\infty \times$ CLPO) For every CLPO $\mathcal{C}$ and every finite or infinite word $\gamma \in \Sigma^\infty$,

$$\gamma \models \mathcal{C} \text{ iff } \exists \mathcal{L} \in \textit{lpo\_expand}(\mathcal{C}) : \gamma \models \mathcal{L}$$

The language defined by $\mathcal{C}$, denoted $\mathscr{L}(\mathcal{C})$, is its set of linearizations, i.e.

$$\bigcup_{\mathcal{L} \in \textit{lpo\_expand}(\mathcal{C})} \mathscr{L}(\mathcal{L})$$

■

There is an extension of the operational semantics presented above to CLPO.

Definition 3.15 (CLPO - Transition System) The transition system associated with a CLPO $\mathcal{C}$ is $(S_\mathcal{C}, \rightarrow_\mathcal{C})$, where $S_\mathcal{C} = \{c | \mathcal{L} \in \textit{lpo\_expand}(\mathcal{C}) \wedge c \in \textit{cuts}(\mathcal{L})\}$ and $c \xrightarrow{a}_\mathcal{C} c'$ iff $\exists \mathcal{L} : c \xrightarrow{a}_\mathcal{L} c'$.

The operational semantics of CLPOs is

$\mathcal{O}(\mathcal{C}) = \{w \in \Sigma^* | \exists s : \{\emptyset\} \xrightarrow{w} s \wedge s$ contains a cut c full for some $\mathcal{L} \in \textit{lpo\_expand}(\mathcal{C})\}$

■

Th. 3.9 can be lifted to CLPOs, to show that the "operational semantics" coincides with the semantics, stated directly in terms of languages.

Theorem 3.16 $\mathscr{L}$(C) = O(C) ■

Proof 3.16

Let $w \in \Sigma$. $w \in \mathcal{O}(\mathcal{C})$ iff there is a sequence of transitions from $\emptyset$ to c, full for some $\mathcal{L} \in$ *lpo_expand*, iff (by Theorem 3.9), w is a linearization in $\mathcal{L}$ iff $w \in \mathscr{L}(\mathcal{C})$ (by definition of $\mathscr{L}(\mathcal{C})$). □

Every basic chart (with choice) can be turned to a CLPO. This is done in the obvious way, extending our previous technique to subcharts: a point in a basic chart can correspond to a subchart. If a subchart ranges over several instances, it may only be entered when all instances have reached this point. This forces this location to be greater than all the previous locations of those instances. Then, we recursively translate the subchart and obtain CLPOs to label the point corresponding to the subchart. We have thus a convenient mathematical object for describing basic scenarios, with choice.

So far, we did not introduce the two main features which motivated our choice of LSCs as a specification language, namely behavioural modality and message abstraction. We introduce this now; this completes the definition of the LSC abstract syntax and their semantics, as well.

Definition 3.17 (LSC - abstract syntax) There are two forms of Live Sequence Charts: existential LSC and universal LSC.

- A *Universal Live Sequence Chart* (ULSC) is a couple

  $$\Box(P, M),$$

  where P and M are Σ_R-CLPOs. P is called *prechart* and M is called *main chart*. Events in Σ_R are called *restricted*.

- An *Existential Live Sequence Chart* (ELSC) is

  $$\Diamond(M),$$

  where M is a Σ_R-LPO.

■

As said previously, in scenario-based software engineering, we are interested in gathering scenarios from several stakeholders and put them together in a "big bag of requirements". This set is deemed a specification and the semantics of the language will take care of how these bits of requirements shall be composed.

Definition 3.18 (ULSC-Spec) A *ULSC specification* (ULSC-Spec) is a finite set of ULSCs. ■

The semantics of LSC can be defined, on the basis of "linearizations".

Definition 3.19 (LSC Semantics) LSCs are interpreted against infinite words ($\gamma \in \Sigma^\omega$).

- $\gamma \models \Box(P, M)$ iff, whenever the prechart is matched in γ, the main chart is matched afterwards:

$$\forall u, v \in \Sigma^* : \forall \gamma' \in \Sigma^\omega : (uv\gamma' = \gamma \text{ and } v \models P) \implies \gamma' \models M.$$

- $\gamma \models \Diamond(M)$ iff it is eventually matched in γ:

$$\exists u \in \Sigma^* : \exists \gamma' \in \Sigma^\omega : u\gamma' = \gamma \text{ and } \gamma' \models M$$

The notion of LSC model is lifted to languages: a language L satisfies a ULSC ($L \models \Box(P, M)$) if $\forall \gamma \in L : \gamma \models \Box(P, M)$. A language L satisfies an ELSC ($L \models \Diamond(M)$) if $\exists \gamma \in L : \gamma \models \Diamond(M)$.

The language defined by a universal LSC is

$$\mathscr{L}(S) = \{\gamma \in \Sigma^\omega | \gamma \models S\}.$$

The language defined by a ULSC-Spec is the conjunction of the semantics of its components. Formally, for some ULSC-Spec S,

$$\mathscr{L}(\mathcal{S}) = \bigcap_{L \in \mathcal{S}} \mathscr{L}(L)$$

■

Universal LSCs can be used to model the properties of agent systems in an intuitive way. Every agent receives powers, in the sense that it can trigger events. With these powers come responsibilities: it must ensure that its events occur at the right time. More particularly, that they do not occur in situations where they would cause a violation of the specification. But, that they do occur whenever their absence would also cause a violation of the specification. Actually, LSCs can be equivalently expressed as the conjunction of these two conditions, called safety and liveness conditions. Safety asserts that at every position in an execution, if some event e is not allowed by the ULSC at this position,then e does not occur at the next position. Let us fix some ULSC $\Box(P, M)$, with restricted events Σ_R, for the rest of this section. Liveness says that at every position in γ, if e is a possible continuation at this position, then some restricted event must occur. We do not oblige e to occur, any restricted event will do. This is because of choices: if there is a choice between two events, e and e', then both of them are required at the same time. Although, only one of them must occur to resolve the choice and ensure progress in the chart.

Definition 3.20 (Forbids - Safe) Consider a word $w \in \Sigma^*$. We say that it *forbids* some event $e \in \Sigma_R$ iff there is a decomposition upv of w such that

1. $p \models P$
2. $\forall m \sqsubseteq v : m \not\models M$,
3. $\nexists v' : vev' \models M$.

An infinite run γ is *e-safe* iff, for every $uv \sqsubset \gamma$, if v forbids e, then $uve \not\sqsubset \gamma$. ■

Definition 3.21 (Requires - Live) A word $w \in \Sigma^*$ *requires* an event $e \in \Sigma_R$ iff there is a decomposition upv of w such that

1. $p \models P$,
2. $\forall m \sqsubseteq v : m \not\models M$,
3. $\exists v'' : vev'' \models M$.

An infinite run γ is *e-live* iff, for every $uv \sqsubset \gamma$, if v requires e, then there are $w \in \Sigma^*$ and $e' \in \Sigma_R$ such that $uvwe' \sqsubset \gamma$. ■

These constraints, which are given event-by-event, are equivalent to the semantics of ULSCs.

Theorem 3.22 (ULSC = safe + live) For every $\gamma \in \Sigma^\omega$ and every ULSC $S = \square(P, M)$,

$$\gamma \models S \iff \forall e \in \Sigma_R : \gamma \text{ is } e\text{-safe and } e\text{-live}$$

■

Proof 3.22

($\Rightarrow$)

Suppose that $\gamma \models S$ but γ is not e-safe or not e-live.

not *e*-safe Since γ is not e-safe, by definition, we find a decomposition $upve\gamma'$ of γ such that

1. $p \models P$,
2. $\nexists v' \sqsubseteq v : v' \models M$,
3. $\forall w \in \Sigma^* : vew \not\models M$.

But, because $\gamma \models S$ and $p \models P$, it must be the case that $ve\gamma' \models M$. Therefore, by definition, there is a prefix m of $ve\gamma'$ such that $m \models M$. This contradicts one of items 2 or 3. Thus, we reach a contradiction.

not *e*-live Again, we find a decomposition of γ into $upv\gamma'$ such that

1. $p \models P$,
2. $\nexists v' \sqsubseteq v : v' \models M$,
3. $\exists w \in \Sigma^* : vew \models M$,
4. $\forall w : w \sqsubset \gamma' : w$ does not contain any restricted event.

Item 4 implies that for all prefixes w of γ', $vw|_{\Sigma_R} = v|_{\Sigma_R}$. Hence, since $v \not\models M$ (by item 2, $vw \not\models M$, neither). Hence, $v\gamma' \not\models M$. We reach a contradiction.

($\Leftarrow$)

Suppose that $\gamma \not\models S$. We show that γ is either not e-safe or not e-live, for some $e \in \Sigma_R$. First, $\gamma \not\models S$ means that we can find a decomposition $up\gamma'$ of γ such that $p \models P$ and $\gamma' \not\models M$. We perform the following case split, about γ'. Their disjunction is a tautology.

$\forall r \sqsubset \gamma : \exists w \in \Sigma^* : rw \models M$ Then, γ is not e-live, for some e. Indeed, there must be some $r' \sqsubset \gamma$ such that no restricted event occurs after r'. Otherwise, there would be more restricted events than the number of locations in M and M should be eventually matched, or could never be matched afterwards.

$\exists r \sqsubset \gamma : \forall w \in \Sigma^* : rw \not\models M$ Then, the run is not e-safe, if we take the smallest r satisfying the condition above and let $r = er'$.

One of the two conditions must be true, which implies that the run is not live or not safe. □

3.3 Expressiveness and Succinctness

In Section 3.2, we have introduced a scenario-based specification language, named "Universal Live Sequence Charts". This specification language has been formally defined: it has a formal abstract syntax and a formal semantics, given respectively in terms of CLPOs and in terms of languages of infinite words. Having given such a formal semantics to the language, we are now in position to study its properties. In this section, we will consider its *expressiveness* and its *succinctness*.

These two properties are not absolute but relative. Hence, our study will be based on the comparison of LSCs with other well-known modeling languages. Of course, languages can only be compared if they have the same semantic domain. In this thesis, we have tackled this problem by focusing on languages of infinite words as a semantic domain. Therefore, we only consider linear-time semantics.

The results of this section are summarized in the graph of Fig. 3.32. Nodes represent classes of languages. There is a solid-line arrow between from a class L_1 to a class L_2 if, and only if, $\mathsf{L}_1 \subset \mathsf{L}_2$. Bidirectional (two-head) arrows denote the equality of the two classes. Let us first briefly recall the meaning of the various acronyms.

NBA non-deterministic Büchi automata.

LTL linear temporal logic;

DBA deterministic Büchi automata;

ALA alternating linear automata;

ULA universal linear automata;

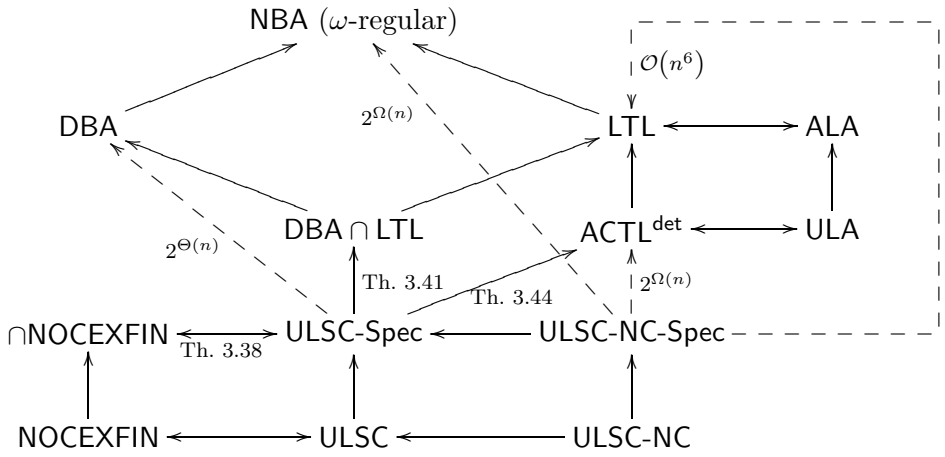


Figure 3.32: Expressiveness and Succinctness

ACTL^{det} the common fragment of LTL and ACTL. Roughly, ACTL is CTL with only A as path quantifier.

NOCEXFIN no finite-counter example. This is an awkward class of languages, with the sole interest of being exactly equivalent to universal LSCs, with choice.

ULSC Universal LSC, see Sec.3.2.3.

ULSC-NC Universal LSC without choice.

ULSC-Spec Specification expressed as ULSCs, i.e. finite conjunctions of ULSC.

ULSC-NC-Spec Specification expressed as ULSC-NC, i.e. finite conjunctions of ULSC-NC.

NBA denotes the class of all languages that can be recognized by nondeterministic Büchi automata. It is very important here to understand that some of these classes are *syntactic* while others are *semantic*. On the one hand, syntactic classes are defined on the basis of a certain formalism, for instance Büchi automata, and the addition of certain syntactic constraints. For instance, deterministic Büchi automata (DBA) are Büchi automata with a functional transition relation. The language class DBA contains all languages which are recognizable by a DBA. Again, we recall that there is a difference between formalisms, that are syntactic classes, and the class of languages that these formalisms define. We use two different types of fonts to emphasize this difference: the set of all constructs belonging to a formalism is written in plain Roman font (LTL, DBA, ...) while classes of languages are written in sans-serif font (LTL, DBA,...).

According to this convention, we can already talk about the expressiveness of languages. This is, as announced, a relative property: one can wonder whether a language is *more* expressive than another language.

Definition 3.23 (Expressiveness) Consider two formalisms, L_1 and L_2, and let their classes of languages be $\mathsf{L_1}$ and $\mathsf{L_2}$. If $\mathsf{L_1} \subseteq \mathsf{L_2}$, L_2 is as *expressive* as L_1. If the inclusion is strict, L_1 is *strictly less expressive* than L_2. ■

On the other hand, the definition of semantic classes is based on the "semantic" properties of the languages they contain. In our diagram of Fig. 3.32, there are three semantic classes: NOCEXFIN, ∩NOCEXFIN, i.e. finite intersections of languages from NOCEXFIN, and DBA ∩ LTL. In comparison, when proving containment of syntactic classes, we often come up with a computable function, translating every model of the former language to an equivalent model of the latter. This is practically more interesting, as showing that models in a formalism L_1 can be translated to L_2, all tools developed to deal with models of L_2 can be reused to analyze models from L_1. Of course, the cost of this translation shall be investigated. The cost of such a translation is called "succinctness".

Definition 3.24 (Succinctness) Consider two formalisms L_1 and L_2. Suppose that their semantics is given in terms of languages, through semantic functions $\mathscr{L}_i(.) : L_i \to \Sigma^\omega$ $(i = 1, 2)$. Let $\mathcal{G}$ be a set of functions $\mathbb{N} \to \mathbb{N}$ (called "a bound"). L_1 is $\mathcal{G}$-as succinct as L_2 iff there is some $g \in \mathcal{G}$ such that, there exists a computable function $f : L_1 \to L_2$ with,

1. f is correct: $\forall t \in L_1 : \mathscr{L}(t) = \mathscr{L}(f(t))$.
2. the size of f outputs remains within the bound: $\forall t \in L_1 : |t| \leq |f(t)|$.

Typical bounds are "exactly" n, polynomially $\mathcal{O}(n^k)$ and exponentially $\mathcal{O}(2^{n^k})$. ■

The rest of this chapter will be devoted to the proof of the results presented in the graph of Fig. 3.32.

3.3.1 Classical Results

We start by recalling well-known results from the literature.

Theorem 3.25 (DBA ⊂ NBA) DBA are strictly less expressive than NBA. ■

Proof 3.25

See [156]. The language $\{a, b\}^* \cdot \{a\}^\omega$ is not recognized by any DBA. □

Theorem 3.26 (LTL ⊂ NBA) LTL is strictly less expressive than NBA. ■

Proof 3.26

See [181]. LTL coincides with the class of languages recognized by counter-free non-deterministic Büchi automata. □

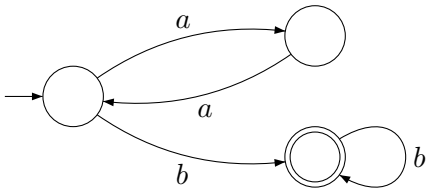
As a corollary of these two theorems, it comes that

Corollary 3.27 (LTL and DBA incomparable) LTL is not as expressive as DBA and DBA is not as expressive as LTL. ■

Proof 3.27

LTL $\not\subseteq$ DBA: the language presented in the proof of Theorem 3.25 is defined by the LTL formula $\Diamond\Box a$.

DBA $\not\subseteq$ LTL there is a language expressible as a DBA which is not counter-free. For instance, $(aa)^*b^\omega$ is recognized by the following DBA and we just have to show that this language is not counter-free.



Take any DBA recognizing this language. A counter for $u \in \Sigma^*$ is a sequence of states $q_1 \dots q_n$ such that, from q_i, u leads to $q_{i+1 \mod n}$, and from these states, different ω-words are accepted. Assume that $\mathcal{A}$ is counter-free and recognizes the language considered. Pick some n larger than the number of states of $\mathcal{A}$. Then, $a^{2n}b^\omega$, which belongs to the language, must visit twice the same state (i.e. follow a loop), by the pigeonhole principle. In other words, the (unique) infinite sequence of states followed on $a^{2n}b^\omega$ is $q_0 \dots q_{2n+1}q_{2(n+1)} \dots$ with

1. $q_{2n} \xrightarrow{b} q_{2n+1} \xrightarrow{b} \dots$,
2. $q_{i-1} \xrightarrow{a} q_i$, $1 \leq i \leq 2n$,
3. there are j, k, with $1 \leq j < k \leq 2n$, such that $q_j \dots q_k$ and $q_j = q_k$. Let $l = k - j$.

Thus, a^{l+1} is a counter for $q_j \dots q_{k-1}$ if, different words can be accepted from q_{k-1} than from q_j. Since $\mathcal{A}$ is assumingly counter-free, q_j and q_{k-1} recognize the same language. Hence, $a^j a^l a^{2n-l-j} b^\omega = a^{2n} b^\omega$ and $a^j a^{l+1} a^{2n-l-j} b^\omega = a^{2n+1} b^\omega$ should both be accepted by $\mathcal{A}$, whereas the latter $a^{2n+1} \notin (\{a\} \cdot \{a\})^* \cdot \{b\}^\omega$. Contradiction.

□

From this, it comes that the the semantic class DBA∩LTL is a *strict* sub-class of both LTL and DBA.

Theorem 3.28 (LTL = ALA) LTL is exactly as expressive as ALA (Alternating Linear Automata). Furthermore, LTL is linearly as succinct as ALA and vice-versa. ■

Proof 3.28

The proof is provided in [109] and [151]. □

Theorem 3.29 (ACTL ∩ LTL = ULA) The class of languages definable in LTL and ACTL is exactly the class of languages recognized by universal linear automata (ULA). ■

Proof 3.29

Maidl shows that a formula $\phi \in$ LTL is expressible in ACTL (i.e. $\exists \phi' \in$ ACTL : $\mathscr{L}(\phi) = \mathscr{L}(\phi')$) iff $\overline{\mathscr{L}(\phi)}$ is recognized by a non-deterministic linear Büchi automaton.

We simply show that the complement of a non-deterministic linear automaton is a ULA. Take any non-deterministic linear automaton and simply change its status: all states become universal instead of existential. Then, swap accepting and non-accepting states. A word is rejected by a non-deterministic linear automaton iff all the runs on this word visit only finitely often accepting states. Since the automaton is linear, all infinite runs must stabilize in some state, i.e. remain in that state forever. Hence, a run visits only finitely often accepting states iff it visits infinitely often *non*-accepting states. We are done. □

3.3.2 Expressiveness

We start by characterizing exactly the languages definable by Universal LSCs. We come up with an artificial language class, dubbed ∩NOCEXFIN, the sole interest of which is to coincide with ULSC-Spec and to facilitate some of the future proofs.

We introduce the following class, called FCmP . Intuitively, a language L is in FCmP if projecting every word of L onto A yields a finite language.

Definition 3.30 (A-Finite/Closed modulo Projection Languages) A language $L \subseteq \Sigma^*$ is *Finite/Closed modulo Projection* (FCmP) if, for the given set of symbols $A \subseteq \Sigma$, letting $w \sim_A w' \iff w|_A = w'|_A$,

1. L is closed under $\sim_A$ $(w \sim_A w' \implies (w \in L \iff w' \in L))$
2. L has a finite number of $\sim_A$ equivalence classes: ($[L]_{/\sim_A}$ is finite and non-empty).

We let FCmP denote the class of A-FCmP languages, for all A. ■

Lemma 3.31 (CLPO = FCmP) A language L is definable by an A-CLPO iff L is in $A-$ FCmP . ■

Proof 3.31

Consider some A-LPO $\mathcal{L}$. Its (finite) language is the set of words $\{w \in \Sigma^* | w|_A$ is a linearization of $\mathcal{L}\}$. First, this language is closed under projection.

Second, $\mathcal{L}$ has only a finite number of linearizations. Therefore, $\mathscr{L}(\mathcal{L})$ is in A-FCmP . Since A-CLPO languages are unions of A-LPO languages, the two conditions hold, too. In the other direction, we build a CLPO from an arbitrary language $L \in$ FCmP . To do so, we let $L' = L|_A$. Remark that L' is finite, i.e. can be written as a finite union of singletons $\bigcup_{w \in L}\{w\}$. Each singleton defines a total order, thus an LPO. Putting together all these LPOs into some big CLPO yields the desired CLPO. □

We forge a new class of languages, NOCEXFIN, to correspond to ULSC. The languages in this class all assert that it is forbidden to match a "prechart" without matching a "main chart" afterwards.

Definition 3.32 (NOCEXFIN) A language L is in *NOCEXFIN* (no finite counterexample) iff, for some alphabet $A \subseteq \Sigma$, there are two A-FCmP languages V and W, such that

$$L = \overline{\Sigma^* \cdot V \cdot \overline{W \cdot \Sigma^\omega}}.$$

■

Proposition 3.33 (NOCEXFIN not boolean closed) The class NOCEXFIN is not closed under any boolean operation. ■

Proof 3.33

Closure under complement. Let $L = \Sigma^* \cdot a \cdot \overline{b \cdot \Sigma^\omega}$. It is clearly the complement of a language in NOCEXFIN. We start by remarking the following fact above L: for every $\gamma \in \Sigma^\omega$, $aa\gamma \in L$. This is because $a\gamma \in \overline{b \cdot \Sigma^\omega}$. Suppose that NOCEXFIN is closed under complement, then there are two FCmP languages, V, W, defining a NOCEXFIN language $L' = \overline{\Sigma^* \cdot V \cdot \overline{W \cdot \Sigma^\omega}}$ such that $L' = L$. Remark that $L' \neq \Sigma^\omega$, which implies that its complement is nonempty. Hence, we can take some arbitrary $\gamma' \notin L'$. Then, by hypothesis, $\gamma' \notin L$ neither. Because $aa \in \Sigma^*$, it is the case that $aa\gamma' \notin L'$. But, we remarked above that $aa\gamma' \in L$, for every $\gamma \in \Sigma^\omega$. We reached a contradiction, and are obliged to conclude that NOCEXFIN is not closed under complement.

Closure under intersection. Consider the two following languages, over $\Sigma = \{a, b, c\}$: $\overline{\Sigma^* \cdot a \cdot \overline{b \cdot \Sigma^\omega}}$ and $\overline{\Sigma^* \cdot b \cdot \overline{a \cdot \Sigma^\omega}}$. Their intuitive meaning is "whenever an a (resp. b) occurs, it must be followed by a b (resp. an a)". There is no NOCEXFIN language recognizing their intersection. The argument is tedious but simple. First, we have to recognize that in the candidate language, that we let $\overline{\Sigma^* \cdot V \cdot \overline{W \cdot \Sigma^\omega}}$, V, W are $\{c\}$-FCmP language, i.e. occurrences of c are projected away. Then, we remark that V must contain a and W must contain b. To ensure this, we simply remark that ac^ω is not in the considered language. Neither is $aa\{c\}^\omega$, but $\Sigma^* abc^\omega$ i, which rules out the possibility that $V = \emptyset$ and $W = \{a, aa\}$. The same argument is applied to obtain that V contains b and W must contain a.

Then, it comes easily that aac^ω is in the language, which contradicts our hypothesis.

Closure under union. Follows the same argument as above. We let V_1 be $\{c,d\}^*\{a\}\{c,d\}^*$ and $W_1 = \{c,d\}^*\{b\}\{c,d\}^*$. Thus, the first NOCEXFIN language obliges every a to be followed by some b, ignoring all occurrences of c and d. The second language is the same, except that c,d and a,b exchange their roles. Then, it is simple to show that their complement is not definable within coNOCEXFIN, i.e. by some language $\Sigma^* V \overline{W\Sigma^\omega}$, because there can be any number (≥ 2) of a in the word. The same is true of c. Yet, their must be at least two a. Thus, V must be infinite.

□

We can characterize the languages definable by universal LSC. As announced, it is a strict subclass of the languages (already of a low Borel level) used in Prop. 3.36.

Theorem 3.34 (ULSC = NOCEXFIN) Universal LSCs define exactly languages in NOCEXFIN. ■

Proof 3.34

($\Rightarrow$)

Suppose that X is definable by some universal LSC, $\Box(P,M)$, with restricted events Σ_R. We let $V = \mathscr{L}(P)$ and $W = \mathscr{L}(M)$. V and W are Σ_R-$FCmP$ languages by Lem. 3.31. It remains to show that

$$\overline{\Sigma^* \cdot V \cdot \overline{W \cdot \Sigma^\omega}} = X.$$

We prove that their complement coincides, for it makes the argument simpler:

$$\Sigma^* \cdot V \cdot \overline{W \cdot \Sigma^\omega} = \overline{X}.$$

Let $\gamma \in \overline{X}$. Then,$\gamma \not\models \Box(P,M)$ and thus, γ can be decomposed into $u \in \Sigma^*$, $p \models P$ and $\gamma' \not\models M$. Since $p \models P$, $p \in V$ and, because $\gamma' \not\models M$, there is no prefix $w \sqsubset \gamma'$ such that $w \models M$, i.e. $\gamma' \in \overline{W \cdot \Sigma^\omega}$. The other direction is similar: consider some $up\gamma'$ with $p \in V$ and $\gamma' \in \overline{W \cdot \Sigma^\omega}$. It comes immediately that $up\gamma' \not\models \Box(P,M)$, i.e. $up\gamma' \notin X$.

($\Leftarrow$)

Take any language $\overline{\Sigma^* \cdot V \cdot \overline{W \cdot \Sigma^\omega}}$ and show that it is definable by some universal LSC. Since V and W are both A-FCmP languages, there are CLPOs defining them, by Lem. 3.31, say $\mathcal{C}_V$ and $\mathcal{C}_W$, respectively. The desired ULSC is then $\Box(\mathcal{C}_V, \mathcal{C}_W)$. The correctness can be proved using the same argument as in the first part of the proof. □

It is interesting to note that the two parts upon which LSCs are built, namely their precharts and main charts, describe in fact very simple languages.

Those languages are finite, modulo some technicalities. Thus, they are much poorer than regular languages, for instance, as they impose a very restricted use of Kleene's star.

Corollary 3.35 Languages definable by universal LSCs are *strictly* included in the languages of the form

$$\overline{\Sigma^* \cdot V \cdot \overline{W \cdot \Sigma^\omega}},$$

with V, W being regular languages. ■

We will now prove that the semantics of an LSC, which is an ω-language can be recognized by a Deterministic Büchi Automaton (DBA). By corollary 3.35, we thus obtain strict inclusion of LSCs in DBA. In fact, we prove a stronger result, which is expected, from the results of [22] on languages of low Borel level.

Proposition 3.36 If $V \subseteq \Sigma^*$ and $W \subseteq \Sigma^*$ are regular languages, then

$$\overline{\Sigma^* \cdot V \cdot \overline{W \cdot \Sigma^\omega}}$$

is recognized by some DBA. ■

Proof 3.36

Since W is regular, there is a DFA

$$\mathcal{A}_W = \langle \Sigma, Q_W, q_0^W, \Delta_W, F_W \rangle$$

recognizing it. By definition, $w \in W \cdot \Sigma^\omega$ iff $\exists v \in \Sigma^* : \exists w' \in \Sigma^\omega : v \cdot w' = w \wedge v \in W$. Membership to $\overline{W \cdot \Sigma^\omega}$ is thus characterized as

$$w \notin W \cdot \Sigma^\omega \iff \forall v \in \Sigma^* : \forall w' \in \Sigma^\omega : v \cdot w' = w \implies v \notin W.$$

If $q_0^W \in F^W$, then $\overline{W \cdot \Sigma^\omega} = \emptyset$, and it is possible to build a DBA recognizing $\emptyset$.

Otherwise, we claim that the following DBA recognizes $\overline{W \cdot \Sigma^\omega}$, provided that $\#$ is a fresh state name,

$$\mathcal{A}'_W = \langle Q'_W, q_0^W, \Delta'_W, F'_W \rangle$$

with

$$Q'_W = Q_W \cup \{\#\} \setminus F_W,$$

$$\Delta'_W = (\Delta_W \setminus \{(q, a, q') | q' \in F_W\}) \cup \{(q, a, \#) | (q, a, q') \in \Delta_W \wedge q' \in F_W\},$$

$$F'_W = Q_W \setminus F_W$$

This automaton is essentially the same as $\mathcal{A}_W$, except that every transition which previously ended in a final state is now redirected to a sink state. Thus,

a run of this automaton is accepting if it never reaches a final state of $\mathcal{A}_W$. Precisely, none of its prefixes is accepted by $\mathcal{A}_W$.

Moreover, $\mathcal{A}'_W$ is weak. Indeed, its states can be partitioned in two classes: $Q_W \setminus F_W$ and $\{\#\}$, such that it is not possible to go back from $\{\#\}$ to $Q_W \setminus F_W$ and these classes contain either only accepting states ($Q_W \setminus F_W$) or non-accepting states ($\{\#\}$).

Of course, $\Sigma^* \cdot V$ is regular and recognized by a DFA $\mathcal{A}_V$. The concatenation of $\mathcal{A}_V$ and $\mathcal{A}'_W$ yields a nondeterministic weak Büchi automaton, since all states of $\mathcal{A}_V$ are made non-accepting and it is impossible to go back from $\mathcal{A}'_W$ to $\mathcal{A}_V$.

Following [103], it is possible to use the *breakpoint construction* in order to obtain a deterministic co-Büchi automaton $\mathcal{B}$ recognizing $\Sigma^* \cdot V \cdot \overline{W \cdot \Sigma^\omega}$. Considering this automaton as a Büchi automaton implies, by the fact that it is deterministic, that it recognizes the complement of the language of $\mathcal{B}$, i.e. $\overline{\Sigma^* \cdot V \cdot \overline{W \cdot \Sigma^\omega}}$. □

Definition 3.37 (∩NOCEXFIN) A language L belongs to $\cap NOCEXFIN$ iff there exists $L_1, \ldots, L_n \in \mathsf{NOCEXFIN}$ such that

$$L = \bigcap_{i=1} L_i.$$

■

Theorem 3.38 (∩NOCEXFIN = ULSC-Spec) The class of all languages definable by LSC specifications coincides with ∩NOCEXFIN. ■

Proof 3.38

By the obvious symmetry of Def. 3.19 and Def. 3.37, this theorem follows from Theorem 3.34. □

Thus, we obtain, as desired, that

Theorem 3.39 (ULSC-Spec ⊆ DBA) Every language definable by ULSC specifications is recognized by some DBA. ■

Proof 3.39

Combination of Theorem 3.38 and Prop. 3.36. □

Now, we turn to the comparison between ULSC Specifications and LTL and first demonstrate that LSCs can be translated to LTL. The following fact is known from [72, 102].

Theorem 3.40 (ULSC-Spec ⊆ LTL) Every language definable by a universal LSC specification is definable in LTL. ■

Proof 3.40

We only prove inclusion and delay the proof of strictness until later in this section. This proof is constructive, we exhibit the translation between LSCs and LTL. Consider a universal LSC $S = \Box(P, M)$, with restricted events Σ_R the LTL formula expressing it is φ_S

$$\varphi_S = \Box \bigwedge_{w \in prech} \left(\phi_{\Sigma_R}(w) \rightarrow \bigvee_{w' \in w \cdot main} \phi_{\Sigma_R}(w') \right),$$

where

$$\begin{aligned} prech &= \{w|_{\Sigma_R} \in \Sigma^* | w \text{ is a linearization of } P\} \\ main &= \{w'|_{\Sigma_R} \in \Sigma^* | w' \text{ is a linearization of } M\} \end{aligned}$$

and the function $\phi_{\Sigma_R} : \Sigma^* \rightarrow \text{LTL}$ is defined inductively, on the length of its argument as:

$$\begin{aligned} \phi_{\Sigma_R}(\epsilon) &= \top \\ \phi_{\Sigma_R}(a \cdot w) &= a \wedge \bigcirc \left(\left(\bigwedge_{e \in \Sigma_R} \neg e \right) \mathcal{U} \left(\phi_{\Sigma_R}(w) \right) \right) \end{aligned}$$

The formula corresponding to a specification $\mathcal{S} = \{S_1, \ldots, S_n\}$ is simply the conjunction of the formulae of its components:

$$\varphi_{\mathcal{S}} = \bigwedge_{S \in \mathcal{S}} \varphi_S.$$

We need to check that $\mathscr{L}(\varphi_{\mathcal{S}}) = \mathscr{L}(\mathcal{S})$. This is rather obvious. First, $\phi_{\Sigma_R}(w)$ is equivalent to projecting w on Σ_R: $\gamma \models \phi_{\Sigma_R}(w) \iff \exists u \sqsubset \gamma : u|_{\Sigma_R} = w$. This comes from a simple induction on w, which we skip.

Then, φ_S is the encoding of the semantics of universal LSCs, from def. 3.19 and the same holds of $\varphi_{\mathcal{S}}$: it is an immediate translation of the semantics of an LSC specification. □

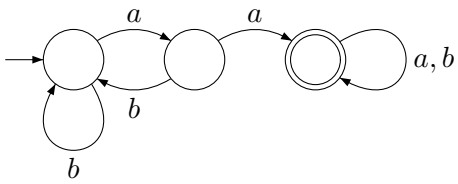
DBA are not as expressive as NBA. We have already stated, as illustrated by Fig. 3.32, that the languages definable both in DBA and LTL are even a lower class. Now, we prove that the expressiveness of LSC specifications is even less expressive.

Theorem 3.41 (ULSC-Spec $\subset$ DBA $\cap$ LTL) Every language definable by a ULSC Specification is also recognized by a DBA *and* definable by an LTL formula. ■

Proof 3.41

Inclusion follows from the conjunction of Theorems 3.40 and 3.36.

Strictness is implied by Lemma. 3.42. Actually, the language $(a|b)^*aa(a|b)^\omega$ is definable in LTL (by the formula $\Diamond(a \wedge \bigcirc a)$ and recognized by a DBA, namely



Consider any ULSC Specification, $\{S_1, \ldots, S_n\}$ and suppose without loss of generality, that, $\forall i : 1 \leq i \leq n : \mathscr{L}(S_i) \neq \Sigma^\omega$. Otherwise, remove all these scenarios without changing the semantics of the specification. By Lemma 3.42, in every scenario S_i, there is a word in $(a|b)^*aa(a|b)^\omega$. which is missing from $\mathscr{L}(S_i)$. Since $\mathscr{L}(\mathcal{S}) = \bigcap_{i=1}^{n} \mathscr{L}(S_i)$, all these words are also missing from the intersection of languages.

□

For the sake of completeness, we prove the lemma used in the proof of Theorem 3.41.

Lemma 3.42 For every ULSC S, either $\mathscr{L}(S) = \Sigma^\omega$ or there is a word in $\Sigma^* \cdot \{aa\} \cdot \Sigma^\omega$ which does not belong in $\mathscr{L}(S)$. ■

Proof 3.42

Assume that $\mathscr{L}(S) \neq \Sigma^\omega$. Then, we show that there is a word $w \notin \mathscr{L}(S)$ but $w \in \Sigma^* \cdot \{aa\} \cdot \Sigma^\omega$. Since S is a ULSC, Theorem 3.34 implies that $\mathscr{L}(S) \in$ NOCEXFIN. Thus, there are two A-FCmP languages V, W, with

$$\mathscr{L}(S) = \overline{\Sigma^* \cdot V \cdot \overline{W \cdot \Sigma^\omega}}.$$

Thus, $\gamma \notin S$ iff $w \in \Sigma^* \cdot V \cdot \overline{W \cdot \Sigma^\omega}$. Clearly, if $\gamma \notin S$, then, for every $u \in \Sigma^*$, $u \cdot \gamma \notin \mathscr{L}(S)$. In particular, since, $\mathscr{L}(S) \neq \Sigma^\omega$, then there is a $\gamma \notin \mathscr{L}(S)$, which implies that $aa\gamma \notin \mathscr{L}(S)$. However, $aa\gamma \in \Sigma^* \cdot \{aa\} \cdot \Sigma^\omega$. □

Lemma 3.43 (ULSC-Spec $\subseteq$ ULA) Every ULSC-Spec defines a language that is recognized by some Universal Linear Automaton. ■

Proof 3.43

We show that the complement of every ULSC-Spec definable language is recognized by a non-deterministic linear Büchi automaton. Then, by Theorem 3.29, we obtain the desired result. Consider an arbitrary ULSC Specification $\mathcal{S} = \{S_1, \ldots, S_n\}$ and a word $\gamma \notin \mathscr{L}(\mathcal{S})$. By Def. 3.19, this is equivalent to

$$\exists i : 1 \leq i \leq n : \gamma \notin \mathscr{L}(S_i).$$

For every $1 \leq i \leq n$, we build a linear NBA recognizing $\overline{\mathscr{L}(S_i)}$. A word γ violates a ULSC if, at some point in γ, the prechart is matched but is not followed by the main chart. This automaton is

$$\langle Q, q_0, \Delta, F \rangle,$$

where

- $Q = \{c | c \text{ is a cut in some } \mathcal{C} \in \mathit{lpo\_expand}(P) \cup \mathit{lpo\_expand}(M)\} \cup \{\text{Start}, \text{Fail}\}$,
- $q_0 = \text{Start}$,
- $\Delta(q, a, q')$ iff one of the following conditions holds
  - $q \xrightarrow{a} q'$ and q' is not a full cut in P,
  - there is some q'' such that $q \xrightarrow{a} q''$, q'' is a full cut in P and $q' = \emptyset$ (the first cut in M).
  - $q = \text{Start} = q'$,
  - $q = \text{Start}$ and $\emptyset \xrightarrow{a} q'$, in P,
  - q contains all prechart locations ($\exists c \in \mathit{lpo\_expand}(P) : c \subseteq q$), q is not full ($\nexists c \in \mathit{lpo\_expand}(L) : c = q$), there is no q'' such that $q \xrightarrow{a} q''$ and $q' = \text{Fail}$.
  - $q = \text{Fail} = q'$.
- $F = \{q | \exists c \in \mathit{lpo\_expand}(P) : c \subseteq q \text{ and } \nexists c \in \mathit{lpo\_expand}(L) : q = c\} \cup \{\text{Fail}\}$.

The automaton waits in its initial state and nondeterministically guesses when the prechart is to be matched. It checks that the prechart is indeed matched but is not followed by the main chart. There are two ways to fail the main chart: (1) an unexpected restricted event occurs or (2) a non-full cut is reached and never left. The former case is dealt with by transitions to Fail and the latter is tackled by accepting states. By a simple inspection, one can check that this automaton is indeed linear and non-deterministic. Linearity follows from the definition of the operational semantics of CLPOs. Then, all $\mathcal{A}_i$ can be combined into a single linear NBA, which starts its execution by guessing which $\mathcal{A}_i$ to simulate. □

Theorem 3.44 (ULSC-Spec $\subset$ ACTL$^{\text{det}}$) All languages definable by ULSC Specifications are also definable in ACTL$^{\text{det}}$, the common fragment of LTL and ACTL. This inclusion is strict. ■

Proof 3.44

We only need to demonstrate strictness, as inclusion is deduced from Lemma 3.43 and Theorem 3.29. Strictness is shown in a way very similar to Theorem 3.41. Let $L = \{b\}^* \cdot \{a\} \cdot \{a\}^* \{b\} \cdot \{a, b\}^\omega$. This language is defined by $\Diamond(a \wedge \Diamond b)$ in LTL

and $A\Diamond(a \wedge A\Diamond b)$ in ACTL. We rely on a lemma stating that, for every ULSC S, either $\mathscr{L}(S)$ misses some word from L or $\mathscr{L}(S)$ is useless in a specification, for $\mathscr{L}(S) = \Sigma^\omega$. This lemma is formally stated and proved below. With such a lemma at hand, it comes immediately that, in every ULSC Specification $\mathcal{S}$, some word from L must also be missing in $\mathscr{L}(\mathcal{S})$, unless $\mathscr{L}(\mathcal{S}) = \Sigma^\omega \neq L$. □

Lemma 3.45 There is a language $L \in \mathsf{ULA}$ such that, for every ULSC S, either $\mathscr{L}(S) = \Sigma^\omega$ or $\exists \gamma \in L : \gamma \notin \mathscr{L}(S)$. ■

Proof 3.45

Let $L = \{b\}^* \cdot \{a\} \cdot \{a\}^* \{b\} \cdot \{a, b\}^\omega$. Consider some ULSC S. Its language is of the form $\overline{\Sigma^* \cdot V \cdot \overline{\Sigma^\omega}}$, with V, W A-FCmP languages. Suppose that $\mathscr{L}(S) \neq \Sigma^\omega$. Then, there is a word $\gamma \notin \mathscr{L}(S)$. By definition, $ab\gamma \notin \mathscr{L}(S)$, neither, since $\gamma \notin \mathscr{L}(S)$ iff $\gamma \in \Sigma^* \cdot V \cdot \overline{W \cdot \Sigma^\omega}$. But $ab\gamma \in L$. Thus, we obtain the desired result. □

3.3.3 Succinctness

Expressiveness provided us with a first criterion to compare languages. We showed that ULSC-Spec is strictly less expressive than many other languages, either automata-based (ULA, DBA, ...) or based on logic, like LTL. Our goal, in the previous section, was to sustain the first part of our thesis: those classical formal languages are probably too expressive, for real-world usage. ULSC-Spec enjoys a "reduced expressiveness", which would make it more suitable.

In this section, we consider a more fine-grained criterion to compare ULSC-Spec with other linear-time specification formalisms, viz. *succinctness*. Succinctness is a *quantitative* notion. Intuitively, a language L_1 is as succinct as a language L_2 (within some to-be-defined bound) if there is a computer program translating every model of L_1 to an equivalent model of L_2, with only some reasonable size inflation. The difficulty of definition 3.24 is to introduce the notion of "reasonable size inflation". As a first approximation, one could ask the model in L_2 not to be larger than L_1. However, this is too drastic as, for instance, if there is a bijective mapping between L_1 and L_2 constructs, we would expect these two languages to be just as succinct as each other. Yet, if keywords in L_2 are longer than in L_1, we will not get this result. The situation is even worse if some L_1 keywords are longer than L_2's but the reverse holds for other constructs. Then, two "intuitively equivalent" formalism would become incomparable. Thus, we are interested in "rough" bounds, as in computational complexity. Secondly, defining what "the size of a model" is, can raise subtle difficulties. For instance, in the case of LTL, we will have to consider two cases: the number of operators appearing in a formula and the number of distinct sub-formulae [54].

The translations from ULSC-Spec to other languages proposed in the previous section are all computable. However, none of them is efficient. Actually, all of them is exponential. They give us a first theorem: all the languages discussed

in the previous section (DBA, NBA, LTL, ULA, ACTL, ACTL$^{\text{det}}$, ULA) are exponentially as succinct as ULSC-Spec. On the one hand, it is interesting to know that ULSC-Spec can be embedded in all these languages, as it opens the way for tool reuse. On the other hand, an exponential upper-bound is neither very informative (what if there is a more efficient translation) nor very encouraging (even efficient tools are not of practical use if their input is gigantic). In this section, we show that these bad translations are actually optimal for all translations, but for the translation of ULSC-Spec to LTL and, consequently, to ALA.

From the point of view of tool reuse, this is bad news. However, this result sustains very well our thesis about the naturalness of ULSC-Spec. It can be seen as a "specialized" language, capable of expressing some facts in a much more compact way than other "wide-spectrum" languages.

The translation from ULSC-Spec to LTL presented in the proof above can yield formula that are exponentially larger than the specification. This statement should be understood as "there is a family of ULSC-Spec $F = \{\mathcal{S}_i\}_{i=0,1,\ldots}$ such that the function associating $|\mathcal{S}_i|$ to i is in $\mathcal{O}(n)$ but the size function of $\{\varphi_{\mathcal{S}_i}\}_{i=0,1,\ldots}$ is in $2^{\Omega n}$. This construction can be made more efficient, using a counting automaton. In order for this construction to be correctly defined, we need to introduce some technical subtlety, viz. the concept of deterministic LPO. An LPO is deterministic if its unordered locations have different labelings. Or, equivalently, if all locations with identical labels are ordered. These LPOs are called deterministic because their associated transition system is deterministic (and vice-versa: an LPO with a deterministic transition system is necessarily deterministic). We do not prove this fact here, because it will not be used in the succinct translation between ULSC-NC and LTL. The interested reader will find the proof in [24] and [27].

In full generality, deterministic LPO is strictly less expressive than CLPO.

Proposition 3.46 (DLPO $\subset$ LPO) There is an LPO $\mathcal{L}$ such that no deterministic LPO has the same set of linearizations as $\mathcal{L}$. ■

Proof 3.46

Consider the following LPO: $\mathcal{L} = \begin{pmatrix} l_1 : b & & l_2 : c \\ \downarrow \leq & & \downarrow \leq \\ l_3 : a & & l_4 : a \end{pmatrix}$

Its set of linearizations is $\{bcaa, baca, cbaa, caba\}$. Now, assume that some deterministic LPO has the same linearizations. It necessarily has 4 locations, with the same labeling. First, its two a labeled locations must be ordered $l_3 < l_4$. Secondly, remark that l_1 must be ordered with either l_3 or l_4. Otherwise, there would be a linearization in which b occurs after two a. Assume thus that $l_1 < l_4$, because there is one a occurring before b in $caba$, thus $l_1 < l_3$ is not possible. Apply the same reasoning to l_2 and deduce that $l_2 < l_4$, too (as otherwise, $baca$ would not be possible). However, we then obtain that $abca$ is a possible linearization, which is a contradiction. □

Nevertheless, when considering LPOs obtained from ULSC-NC, they are equivalent. Indeed, two equivalently labeled locations are necessarily ordered, because they must appear on the same lifeline or on the ends of the same arrow, unless they belong to the same coregion. If they do belong to the same coregion, they can be arbitrarily ordered without changing the semantics of the ULSC-NC.

We will show that, for the subset of ULSC-NC, that are thus mapped on deterministic LPOs instead of CLPOs, there is a translation to a linear alternating automaton, of size polynomial in the LSC. A linear alternating automaton is an alternating automaton, i.e. its states are partitioned between universal and existential states, with the constraint that its transition relation induces a partial order on states. In words, one can assign "levels" to the states. In every state, only transitions to strictly lower levels or self-loops are allowed.

If one considers the size of an LTL formula as its number of *distinct* subformulae, then linear alternating automata can be linearly translated to LTL [151, 109]. The translation is inductive, going "upwards" from lowest levels. Defining the size of an LTL formula as its number of distinct subformulae is not as unnatural as it might sound. Actually, the complexity of algorithms deciding satisfiability for LTL depends merely on this parameter [149, 171].

Consider a deterministic LPO (recall def. 3.1) and a finite word. Under what condition is that finite word a linearization of this DLPO? The answer is quite simple: when all locations appear in order. That is, suppose that location l appears at the j-th position in this word, then all predecessors of l must have appeared before the j-th position. Furthermore, this word shall be just as long as the size of the LPO. As a matter of fact, if $e_1 \dots e_n$ is a linearization of a DLPO, it is possible to map easily some location l to the position at which it "appears" in $e_1 \dots e_n$, by counting the number of $\lambda(l)$ labeled locations occurring before l in the DLPO.

Example 3.47 The sequence *acbcda* is a linearization of the DLPO of Fig. 3.33. It corresponds to the sequence of locations $l_0 l_1 l_2 l_3 l_4 l_5$. Now, take *abcdca*, matching $l_0 l_2 l_1 l_4 l_3 l_5$. Notice that, in both sequences, location l_3 appears at the position of the *second* occurrence of c. This is natural, as there are exactly 2 c-labeled locations $\leq l_3$. Hence, in every linearization, l_3 must be the second occurrence of c. This is true of every location: we can retrieve the position at which it appears in the word by counting the number of occurrences of its label. In a DLPO, we call the index of a location l ($idx(l)$) the number of locations bearing the same label as l and smaller than or equal to l. The translation used in the proof of Th. 3.48 relies on this fact. ■

Theorem 3.48 (ULSC-NC-Spec polynomially as succinct as LTL) ULSC-NC-Spec is $\mathcal{O}(n^6)$-as succinct as LTL, i.e. specifications made of universal LSCs *without choice* can be translated to LTL with a polynomial blow-up. ■

Proof 3.48

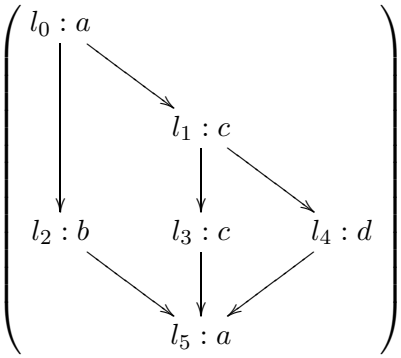


Figure 3.33: Deterministic LPO

The LTL formula that we build from an LSC $\Box(P, M)$ is of the form

$$\Box(\mathit{nprech} \vee \mathit{mainch}),$$

where

1. *nprech* is a formula that asserts that the prechart will not be matched by the subword starting at the current position. It is of the form

$$\bigvee_{l \in P} \mathit{notoccurs}(l) \vee \mathit{notorder}(l),$$

   where $\mathit{notoccurs}(l)$ asserts that there will not be $\mathit{idx}(l)$ occurrences of $\lambda(l)$ before having seen $|P|$ occurrences of restricted (Σ_R) events, and *notorder* is a disjunct over all direct predecessors of l. For every direct predecessor l', it says that the number of occurrences of $\lambda(l')$ is smaller than $\mathit{idx}(l')$ when the $\mathit{idx}(l)$-th occurrence of $\lambda(l)$ is encountered. Again, we verify this property within $|P|$ steps. This formula is of size $O(|P|^4)$, because we need 3 counters, ranging over $|P|$, and the outermost disjunction is over all prechart locations.

2. *mainch* is a formula asserting that, after $|P|$ occurrences of restricted events (i.e. exactly the prechart), for every l and l', where l' is a predecessor of l, l occurs after l' has occurred, yet within $|M|$ steps. Determining the position of l and l' relies on counting $\mathit{idx}(l)$ and $\mathit{idx}(l')$ occurrences of $\lambda(l)$ and $\lambda(l')$, respectively. Again, this formula is of size $O(|M|^5)$.

□

The translation proposed in the proof above is a generalization of the "pivot" technique proposed by the author to generate smaller formulae from universal LSCs in [23]. The original pivot technique considered only total order points and, in the worst case, could not help in reducing the size of generated formulae. Kugler *et al.* proposed a polynomial translation of LSCs to LTL [102] under the restriction that no event appears twice in the chart.

Theorem 3.48 states that there is a polynomial "reduction" of ULSC-NC-Spec to LTL. Hence, every decision problem on LSC will belong to the same complexity class as the same decision problem over LTL.

We will now prove that the same does not hold for automata-based formalisms, without alternation. There is a case in which LSCs always beat non-alternating automata. For achieving greater compactness, ULSCs *need* to use, in a crucial way, their ability to tell about the possible orderings of a sequence of events. Next, we thus introduce a sequence of languages, parameterized by some natural number n, called $CopyCat_n$. Depending on n, we let the alphabet be made of three types of events: $a_1, \ldots, a_n$, which are "triggers", \$, a marker, and $b_1, \ldots, b_n$, that are "responses". $CopyCat_n$ imposes the following rule: "whenever some permutation of $a_1, \ldots, a_n$ occurs, the marker must follow, and, immediately, the same permutation, over b." The ULSC-Spec-NC $CopyCat_n$ defines a language with such a property.

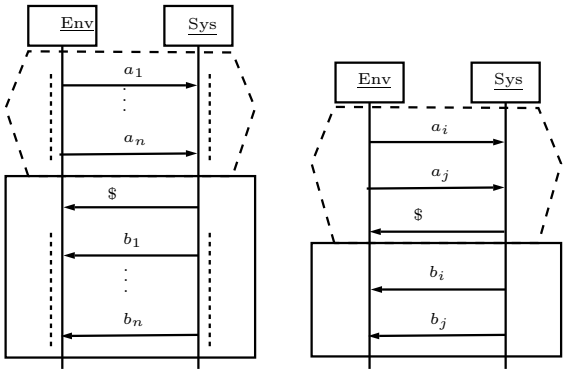


Figure 3.34: LSC specification $CopyCat_n$

Lemma 3.49 ($CopyCat_n$ can start with any permutation of $a_1 \ldots a_n$) For every n, and every permutation w of $a_1 \ldots a_n$, there is some $\gamma \in \Sigma^\omega$, such that $w\gamma \in \mathscr{L}(CopyCat_n)$. ■

Proof 3.49

Remark that all precharts of $CopyCat_n$ terminate with a marker \$. Since any permutation of $a_1 \ldots a_n$ does not include such a marker, they never activate any ULSC and thus are unconstrained by $CopyCat_n$. □

Lemma 3.50 ($CopyCat_n$ imposes matching order) For every n and every $\gamma \in \mathscr{L}(CopyCat_n)$, if there is some permutation of $a_1 \ldots a_n$, say $w = a_{i_1} \ldots a_{i_n}$ such that $w \sqsubset \gamma$, then $w\$b_{i_1} \ldots b_{i_n} \sqsubset \gamma$. ■

Proof 3.50

After w, the prechart of the first scenario is matched. Since $w \sqsubset \gamma$, this first scenario implies that there is some permutation of $b_1 \ldots b_n$, say v, such that $w\$v \sqsubset \gamma$.

Now, assume that events in w and v are not in matching order. We prove that this hypothesis leads to a contradiction. Precisely, w and v's events not being in "matching order" means that there are some indices j and k with $1 \leq j < k \leq n$, i.e. a_k precedes a_j in w, but b_k follows b_j in v. This is not possible, as this violates the requirement stating that if a_j precedes a_k then, b_j shall precede b_k, after the occurrence of \$ from $CopyCat_n$. This constraint corresponds to the right-hand side scenario of Fig. 3.34, where $i = k$. Therefore, $w\$v$ could not be a strict prefix of γ. □

Lemma 3.51 Consider a family of NBA $(\mathcal{A}_n)_{n=0,1,\ldots}$, with

$$\mathscr{L}(\mathcal{A}_n) = \mathscr{L}(CopyCat_n).$$

Then, there is some k such that, $\forall i \geq k.|\mathcal{A}_i| \geq 2^{i \log i}$. ■

Proof 3.51

Suppose that $\mathcal{A}_n$ does not grow exponentially, but polynomially. Assume, without loss of generality, that $\mathcal{A}_n$ does not contain any dead state (i.e. states from which no accepting state is reachable).

Because the growth of $\mathcal{A}_n$ size is only polynomial in n, at some point, there are less states in this automaton than there are permutations of $a_1 \ldots a_n$, because permutations of n grow like $2^{n \log n}$.

Consider two permutations w and w' of $a_1 \ldots a_n$, such that $w \neq w'$ and, yet, $\mathcal{A}_n$ reaches the same state after w and w', say q. Because $\mathscr{L}(\mathcal{A}_n) = \mathscr{L}(CopyCat_n)$, lem. 3.49 imposes the existence of q. Since there are no dead states in $\mathcal{A}_n$, there is an accepting infinite path starting at q, labeled by γ. Thus, $w\gamma \in \mathscr{L}(\mathcal{A}_n)$, but also $w'\gamma \in \mathscr{L}(\mathcal{A})$. Though, lem 3.50 imposes that γ starts with \$, followed by a permutation $b_1 \ldots b_n$ matching the permutation of w. The same holds true of w', which implies that $w'\gamma \notin \mathscr{L}(CopyCat_n)$. We reach a contradiction and are obliged to conclude that the size of $\mathcal{A}_n$ grows at least like $2^{n \log n}$. □

We have proved that there is no small automaton recognizing the same language as $CopyCat_n$. Yet, we need to check that the growth rate of $CopyCat_n$ is small, i.e. polynomial. This is achieved simply by counting the number of locations in the scenarios of Fig. 3.34.

Proposition 3.52 The size of $CopyCat_n$ is quadratic in n. Thus, it grows only linearly in n:

$$|CopyCat_n| = 5n^2 + 2n + 1.$$

■

As a corollary, they are also at least exponentially as succinct as DBA. It is quite easy to apply the proof of Lemma 3.51 to ULA as well. Hence, from the results of [114], which proved that ACTL^{det} is linearly as succinct as ULA, it

also comes that every translation from ULSC-NC-Spec to ACTL$^{\text{det}}$ also implies an exponential blow-up.

To conclude this section, we prove that there is a translation of ULSC-Spec to DBA which involves an exponential blow-up. The translation provided by the previous section were not concerned about the size of the resulting automaton, and yielded double-exponentially large Büchi automata. Thus, we will have shown that (1) ULSC-NC-Spec are exponentially as succinct as DBA and (2) ULSC-NC-Spec are exponentially more compact than DBA. Therefore, any translation of LSCs to DBA implies an exponential blow-up. We will also note that this translation requires simply exponential time.

We can use these results to show that ULSC-Spec can be translated to DBA, with only a single exponential blow-up. From a single ULSC $\Box(P, M)$ we build an automaton recording the last $|P|+|M|$ symbols of the word read. According to these symbols, it decides whether to allow a transition, if the next symbol read is not forbidden. The liveness condition is encoded in the acceptance condition.

Definition 3.53 ($\mathcal{A}_S$)

$$\mathcal{A}_S = \langle \{0,1\} \times ((\Sigma_R \cup \{\sqcup\})^n), \sqcup^n, \Delta, \{1\} \times (\Sigma_R \cup \{\sqcup\})^n \rangle,$$

with the transition relation being defined as follows: $\Delta((i,w),a,(i',w'))$ iff, let $b \in \Sigma \cup \{\sqcup\}$ and $w = bv$,

- Update sliding window:
  - if $a \in \Sigma_R$, then
    - $w' = va$,
    - whenever the prechart is matched and the main chart is *being* matched, there is a continuation starting with a (viz. at): $\forall pu \sqsupseteq w$: if $p \models P$ and $\nexists m \sqsubseteq u : m \models M$, then $\exists t \in \Sigma^* : uat \models M$.
  - if $a \notin \Sigma_R$, then $w' = w$.
- Update liveness counter:
  - if $i = 0$ then $i' = 1$ if either a is restricted ($a \in \Sigma_R$) or whenever the prechart is matched, the main chart is also matched afterwards: $\forall pu \sqsupseteq w : p \models P \implies \exists m \sqsubseteq u : m \models M$.
  - if $i = 1$ then $i' = 0$.

■

Lemma 3.54 ($\mathcal{A}_S$ is safe) $\mathcal{A}_S$ has a run on γ iff γ is Σ_R-safe. ■

Proof 3.54

Take some run γ which is not Σ_R-safe. It means that there is a prefix $w \sqsubset \gamma$ which forbids some $e \in \Sigma_R$ but $we \sqsubset \gamma$. Consider, without loss of generality,

the smallest such w, wrt $\sqsubset$ ordering. Remark that this implies that all prefixes of w are Σ_R-safe. We prove that w forbids e iff the subword of size n of w forbids e. By definition, w forbids e iff there is a decomposition of w in uv with (i) some $p \sqsubseteq v$ linearizing the prechart, (ii) no $m \sqsubseteq v$ linearizing the main chart and (iii), for every r, ver not linearizing the main chart. By definition, the length of v is smaller or equals to the maximal number of locations in the chart, i.e. n. Hence, only the last n elements of w determine the truth of the previous clauses.

Then, remark that the automaton records the last n restricted events of the word read, i.e. for every $w \in \Sigma^*$, if A_S reads w and reaches some state q, then, $w = vq$, after removing all blank ($\sqcup$) symbols from q.

Finally, consider a word $w \in \Sigma^*$. Let q be the state reached by A_S on w. We claim that a transition is defined from q on e iff e is not forbidden by w. By the observation above, w forbids e iff its n last restricted events forbid e, i.e. q forbids e. Remark that the transition condition embeds the safety definition, to ensure the desired result. Thus, $\mathcal{A}_S$ has a run on γ iff for every $a \in \Sigma_R$, for every prefix $wa \sqsubset \gamma$, w is a-safe iff γ is Σ_R-safe. □

Proposition 3.55

$$\mathscr{L}(\mathcal{A}_S) = \mathscr{L}(S)$$

■

Proof 3.55

We have to prove that $\mathcal{A}_S$ has an accepting run on γ iff $\gamma \models S$. Lemma 3.54 asserts that γ is safe iff $\mathcal{A}_S$ has a run on γ. Since $\mathcal{A}_S$ has an accepting run on γ, this run visits infinitely often states flagged with 1. This is only possible if either (i) some Σ_R symbol appears infinitely often in γ or (ii) ultimately, no $e \in \Sigma_R$ gets required by γ. The second clause can be checked by an argument similar to the proof of Lemma 3.54, by ensuring that only the n last restricted events in any word w determine whether w requires e or not. Then, since the automaton records exactly those last n events and the condition for flagging states embeds the liveness condition, we get that the run is accepting iff γ is Σ_R-live as well. □

The automaton $\mathcal{A}_S$ has three essential properties:

1. it is deterministic; This is easily checked, since, in $w' = va$ defines uniquely w', from the transition relation defined in 3.53.
2. its size is $2^{\mathcal{O}(n \log(n+1))}$. Remark that there are as many states as there are permutations of $\Sigma + 1$ elements of size n (n-tuples of $\Sigma \cup \{\sqcup\}$). There are $(|\Sigma| + 1)^n$ such permutations. Since $|\Sigma| = \mathcal{O}(n)$, we get that there are $\mathcal{O}((n+1)^n)$ states, i.e. $2^{\mathcal{O}(n \log n + 1)}$.
3. computing $\mathcal{A}_S$ from S is in EXPTIME. We simply need to show that, for every state q and every a in Σ_R, checking that $\Delta(q, a)$ can be done in EXPTIME. This is left to the reader.

Thus, we have shown that (1) every ULSC-Spec can be translated to an NBA, with an exponential blow-up and (2) this blow-up is unavoidable.

Theorem 3.56 ULSC-Spec is $2^{\Theta(n \log n)}$-as succinct as NBA (and DBA, and ULA). ■

3.4 Conclusion

In this chapter, we have formally defined the subset of Live Sequence Charts that will be used in the rest of this thesis. It is a graphical language and is given a very precise semantics, based on the classical logical concepts of interpretation and model. As a matter of fact, LSCs are a graphical form of temporal logic.

Thus, we can fall back on classical approaches to evaluate languages. In particular, we have compared LSCs with automata-based formalisms (NBA and DBA) and logic-based formalisms (LTL and ACTL^{det}). We assessed LSCs with respect to two criteria: expressiveness and succinctness. The former tells about the ability to describe more behaviours/objects/concepts in a language than in another. It turns out that LSCs are less expressive than all other languages to which they have been compared. We claim that they enjoy an *adapted* expressiveness, as it is still possible to model real-world problems with LSCs. The latter criterion, succinctness, allows us to tell whether a language can express some facts more compactly than another language. We proved that LSCs are more succinct than automata. They are also more succinct than ACTL^{det}.

In the next chapter, LSCs will be inserted into the development process of reactive systems. It will be explained *how* LSCs can be used and *when* we propose to use them, in a process life-cycle. Since modern software development is becoming ever more model-based [127], different models specialized for different phases of the software development life cycle, and some relationships between these models will be described. As we are dealing with formal languages, these relationships can be automated. Because LSC is a simple language, one can hope that this automation is practically feasible. We will show that this sensible expectation is actually false.

Chapter 4

Analysis Problems

Contents

What could you say to a girl who asked a question of such simplicity, when the answer was of great complexity?

A. Christie, "At Bertram's Hotel"

4.1 Introduction

Fig. 4.1 presents Harel's dream of supporting scenario-based software engineering through software tools. As is highlighted in [71], much research has been devoted to linking executable models, such as Statecharts, to code and vice-versa. This problem, although not obvious, amounts to high-level compilation.

In this chapter, we will consider the relation between "requirements", formalized as ULSC and a behavioural model. This model describes the behaviour of system components. Every component is a reactive system, i.e. reacting to changes in its environment. We will consider them from a very abstract point of view, as *strategies*. We have already defined the language of Live Sequence Charts and shown that this language is simpler than LTL and more compact than automata. On an example, we have illustrated that it is indeed possible to manually translate use cases, expressed in a textual form, to an LSC specification.

We will now investigate the problems related to adopting the dual inter/intra-agent view on behaviour. Each view has its own advantages: ULSCs are closer to informal use cases and can easily be matched against user's requirements, whereas strategies (automata) offer a component-based view which is easier to translate to a program fitting a particular software architecture. However, having two different views on a single conceptual model opens the way for inconsistency. The two views might be contradictory, in that they do not describe exactly the same underlying concept. Hence, engineers should be provided with tools that support their work and detect these inconsistencies as soon as possible. This is in essence what is proposed in [71]. We will consider three

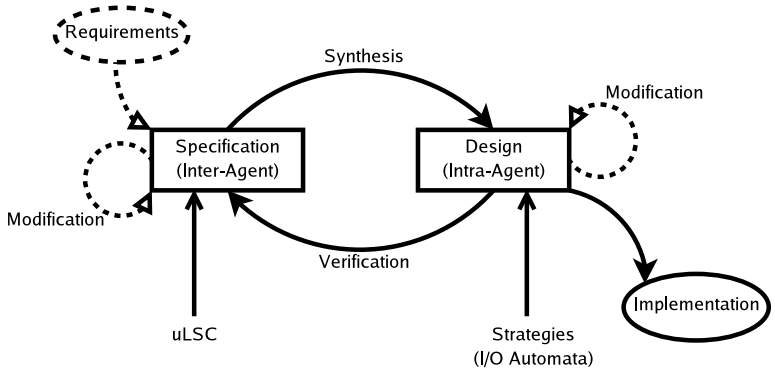


Figure 4.1: From Requirements to Code

problems that such tools could support: use case checking, synthesis and verification. Inter-relationships between these tools, models and a basic software development process are shown in Fig. 4.1.

1. *Use case checking* will be used by engineers when they are deriving an inter-agent specification from use cases. They constrain the future system's behaviour by adding more and more rules. Every new universal LSC restricts a little bit more the set of possible executions of the future system. At some point, it makes sense to check that it is *still* possible to perform a certain sequence of operations. If it is not, engineers need to relax the behaviour. This problem will be investigated in section 4.3.

2. When engineers are satisfied with their requirement specification, they have to build an intra-object design model of their system. Currently, this is a manual operation: analysts have to consider the whole specification, focus on a single agent and write down a state-machine implementation for it. This tedious and error-prone process can be automated thanks to *synthesis*. Synthesis is *not* a simple compilation problem; technically, it is a *constructive* proof that the specification is implementable, that it does not contain any conflicting requirements. This problem will be studied in section 4.5.

3. The design model, be it built automatically or manually, will often be modified by engineers. The specification is likely to be updated after the design model has been built, too. This phenomenon is known in Requirements Engineering as "requirements creep": during the life of a software project, new requirements are discovered and must be integrated to the software product [179]. As a matter of fact, the two artifacts, specification and design, will grow independently. Nevertheless, ultimately, the design model will *have to* comply with the requirements. Therefore, engineers need tools that will help them *verify* this compliance. We will analyze this problem in section 4.4.

4.2 Models

In this section, we present the abstract models that are used in Fig. 4.1. So far, we have only presented the language of LSCs, but we also need a language for describing the structure of the system to be built and the behaviour of these components.

4.2.1 Structure

The structural view of the system provides a static view of it. It lists the various agents and shows what events they trigger (send) or sense (receive).

Definition 4.1 (System Structure) A *system structure* (in abstract syntax) is a tuple

$$\langle Ag, (\Sigma_a^s)_{a\in Ag}, (\Sigma_a^r)_{a\in Ag}, Sys\rangle\,,$$

where

- Ag is a finite set of *agent* names.
- Σ_a^s gives, for every agent, the events that it sends. These sets are disjoint, as two agents may not control the same event.
- Σ_a^r is the set of events received by agent a. Again, these sets must be disjoint.
- $Sys \subseteq Ag$ is the set of *system agents*. All agents not in Sys are called *environment agents*: $Env \stackrel{\triangle}{=} Ag \setminus Sys$.

We let the set of all events be Σ and require that $\bigcup_{a\in Ag} \Sigma_a^r = \bigcup_{a\in Ag} \Sigma_a^s = \Sigma$. We let $\Sigma_a = \Sigma_a^r \cup \Sigma_a^s$. ■

In the rest of this section, we will refer to this structure and assume that it is fixed.

Concretely, a system structure is specified thanks to diagrams, which are a slight variation of *context diagrams* [91] and object diagrams [130] as shown in Fig. 3.2. Agents are drawn as boxes, in which their name is written. Ag is the set of all names appearing in boxes. System agents are tagged with an icon representing a computer terminal.

In order to describe Σ, we use *interfaces*. An interface is an ellipse, in which some event names are listed, separated by commas. If the list is too long, an alias can be given and expanded in an appendix, see Tab. 3.2. Let us denote with $I \subseteq \Sigma$ the set of events belonging to a given interface. There is an arrow going from the ellipse to agent a's box if $I \subseteq \Sigma_a^r$. There is a line (without arrow) linking agent a's box with the ellipse if $I \subseteq \Sigma_a^s$.

4.2.2 Inter-Agent Behaviour

Inter-agent behaviour is specified thanks to Live Sequence Charts. Every piece of textual requirements gets translated and generalized to some ULSC. These ULSCs are then gathered in a ULSC Specification. As ULSCs come with a pre-built notion of scenario integration, through conjunction, they form a consistent whole, specifying the desirable behaviours of the future system.

Of course, there are syntactic requirements, in order to be sure that the LSC specification complies with a given structure.

Definition 4.2 (Inter-agent specification) An *inter-agent specification* is a couple

$$\langle S, \mathcal{S} \rangle ,$$

with S being a system structure and $\mathcal{S}$ a ULSC-Spec, over alphabet Σ. It is required that only agents from Ag take part in $\mathcal{S}$ and that they respect their interfaces, i.e. an arrow from agent a_1 to a_2 may only be labeled by events from $\Sigma^s_{a_1} \cap \Sigma^r_{a_2}$. ■

4.2.3 Intra-agent Behaviour

System structure specifies which agents belong to the system (*Sys*) and which agents are part of their environment (*Env*). *Sys* implementation will be deployed among *Env* agents that provide thus the model-time context of the specification. It also tells what every agent *can* do. These abilities should be rendered in the behavioural specification of agents.

We consider abstractly agents as "reactive (sub-)systems". A reactive system keeps an ongoing relationship with its environment, reacting to environment inputs (stimulus) by producing outputs (responses). These reactuisThey react to changes in their environments, by sending new events. Our abstraction highlights that every agent has some powers: it can make some events happen, because they are under its control. Other events, which are beyond its control, cannot be handled or constrained by this agent. Thus, a single agent *cannot* force to nor prevent its environment from performing an action. Agents can only select events in such a way that, according to their knowledge of the environment, they will lead the environment to behave in a certain fashion. Our abstract view of agent a is a *strategy* $f : \Sigma^* \to 2^{\Sigma^s_a}$. A strategy f represents an agent for which it is advisable to perform any action in $f(w)$ after some history w. Although this view is very appealing from a mathematical point of view, we will have to focus on strategies which are representable within computers. We use the notion of input/output automata for this purpose [110].

An input-output automaton for agent $a \in Ag$ is a finite automaton, the alphabet of which is Σ_a. A distinction is made between input events (Σ^r_a) and output events (Σ^s_a). Syntactically, an I/O automaton for agent a must be *input-enabled*: in every state q, agent a should have one transition labeled by every input event. In other words, a may never block incoming events. This fulfills our desire to model agents powers, as explained above.

Definition 4.3 (Input/Output Automaton) An *input/output automaton* (I/O Automaton) is a tuple

$$\langle q_0, Q, \Delta, \Sigma_i, \Sigma_o \rangle,$$

where

- Q is a set of states,
- $q_0 \in Q$ is an initial state,
- $\Delta \subseteq Q \times (\Sigma_i \cup \Sigma_o) \times Q$ is a transition relation. The transition relation should fulfill the two following constraints:
  - $\forall q \in Q : \forall e \in \Sigma_i : \exists q' \in Q : (q, e, q') \in \Delta$,
  - $\forall q \in Q : \exists e \in \Sigma_o : \exists q' \in Q : (q, e, q') \in \Delta$.
- Σ_i is a set of *input events*,
- Σ_o is a set of *output events*.

The second constraint on Δ is not standard, but has been added to ensure that I/O automata actually correspond to totally defined strategies. Practically, a simple "stutter" move, labeling self-loops, can be added to Σ_o to satisfy this requirement. ■

A run of an I/O automaton is an infinite path in the automaton, following the transition relation and starting from the designated initial state. A fair run is a run in which infinitely many transitions labeled by output events are taken. The word generated by a run is the infinite sequence of events encountered along the transitions of the run. The language of an I/O automaton $\mathcal{A}$, denoted $\mathscr{L}(\mathcal{A})$, is the set of words generated by $\mathcal{A}$'s fair runs. The composition of two I/O automata $(\mathcal{A}_1 \times \mathcal{A}_2)$ is defined as a variation of the classical synchronous product of automata, see [110] for details. We quickly recall how this operation works and some of its basic properties.

Definition 4.4 (Composition of I/O automata) The composition of two automata $\mathcal{A}^1$ and $\mathcal{A}^2$ is defined if their output events are distinct $(\Sigma_o^1 \cap \Sigma_o^2 = \emptyset)$. In that case, $\mathcal{A} = \mathcal{A}^1 \times \mathcal{A}^2$ is

1. $Q = Q^1 \times Q^2$;
2. $q_0 = (q_0^1, q_0^2)$;
3. $\Sigma_i = (\Sigma_i^1 \setminus \Sigma_o^2) \cup (\Sigma_i^2 \setminus \Sigma_o^1)$ i.e. only input events controlled by neither agents are input events of the composition;
4. $\Sigma_o = \Sigma_o^1 \cup \Sigma_o^2$: "local events" are not hidden, in order to ensure associativity;
5. $\Delta((q^1, q^2), e, (s^1, s^2))$ iff

- $e \in \Sigma^1 \cap \Sigma^2$ and $\Delta^i(q^i, e, s^i)$, for $i = 1, 2$;
- or, $e \in \Sigma^1 \setminus \Sigma^2$, $q^2 = s^2$ and $\Delta^1(q^1, e, s^1)$ or vice-versa.

■

The composition operation enjoys the following properties:

Lemma 4.5 For every I/O Automata $\mathcal{A}^1, \mathcal{A}^2, \mathcal{A}^3$, provided composition is defined, we have

Associativity: $\mathcal{A}^1 \times (\mathcal{A}^2 \times \mathcal{A}^3) = (\mathcal{A}^1 \times \mathcal{A}^2) \times \mathcal{A}^3$.

Commutativity: $\mathcal{A}^1 \times \mathcal{A}^2 = \mathcal{A}^2 \times \mathcal{A}^1$.

Refinement (Trace inclusion): $\mathscr{L}(\mathcal{A}^1 \times \mathcal{A}^2) \subseteq \mathscr{L}(\mathcal{A}^1)$

■

Proof 4.5

Associativity and commutativity are shown in [110]. The former relies on the fact that $\mathcal{A}^1$ output events caught by $\mathcal{A}^2$ are not hidden. Trace inclusion comes from the fact that $\mathcal{A}^2$ cannot block an $\mathcal{A}^1$ transition in the composition, by input-enabledness (see def.4.3) . Therefore, fairness is preserved. □

A finite state I/O automaton represents a finite-memory strategy for agent a. Formally, a (non-deterministic) strategy for agent a is a function $f : \Sigma^* \to 2^{(\Sigma_a^s)}$. It is of finite memory if there is an equivalence relation $\simeq$ on Σ^* such that (1) $\simeq$ is of finite index and (2) $\forall w \simeq w' : f(w) = f(w')$. The size of the memory is the index of the smallest such equivalence relation. Clearly, every finite memory strategy can be translated to an I/O automaton. Conversely, every I/O automaton can be turned into a strategy. The *outcome* of a strategy f is the set of all runs in which Σ_a^s events appear only according to the strategy:

$$Out(f) = \{u_0 e_0 u_1 e_1 \ldots | \forall i \geq 0 : u_i \in (\Sigma \setminus \Sigma_a^s)^* \text{ and } e_i \in f(u_0 e_0 \ldots u_i)\}.$$

Agents can be organized in societies. A society is a set of agents $A \subseteq Ag$. Its triggered events and sensed events are the union of all triggered/sensed events of its composing agents: $\Sigma_A^s = \bigcup_{a \in A} \Sigma_a^s$ and $\Sigma_A^r = \bigcup_{a \in A} \Sigma_a^r$. The strategy of A is also the union of its agent's strategies: $f_A(w) = \bigcup_{a \in A} f_a(w)$.

It is easy to check the following proposition and we leave it to the reader:

Proposition 4.6 Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two I/O automata representing strategies f_{a_1} and f_{a_2}. Then, $\mathcal{A}_1 \times \mathcal{A}_2$ represents $f_{\{a_1, a_2\}}$. ■

Definition 4.7 An *intra-agent specification* is a couple

$$\langle S, f_{Sys} \rangle ,$$

where S is a system structure and f_{Sys} is a strategy for society *Sys*. ■

We are in position to define when a society of agents is behaving correctly, wrt some given LSC specification. Intuitively, agents within A are only required to respect the specification if agents outside A also do so. This is similar to the well-known assume/guarantee principle in Computer Science. In summary, agents are only responsible for the correct occurrence of their *own* events.

Definition 4.8 (Correct Implementation) An intra-agent specification $(S, \mathcal{F})$ associated to a society of agents *Sys* is a *correct implementation* of an inter-agent specification $(S, \mathcal{S})$ iff

$$\forall \gamma \in Out(f_{Sys}) : \left\{ \begin{array}{l} \gamma \text{ is } \Sigma_{Env}\text{-live} \implies \gamma \text{ is } \Sigma_{Sys}\text{-live} \\ \gamma \text{ is } \Sigma_{Env}\text{-safe} \implies \gamma \text{ is } \Sigma_{Sys}\text{-safe} \end{array} \right.$$

■

The reader will probably wonder about the condition above, because one is more naturally led to formalize our intent as:

$$\left(\begin{array}{cc} & \gamma \text{ is } \Sigma_{Env}\text{-safe} \\ \text{and} & \gamma \text{ is } \Sigma_{Env}\text{-live} \end{array} \right) \implies \left(\begin{array}{cc} & \gamma \text{ is } \Sigma_{Sys}\text{-safe} \\ \text{and} & \gamma \text{ is } \Sigma_{Sys}\text{-live} \end{array} \right)$$

This condition would allow the system to make unsafe moves disabling the possibility for the environment to be live. We give an example of this situation. We are asked to build a controller for a car lift. The controller can set the engine to three positions: up, neutral or down. It is expected that, when the engine is on position "down", the car lift will be going downwards. This is a liveness assumption on the environment. However, we require that the engine must be on "neutral" position when the brakes are on. Otherwise, the transmission breaks down and the car lift will never move anymore. It is possible to design an implementation fulfilling the condition above, yet naturely incorrect: the controller first breaks the transmission by setting the brakes on and putting the engine to position "up"'. Then, the controller sets the engine to position "down". Since the environment cannot fulfill its liveness constraints (the car lift cannot go down, because the transmission is broken), the condition above is vacuously verified, hence the implementation is correct. Our condition in Def.4.8 is stronger and rules out that situation. Def 4.8 clearly separates constraints that can be finitely falsified (safety) from conditions that can only be falsified by infinite runs (liveness) [4].

4.3 Use Case Checking and Refinement

The first problem we consider is whether an LSC specification is compatible with an existential LSC.

Problem 4.9 (Reachability) Given an ELSC $\Diamond(M)$ and an inter-agent specification $(S, \mathcal{S})$, decide whether

$$\mathscr{L}(\mathcal{S}) \models \Diamond(M).$$

■

REACHABILITY checks that a certain specification, together with assumptions over the domain still makes it possible to achieve a certain behaviour. In software engineering terms, REACHABILITY is used when one wants to check that the future system specification does not disallow a certain use case.

Theorem 4.10 REACHABILITY is complete for PSPACE. ■

Proof 4.10

(Membership) The proof of statisfiability from [149] is adapted to LSCs. Let $\mathcal{S} = \{L_1, \ldots, L_m\}$. Firstly, it is easy to devise a nondeterministic Büchi automaton recognizing all words $\gamma \in \Sigma^\omega$ such that $\gamma \models L$. This automaton, that we call $\mathcal{A}_E$, skips some finite input in its initial state and nondeterministically guesses when the LSC should be matched. Then, it follows the "cut transition system" and, as soon as some full cut is reached, it accepts all input. Secondly, we have to find a finite path π in the Büchi product $\mathcal{A}_E \times \prod_{i=1}^{m} \mathcal{A}_{L_i}$, where $\mathcal{A}_{L_i}$ is the tableau automaton defined in def. 3.53, such that

1. π is lasso-shaped, with a head of size j. The first state within the loop is the j-th state. Thus, there is a transition between the last state of π and its j-th state.
2. There is an accepting state in the segment between the j-th state and the last one. There is thus an accepting state visited infinitely often.

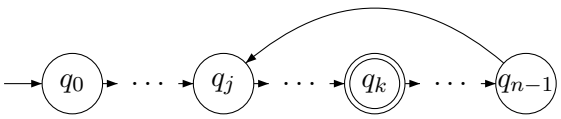


Figure 4.2: A satisfying run of length n

We may restrict ourselves to simple paths in $\mathcal{A}_E \times \prod_{i=1}^{m} \mathcal{A}_{L_i}$, i.e. paths in which no state appears twice. Actually, in all non-simple paths, there is a "shortcut" that skips the loop and is equivalent with respect to language emptiness. There is a nondeterministic PSPACE TM finding such a path. The reader shall notice here that π has a length upper-bounded by $(\log m + 1)2^{(m+1)\cdot n}$, with n the size of the largest scenario in the specification, because it is an upper bound on the number of states in $\mathcal{A}_E \times \prod_{i=1}^{m} \mathcal{A}_{L_i}$. This element can be recorded with $\log((\log m + 1)2^{(m+1)\cdot n})$ bits, i.e. $\mathcal{O}(m \cdot n)$ bits. The TM initializes itself by guessing the following information:

- the length of π,
- the value of j.

and then it runs to find such a π. The TM records the following information:

- current state in $\mathcal{A}_E \times \prod_{i=1}^{m} \mathcal{A}_{L_i}$,
- next state in $\mathcal{A}_E \times \prod_{i=1}^{m} \mathcal{A}_{L_i}$,
- current position (index) in π.
- the length of π,
- the value of j,
- the state at position j in π.
- whether some final state has been visited.

For $i = 1$ to $|\pi|$, the algorithm performs the following operations.

- if $i - 1 = j$, it stores the current state to q_j.
- it nondeterministically guesses the next state (q_i) and checks that there is a transition between q_{i-1} and q_i.
- If $i > j$ and q_{i-1} is accepting, it records that some accepting state has been visited in the loop.
- If $i = |\pi|$, it checks that $q_i = q_j$, i.e. the loop properly closes.

Then, it accepts. If any of the conditions above fails, it rejects. This algorithm uses only polynomial space and is nondeterministic. Since NPSPACE = PSPACE, we have shown that REACHABILITY is in PSPACE.

(Hardness) We encode the execution of a PSPACE Turing Machine on the blank input within an LSC specification. Assume that the control locations of the TM are taken from a a finite set Γ. Furthermore, suppose that the TM has been modified in such a way that, when it moves the tape head beyond the input, it loops forever in some non-halting state. We let the alphabet of the tape cell be the binary alphabet $\{0, 1\}$. Finally, we suppose that, among Γ, the halting location is γ_h, which is never left once it is reached. Since it is a PSPACE TM, it uses at most n cells of memory. The run of the TM will be encoded as an infinite word over the alphabet:

$$(\Gamma \cup \{in, \$\} \cup \{0, 1\}) \times \{0, \ldots, n\}.$$

The LSC specification contains only one agent; we will thus omit it in the rest of the proof. A correct encoding will have the following form $init \cdot exec$, where

$$init = (in, 0)(0, 0)(in, 1)(0, 1) \ldots (in, j)(0, j) \ldots (in, n)(0, n)(\gamma_0, 0) \tag{4.1}$$

The "init" sequence ensures that, at the beginning of the run, the tape cell contains n blank cells and the initial location is γ_0, with the tape head on cell 0. An event (in, j) requires the agent to perform $(0, j)$, i.e. to immediately initialize the j-th tape cell to 0.

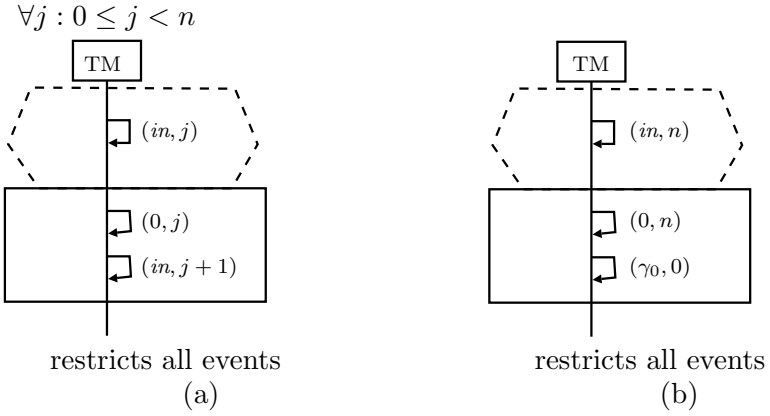


Figure 4.3: Initialization Sequence of PSPACE TM

We express this "initialization sequence" using the LSCs in Fig.4.3, which restrict *all* events.

Consider an arbitrary configuration of the TM: $C = (T, \gamma, i)$, where T is the tape content, γ is the control location and i is the tape head position. We say that it is encoded by a word w if

1. $\exists v \in \Sigma^* : w = v(\gamma, i)$
2. $\forall j : 1 \leq j \leq n : T[j] = a \implies \exists u, v \in \Sigma^* : w = u(a, j)v$ and neither $(0, j)$ nor $(1, j)$ appears in v.

Notice that, when w fulfills these conditions, C can be unambiguously retrieved from w.

It is easy to check that *init* encodes the initial configuration $C_0 = (T_0, \gamma_0, 0)$, where $T_0[j] = 0$, for all j. We need to express the successor relation between two configurations $C \rightarrow_M C'$.

Suppose without loss of generality, that $C = (T, \gamma, i)$, $T[i] = 0$ and $C' = (T', \gamma', i+1)$, where T' is like T, except that 1 has been written at the i-th position. Assume that C is encoded by some word w. By definition of configuration encoding, $w = v \cdot (\gamma, i)$, and the last occurrence of either $\{(0, i), (1, i)\}$ is $(0, i)$ in w. The transition will be encoded as the following continuation:

$$w' = v \underbrace{(\gamma, i)(0, i)(\$, i)(1, i)(\gamma, i+1)}_{u}.$$

One can check that w' is indeed an encoding of C', by noting that

1. it ends with $(\gamma, i+1)$;
2. in u, no event of the form $(0, j)$ or $(1, j)$ $(j \neq i)$ has been added. Hence, the tape content of the configuration encoded by w does not differ from that of C on these cells.

The proof is almost over, we simply need to describe all sequences of the form above with a conjunction of LSCs. This is achieved with the scenarios of Fig. 4.3 to 4.6. The first one retrieves the last occurrence of an event of the form $(0, i)$ or $(1, i)$. It is copied immediately after (γ, i). This retrieval is presented in Fig.4.4. One should take care of a detail, here: we want to be sure that after (γ, i), only *one* occurrence of $(0, i)$ will be repeated. This is achieved by using *no-scenarios*, the prechart asserts that matching a sequence of the form $(a, i)(a', i)(\$, i)$, where $a, a' \in \{0, 1\}$, should cause a contradiction in the specification. Therefore, such a "bad" encoding is forbidden.

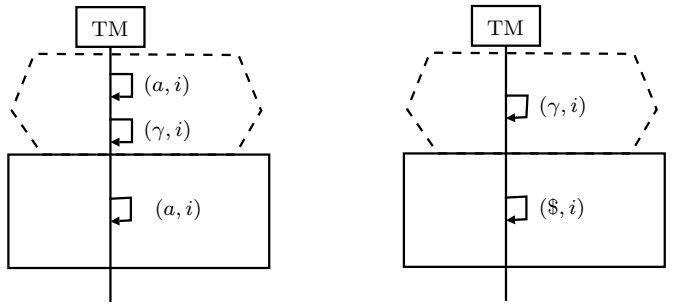


Figure 4.4: Retrieving tape cell content

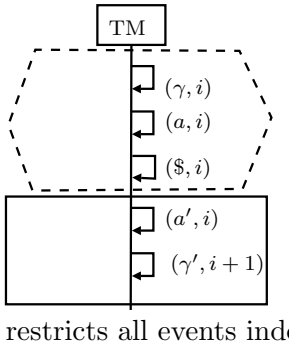


Figure 4.5: Transition $(\gamma, a, a', r, \gamma')$

A third scenario encodes the rest of the transition, i.e. writing to the i-th cell and moving the tape head to the right. This scenario is shown in Fig.4.5.

To conclude, we use the existential LSC to encode the property that, after having been initialized, the TM eventually halts. This scenario, in Fig.4.6, ignores all events, but the two in it.

□

Another natural problem pertaining to the analysis of inter-agent specifications is specification refinement checking. This occurrs naturally in the framework of a progressive software development approach. Given a certain abstract

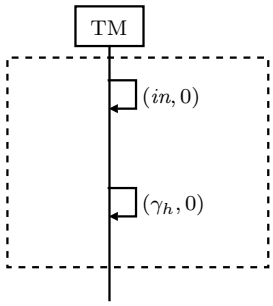


Figure 4.6: Existential scenario: TM initializes and eventually halts.

specification $(S, \mathcal{S})$, a more precise specification $(S, \mathcal{S}')$ is designed and we want to verify that every behaviour induced by $(S, \mathcal{S}')$ is a legal behaviour of $(S, \mathcal{S})$. Logically, this boils down to verifying the validity of $\mathcal{S}' \rightarrow \mathcal{S}$, or in language terms, that $\mathscr{L}(\mathcal{S}') \subseteq \mathscr{L}(\mathcal{S})$.

Problem 4.11 (LSC-Impl) The problem of implication of LSC specifications (LSC-Impl) is given two inter-agent specifications $(S, \mathcal{S})$ and $(S, \mathcal{S}')$, to decide whether

$$\forall \gamma \in \Sigma^\omega : \gamma \models \mathcal{S} \implies \gamma \models \mathcal{S}'.$$

■

Satisfiability of LSC specifications is polynomial-time reducible to reachability. One can add a scenario obliging the machine to perform an infinity of computations: every time it reaches the halting location, it is launched again, from the initial location. Hence, only runs in which the machine can "halt" from the initial location will be models of the specification.

Problem 4.12 (LSC-SAT) The problem of LSC satisfiability (LSC-SAT) is to decide, given an LSC specification $\mathcal{S}$, whether

$$\exists \gamma \in \Sigma^\omega : \gamma \models \mathcal{S}.$$

■

Hence, the two problems also considered in this section are as difficult as reachability. This is not surprising as reachability is an important primitive of most verification algorithms.

Corollary 4.13 LSC-SAT and LSC-Impl are PSPACE-complete. ■

In 2001, Harel and Marelly introduced an algorithm and an approach to the validation of LSC-based specifications, called *play-out* [75]. The specification is immediately executed, without generating any code from it, but using an animation engine instead. This animation engine uses a super-step approach: when the environment or user inputs some new event, by performing some action

on the graphical user interface, the engine performs all system-controlled events that become required, until it reaches some stable status, in which no event is required anymore. There may be more than one event required and enabled at the same time. Thus, several supersets may exist. The theorems provided in this section can be adapted to show that computing whether a finite superstep exists is PSPACE-complete. A technique named *smart play-out* as been developed by Kugler, Harel, Marelly and Pnueli to find a superstep avoiding deadlocks and divergence, if it exists. This technique translates the LSC to a transition system and, thanks to a model checker, finds an execution of this system that succesfully terminates. This execution is then used to drive play-out.

4.4 Verification

In this section, we will investigate the problem of agent verification. Informally, this problem is to check that an implementation of a society is correct. We will consider several consecutive problems. The most general case considers that the society *Sys* consists of at least one agent, and that there might be agents out of *Sys* interacting with them. We will investigate "degenerated" versions, along the following axes:

1. *Sys* consists of a single agent or several agents (viz. centralized vs distributed agent verification);
2. *Env* is empty or not (viz. closed vs open agent verification).

We will start with the simplest problem and progressively consider more difficult ones.

Problem 4.14 (CCMC) CCMC (Closed Centralized Model Checking) is the problem of verifying that an intra-agent specification (S, f_{Ag}) is a correct implementation of an inter-object specification $(S, \mathcal{S})$, with $Sys = Ag$ and f_{Ag} being presented as an (I/O) automaton $\mathcal{A}$. ■

In the hardness proof of "closed centralized model checking", we will make use of the fact that the complement of "Traveling Salesman Problem" is coNP-complete.

Problem 4.15 (coTSP) The *Complement Traveling Salesman Problem* (coTSP) is to decide whether, for some given constant k, in a given complete graph G, with weights on edges d_{ij}, all tours have a total weight $\geq k$. The weights are all polynomial in $|G|$. ■

Even with the additional assumption that weights are polynomial in $|G|$, this problem is coNP-complete. Indeed, by inspecting the hardness proof in [132], it actually suffices to consider weights bounded by 2 to obtain coNP-hardness.

Theorem 4.16 CCMC is complete for coNP. ■

Proof 4.16

(Membership) A counter-example is a path in which (i) the prechart is matched and (ii) the main chart never finishes or a safety condition is not met. Such a violation must occur in at most n steps, where n is the number of locations in the Live Sequence Chart. The non-deterministic algorithm guesses the following elements: the LSC L to violate, a state q in $\mathcal{A}$ and a simple path in the synchronous product $\mathcal{A}\times\mathcal{A}_{\neg L}$, with $\mathcal{A}_{\neg L}$ is the linear nondeterministic Büchi automaton recognizing all counter-examples of L. Remark that the simple path is at most of length $n \times |\mathcal{A}|$, which suffices to obtain that the algorithm runs in time polynomial in the size of the intra- and inter-agent specifications.

(Hardness) There is a polynomial reduction of Complement TSP (see [132]) to CCMC. Here, we consider a special case of CCMC, in which all events are system-controlled. A graph G, with a distance d_{ij} is turned into an automaton having states of the form (vertex, counter). The counter sums the weight of the current path, up to the current state. Of course, this counter is bounded by the longest possible path in G. It is thus polynomial in $|G|$, too. The alphabet is the set of vertexes from G. From a state (v, n), there is a transition $(v, n) \xrightarrow{v'} (v', n + d_{qq'})$, iff the edge between q and q' in G has weight $d_{qq'}$. Thus, a path $(v_0, i_0) \dots (v_j, i_j)$ in the automaton corresponds to a path $v_0 \dots v_j$ in G. Furthermore, the total weight of $v_0 \dots v_j$ is i_j.

In any state, there is also a transition to the "billing" states: $(v, n) \xrightarrow{\$} (\$, n)$. From these states, the automaton counts down, decreasing the counter by one unit at a time, until its counter equals 0: for $n > 0$, $(\$, n) \xrightarrow{\text{tick}} (\$, n - 1)$. When zero is reached, the automaton reads an infinite sequence of "end" events: $(\$, 0) \xrightarrow{\text{end}} (\$, 0)$. Finally, we add an initial state q_0, with a transition $q_0 \xrightarrow{\text{init}} (q, 0)$, for all q. This automaton has $2 + D \cdot (|G| + 1)$ states, where D is the maximal distance. It is thus polynomial in the size of the original graph.

The fact that all tours have length $\geq k$ is encoded in an LSC as follows: the prechart contains $\{q_1, \dots, q_n, \$\}$, where q_i's are unordered, whereas \$ is greater than all q_i.

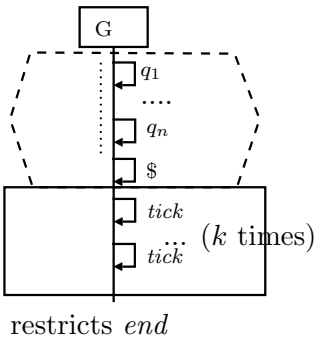


Figure 4.7: All tours have length $\geq k$

The prechart is matched when all vertexes have occurred exactly once and, then, the automaton has announced that it will start down-counting. Then, the

main chart checks that tick occurs k times, without any end event in between. It is clear that there is a tour of total weight $< k$ iff the automaton violates the LSC, i.e. the prechart is matched (we found a tour), but the main chart is violated afterwards. Violating the main chart means that, before k ticks, the "end" event occurs. Hence, the total weight of the tour is smaller than k. □

A first extension to this problem is to consider that some agents belong to the environment, while others are system agents. Then, we are presented with an implementation of system agents only and the question becomes: "whenever environment agents do behave correctly, does this implementation behave appropriately?".

Problem 4.17 (OCMC) OCMC (Open Centralized Model Checking) is the following problem: "Decide whether (S, f_{Sys}) is a correct implementation of $(S, \mathcal{S})$, where f_{Sys} is represented as an I/O automaton $\mathcal{A}$". ■

Theorem 4.18 OCMC is complete for PSPACE. ■

The proof of this theorem is similar to the proof provided in Sec. 4.3. The computations of a PSPACE Turing Machine can be encoded in an LSC specification, in polynomial-time and logarithmic space. The automaton generates only traces starting with an initialization event and, eventually, emitting a halting event.

The second restriction imposes that we consider monolithic systems only, made of a single component. As it was clear from the introduction, we are mostly interested in distributed systems. The design-time specification of such systems will typically be presented as a "network" of automata, one for each agent. Every automaton prescribes how its owner shall behave.

Problem 4.19 (CDMC) CDMC (Closed Distributed Model Checking) is stated as follows: "Decide whether an intra-agent specification (S, f_{Sys}) is a correct implementation of an inter-agent specification $(S, \mathcal{S})$, with f_{Sys} represented as a set of I/O automata $\{\mathcal{A}_a\}_{a \in Sys}$. ■

Unfortunately, as usual in verification, distribution makes model checking more complex [78]. The problem becomes PSPACE-complete instead of coNP-complete. Remark that , CDMC is a degenerated problem, because the specification contains only *one* ULSC. Considering an actual specification is not harder. Actually, there is an immediate nondeterministic PSPACE algorithm deciding the complement of the problem: pick nondeterministically one scenario in the specification and check that the implementation violates it. This problem is exactly the complement of CDMC, which is thus in coPSPACE=PSPACE, by Savitch's theorem [146].

Theorem 4.20 CDMC is PSPACE-complete. ■

Proof 4.20

(Membership) Let m be the size of $\mathcal{A}_i$'s and the LSC be of size n. By Savitch's theorem, it suffices to build a nondeterministic PSPACE Turing machine deciding the complement of the distributed model checking problem. This algorithm guesses an initial state and a path in the product of the automata. As this path needs to be ultimately periodic, it also guesses the following elements: the index in the path at which the loop is entered and the length of the path, as in [149]. We then check that the transition relation of the LSC is correctly followed , thus only two configurations need to be saved, plus the configuration at the entry of the loop. Within the loop, either no environment event occurs, but no such event is required, or some event occurs infinitely often.

(Hardness) Consider an arbitrary PSPACE Turing machine. Assume that its set of control locations is Γ and its symbols are Σ. One can without loss of generality, assume that the machine uses only its input space. Otherwise, the input can be padded with n^k blank spaces, see In-Place Acceptance in [132]. For every cell tape, we build an automaton, say $\mathcal{A}_i$. The alphabet of the system is $\{\text{init}, \text{halt}\} \cup (\Gamma \times \{1, \ldots, n\})$. An event (γ, i) means that the tape head moves to cell i and the control location becomes γ. $\mathcal{A}_i$ has two types of control locations, to record the fact that the tape head is on its cell or not. The former is of the form $(a, \gamma) \in \Sigma \times \Gamma$ and the latter of the form $a \in \Sigma$. Assume that we want to encode a transition $(\gamma, a, r, a', \gamma')$, i.e. when the TM control location is γ and it reads a from the cell on which the tape head resides, the TM writes a', moves the tape head to the right and the control location becomes γ', of the Turing machine. Let the tape head be on cell i. Then, $\mathcal{A}_i$ will contain a transition $((a, \gamma), (\gamma', i+1), a')$, while $\mathcal{A}_{i+1}$ has a transition $(b, (\gamma', i+1), (b, \gamma'))$. All automata synchronize on a first common event "init". The "init" event is caught by the prechart. The main chart then asserts that "halt" will eventually occur. □

Combining distribution and openness does not increase the problem complexity; it is still PSPACE-complete.

Theorem 4.21 ODMC is PSPACE-complete. ■

4.4.1 Related Work

There has been much work on *model checking* and we only refer the reader to recent surveys [133, 38]. This technique has undergone many optimizations to match industrial standards, such as partial-order reduction [62, 182], symbolic state space exploration [124] or SAT-based model checking [19]. In 1996, a collegial paper by Clarke, Wing *et al.* reported many industrial success stories [39].

There has been some work on the verification of scenario-based languages. Alur and Yannakakis investigated the problem of model checking MSC [11]. Schäffer and Knapp build a model checker to verify that state-machine implementations allow traces specified by collaboration diagrams [147].

Klose and colleagues use LSC as a specification language and verify that reactive systems specified with STATEMATE satisfy them, using model checking [98, 97]. Their technique can deal with real-time constraints and has been applied to the validation of a radio-based train system [21]. The tool developed on the basis of this work is commercially available with Rhapsody.

Bunker uses LSC to specify the properties of a hardware VCI (Virtual Component Interface) bus [33, 32]. The protocol specified in this standard is modeled as LSC, which strengthens guarantees that the formal model reliably captures the intent of the standard. These properties are then translated to a series of assertions in temporal logic, using a tool named `lscAssert` and a register transfer level model of the implementation can be model checked.

Kupferman and Vardi study the problem of verifying open systems against temporal logic formulae in LTL, CTL and CTL*. This problem is called *module checking*[104]. The main difference here is that modules are open: they are finite structures in which states are partitioned into environment and system states. In environment states, choices are external. Properties shall thus be verified against *every* environment. Kupferman and Vardi show that this problem is substantially harder than model checking for CTL and CTL*. However, it coincides with model checking when one considers universal temporal logics, i.e. formulae should hold along all paths in the computation tree. This is the case of LTL and ACTL. It is also the case of ULSC specifications: they are linear-time properties by definition, hence universal temporal properties.

4.5 Synthesis

In this section, we turn to the last class of problems that we will consider. We want to determine whether agents can indeed be implemented in order to satisfy the protocol. Ideally, the proof of implementability should be constructive: some strategy, for every agent, must be built. Would this implementation be compact and readable, the burden of designing the system would be taken away from engineers. This would achieve Harel's "achievable dream" [71].

4.5.1 Centralized Synthesis

As in the previous section, we will consider two versions of this problem. The first version requires us to build a strategy for Sys, say f_{Sys}, which is represented as a single automaton $\mathcal{A}$. The second version, that we call "distributed", obliges us to find a "distribution" of f_{Sys} into $(f_a)_{a \in Sys}$. This problem turns out to be undecidable.

Remark that we will not be considering the problem of synthesizing *closed* agent systems. This is because this problem is rather trivial. It suffices to test whether $\mathscr{L}(\mathcal{S})$ is nonempty, which is formally equivalent to LSC-SAT.

We are more interested in the design of open agent systems. They are going to be deployed in adversarial environments. Under these conditions, the problem of implementability is not equivalent to satisfiability [2]. The question is more accurately posed as "is there an implementation of system agents

such that, when deployed in any possible environment, the specification will be respected?". Satisfiability asks whether there is a benevolent environment in which some implementation can be deployed in order to meet the specification. When considering implementability, engineers have to consider malevolent environment, that will always try to drive the system into faulty states. However, remember that, in our approach, not every environment needs to be considered, since ULSCs enable analyst to make explicit safety and liveness assumptions on environment agents. Hence, only environment agents satisfying those hypothesis must be considered. COAD is therefore concerned with building an implementation of a system that will work in *adverse conditions*.

Problem 4.22 (COAD) *Centralized Agent Design* (COAD) is the problem of deciding, whether there exists an intra-agent specification (S, f_{Sys}) that implements correctly a given inter-agent specification $(S, \mathcal{S})$. ■

It is important to remark that a *single* monolithic strategy, for all *Sys* is enough to solve COAD. This problem makes the *perfect information* hypothesis. This means that agents may observe every event and every agent knows instantaneously in what state other agents are.

There is an exponential-time algorithm solving COAD. A two-player game graph, with a one-pair Streett winning condition is built, with the property that a winning strategy for player 0 exists if the specification is implementable. The game graph is exponentially larger than the initial specification. So, let us fix $\mathcal{S} = \{L_1, \ldots, L_m\}$. We let every ULSC be of size $\mathcal{O}(s)$. We let n represent the size of $\mathcal{S}$. By definition, we have $\mathcal{O}(n) = m \cdot \mathcal{O}(s) = \mathcal{O}(m \cdot s)$, because there are m ULSCs of size $\mathcal{O}(s)$. The game graph is the product of the deterministic automata obtained from every LSC, see Def. 3.53. We will denote by $\mathcal{A}_i = \langle Q_i, q_i^0, \Delta_i, F_i \rangle$ the DBA recognizing all models of L_i. Remember that, in this automaton, states are finite words (of length $\leq |L_i|$). Thus, we may say that "a state requires/forbids a certain event", without any problem. Furthermore, we extend the alphabet Σ with a fresh environment-controlled event τ, which stands for environment stutters. This event is used by *Env* to warn the other player of a "turn change". When τ occurs, *Sys* has to perform an event afterwards. To ensure that *Env* lets *Sys* play infinitely often, a scenario is added to the specification, as shown by Fig. 4.8.

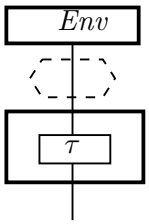


Figure 4.8: Ensuring that turns alternate

We also assume that there are two events, say λ_{Sys} (resp. λ_{Env}), in Σ^s_{Sys} (resp. Σ^s_{Env}) that are not restricted by any scenario in $\mathcal{S}$. This allows us to

deal easily with safety conditions by removing some pathological conditions and corresponds intuitively to the situation in which the system (resp. its environment) does nothing. In this setting, the following fact is easy to prove: every run in which no safety condition has been violated can be prolongated to a safe infinite run, by adding an infinity of "do nothing" steps.

Proposition 4.23 For every finite word $w \in \Sigma^*$, w is Σ-safe iff $w\lambda_j^\omega$ is Σ-safe as well. ■

When regarding only safety conditions, the first player to perform an unsafe move will be losing the game [1]. The first proposition asserts that, if *Env* performs the first bad move, then, there is a simple correct implementation that can continue the play from that situation.

Proposition 4.24 There is a strategy h_{Sys} such that for every $w \in \Sigma^*$, if w is Σ^s_{Sys}-safe but not Σ^s_{Env}-safe, $Out(h_{Sys}) \cap (w \cdot \Sigma^\omega)$ are all correct (see Def. 4.8). ■

Proof 4.24

Since all runs in $w \cdot \Sigma^\omega$ are not Σ^s_{Env}-safe, the strategy h_{Sys} has to check that the liveness condition is matched. The strategy does this in a simple way: it performs a restricted event for every scenario, in turn. Thus, in all outcomes, every scenario has some restricted event which occurs infinitely often. Formally, h_{Sys} is specified as the following I/O automaton:

$$\langle \{0, \ldots, m-1\}, 0, \Delta, \Sigma^s_{Sys}, \Sigma^r_{Sys} \rangle,$$

where $\Delta(i, a, i)$, for every $a \in \Sigma^r_{Sys}$, and $\Delta(i, b, j)$, with $b \in \Sigma_{R_i} \cap \Sigma^s_{Sys}$ and j is the first index $> i$ (modulo m) for which L_j restricts some event controlled by *Sys*.

□

The second proposition says that no correct implementation contains a trace in which *Sys* violates first some safety condition.

Proposition 4.25 Let f_{Sys} be a correct implementation. Then, there is no $w\gamma \in Out(f_{Sys})$ such that w is Σ^s_{Env}-safe but not Σ^s_{Sys}-safe. ■

Proof 4.25

By contradiction. Suppose that such a $w\gamma$ exists. Hence, substituting in γ all occurrences of Σ^s_{Env} events by λ_{Env} yields a run $w\gamma'$ that (i) belongs to $Out(f_{Sys})$ but (ii) is Σ_{Env}-safe and not Σ_{Sys}-safe. Hence, f_{Sys} is not a correct implementation. □

The game graph built to solve COAD is, as specified in Sec. 1.3, an object of the following form:

$$\langle V, V_0, \Delta, \Omega \rangle,$$

where, in our particular case,

- The set of vertexes is

$$V = \{Sys, Env\} \times \prod_{i=1}^{m} Q_i \times \{0, \ldots, m\} \times \{0, \ldots, m\}$$

  Thus, in addition to the product of the states from the ULSC "tableau", we add a flag, indicating whose turn it is to play (*Sys* or *Env*) and two counters, that will take liveness into account, as in the synchronous product of Büchi automata.

- Player 0's vertexes are all vertexes flagged with *Sys*:

$$V_0 = \{v \in V | \pi_1(v) = Sys\}.$$

- Edges are $\Delta \subseteq V \times \Sigma \times V$. There is a transition $\Delta(q, e, q')$ iff, if we let $q = (i, q_1, \ldots, q_n, c_{Sys}, c_{Env})$ and $q' = (i', q'_1, \ldots, q'_n, c'_{Sys}, c'_{Env})$, the following conditions hold:
  - Player *Env* passes his turn by making a τ move. Player *Sys* may only perform one move:
    - if $i = Env$ and $e = \tau$, then $i' = Sys$;
    - if $i = Env$ and $e \neq \tau$, then $i' = Env$.
    - if $i = Sys$, then $i' = Env$;
  - e is controlled by the player who it is to play in vertex v: $e \in \Sigma_i^s$.
  - All transitions are legal, with respect to every tableau:

$$\forall j : 1 \leq j \leq m : \begin{cases} q_j \text{ does not forbid } e \\ \Delta_j(q_j, e, q'_j) \end{cases}$$

  - Counters are updated as in Büchi synchronous product (for $j \in \{Sys, Env\}$):
    - $c_j = m \implies c'_{Sys} = 0$, or
    - $c_j < m$ and, if $e \in \Sigma_{R_{c_j+1}} \vee q_{c_j+1}$ does not require any $e' \in \Sigma^s_{c_j+1}$ then $c'_j = c_j + 1$ else $c'_j = c_j$.
  - $\Omega = Streett(E, F)$, where

$$\begin{aligned} E &= \{(i, q_1, \ldots, q_m, c_{Sys}, c_{Env}) \in V | c_{Env} = m\} \\ F &= \{(i, q_1, \ldots, q_m, c_{Sys}, c_{Env}) \in V | c_{Sys} = m\}. \end{aligned}$$

Finally, we obtain $G_{\mathcal{S}} = \langle V', V'_0, \Delta', \Omega' \rangle$, by "completing" the game graph above, i.e. adding new transitions to Δ to ensure that every vertex has a successor. We add two vertexes: $V' = V \cup \{Fail_{Sys}, Fail_{Env}\}$. They both belong to $V'_1 = V' \setminus V_0$ and act as sink states. Their only outgoing transition is labeled by τ: $\forall e \in \Sigma : \Delta'(Fail_j, \tau, Fail_j)$, for $j \in \{Sys, Env\}$. They are entered from a vertex v, when an unsafe event is performed in that vertex:

$$\begin{aligned} &\forall v \in V_0 : \forall e \in \Sigma^s_{Sys} : (\nexists v' \in V : \Delta(v, e, v')) \implies \Delta'(v, e, Fail_{Sys}) \\ \text{and} \quad &\forall v \notin V_0 : \forall e \in \Sigma^s_{Env} : (\nexists v' \in V : \Delta(v, e, v')) \implies \Delta'(v, e, Fail_{Env}) \end{aligned}$$

Finally, the vertex $Fail_{Sys}$ is good for player 1 while state $Fail_{Env}$ is good for player 0: $\Omega' = Streett(E \cup \{Fail_{Sys}\}, F)$. It is easy to see that $Fail_{Sys}$ belongs to the winning region of player 1: the winning strategy is to remain forever in that state. $Fail_{Env}$ is in the winning region of player 0, as staying there forever is winning for player 1.

Remark that $G_{\mathcal{S}}$ is deterministic. This follows easily from an inspection of the conditions above. The conditions defining i' are all mutually exclusive, as well as the conditions for the liveness counters c_{Sys} and c_{Env}. Since all $\mathcal{A}_j$ are deterministic, their product is also deterministic. The size of $G_{\mathcal{S}}$ is exponential in the size of the specification $\mathcal{S}$:

$$\begin{aligned} |G_{\mathcal{S}}| &= 2 \cdot \left(\prod_{j=1}^{m} |\mathcal{A}_j|\right) \cdot (m+1) \cdot (m+2) \\ &= 2 \cdot \left(\prod_{j=1}^{m} 2^{\mathcal{O}(s \log s)}\right) \cdot (m+1) \cdot (m+2) \\ &= \left(2^{\mathcal{O}(m \cdot s \log s)}\right) \cdot \mathcal{O}(m^2) \\ &= 2^{\mathcal{O}(n \log s)} \end{aligned}$$

The following lemma asserts that we are maximal, with respect to safety: the construction of $G_{\mathcal{S}}$ did not remove any safe move. To put it differently, every Σ-safe run corresponds to a path in the V part of $G_{\mathcal{S}}$.

Lemma 4.26 w is the prefix of a Σ-safe run iff the unique path labeled by w from $(Env, q_0^1, \ldots, q_0^n, 0, 0)$ leads to a vertex in V. Thus, v is not one of the two *Fail* states. ■

Proof 4.26

The proof is by induction. Clearly, ϵ leads to the initial state that belongs to V. Take a Σ-safe run γ and assume that the property above holds for a prefix of size n. Let we be the prefix of γ of size $n+1$. Let v be the vertex, by induction hypothesis in V, to which w leads. Then, w forbids some event e iff some $\mathcal{A}_j$ does in v, by construction of $\mathcal{A}_j$. Remark that w cannot forbid e, since we is a prefix of γ. By looking at the definition of Δ, it is clear that there is a continuation e to some vertex $v' \in V$. The other direction follows from Prop. 4.23, which asserts that every safe finite word can be prolongated to a safe infinite word. □

Every run $w\gamma$ where w is not Σ^s_{Sys}-safe but w is Σ^s_{Env}-safe labels a path from $(Env, q_0^1, \ldots, q_0^n, 0, 0)$ to $Fail_{Sys}$.

We show that the Streett acceptance condition encodes faithfully the liveness part of Def. 4.8.

Lemma 4.27 Let $\gamma \in \Sigma^\omega$ be a Σ-safe run. Then, the two following assertions hold.

1. γ labels a path π in $G_{\mathcal{S}}$ starting from $(Env, q_0^1, \ldots, q_0^n, 0, 0)$ that remains forever in V.

2. π visits infinitely often E (resp. F) iff γ is Σ^s_{Env}-live (resp. Σ^s_{Sys}-live).

■

Proof 4.27

First, γ labels a unique path π from $(Env, q_0^1, \ldots, q_0^n, 0, 0)$ to $Fail_{Sys}$, because Δ' is complete and deterministic. Since γ is Σ-safe, π never visits any $Fail_j$ state ($j \in \{Env, Sys\}$).

Second, suppose that E is visited infinitely often along π. By construction of counters, either every scenario has some restricted event that occurs or some state in which no Σ^s_{Env} event is required. In the former case, the run is clearly Σ^s_{Env}-live. We show that it is in the latter case, too. Suppose that $\gamma = w\gamma'$ such that (i) in γ' no restricted event occurs, (ii) there are infinitely many prefixes of γ that do not require any Σ^s_{Env} event but (iii) there is a prefix after w, i.e. $u \sqsubset \gamma$ and $w \sqsubset u$ that requires some Σ^s_{Env}. It is not possible as there is another $u \sqsubset w'$ which does not require some Σ^s_{Env} and, thus, $w|_{\Sigma_R} \neq u|_{\Sigma_R}$ or $w|_{\Sigma_R} \neq w'|_{\Sigma_R}$. This is not possible. □

Theorem 4.28 ($G_\mathcal{S}$ correctness) There is a winning strategy σ on $G_\mathcal{S}$ for player 0 from $(0, q_1, \ldots, q_n, c_0, c_1)$ iff there is a strategy f_{Sys} implementing $\mathcal{S}$. ■

Proof 4.28

($\Rightarrow$)

Suppose that σ is winning on $G_\mathcal{S}$. We show how to build a strategy f_{Sys} which implements $\mathcal{S}$. We define f_{Sys} as a memory strategy. The memory component $\mu : Q \times \Sigma \to Q$ is defined as follows, for every $e \in \Sigma$:

- $\mu(Fail_j, e) = Fail_j$ for $j \in \{Env, Sys\}$.
- $\mu(q, e) = \Delta(q, e)$, for all $q \in V$.

Then, by induction, $\mu^* : \Sigma^* \to Q$ is $\mu^*(\epsilon) = (0, q_1, \ldots, q_n, c_0, c_1)$ and $\mu^*(ua) = \mu(\mu^*(u), a)$.

- $f_{Sys}(w) = \sigma(\mu^*(w))$, if $\mu^*(w) \neq Fail_j$ ($j \in \{Sys, Env\}$).
- $f_{Sys}(w) = h_{Sys}(w)$, otherwise.

Suppose, for the sake of contradiction, that f_{Sys} is not a correct implementation. Then, one of the two following assertions must hold, for some $\gamma \in Out(f_{Sys})$ and we perform a case split.

γ is Σ^s_{Env}-safe but not Σ^s_{Sys}-safe. By Lem. 4.26, $Fail_{Sys}$ is then reached. However, we have already stated that $Fail_{Sys}$ did not belong to player 0 winning region, which contradicts the fact that σ is winning.

γ **is Σ^s_{Env}-live but not Σ^s_{Sys}-live.** If γ is Σ^s_{Sys}-safe, then, from Lem. 4.27, it comes that E is visited infinitely often but F is only visited finitely often. Thus, it does not fulfill the Streett acceptance condition.

Thus, f_{Sys} is a correct implementation.

($\Leftarrow$)

Suppose that f_{Sys} is a correct implementation. We build a winning strategy σ from it. $\sigma(w)$ is just the same as $f(w)$, but on vertexes rather than moves: $\sigma(w) = v' \iff \Delta(\mu^*(w), f(w), v')$. □

This game can be solved in time $\mathcal{O}(|G_{\mathcal{S}}|) = 2^{\mathcal{O}(n)}$. The Streett acceptance condition can be turned into a three-colour parity game graph, with the acceptance condition Ω' defined as

$$\begin{array}{lcll} \Omega'(v) & = & 2 & \text{if } v \in F \\ \Omega'(v) & = & 1 & \text{if } v \in E \setminus F \\ \Omega'(v) & = & 0 & \text{otherwise} \end{array}$$

As stated in Sec. 1.3, a parity game graph can be solved in time polynomial in the number of colours. We have just described an inefficient algorithm for solving COAD:

Lemma 4.29 COAD is in EXPTIME. ■

Nevertheless, this algorithm cannot be asymptotically improved: we prove that COAD is also hard for EXPTIME. We have already showed that *satisfiability* of LSC specifications was PSPACE-hard, see Sec. 4.4. Essentially, synthesis of *open* systems inserts *alternation* into the problem of satisfiability. This is witnessed by our approach for solving COAD, which reduces realizability to solving a two-player game. Since alternation shifts all complexity classes "one level up", PSPACE-hardness of satisfiability implies EXPTIME-hardness of synthesis [36].

Lemma 4.30 COAD is hard for EXPTIME ■

Proof 4.30

We encode an alternating PSPACE Turing machine into an LSC, as we did before (see Th. 4.10). The result will follow from the fact that APSPACE = EXPTIME [36]. The only difference is that we need to distinguish between universal and existential moves of the machine. Since alternation is built in the realizability problem, we can use the two statuses of the player to model the alternation of the Turing machine. In order to do so, we duplicate all events, and assign them to player 0 and player 1. A transition is now of the form $(\gamma, i, A)(a, i, A)\$_i(a, i, A)(\gamma', j, A')$, where $A, A' \in \{\forall, \exists\}$ indicates the status of the current state (universal or existential).

Since there are several possible moves at configurations (by definition of alternation), we need to encode these possible continuations. All bad continuations are encoded in no-scenarios, which imply contradictory requirements on

the player $(\forall, \exists)$ who is about to play. Thus, if this player decides to pick such a bad continuation, the outcome will certainly not respect the LSC specification. This is equivalent to complete "a priori" the TM transition relation, without altering its language.

Surprisingly, we assign existential moves of the TM to player 1 (the "universal player" in our two-player synthesis "game") and universal moves to player 0 (the "existential player" of the synthesis "game"). A scenario is added, ensuring that player 0 loses as soon as a halting configuration is met. The specification is not realizable iff the machine has an accepting computation. Actually, player 1 can pick existential moves such that the computation tree halts on all its paths (otherwise, player 0 would have a winning strategy to escape). □

Combining these two lemmas, we obtain that COAD is EXPTIME-complete. This proves our claim that, because LSCs are less expressive than LTL, some problems are easier on LSCs than on LTL. Actually, centralized realizability is 2EXPTIME-complete for LTL [134] and EXPTIME $\subset$ 2EXPTIME.

Theorem 4.31 COAD is complete for EXPTIME. ■

The algorithm presented above is computationally expensive, yet optimal. However, it suffers from another problem: it yields design models, as automata, that are exponentially larger than the specification. This is a hindrance for readability and a possible flaw of our algorithm. A synthesis algorithm yielding always small implementations would be much better. Nevertheless, this is hopeless: strategies realizing ULSC specifications *need* exponentially large memory. Therefore, our algorithm is optimal, in the sense that every algorithm solving COAD will necessarily build exponentially large implementations. We show this in the following theorem.

Theorem 4.32 (Memory Lower-Bound) There is a family of LSC specification, namely $(S, (CopyCat_n)_{n>0})$ such that every intra-agent specification realizing $CopyCat_n$ has a memory of size $2^{\Omega(n \log n)}$. ■

This result is similar to the fact that LSC specifications are exponentially more succinct than Büchi automata. This similarity is highlighted by the fact that we use $CopyCat_n$ to prove it.

4.5.2 Constrained and Distributed Synthesis

The centralized agent design problem (COAD) presented in Sec.4.5.1 is lacking some features, which lessens its applicability

1. It would be interesting to come up with an implementation which satisfies the specification and guarantees that additional requirements will be met as well. This is especially interesting if the specification is too abstract or too loosely defined to ensure the requirements, but the analyst thinks that it is possible to refine it in a way that would fulfill the

requirements. The problem of deciding whether there is such a particular implementation, which we call *constrained* centralized agent design is 2EXPTIME-complete, when we consider LTL as a language for expressing requirements.

2. It does not take agent interfaces into account, because it assumes that the "perfect information" hypothesis holds. Hence, agents are not obliged to consider only events occurring at their interfaces. It seems necessary to extend the centralized version of the problem to take this into account. This variant is called *distributed* agent design. As for LTL, this problem is undecidable [135].

Problem 4.33 (LTL-Cons-COAD) The problem of *LTL-Constrained Centralized Open Agent Design* (LTL-Cons-COAD) is, given an inter-agent specification $(S, \mathcal{S})$ and an LTL formula φ, to decide whether there is an intra-agent specification (S, f_{Sys}) such that

1. (S, f_{Sys}) is a correct implementation of $(S, \mathcal{S})$;
2. $Out(f_{Sys}) \models \varphi$.

■

Theorem 4.34 LTL-Cons-COAD is complete for 2EXPTIME ■

Proof 4.34

For membership, translate LSCs to DBA, via the tableau procedure. Then, combine these automata with the automata built from the LTL formula in [134]. Hardness comes from the fact that this problem generalizes LTL realizability, which is 2EXPTIME complete [134]. □

The problem of distributed agent design is to build a strategy for every agent in a society such that

1. agents respect their *interfaces*, i.e. agent a senses events from Σ_a^r only.
2. the society is well-behaving, with respect to an LSC specification.

Surprisingly, this problem is undecidable. Furthermore, the proof uses LSCs without any fancy constructs: no loops, no alternatives, no conditions, ...

Problem 4.35 (DOAD) The *Distributed Open Agents Design* (DOAD) problem is defined as: "Given an inter-agent specification $(S, \mathcal{S})$, is there an intra-agent specification (S, f_{Sys}) such that

1. (S, f_{Sys}) is a correct implementation of $(S, \mathcal{S})$
2. f_{Sys} is the composition of $(f_a)_{a \in Sys}$, with, for every f_a

(a) $f_a : \Sigma^* \to (\Sigma_a^s)$;

(b) $\forall w, w' \in \Sigma^* : w|_{\Sigma_a} = w'|_{\Sigma_a} \implies f_a(w) = f_a(w')$, i.e. if w and w' are the same, from a's point of view, then a shall behave the same way after w or w';

■

Theorem 4.36 DOAD is undecidable. ■

Proof 4.36

We reduce Post's Correspondence Problem (PCP) to the problem of deciding whether the specification is not implementable, following [160].

We first recall the definition of PCP. A PCP instance is a list of pairs of words
(w_1, u_1) ,..., (w_n, u_n), such that, for all i, $w_i \neq u_i$ and $w_i, u_i \in \Theta^*$ (for some finite alphabet Θ). A *solution to a PCP instance* is a finite sequence of indexes $i_1 \dots i_m$ ($m \geq 1$ and $1 \leq i_j \leq n$, for all j) such that $w_{i_1} w_{i_2} \dots w_{i_m} = u_{i_1} u_{i_2} \dots u_{i_m}$. The problem of telling whether any PCP instance admits a solution or not is undecidable.

Let us fix an arbitrary PCP instance. We show how to reduce the problem of determining whether this PCP instance admits a solution to DOAD. The alphabet of our LSC specification is $\Theta \cup \{k_1, \dots, k_n\} \cup \{\$\} \cup \{0, 1\} \cup \{A_0, A_1\}$, plus an arbitrary finite number of events that can be exchanged between system agents, say $\{s_0, \dots, s_q\}$. The system is made of two agents: a_1 and a_2. The first agent may observe $\Theta \cup \{\$\}$, whereas the second can observe $\{k_1, \dots, k_m, \$\}$. All these events, but $\{A_0, A_1\}$ and the additional system events $\{s_0, \dots s_k\}$ are controlled by the environment. A play proceeds as follows. First, the environment picks either 0 or 1. The former means that the environment chooses to read words in the first component of the pairs of words (viz. the w_i's), the latter means that it will read u_i's. Then, the environment must stick to that choice until the end of the play. Namely, the environment chooses a particular word in the list (say, w_i or u_i, depending on the "column" chosen) and indicates the index of this word to the system, by performing k_i. The environment must then enumerate the letters in w_i, which are thus published to agent a_1. The game goes on until the environment performs \$. At this point, the system is required to output A_0 or A_1, depending on what index (0 or 1) the environment had chosen in the first place.

We claim that the PCP instance has a solution iff this specification is not implementable. Assume that PCP has a solution $i_1 \dots i_m$ but there is a winning strategy for the system. Then, upon $0 i_1 w_1 \dots i_m w_m \$$, the system answers with 0. Nevertheless, the strategy of the system shall also answer 0 to $1 i_1 u_i \dots i_m u_m \$$, because the projection of the two words on agent's alphabets are the same. Therefore, there is no winning strategy.

If PCP has no solution, then, the two system agents can get together and compare the submitted run. Agent a_2 sends the sequence of indexes that it has been presented with to a_1 (using some protocol on which they agreed, based

on $\{s_0, \ldots, s_p\}$). This agent can then build $w_{i_1} \ldots w_{i_m}$ and compare it with the word that he has received from the environment. Since PCP has no solution, either they are the same and a_1 shall answer 0 or the two words differ and a_1 replies with 1. □

4.5.3 Merciful Synthesis

Streett pairs have been used to encode games with assume/guarantee principle. We used Streett pairs (E, F) to describe, in the E part, the liveness assumptions that the environment had to fulfill, and, in the F part, the liveness properties the system had to ensure once they have been met.

This approach is common in software engineering, where components are never developed nor deployed in isolation. Rather, some hypothesis about their environment are needed. In the transformational paradigm they are usually given as preconditions. In the reactive world, they must be stated as assumptions on the behaviour of the environment.

However, this behaviour is not only defined by the environment, but emerges as a result of the interaction between the environment and the system. It is thus possible that the environment behaviour does not meet the assumptions, not because of an environment's mistake, but because the system does not allow him to do so!

As an example, consider an Automated Teller Machine (ATM). When a transaction is finished, the system hands the card back to its owner, which has to take it (assumption) and then, the system will be ready for serving a new customer (guarantee). Now, suppose that the system does not let the card owner take his card. Everytime the customer tries to reach the card, the machine swallows it back and then hands it back to him. This frustrating system is correct, as it invalidates the hypothesis about the environment: the customer never retrieves his card.

In this chapter, we introduce the notion of mercifulness, which rules out such problematic specifications and solutions. This property is not a linear property anymore; it is inherently branching and cooperative.

Not every winning strategy is good. Remember that the winning condition is of the form of a Streett condition (E, F). Thus, a run which visits neither E nor F infinitely often is winning. However, intuitively, it seems that a strategy preferring such runs over runs visiting both E and F infinitely often, should be ruled out. Indeed, we do not want the system to use such a strategy when a more "merciful" solution exists.

In this section, we consider a game played on a game graph $G = \langle V, V_0, \Delta, \Omega \rangle$, with

- $V_1 \stackrel{\triangle}{=} V \setminus V_0$;
- $\Delta \subseteq (V_0 \times V_1) \cup (V_1 \times V_0)$ and $\forall v \in V : \exists v'' \in V : \Delta(v, v')$, i.e. we assume that the initial game graph is bipartite and that there is no dead end;

- $\Omega : \mathit{Streett}((E, F))$

Requiring game graphs to be strictly alternating between players is done without loss of generality. Stutter steps can always be added to meet this requirement.

By $\Pi_G(v)$, we will denote the set of all infinite paths $v_1 \cdot \ldots v_n \cdot \ldots$ in G that are rooted in v.

Definition 4.37 ($\Pi_G(v)$)

$$\Pi_G(v) = \{v_0 v_1 \ldots | v_0 = v \wedge \forall i : i \geq 1 : \exists e \in \Sigma : (v_{i-1}, e, v_i) \in \Delta\}$$

■

Definition 4.38 (Merciful strategy) A strategy f is said to be *merciful* iff, for every prefix $w \in V^+$, if there is some $w' \in V^\omega$ such that $w \cdot w' \in \mathit{Out}(f)$, then

$$\exists w' \in \Pi_G(\mathit{last}(w)) : \inf(w \cdot w') \cap E \neq \emptyset$$
$$\Longleftrightarrow$$
$$\exists w' \in \Pi_G(\mathit{last}(w)) : \inf(w \cdot w') \cap E \neq \emptyset \wedge w \cdot w' \in \mathit{Out}(f)$$

■

This definition can be expressed in the so-called *Game Logic* of [9]. The model-checking problem for this logic is decidable and relies on iterated applications of the module-checking procedure of [104]. However, this procedure requires doubly-exponential time. Here, we exhibit a direct solution to the problem of solving merciful games which causes only a polynomial blow-up in the size of the game graph. For the particular problem of one Streett pair merciful games, the solution is polynomial, as the number of colours of the underlying game graph is fixed to four.

A state is mercifully winning if there is a merciful winning strategy starting from that state. The merciful winning region is the set of states from which player 0 can use a merciful strategy and win, with respect to the Streett condition. We will say that we solve a merciful game if

1. We decide membership of states to merciful winning regions,

2. We construct a merciful strategy winning from each of these states.

By definition, the merciful winning region is included in the winning region. One could wonder whether the two regions do not coincide. This would solve trivially the first part of the problem. As shown in fig. 4.9, winning regions and merciful winning regions do not coincide. Winning states (for player 0) are gray states. However, there are no merciful winning vertices in this game: as soon as player 0 chooses to move right and let his opponent meet an E state, player 1 can force the game to enter the rightmost vertex, and win. Any winning strategy is thus merciless.

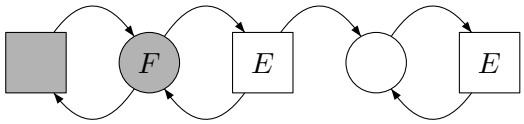


Figure 4.9: Every winning state is not merciful

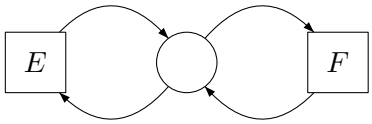


Figure 4.10: Mercifulness requires memory

Furthermore, merciful strategies are more complex than simple winning strategies. In order to solve mercifulness games, some memory is needed. Consider the game of fig. 4.10. Player 0 needs some memory, to remember to alternate between E and F states. Otherwise, his strategy would be either merciful and losing or merciless and winning. Using a graph-based technique to solve these games allow us to quantify precisely the amount of memory needed to be merciful: three bits.

Given a game graph G, we define an extended game graph G'. It is a parity game graph, "simulating" G. Basically, we aim at obtaining the following result about G', for every G' vertex x: $x \in W'_0$ iff, given some appropriate translation s of x to G, $s(x)$ is in the merciful winning region of G. Moreover, the translation shall apply to the winning strategy on G' as well, and deliver a merciful winning strategy on G.

Roughly, we now have three games, instead of one. They are schematically presented in fig. 4.11.

the *play* **game:** this is the usual game. Player 0 tries to fulfill the Streett condition. However, at any point, he can claim that the state is vacuously merciful: there is no path starting at the current vertex and visiting E infinitely often. This claim is made by choosing to play the *lost* game onwards.

the *lost* **game:** player 1 plays alone and wins iff he can find a path running infinitely often through an E state.

the *show* **game** is entered by player 1. The goal for player 0 is to find a path, starting at the current vertex which visits infinitely often both E and F.

Thus, player 0 will play twice in a row: first, he plays on behalf of player 1 and then, plays his own move. However, player 1 can then decide to switch back to the original game. This accounts for the fact that merciful winning strategies must also be winning in the original game.

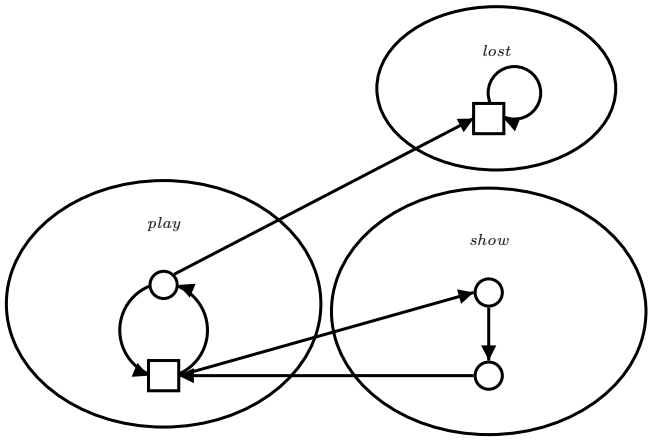


Figure 4.11: Schematic view of mercifulness reduction

Definition 4.39 (Extended game graph for mercifulness)

$$G' = \langle V', V_0', \Delta', \Omega' \rangle,$$

where

- $V' = (V \times \{play, show\} \times \{0,1\}) \cup (V \times \{lost\})$
- $V_0' = V_0 \times \{play\} \times \{0,1\} \cup (V \times \{show\} \times \{0,1\})$
- For all $q, q' \in V'$, $\Delta'(q, q')$ if one of the following statements holds
  1. $q = (u, lost)$, $q' = (v, lost)$ and $\Delta(u, v)$;
  2. $q = (u, play, i)$, $q' = (u, lost)$, for $u \in V_0$;
  3. $q = (u, play, i)$, $u \in V_0$, $q' = (v, play, \kappa(i, u))$ and $\Delta(u, v)$;
  4. $q = (u, show, i)$, $u \in V_0$, $q' = (v, play, \kappa(i, u))$ and $\Delta(u, v)$;
  5. $q = (u, play, i)$, $u \in V_1$, $q' = (v, play, \kappa(i, u))$ and $\Delta(u, v)$;
  6. $q = (u, play, i)$, $u \in V_1$, $q' = (u, show, i)$.
  7. $q = (u, show, i)$, $u \in V_1$, $q' = (v, show, \kappa(i, u))$ and $\Delta(u, v)$.

  where

$$\kappa(i, u) = \begin{cases} 0 & \text{if } u \in F \wedge i = 1 \\ 1 & \text{if } u \in E \wedge i = 0 \\ i & \text{otherwise} \end{cases}$$

- $\Omega : V' \to [4]$ is defined as follows

1.
$$\Omega((u, \mathit{lost})) = \begin{cases} 3 & \text{if } u \in E \\ 2 & \text{otherwise} \end{cases}$$

2.
$$\Omega((u, x, 0)) = \begin{cases} 2 & \text{if } x = \mathit{play} \wedge u \in V_0 \\ 1 & \text{otherwise} \end{cases}$$

3.
$$\Omega((u, x, 1)) = \begin{cases} 4 & \text{if } u \in F \\ 3 & \text{if } u \in E \setminus F \\ 2 & \text{if } x = \mathit{play} \wedge u \in V_0 \setminus (E \cup F) \\ 1 & \text{otherwise} \end{cases}$$

■

Lemma 4.40 For all path w in G', w is accepted iff only one of these conditions holds:

1. From some point on, this path remains in "*lost*" and no state in E is visited infinitely often.
2. This path visits infinitely many states in $V_0 \times \{\mathit{play}\} \times \{0, 1\}$ and satisfies the Streett condition (E, F).
3. It visits only finitely many times $V_0 \times \{\mathit{play}\} \times \{0, 1\}$, but visits infinitely many times $V_0 \times \{\mathit{show}\} \times \{0, 1\}$ and goes infinitely often through both E and F.

■

Proof 4.40

These conditions are clearly mutually exclusive. To see this, note that a path hitting a "*lost*" state cannot go back to a "*play*" or "*show*".

"If" direction

We show that a path w fulfilling none of the conditions is rejected. By a simple boolean manipulation, w fulfills none of the conditions above iff it fulfills one of the following:

1. w remains in "*lost*" and $\inf(w) \cap E \neq \emptyset$: Then, the maximal color is 3 and w is rejected in G'.
2. $\inf(w) \cap V_0 \times \{\mathit{play}\} \times \{0, 1\} \neq \emptyset$, w visits infinitely often E and finitely often F. Then, w remains stuck with its last component at 1. Since it visits infinitely often E, the maximal color is 3 and w is rejected.
3. $\inf(w) \cap V_0 \times \{\mathit{play}\} \times \{0, 1\} = \emptyset$ but $\inf(w) \cap V_0 \times \{\mathit{show}\} \times \{0, 1\} \neq \emptyset$ and it goes finitely often either through F or E. Then, w remains stuck with its last component either at 0 or 1, by definition of κ.

- If it remains at 0, because $\inf(w) \cap V_0 \times \{play\} \times \{0,1\} = \emptyset$, the maximal color is 1.
- If it remains at 1, then F occurs only finitely often and the maximal color is either 3 (E occurs infinitely often) or 1 (E does not occur infinitely often but $\inf(w) \cap V_0 \times \{play\} \times \{0,1\} = \emptyset$).

"Only if" direction

A path behaving as described by condition 1 is accepted, since the highest colour appearing infinitely many times is 2.

A path behaving as prescribed by condition 2 will also be accepted. To see this, consider these two cases: either F is seen infinitely often or E only occurs finitely often.

F inf. often: Then, the play visits infinitely many states the last component of which is 0. In that case, either it does not visit infinitely often 1 components, or it does. In the former case, the maximal color is 2, because inf. many states in $V_0 \times \{play\} \times \{0,1\}$ are met. In the latter, the maximal color is 4 (because F inf. often).

E fin. often In that case, at some point, the run remains within 0 and the maximal color is 2. So, the run is accepted.

Finally, condition 3 imposes that infinitely many visits to E and F are performed. Clearly, in that case, the run alternates between 0 and 1 states, and the maximal color is 4.

□

Without loss of generality, we assume that there is no "*lost*" states in W_0'. If there are such states, then, remove them from G, because they are trivially merciful, i.e. on all paths E is seen only finitely often. So, assume from now on that, from every state u, there is a path going infinitely often through E.

Theorem 4.41 For all states u in G, u is mercifully winning for player 0 iff $(u, play, 0)$ is winning for player 0 on G'. ■

Proof 4.41

"If" direction

Assume that we are given a mercifully winning strategy σ. We build a winning strategy on G'. For the sake of simplicity, we have assumed that every prefix in $Out(\sigma)$ can be extended to some run visiting E infinitely often above.

For every $w \in V^*$, such that $\exists w' \in Out(\sigma) : w \sqsubseteq w'$, we let $\gamma_w \in V^\omega$ be such that

1. $w \cdot \gamma_w \in Out(\sigma)$

2. $\inf(w \cdot \gamma_w) \cap E \neq \emptyset$
3. $\forall w' \in V^* : (w \sqsubseteq w' \wedge w' \sqsubseteq w \cdot \gamma_w) \implies w \cdot \gamma_{w'} = w \cdot \gamma_w$

It is easy to show that a γ_w always exists, for every σ-prefix w , as we assumed that every σ-prefix could be extended to some well-behaved path. Hence, this prefix can also be extended to some well-behaved path *in* σ, for σ is merciful.

Note now that every path $w' \in (V')^\omega$ can be turned into a path $w \in V^\omega$, by removing the "stuttering" moves (when player 1 asks for a *show* move) and projecting it on the first component.

For any $w' \in (V')^+$, letting $w \in V^+$ be w' transformed as explained above,

- If $\mathit{last}(w') = (u, x, i)$ with $u \in V_0$, then $\sigma'(w') = (\sigma(w), \mathit{play}, \kappa(u, i))$
- If $\mathit{last}(w') = (u, \mathit{show}, i)$ with $u \in V_1$, then $\sigma'(w') = (\mathit{first}(\gamma_w), \mathit{show}, \kappa(u, i))$,

To show that this strategy is winning, we prove that every $w' \in \mathit{Out}(\sigma')$ is winning. We refer to lemma 4.40 and perform a case split, for an arbitrary $w' \in \mathit{Out}(\sigma')$:

1. There are infinitely many "*lost*" states. This case cannot occur (we removed vacuously merciful states).
2. There are infinitely many $V_0 \times \{\mathit{play}\} \times \{0, 1\}$ states occurring. Then, since we follow σ in the first component and σ is winning, the outcome fulfills the Streett condition, which results in w' being accepted.
3. At some point onwards, only "*show*" states occur in the second component, when the first part is at V_0. Then, from this point on, the first component follows a certain γ_w, by definition of σ', where w is w' operating on G. Since γ_w sees infinitely many E and w is winning, w' also visits infinitely many F, in its first component, which implies that w' is accepted.

"Only if" direction

Given a memoryless strategy for player 0 σ' on G', we show how to construct a winning merciful strategy with finite memory $M = \{\mathit{play}, \mathit{show}\} \times \{0, 1\}$, acting on G.

The memory update function $\tau : \Delta \times M \to M$ is responsible for updating the memory, on every transition followed in G. This memory update function is defined according to the given strategy σ'.

- If $u \in V_0$, $\tau((u, v), (x, i)) = (\mathit{play}, \kappa(u, i))$.
- If $u \in V_1$,

$$\tau((u, v), (x, i)) = \begin{cases} (\mathit{play}, \kappa(u, i)) & \text{if} \quad (v, \mathit{show}, \kappa(u, i)) \neq \sigma'((u, \mathit{show}, i)) \\ (\mathit{show}, \kappa(u, i)) & \text{if} \quad (v, \mathit{show}, \kappa(u, i)) = \sigma'((u, \mathit{show}, i)) \end{cases}$$

The game with memory is played on $G_\tau = \langle V \times M, V_0 \times M, \Delta_\tau, \Omega_\tau \rangle$, where

- $\Delta_\tau((u,m),(v,m'))$ iff $\Delta(u,v)$ and $m' = \tau((u,v),m)$.
- Ω_τ is *Streett*$((E \times M, F \times M))$.

We let $\sigma(u,(x,i)) = (v, \tau((u,v),(x,i)))$, with $\sigma'(u,x,i) = (v,x',i')$.

Clearly, if σ is mercifully winning on G_τ, one obtains that σ is an automaton strategy, also winning and merciful on G. This part is easy, simply show that G is a game reduction of G_τ and adapt the proof that game reduction preserves winning strategies, to merciful winning strategies [159].

Now, we show that σ is winning on G_τ. In order to do so, we transfer every σ-play to a σ'-play, as follows:

$$\begin{array}{lrcl} \textit{if } u \in V_0, & f((u,(\textit{play},i)),(v,(\textit{play},i'))) & = & (v,\textit{play},i') \\ \textit{if } u \in V_1, & f((u,(\textit{play},i)),(v,(\textit{play},i'))) & = & (v,\textit{play},i') \\ \textit{if } u \in V_1, & f((u,(\textit{play},i)),(v,(\textit{show},i'))) & = & (u,\textit{show},i)\cdot(v,\textit{show},i') \end{array}$$

Given a G_τ-play $w = u_1u_2u_3u_4\ldots$, we let

$$f(w) = (u_1,\textit{play},0)\cdot f(u_1,u_2)f(u_2,u_3)\ldots f(u_{i-1},u_i)f(u_i,u_{i+1})\ldots$$

By inspection of these rules, one can show that

1. f is injective;
2. for every $w \in \mathit{Out}(\sigma)$, $f(w) \in \mathit{Out}(\sigma')$. (At every V_0' positions, w follows σ', because, when the game moves to a $(v,(\textit{play},i))$ position, σ chooses the same move as σ', whereas, when the game moves to a $(v,(\textit{show},i))$ position, then, by definition of τ, $(v,\textit{show},i) = \sigma'(u,\textit{show},i)$);
3. f is invertible if we restrict the range to the set of all runs w such that "*play*" never appears on V_0 positions. (A run w in G' that always follows "*show*" corresponds directly to the same run on G_τ, because, at V_1 positions, the moves obey σ, which ensures that the memory is updated to "*show*" by τ.)

We want to prove the following: Assuming that σ' is a winning strategy on G', then σ is winning and merciful (on G_τ).

σ is winning on G_τ: assume that σ is losing. Then, there is some path $w \in \mathit{Out}(\sigma)$ such that $\inf(w) \cap (E \times M) \neq \emptyset$ but $\inf(w) \cap (F \times M) = \emptyset$. Recall that $f(w) \in \mathit{Out}(\sigma')$. Clearly, along $f(w)$, infinitely many E states occur while only finitely many F states are visited. Since no "*lost*" states are ever met, and $f(w)$ is winning, then, by lemma 4.40, the Streett condition should be fulfilled. We reach a contradiction (σ' is winning but $w \in \mathit{Out}(\sigma')$ is not accepted) and conclude that σ is winning.

σ is merciful on G_τ: We take w such that w is a σ-prefix. Then, $f(w)$ is a σ'-prefix. Assume, without loss of generality, that $last(f(w)) \in V_1'$. Note that there is a $\gamma_{show} \in (V')^\omega$ such that, at every V_0 position, the second component indicates "*show*" and $f(w) \cdot \gamma_{show} \in Out(\sigma')$. Since σ' is winning, $f(w) \cdot \gamma_{show}$ is accepted, which, in turn, implies that E and F are encountered infinitely many times in the first component (lemma 4.40). Furthermore, by the form of γ_{show}, f is invertible on it, delivering a σ-play $f^{-1}(\gamma_{show})$ such that $w \cdot f^{-1}(\gamma_{show})$ visits infinitely often $E \times M$. Hence, we have shown that, from every σ-prefix w, it was possible to extend it into a well-behaved prefix, within σ, i.e. σ was merciful.

□

4.5.4 Related Work

There has been much work in synthesis of concurrent reactive systems. The initial question is due to Church [37] and has been solved by Büchi and Landweber in [29]. Pnueli and Rosner describe how to synthesize reactive modules from LTL specifications, either in a synchronous or asynchronous setting [140]. They reduce the realizability problem for LTL to the satisfiability problem for CTL*. Basically, the set of variables occurring in an LTL formula is partitioned between environment variables (say, x) and system variables (say, y). The full computation tree is a tree such that, for every x value, each node has a successor "corresponding" to x. Formally, if X is the domain of x, the full tree on X is X^*. The problem amounts to finding such a tree whose nodes are labeled by y values, such that every path in the tree fulfills the given LTL formula.

This work is extended to cope with partial information, generalized parallelism and real-time systems by [184]. The approach is trace-based and the realizability problem is exactly given as here: find a strategy for the program such that whatever strategy the environment chooses, the specification is met. Again, Rabin tree automata are at the heart of the approach for the finite-state case, together with Safra's determinization procedure [145]. This makes the procedures hard to implement. Harding *et al.* propose an algorithm for solving LTL games that avoids using determinization [68]. Their method is not complete. They characterize the conditions under which completeness is guaranteed and argue that the counter-examples to this condition are rare enough to justify the practicability of their approach. Their algorithm can be implemented symbolically using Ordered Binary Decision Diagrams (OBDDs) [28]. We are currently working on the implementation and evaluation of a symbolic solution to LSCs centralized realizability, basing our work on the framework of Wallmeier *et al.*, who provide a symbolic implementation of several game solving algorithms [173], and Harel *et al.* who encode the semantics of LSCs as a boolean relation in order to perform "smart play-out" [69].

We have shown how to decide the realizability property for LSCs, using existing game theoretical algorithms and automata-theoretic techniques. Our solution, for the particular case of LSCs, is more interesting than first translating LSCs to LTL [23] and then using the procedure of [134] to check its

realizability, because it is more direct. In particular, we have shown that our solution is simply exponential. If we first translate LSCs to LTL, we will have to build an acyclic automaton, which already contains exponentially more states than the LSC. Then, this automaton will be traversed to construct an LTL formula [23]. Here, we build exactly the same automaton but directly use it to solve the problem in which we are interested.

The so-called "supervisory control synthesis" problem for Discrete Event Systems (DES) focuses on the controllability of finite languages, generated by a plant [138]. Basically, a plant generates some prefix-closed language L, a subset of which is marked, L_m. We are given some language $K \subseteq L_m$. Then, we would like to know whether we can build a strategy, disabling some transitions of the plant, such that the marked "closed-loop" behavior of the plant is K. The computational complexity of this problem makes it difficult to apply practically, hence horizontal and vertical decomposition have been devised as means to make solutions more tractable. In [152], a survey of this area is presented. The problem of DES supervisory control has been extended to real-time [108], branching-time specifications [15], infinite behaviors [154, 155] and partial information/decentralized supervision for which results are mostly negative [160, 106, 137]. In spirit, our solution is not too different from the one presented in [154, 155]. The basic difference between our work and theirs is that we do not require the "plant" and the "legal language" to be explicitly defined. On the contrary, they are described in a succinct and intuitive way, using LSCs. Actually, we could have reduced our problem to a supervisory control problem for DES, but we chose to follow a reduction to game solving problems, with which we are more familiar and, we believe, are easier for the reader to grasp. The two areas are closely related, differing mostly in terms of vocabulary. Here, we deal with pure liveness assumptions. These assumptions can be encoded in the plant as in [154]. Because of the restricted form of our acceptance condition, we have an efficient algorithm (wrt the size of the plant). Another approach to obtain efficient algorithms, as in [153], is to add a strong fairness assumption, which makes the synthesis of controllers for Rabin automata computable in polynomial time, modulo a modification of the synthesis routine.

Our solution synthesizes a global strategy, for all agents of the reactive systems. Nevertheless, it seems that practically, synthesizing a distributed implementation would be of greater interest. It turns out that the realizability problem over fixed architectures is undecidable for almost every interesting architecture [140]. Rosner presented a type of architecture for which the problem becomes decidable, which he calls *hierarchical architectures*. In this architecture, one agent receives all inputs and the information flow between agents induces a tree, hence the term "hierarchical". Observing that decidability in this case comes from the fact that the "root" agent can always simulate the behavior of every other agent, Kupferman and Vardi came with other architectures for which realizability is also decidable [103]. These architectures simply require the agents to be ordered either linearly or in cycle. More generally, Madhusudhan and Thiagarajan present three properties making the distributed controller synthesis problem decidable [112]. These restrictions are: trace-closure of spec-

ification, strategy must be com-rigid and clocked. As soon as one of them is dropped, the problem becomes undecidable. This result generalizes their work on the synthesis of distributed controllers from *local* specifications, which are, by definition, trace-closed [111]. LSC specifications are typical global-specifications and are not trace-closed. Gastin *et al.* study the problem of solving distributed games and thus, synthesizing distributed controllers for asynchronous systems [57]. Again, the conditions set for decidability do not apply here. In [160], Tripakis studies the problem of distributed observability, which amounts to, given n alphabets $A_1, \ldots A_n$ ($A_i \subseteq \Sigma$, for $i = 1, \ldots, n$) and a regular language $R \subseteq \Sigma^*$ building a function $o : A_1^* \times \ldots A_n^* \to \{0, 1\}$ such that for every $w \in \Sigma^*$, $w \in R \iff o(w|_{A_1}, \ldots, w|_{A_n}) = 1$. This problem is shown to be undecidable. We adapted Tripakis' proof to the case of LSCs, in the proof of Th. 4.36.

Facing this undecidability result, one could tackle the "distribution" problem by first synthesizing a global strategy and then trying to distribute it over the various agents [118, 52]. This technique is sound and not complete, in the sense that there might exist some distributed implementation even though the centralized solution is not distributable.

In the realm of scenarios, previous research on "synthesis" followed three directions: induction, compilation and controller synthesis. Every approach finds a justification, depending on the purpose of the scenarios and the phase of the software life cycle in which it is used [13, 174].

Induction: Typically, scenarios are considered as a partial view on the behavior of the future system. A set of scenarios is thus a finite set of example computations. The problem is to induce, from these examples, a universal rule describing every acceptable behavior of the system. The user has some (regular) set of behaviors W in mind and gives a finite set of examples $E \subseteq W$. We are asked to build an automaton $\mathcal{A}$ recognizing W. [99] have used an algorithm, due to Biermann and Krishnaswamy [20], which computes the "best" deterministic $\mathcal{A}_E$. At the limit, the language of the synthesized automata converges to W. Practically, this automaton needs to be checked, in order to ensure that it does not contain inappropriate behaviors. It has been integrated to the CASE tool FUJABA (From UML to Java and Back Again) and validated on a case study from manufacturing industry [45].

Minimal Adequate Synthesizer (MAS) is an interactive approach to machine learning from synthesizer which is based on the Minimal Adequate Teacher paradigm. MAS finds missing scenarios in the set of scenarios and asks its operator whether such scenarios should be integrated in the specification or not. It has been implemented by Koskimies and Systä [115] and validated on the "Paderborn New-Rail Technology" Shuttle case study [100].

Hsia *et al.* use example scenarios, given as an execution tree, to build a grammar and analyze the specification, wrt consistency criteria, using automata analysis techniques [87] . By hand, they produce, from such

a set of examples, a prototype of the future system. This prototype can then be used to validate the specification with the end-user.

Compilation: Scenarios can also be seen as a *complete* description of the future system. Thus, the desired behavior W equals the given traces E. This is especially useful if the system to be built is closed. Then, state machines must be built from the various scenarios, projecting them onto every instance [101, 177, 162, 6]. The main problem that arises then is that there might be some discrepancies between the global view of the behavior, induced by the scenario and the local view that every instance has of this behavior. When recomposing the full system from the individual state machines, yielding a set of behaviors C, it might be that $C \supset W$. These additional scenarios, $C \setminus W$, are called "implied scenarios" [6]. Techniques have been developed to detect such implied scenarios and report on them [163, 6]. In [55], the authors give syntactical restrictions, namely *causality*, which ensures that MSCs, with control flow constructs, such as iteration or choice, can be distributed, i.e. that the liveness/safety constraints imposed by lifelines match the global constraint stated by the considered MSC.

One result, which is in spirit close to ours, is due to Desharnais *et al.* [44], except that their setting is state-oriented, whereas most researchers on scenarios, including us, focused on an event-based setting. In [44], the authors present a way to represent scenarios as a relation between states. This presentation can be graphical, thanks to relational transition systems. They make a distinction between environment moves and system moves, allowing moves "within" the environment and "within" the system, as we do it. A scenario is assumed to describe possible environment inputs and all legal system reactions. The authors propose an operator for integrating scenarios, based on the "demonic meet" operator. Like in our work, the integration of two scenarios relative to the same input obliges the system to answer as specified by both scenarios.

Controller synthesis: this approach corresponds to what we have proposed here. One uses an expressive scenario-based specification language and then tries to synthesize a program satisfying it. LSC is so far the only candidate language for this approach, thanks to its higher expressiveness.

The main work in this field is due to Harel and Kugler [72]. They deal with very simple LSCs: precharts contain only one environment message ($\in A_{in}$) and main charts contain only system messages ($\in A_{out}$). They choose the super-step approach of [77]: the environment provides one input and the system answers with a sequence of output messages: $(A_{in}A_{out}^*)^\omega$. They show that realizability is equivalent to a *consistency* condition. This condition asserts the existence of a nonempty regular language $L \subseteq (A_{in}A_{out}^*)^*$ such that

1. L contains one execution for every existential chart,
2. L satisfies all universal charts,

3. for every $w \in L$ and every $a \in A_{in}$, there is some $r \in A_{out}^*$ such that $war \in L$.
4. for every $xyz \in L$ such that $y \in A_{in}$, $x \in L$.

They build a minimal deterministic automaton recognizing the intersection of universal LSCs and progressively prune it to remove "bad states", i.e. states that do not satisfy condition (3). The resulting automaton can then be transformed to some strategy automaton. Our solution is an extension of their work. Firstly, we tackle the problem for more general LSCs, allowing choice constructs, complex precharts and environment messages in the main chart. The latter extension enabled us to use LSCs in assume/guarantee development of reactive systems. Secondly, instead of relying on an ad-hoc technique, we proposed a reduction to parity games, for which a range of results and algorithms is available.

They also propose three strategies to distribute the synthesized strategy among the various agents:

1. build a central controller, driving the execution of the individual agents;
2. duplicate the central controller in every agent;
3. duplicate the central controller and remove states that are not relevant to the object in question.

Harel and Marelly have developed a technique called "play-out" for executing LSC specifications [75]. In this approach, the interpreter (called play-engine) follows a built-in strategy: it selects (in a predefined way) a required event and performs it. Of course, this strategy can lead to deadlocks. To solve this problem, Harel *et al.* have developed a "smart play-out" algorithm [69]: they use model-checking to compute a "non-blocking" strategy. This strategy proposes a sequence of events which leads to a state in which there are no more required events, for the system, if such a sequence exists. However, even though this approach ensures that a successful "super-step" s starting at state q will be found if it exists, it might be that, for some environment input, say i, there is no super-step starting at state q' with $q \xrightarrow{s \cdot i} q'$. This shows an advantage of our method, which ensures that such situations will be avoided, at the price of a higher complexity.

In our work, we *verify* that a specification is consistent, by performing a full search on the state space, while Harel and Marelly use play-out to *validate* a specification, by letting a user execute it. Our work could be integrated with theirs: if the specification is inconsistent, our algorithm builds a *counter-strategy*, i.e. a strategy for the environment, making the system fail. This strategy could be given to the play-out engine to illustrate the flaws of the LSC specification.

4.5.5 Summary

In this section, we have presented several versions of the synthesis problem. Centralized synthesis is the simplest, but it is already intractable, as it is an EXPTIME-complete problem. Constrained realizability is a possible flavour of this problem which gets 2EXPTIME-complete. Finally, we presented the most general, industrially relevant, problem of distributed synthesis and we proved that it is undecidable, just as full linear temporal logic. In the next section, we introduce another version of centralized synthesis, which we call "synthesis under mercifulness assumption". It introduces a qualitative criterion on the synthesized implementation. We exhibit an algorithm for solving merciful games which runs in time polynomial in the size of the game graph.

4.6 Incomplete Approaches

In this section, we investigate more "lightweight" approaches to the verification and synthesis problems. The former class of problems is intractable, because of its high complexity. There are two sources of complexity in verification:

- the formula or the automaton generated from a ULSC needs to be large.
- the intra-agent specification is made of many small automata. However, computing the *product* of these automata is an expensive operation, giving rise to the well-known state explosion problem. This is one of the main problems in verification and there are other solutions, in the spirit of ours, trying to address it [49].

We will propose two small techniques that can help in tackling these problems. They must be combined with other, more efficient, techniques for fighting state-space explosion such as partial order reduction [62, 182], symbolic approaches [28] or bit-state hashing [183].

The latter problem, namely distributed synthesis, is undecidable, as we have shown in the previous section. We will present a sound but incomplete technique for synthesizing distributed implementations. Synthesized implementations are guaranteed to be correct, but our algorithm may fail to find an implementation, even though it exists.

4.6.1 Verification

First, we suggest to use the techniques for minimizing the size of the formulae generated from ULSCs devised in [23, 102]. Typically, a ULSC can be split into several small formulae, in which we only need to check the proper ordering of *pairs* of events, and not all linearizations (as in sec. 4.4).

In order to address the state explosion problem, we suggest to ignore all components that do not participate in the verified ULSC. Suppose that only agents i through k participate in S. Hence, it is sufficient to check that the ULSC is correct, wrt the subsystem composed of agents i through k only. Since we demonstrated that I/O automata composition is a refinement, proving that

the ULSC is satisfied by this reduced system is enough to show that the global system is correct, too:

$$\mathcal{L}\left(\prod_{j=1}^{n}\mathcal{A}_j\right) \subseteq \mathcal{L}\left(\prod_{j=i}^{k}\mathcal{A}_j\right) \subseteq \mathcal{L}(S).$$

Furthermore, when the subsystem can satisfy the LSC on its own, it indicates that the design achieves low coupling: the fulfillment of the property does not depend on components which are not directly involved in it.

However, if model checking fails, it might be a false negative: the counter-example could have been avoided, had we included more agents in the system, which one can try.

4.6.2 Synthesis

Our lightweight algorithm is illustrated in fig. 4.12. The automaton at the bottom of this figure contains dotted-line transitions. They represent "default transitions", which are taken when some input event occurs that does not label any outgoing transition from the current state. The various steps of our lightweight algorithm are detailed in the rest of this section.

4.6.2.1 (1) Agent Selection

As opposed to the previous algorithm, the lightweight algorithm focuses on a single agent at a time. It does not try to find a strategy for all participants in one run. For the rest of this section, let a be the selected agent.

4.6.2.2 (2) Sanity check

All scenarios in which a participates actively are checked to ensure that their causal order matches their visual order. Two locations are causally related if they are sending locations on the same lifeline or if they are the sending and receiving locations of the same event. This is done in polynomial time [10]. Fig.4.13 gives an example of a ULSC which does not fulfill this condition. Clearly, it is not simply distributable for agent `obj3`, because `d` may only be sent after `c` has occurred, which `obj3` cannot see.

If this sanity check fails, the algorithm stops and explains why specification is not distributable.

4.6.2.3 (3) Scenario Projection

All scenarios are projected onto the lifeline of agent a (e.g., the upper part of fig.4.12 illustrates an attempt to synthesize an implementation for `c[1]`). All ULSCs in which a is not required to perform any event are discarded. For instance, the scenario of fig 3.23 would be discarded because `c[1]` does not take part in it. In summary, Step 3 produces a set of non-empty ULSCs, reduced to the lifeline of a, one for each ULSC in which at least one event controlled by a is restricted.

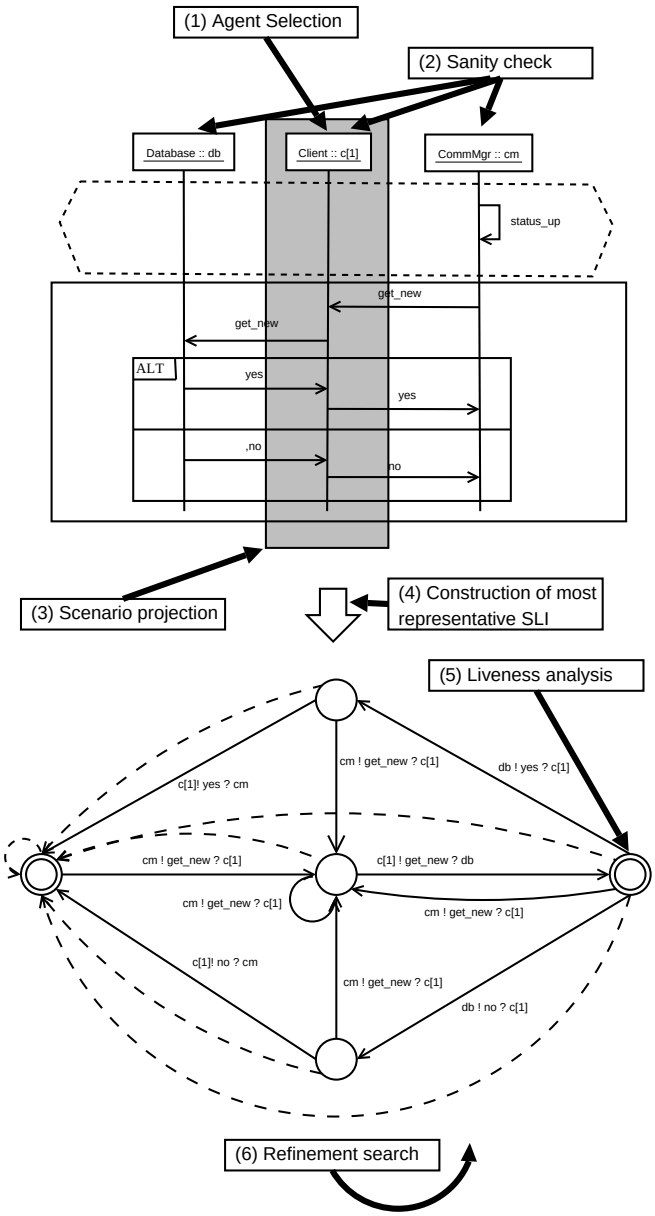


Figure 4.12: Standard Local Implementation (SLI) for `c[1]`

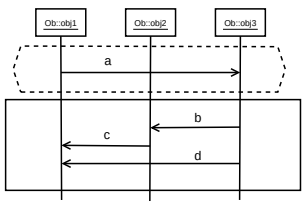


Figure 4.13: Mismatch between causal order and visual order

4.6.2.4 (4) Construction of Most Representative SLI

The I/O automaton built is input-enabled. It records in I all possible cuts of every scenario. The invariant of the automaton is: for every word w, if the automaton reads w and ends up in a state I then, for every cut c, $c \in I$ iff some suffix of $w|_{\Sigma_R}$ linearizes c. For instance, in Fig.4.12, the center state of the I/O automaton records the configuration where the last event was `get_new` (from `cm` to `c[1]`), as all its incoming transitions indicate. This means that the prechart of the projected scenario has been matched and `get_new` (from `c[1]` to `db`) is now required from agent `c[1]`. Since this event is not forbidden at that state, the *Standard Local Implementation (SLI)* rule (see below) allows it to be scheduled.

Definition 4.42 (Standard Local Implementation (SLI)) Let the projected specification be composed of m non-empty ULSCs: $\{S_1, \ldots, S_m\}$. An I/O automaton fulfilling the following constraints is called a *Standard Local Implementation (SLI)*:

$$\langle \Sigma_a^r, \Sigma_a^s, Q, q_0, \Delta, \{\Sigma_a^r\} \rangle$$

where

- $Q = \prod_{i=1}^{m} 2^{2^{L_i}}$, i.e. every state keeps one configuration per ULSC, a configuration being a set of cuts.
- $q_0 = (\{\emptyset\}, \ldots, \{\emptyset\})$,
- $\Delta((I_1, \ldots, I_m), e, (I'_1, \ldots, I'_m))$ implies both the following statements
  1. Δ follows the cuts transition system:
     - if $e \notin \Sigma_R^i$, $I'_i = I_i$;
     - if $e \in \Sigma_R^i$, $I'_i = \{c' | \exists c \in I_i : c \xrightarrow{e} c'\} \cup \{\emptyset\}$. The empty cut is always added, because it is linearized by the empty word, which is a suffix of every word $w \in \Sigma^*$, thus preserving the invariant.
  2. If $e \in \Sigma_a^s$, there is some i such that $c \in I_i$ requires e and, for every j, there is no $c \in I_j$ forbidding e.

■

Such an implementation is called "standard" because it follows the classical way of extracting state machines from MSCs (see sec.4.5.4). It is dubbed "local" because it only considers a single agent, restricting a scenario to the local view of that agent.

Note that there may exist many SLIs for a given specification, because the condition on Δ is only an implication. They differ only in the scheduling of Σ_a^s events. Thus, it is possible to order SLIs: an SLI $\mathcal{A}$ is *more general* than an SLI $\mathcal{A}'$ ($\mathcal{A}' \sqsubset \mathcal{A}$) iff, at every state q, if $\mathcal{A}'$ allows $e \in \Sigma_a^s$ event, then $\mathcal{A}$ allows e, too.

4.6.2.5 (5) Liveness Analysis

The I/O automaton built according to the SLI rule is always safe, because the forbidden events may not be scheduled. To show this, we prove that the hypotheses made by a about the global state are valid:

Lemma 4.43 (SLIs are sound) Let $\mathcal{A}$ be an SLI. Consider a finite run $w \in \Sigma^*$, decomposed in two parts $uv = w$ and a scenario L_j. If $v|_{\Sigma_R}$ linearizes some cut c in L_j and $\mathcal{A}$ has a run on $v|_{\Sigma_a}$ leading to a state $(I_1, \ldots, I_j, \ldots, I_n)$, then I_j contains $c|_{\Sigma_a}$. ■

Proof 4.43

By induction on w. □

Lemma 4.44 (SLIs are safe) All behaviours induced by an SLI are Σ_a^s-safe. ■

Since SLIs guarantee Σ_a^s-safety, it suffices to ensure that the considered automaton is Σ_a^s-live to verify that it is a correct implementation.

Theorem 4.45 Let $\mathcal{A}$ be an SLI. If all runs in $\mathcal{A}$ are Σ_a^s-live, then $\mathcal{A}$ is a correct implementation of a system consisting of agent a. ■

Proof 4.45

By definition 4.8, if $\mathcal{A}$ is Σ_a^s-live (assumption) and Σ_a^s-safe (Lem.4.44), it is a correct implementation. □

In general, liveness is not true of all SLIs, because some required event might be postponed forever, since it is always unsafe. The liveness condition needs to be algorithmically checked; this is done in time quadratic in $|\mathcal{A}|$: $\mathcal{A}$ is analyzed to check that, on all fair infinite paths, there are infinitely many occurrences of e or e is not required in infinitely many states, for every $e \in \Sigma_a^s$. In fig. 4.12, the states in which no event is required are drawn with a double line. This SLI example is live for agent `c[1]`.

4.6.2.6 (6) Refinement Search

If $\mathcal{A}_\top$ is not a correct implementation, i.e. it is not live, we can try to find another SLI $\mathcal{A}$ such that $\mathcal{A} \sqsubset \mathcal{A}_\top$ and $\mathcal{A}$ is live. In order to do so, we consider refinement as a two-person game, between a "protagonist" and an "antagonist". The protagonist may remove some edges labeled by Σ_a^s events while the antagonist tries to prove that the resulting automaton is still not live. If the protagonist has no winning strategy in this game, there is no live SLI for agent a. This game can be solved using classical algorithms, in time polynomial in the size of the graph [95].

Remark 4.46 (Safety Assumptions) An SLI allows agents to make safety assumptions about their environment, which makes compositional reasoning feasible [1]. For instance, when synthesizing agent i, we can make use of the fact that we know *beforehand* that agents $1, \ldots, k$ will also be synthesized using the same method. In that case, when agent i receives an event from another "to-be-synthesized" agent j, he knows that some cuts of the configuration are not valid anymore. Indeed, if agent j sends this message, he must be required to do so. Now, if there is only one scenario which requires him to send j, the agent we are synthesizing can deduce the exact position in this scenario. For synthesizing the SLI of fig.4.12, our algorithm used this assumption. ■

Remark 4.47 (Efficiency) By construction, the I/O Automaton built here is *necessarily* smaller than the automaton constructed by the algorithm of Sec.4.5.1. This justifies our claim that this localized technique can sometimes be more efficient than the exact centralized one. However, in the worst case, the SLI is as big as the solution for the centralized case (and thus, exponential in the size of the specification, see Th. 4.32).

Our running example is specified with 25 scenarios and contains 8 components. Our implementation of the centralized synthesis algorithm fails to analyze it, because of its size. However, the implementation of the lightweight algorithm successfully synthesizes an SLI for every component, but `cm` and `db`. `cm` cannot be synthesized because it participates in all scenarios; projecting the specification on it does not drastically reduce the size of the specification. The SLIs that we obtained had less than 20 states each and their synthesis took only a couple of seconds. This synthesis relied on the additional safety assumptions explained above. ■

4.7 Conclusion

In this chapter, we have presented and analyzed the "dream" of moving from requirements to code in an automated way. This dream relies on several artifacts: formal modeling languages, precise definitions of the problems to automate and tools supporting this automation.

The languages should enable analysts to cope efficiently with the tasks relative to the various software life cycle development phases. When considering requirements analysis and design, we advocated that two usual views of the behaviour of distributed reactive systems should be used. The first specification language is scenario-based and provides analysts with an inter-agent view. The second language is state-based and makes it possible to describe the complete behaviour of every agent/component/object individually. We have argued that the former view makes it easier to validate the model of the system, by relating informal user requirements with a formal behavioural description, while the latter is closer to code, fitting a particular architecture.

We have introduced abstract models of reactive systems, in two forms: intra-agent specifications and inter-agent specifications. They both describe the behaviour of distributed reactive systems. This behaviour takes place on some

structure, which is common to the two modes of specification. Intra-agent specifications are mathematically considered as (non-deterministic) strategies and concretely represented by Input/Output Automata, while inter-agent specifications are expressed by universal Live Sequence Charts. Our intra-agent specification language supports composition. Furthermore, composition enjoys several nice properties, the most significant of which is that it is a linear-time refinement. This means that properties proven about a single component cannot be broken by plugging this component into a larger architecture.

In order to benefit fully from the use of these languages, tools are needed to assist the analyst. These tools should be able to detect inconsistencies between the inter- and intra-agent views. We have identified three classes of features that these tools shall provide:

use case checking and refinement, which are centered on inter-agent specifications.

synthesis of an intra-agent specification from an inter-agent specification. Underlying this functionality is the problem of deciding whether the inter-object specification is actually meaningful and implementable.

verification that an intra-agent specification complies with an inter-agent specification.

The formality and rigour of our framework allowed us to define precisely the meaning of each of these features, as well as to investigate the complexity of each of them. It turned out that all these problems are difficult. The simplest, and least interesting, an unrealistic flavour of verification, viz. verification of closed and centralized systems, is already coNP-complete. The most complex is the industrially relevant problem of distributed synthesis, which is undecidable. In-between, we found that use case checking is PSPACE-complete and synthesis of centralized systems is EXPTIME-complete.

Facing the intractability and undecidability of these problems, we proposed some partial approaches. These approaches are imprecise, in the sense that they sometimes provide the analyst with false negatives. This thesis does not claim to solve, or even try to cope with, the infamous state explosion problem in verification. We only introduced a small technique, relying on the properties of agent composition, in our setting. This technique can be combined with other, more advanced techniques for dealing with state-space explosion. We also proposed an algorithm that synthesizes a distributed implementation from an inter-agent specification. On our running example, an implementation of this algorithm has been able to generate some intra-agent models. In comparison, our implementation of the centralized algorithm, which is presented in Chapter 6 ran out of memory on this example.

The high complexity of these problems has to be compared with the low expressiveness of our specification language, as investigated in Chapter 3. This complexity is surprising, as most researchers and practitioners are naturally led to believe that "analyzing simple diagrams built from a few lines and arrows should not be that difficult".

The next chapter presents some extensions to the LSC language and study their impact on the complexity of the problems introduced in this chapter. Chapter 6 will describe an implementation of the algorithms defined in this chapter.

Chapter 5

Language Extensions

Contents

And now, for something completely different...

Monty Python's Flying Circus

5.1 Introduction

In this chapter, we introduce three extensions to the language of LSCs: conditions, real-time and symbolic instances. Conditions were part of the initial definition of the language, by Damm and Harel [42], while the two latter features have been introduced later [75, 74, 119].

5.2 Conditions

5.2.1 Introduction

So far, we have considered our systems as being purely event-based. This can be the right abstraction for *designing* interacting reactive systems. However, in some situations, it is natural to express behavioural constraints that depend on *state-based conditions*. For instance, in the CTAS example, the following requirement is quite obvious: "if the user interface (weather control panel) is disabled and the user clicks on it, it has no effect" or "when a disconnected client sends a message to the idle communication manager, asking to become connected, the communication manager must acknowledge this query and start an initialization process". Statements as "disconnected client", "disabled user interface", "idle communication manager" refer to the state of these objects, or of the overall system.

In LSCs, conditions can be used to refer to states. Thus, in addition to events, it is also possible to constrain the possible system behaviours thanks to conditions. The original language made a distinction between two modes of conditions: hot conditions and cold conditions. Hot conditions *must be true* in order to be traversed whereas a cold condition *may be false*, forcing the execution to abort prematurely but successfully.

In [75], the authors discuss several possible design choices as to when conditions should be evaluated. When the execution reaches a condition, there are three possibilities: evaluate the condition immediately, wait for some "random" amount of time before evaluating it, or wait until the condition becomes true. They took the following decision, with respect to the operational semantics of LSCs, as implemented by the play-out engine: cold conditions are evaluated as soon as possible whereas the animator waits until hot conditions become true. The former decision is justified on the grounds of the fact that users find it more intuitive. The latter decision is simply considered "easy". It is true that, in the framework of play-out, evaluating a hot condition to false leads to an irremediable error and execution must be halted abruptly. However, our interest is on the use of LSCs as a behaviour specification language. As such, it must be able to draw a hard line between allowed and forbidden behaviours. Having said that, bad executions are at least as interesting for us as good executions are. Thus, we do not want to rule out or try to avoid the possibility of hot conditions being evaluated to false.

Therefore, we propose to consider two types of conditions in addition to the usual hot/cold distinction: *ASAP* conditions and *eventual* conditions. The former are going to be evaluated as soon as possible and the latter will be eventually evaluated. We underline the fact that ASAP means "as soon as the location labeled by the considered condition is enabled, i.e. all its predecessors have been reached" while, by eventual, we mean "as soon as (i) all the predecessors have been reached and (ii) the condition becomes true". The latter definition will thus oblige the condition to be matched without delay, thus eliminating nondeterminism regarding conditions. We oblige cold conditions to be ASAP. We are still backwards-compatible with the decisions taken by Harel and Marelly in [75].

5.2.2 LSC with Generic Conditions

In order to use conditions, we need some basic language in which state formulae are expressed. We will start with a generic setting that will be instantiated shortly. We thus assume that some logic is given. We see a logic as a triple $(\Psi, \mathcal{I}, \models)$ with Ψ a set of sentences, $\mathcal{I}$ their interpretations and $\models \subseteq \mathcal{I} \times \Psi$ a model relation, with $i \models \psi$ representing that ψ is true under interpretation i[1]. States, in our approach, will be interpretations. We will also require to have at our disposal some way to link executions, that are traces of events with states. This is achieved through *state functions* $\zeta : \Sigma^* \to \mathcal{I}$.

The set of conditions that may be used in ULSC is Φ. As said above, it contains cold (Φ_c) and hot conditions (Φ_h). The sets of ASAP and eventual conditions are respectively denoted by Φ_A and Φ_E. Remark that all these conditions are built using the same underlying logic, but we use disjoint union ($\uplus$) to ensure that conditions of a different nature are distinguishable. In other words, it is possible to tell whether some sentence is taken from Φ_c or Φ_h.

[1] The model relation is conventionnally written in infix notation: $i \models \psi$ represents $(i, \psi) \in \models$.

Definition 5.1 (Conditions (Φ))

$$\begin{aligned} \Phi_c, \Phi_h^A, \Phi_h^E &\triangleq \Psi \\ \Phi_h &\triangleq \Phi_h^A \uplus \Phi_h^E \\ \Phi_A &\triangleq \Phi_c \uplus \Phi_h^A \\ \Phi_E &\triangleq \Phi_h^E \\ \Phi &\triangleq \Phi_c \uplus \Phi_h \end{aligned}$$

■

We need to adapt the notion of LPO and linearization to introduce conditions to LSCs. All definitions, provided in Chapter 3 can then be used without modifications.

Definition 5.2 An LPO is a tuple

$$\langle L, \leq, \lambda, \Sigma_R \rangle,$$

such that

- L is a finite set of locations,
- $\lambda : L \to \Sigma_R \cup \Phi$,
- $\leq \subseteq L \times L$ is a partial order on L,
- $\Sigma_R \subseteq \Sigma$.

■

Matching finite words with LPOs was previously simple: we checked that the events occurred in the proper order, with respect to locations. We keep the same idea here, except that several locations can be mapped onto the same event now, because of ASAP conditions. Furthermore, cold conditions can cause premature, but not invalid, abortion of the execution.

A finite word $e_1 \dots e_n$ is a linearization, under the interpretation of some state function ζ, of an LPO (with conditions) if there is a partial function $f : L \not\to [n]$ such that

1. Some location is mapped on the last index of the finite word. Thus, there is no "trailing garbage" in the word:

$$\exists l \in L : f(l) = n.$$

2. location ordering is respected in the linearization:

$$\forall l, l' \in dom(f) : l \leq l' \implies f(l) \leq f(l').$$

3. only unrestricted events may be skipped:

$$\forall i \in [n] : i \notin ran(f) \implies e_i \notin \Sigma_R.$$

4. events in the finite word match the labeling of event-labeled locations, according to the indexes returned by f:

$$\forall l \in dom(f) : \lambda(l) \in \Sigma \implies \lambda(l) = e_{f(l)}.$$

5. event-labeled locations must be strictly ordered by f:

$$\forall l, l' \in dom(f) : (\lambda(l) \in \Sigma \wedge \lambda(l') \in \Sigma \wedge l < l') \implies f(l) < f(l').$$

6. premature abortion is only possible if a cold condition is violated

$$dom(f) \subseteq L \implies (\exists l \in L : f(l) = n \wedge \lambda(l) \in \Phi_c \wedge \zeta(e_0 \dots e_n) \not\models \lambda(l)).$$

7. hot conditions must be matched:

$$\forall l \in dom(f) : \lambda(l) \in \Phi_h \implies \zeta(e_1 \dots e_{f(l)}) \models \lambda(l).$$

8. non-terminal conditions must be matched:

$$\forall l \in dom(f) : f(l) < n \wedge \lambda(l) \in \Phi \implies \zeta(e_1 \dots e_{f(l)}) \models \lambda(l).$$

9. eventual conditions could not be matched sooner:

$$\forall l \in dom(f) : \lambda(l) \in \Phi^E \implies \nexists i : 1 \leq i \leq n : \begin{cases} f(l) > i \\ \zeta(e_1 \dots e_i) \models \lambda(l) \\ \forall l' \leq l : f(l') \leq i. \end{cases}$$

10. ASAP conditions are actually verified as soon as possible:

$$\forall l \in dom(f) : \lambda(l) \in \Phi^A \implies f(l) = \max(\{0\} \cup \{f(l') | l' < l\}).$$

The last condition is recursive and well-founded, because it follows the partial order on locations downwards. The previous definition of CLPO can be applied with the simple modification to get CLPO with conditions and the notion of linearization is not modified by LPOs with conditions entering the stage.

Definition 5.3 ($\models \subseteq \Sigma^\infty \times (\Sigma^* \to \mathcal{I}) \times$ LPO) LPOs with conditions are interpreted against finite or infinite words and state functions:

- $\gamma, \zeta \models \mathcal{L}$ with $\gamma \in \Sigma^*$ if γ linearizes $\mathcal{L}$,
- $\gamma, \zeta \models \mathcal{L}$ with $\gamma \in \Sigma^\omega$ if there is some $w \sqsubset \gamma$ such that $w, \zeta \models \mathcal{L}$.

■

Henceforth, universal LSCs are easily defined, as a couple of two CLPOs, with the same restricted events (prechart/main chart). An infinite word γ, with a state function ζ, satisfies a universal LSC iff, for every prefix $uv \in \Sigma^*$ of γ, if v linearizes the main chart, under the function ζ_u, that maps x to $\zeta(ux)$, for every w, then there is some w such that uvw is a prefix of γ and w linearizes the main chart, under ζ_{uv}.

Definition 5.4 ($\models \subseteq \Sigma^\infty \times (\Sigma^* \to \mathcal{I}) \times$ CLPO) A CLPO $\mathcal{C}$ is satisfied by a word $\gamma \in \Sigma^\infty$ and a state function ζ, if there is some LPO $\mathcal{L} \in$ *lpo_expand*($\mathcal{C}$) such that $\gamma, \zeta \models \mathcal{L}$. ■

Definition 5.5 ($\models \subseteq \Sigma^\infty \times (\Sigma^* \to \mathcal{I}) \times$ ULSC) A word $\gamma \in \Sigma^\infty$ and a state function ζ satisfy a ULSC $\Box(P, M)$, denoted $\gamma, \zeta \models \Box(P, M)$ iff $\forall u, v \in \Sigma^* :$ $\forall \gamma' \in \Sigma^\omega :$

$$(uv\gamma' = \gamma \text{ and } v, \zeta_u \models P) \implies \gamma', \zeta_{uv} \models M,$$

where $\zeta_w(x) = \zeta(wx)$. ■

5.2.3 Fluents

The language of LSC with conditions presented in the previous section is generic and needs to be instantiated to some particular logic to express conditions. The link between event traces and interpretations need also to be defined, in order to exploit it. In the play-out mechanism, the language is made of conjunctions of basic boolean functions over some primitive data type. Typically, comparisons between integers and equality between other data types is supported. These properties evolve because of the occurrence of events. The user defines what special events set the value of each property. We will follow a similar idea. Of course, in order to keep properties decidable, we will restrict ourselves to finite data type. Actually, we limit ourselves to propositional logic. The basic propositions will be *fluents* [60]. A fluent is a proposition which is associated to two sets of events: setting and unsetting events. When a setting event occurs, the fluent becomes true and remains true until an unsetting event happens. Then, it becomes and remains false until the next setting event occurs. Fluents have proved their value for adding state-based notions to event-based models. For instance, a model-checker has been developed for LTSA, which allows one to verify LTL formulae that mix fluents and events, against FSP (Finite State Processes) which are purely event-based [60].

Let $\mathcal{F}$ be a countable set of fluent names. A fluent formula is simply a propositional logic formula, defined over $\mathcal{F}$. We denote this logic with $\mathcal{B}(\mathcal{F})$ (boolean combinations of fluents).

$$\begin{array}{rcl} \varphi & ::= & p \quad (\text{with } p \in \mathcal{F}) \\ & | & \neg\varphi_1 \\ & | & \varphi_1 \vee \varphi_2 \end{array}$$

An interpretation is a function $i : \mathcal{F} \to \{\top, \bot\}$, that assigns a truth value to every fluent.

Definition 5.6 (Fluent Specification) A *fluent specification* is a triple of functions $\langle \mathit{Fluent}_{in}, \mathit{Fluent}_{on}, \mathit{Fluent}_{\mathit{off}} \rangle$, with $\mathit{Fluent}_{in} : \mathcal{F} \to \{\top, \bot\}$, $\mathit{Fluent}_{on} : \mathcal{F} \to 2^{\Sigma}$ and $\mathit{Fluent}_{\mathit{off}} : \mathcal{F} \to 2^{\Sigma}$. Given a fluent specification

$$FS = \langle \mathit{Fluent}_{in}, \mathit{Fluent}_{on}, \mathit{Fluent}_{\mathit{off}} \rangle,$$

we need to define a state function ζ_{FS}. It is defined recursively as $\zeta_{FS}(\epsilon) = \mathit{Fluent}_{in}$ and $\zeta_{FS}(wa) = \{x \mapsto \mathit{upd}(x, \zeta_{FS}(w)(x), a) | x \in \mathcal{F}\}$ where $\mathit{upd}(x, t, a)$ returns

- $\top$, if $t = \top$ and $a \notin \mathit{Fluent}_{\mathit{off}}(x)$;
- $\bot$, if $t = \bot$ and $a \notin \mathit{Fluent}_{on}(x)$;
- $\neg t$, otherwise.

■

5.2.4 Inter-Agent Specification with Fluents

In previous chapters, we relied heavily on the liveness-safety theorem: it enabled us to share out responsibilities among agents. In case of violations of a specification, it makes it possible to determine which agent is responsible for failure, i.e. whether an assumption on the environment has been falsified or whether the system is flawed. We need of course to adapt this theorem for dealing with conditions. First, we say that a condition *depends on* an event e if this condition contains a fluent p such that $e \in \mathit{Fluent}_{on}(p)$ or $e \in \mathit{Fluent}_{\mathit{off}}(p)$. We extend safety and liveness conditions to formulae of $\mathcal{B}(\mathcal{F})$. This obliges us to consider state functions in the definition as well. The following statements shall be understood as "under the interpretation of a state function ζ". ASAP formulae introduce new safety conditions on events: an event e is forbidden after a partial execution w if *we* violates some hot ASAP condition appearing in the main chart. Eventual conditions introduce liveness conditions on runs: a condition is required by some word w if some suffix v of w linearizes some cut in the main chart that enables a location labeled by this condition. The formalization of this expression is left to the reader.

An infinite run γ, with state function ζ, is e-safe iff, for every prefix $w \sqsubset \gamma$, if w forbids e (wrt ζ), we is not a prefix of γ. An infinite run is e-live iff (i) it is e-live, according to the previous definition, and (ii) there is no prefix $uv \sqsubset \gamma$ such that v linearizes some cut c in some ULSC, in which one successor location is labeled by a condition depending on e and for all subsequent prefixes of $uvv' \sqsubset \gamma$, v linearizes c. For short, a condition, depending on e should be evaluated to true but is never evaluated. The complex definition of liveness is needed in order to take into account LSCs containing conditions *only* or conditions terminating the LSC, combined with alternatives.

With this said, it is clear that we can again share out responsibilities between agents as before. But, first, intra-agent specifications must be adapted to include some fluent specification.

Definition 5.7 (Inter-Agent Specification) An inter-agent specification is a triple

$$\langle S, \mathcal{S}, FS \rangle,$$

where S is a structural model, $\mathcal{S}$ is a ULSC specification and FS a fluent specification. ■

It should be noted that safety and liveness have been defined on the level of LSCs with conditions, *instantiated to fluents*. We did not define it at the generic level, for we have no means at th is higher level to link conditions and events. Hence, we cannot assign the violation of eventual conditions to agents. Because fluents associate propositions with events, by telling which events can make propositions true and false, we can assign responsibilities. We have chosen a rough assignment: every agent owning an event that can change the value of a fluent appearing in some condition is responsible for the eventual occurrence of this condition. It is problematic when conditions mix fluents owned by the system and the environment. This association could be more fine-grained, but this needs more research.

It took us some effort to adapt the notions of liveness and safety to LSCs with conditions. The reward is that the notion of *correct implementation* is not different from the simpler event-based case. Thus, all analysis problems are identical.

Definition 5.8 (Correct Implementation) An intra-agent specification $(S, \mathcal{F})$ associated to a society of agents *Sys* is a *correct implementation* of an inter-agent specification $(S, \mathcal{S}, FS)$ iff

$$\forall \gamma \in Out(f_{Sys}) : \begin{cases} \gamma, \zeta_{FS} \text{ is } \Sigma_{Env}\text{-live} \implies \gamma, \zeta_{FS} \text{ is } \Sigma_{Sys}\text{-live} \\ \gamma, \zeta_{FS} \text{ is } \Sigma_{Env}\text{-safe} \implies \gamma, \zeta_{FS} \text{ is } \Sigma_{Sys}\text{-safe} \end{cases}$$

■

5.2.5 Analysis Problems with Fluents

When adding fluents to our model, it is clear that all our algorithms remain as efficient as before, because automata do not grow in size. Exponential-size automata remain simply exponential. The only problem for which membership in its complexity class relies on an additional argument is CCAV. It is is still in coNP, because exponentially long paths need not be taken into account for model-checking: there are always linear shortcuts. In particular, only the case of conditions can be problematic. Suppose that in the tableau automaton, we need to verify a condition with n fluents. There is no need to follow a path with exponentially many events: at most n events are needed to switch fluents to satisfying values. Actually, this number of events is upper-bounded by the largest Hamming distance between the current valuation and any satisfying valuation.

5.2.6 Summary

In conclusion, conditions can be added to LSCs for free, both from a computational complexity perspective, and from the perspective of definitions. It is remarkable that conditions altered only the lowest level of our definitions, viz. LPOs, but all higher levels, up to ULSC remained unchanged. The reader shall notice that cold conditions appearing in the prechart may cause surprising interpretations. Indeed, if a cold condition is violated, the prechart ends successfully and *is matched*. This is not in line with what Harel and Marelly do in [75]. They systematically use cold conditions in the prechart, because they cannot distinguish between ASAP and eventual conditions. Cold conditions are the only conditions that are evaluated as soon as they become enabled. Thus, they have to make a distinction between basic charts used as a prechart and basic charts used as a main chart: the prechart is satisfied if it is *fully* completed while the main chart can be satisfied by the violation of a cold condition. In order to follow completely their approach, we should define two types of linearizations: complete linearizations, in which the function f defined above is required to be total, and partial linearizations, which are the type of linearizations we defined. Then, we can re-phrase their semantics in our setting. Nevertheless, we stick to our approach and simply remark that using ASAP hot conditions instead of cold conditions in the prechart captures the user intent: "when the prechart is fully matched, which implies that all conditions have been verified immediately, the main chart shall be observed afterwards".

5.3 Time

5.3.1 Introduction

Scenarios can also be used to describe the behaviour of timed systems. In this section, we explain how the previous arsenal can be extended to cope with real-time specifications. This was done by Marelly and Harel, in their study of play-out, too. We will follow Dill and Alur's approach to describing real-time systems. To our discrete-time model, we add real-valued clocks. These clocks can be *reset* and tested for inequalities against rational numbers. As part of the signature of our language, we now assume a countable set $\mathbb{C}_F$ of *formal clocks*. The only constants that can be used in constraints are rational-valued. We let $\mathbb{Q}_{\geq 0}$ denote the set of all non-negative rational numbers.

We will successively introduce Timed Inter-Agent Specifications and Timed Intra-Agent Specifications. We will then adapt the definition of a correct implementation to the timed case and investigate how these changes impact the problems in which we are interested. But, first, we need to define precisely what a timed language is. Since the beginning of this thesis, we identified executions in which observable events happen and words on a given alphabet. This allowed us to use a well-defined and usual semantic domain that was shared with other specification languages, and also to benefit from results and tools from this field. When real-time is introduced in our systems, i.e. the time of occurrence of an event becomes observable, we need to move to timed languages, to go on using

our theoretical setting.

5.3.2 Preliminaries: Timed Languages

The concept of timed language is classical and we briefly recall the definitions [5]. A finite timed word over some alphabet Σ is a sequence $w \in (\Sigma \times \mathbb{R}_{>0})^*$. Every event in the sequence is tagged with the delay that elapsed between its occurrence and the occurrence of the event just before. Intuitively, a sequence $(e_0, \tau_0)(e_1, \tau_1), \ldots, (e_n, \tau_n)$ means that event e_i occurs at time $\sum_{j=0}^{i} \tau_i$. Thus, τ_i is the delay between event e_{i-1} and e_i. The same intuition can be applied to infinite words. An infinite timed word is an infinite sequence $\gamma \in (\Sigma \times \mathbb{R}_{>0})^\omega$, fulfilling the non-Zeno condition. The non-Zeno condition asserts that "time cannot be stopped".

Definition 5.9 (Zeno word) Let $\gamma = (e_0, \tau_0)(e_1, \tau_1) \ldots \in (\Sigma \times \mathbb{R}_{>0})^\omega$, then γ is non-Zeno if $\forall t \in \mathbb{R} : \exists j \geq 0 : t < \sum_{i=0}^{j} \tau_j$. ■

Definition 5.10 (Timed Words ($\mathbb{T}_*(\Sigma)$,$\mathbb{T}_\omega(\Sigma)$,$\mathbb{T}_\infty(\Sigma)$))

- $\mathbb{T}_*(\Sigma) = (\Sigma \times \mathbb{R}_{>0})^*$ is the set of all finite timed words.
- $\mathbb{T}_\omega(\Sigma) = \{\gamma \in (\Sigma \times \mathbb{R}_{>0})^\omega | \gamma \text{ is non-Zeno}\}$ is the set of all infinite timed words.
- $\mathbb{T}_\infty(\Sigma) = \mathbb{T}_*(\Sigma) \cup \mathbb{T}_\omega(\Sigma)$.

■

5.3.3 Timed Inter-Agent Specification

Graphically, a timed LSC is displayed in Fig. 5.1. It enforces that, if no answer is received from the database within 2 minutes[2] the client informs the communication manager that its download failed. This LSC is quite subtle, because it obliges the client to answer "no" *after at least 120 seconds* but sets no upper limit on the time at which a "yes" answer will be sent. This form of LSC is the same as in [74, 75]. Note that, instead of clock resets, assignments are used to freeze time, i.e. store the current time in a real-valued variable. Atomic constraints are of the form $x \sim \texttt{Time} + k$ where x is a clock name, k is a rational number and $\sim \in \{\leq, <\}$. This is the choice of Harel and Marelly. Experience tells us that presenting time constraints in that form improves readability. Nevertheless, the two modes are interchangeable, it suffices to replace all "time freezes" by "clock resets" and every clock constraint, which are of the form $x \sim \texttt{Time}+k$, by $x \sim k$. Clock constraints must be interpreted "ASAP", i.e. we do not allow eventual clock conditions, and clock resets are guarded by an event. We make such a decision to avoid "spontaneous" clock resets, that would not be mapped onto events, and conditions that can be mapped at multiple times. In the spirit of LSCs, this reduces as much as possible nondeterminism and gives a feeling of executability to the specification formalism. This high

[2] We assume that seconds are used as time unit

degree of operationalism is a characteristic of scenario-based languages. It opposes them to approaches relying on declarative approaches, which favour more nondeterministic specifications, but are also more difficult to comprehend.

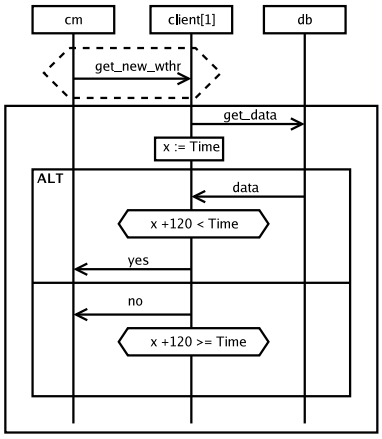


Figure 5.1: Real-time LSC

Formally, we introduce a new labeling of locations, namely *reset*, which are sets of formal clocks. Resets are hidden, i.e. non-observable, events which will be taken as soon as possible, in the spirit of ASAP conditions. We also add clock constraints to our condition language. Atomic clock constraints are of the form $x \sim k$, with $x, y \in \mathbb{C}_F$, $k \in \mathbb{Q}_{\geq 0}$ and $\sim \in \{\leq, <\}$. Clock constraints are boolean compositions of atomic clock constraints. Let Θ denote the set of all clock constraints. We add clock constraints to the language of conditions. Clock constraints may only be used as ASAP conditions, either in cold or hot flavour.

$$\begin{aligned}
\Theta_c, \Theta_h &\triangleq \Theta \\
\Phi_h^A &\triangleq \Theta_h \uplus \Psi \\
\Phi_h^E &\triangleq \Psi \\
\Phi_c^A &\triangleq \Theta_c \uplus \Psi \\
\Phi^A &\triangleq \Phi_h^A \uplus \Phi_c^A \\
\Phi^E &\triangleq \Phi_h^E \\
\Phi_h &\triangleq \Phi_h^A \uplus \Phi_h^E \\
\Phi_c &\triangleq \Phi_c^A
\end{aligned}$$

Clock constraints are evaluated against *clock valuations*. A clock valuation is a function $\nu \in \mathbb{V} = \mathbb{C}_F \to \mathbb{R}_{\geq 0}$. Given some $\nu \in \mathbb{V}$, we define the following operations:

- Time passing of $t \in \mathbb{R}_{\geq 0}$ units: $\nu + t$ denotes the clock valuation that returns $\nu(x) + t$ for every x.
- Reset of $X \subset \mathbb{C}_F$: $\nu[X := 0]$ denotes the clock valuation that maps every $x \in X$ to 0 and every $x \notin X$ to $\nu(x)$.

$\nu \models \phi$ denotes, as usual, the fact that ϕ is true, under interpretation ν.

Definition 5.11 ($\models \subset \mathbb{V} \times \Theta$) The semantics of clock constraints is defined inductively, on the structure of clock constraints.

$$\begin{array}{rcl} \nu \models \neg\phi & \text{iff} & \nu \not\models \phi \\ \nu \models \phi_1 \wedge \phi_2 & \text{iff} & \nu \models \phi_1 \text{ and } \nu \models \phi_2 \\ \nu \models x < k & \text{iff} & \nu(x) < k \\ \nu \models x \leq k & \text{iff} & \nu(x) \leq k \end{array}$$

■

Definition 5.12 (Timed LPO) A timed Σ-LPO is a $\left(\Sigma \uplus \Phi \uplus 2^{\mathbb{C}_F}\right)$-LPO, with the additional constraint that $\forall l \in L : \lambda(l) \subset \mathbb{C}_F \implies \exists l' \in L : \lambda(l') \in \Sigma$ and $l' < l$. Thus, locations can be labeled by events, or conditions that are simple conditions or clock constraints, as explained above, and finite sets of formal clocks, that are reset when the location is reached. The additional constraint states that resets must be associated with observable events. ■

Associating clock resets with observable events is an idea that can be found in Event-clock automata [7, 139]. It is a bit restrictive, with respect to [75]. Nevertheless, all examples of timed scenarios in [75] respect this constraint. Intuitively, it seems very strange to measure time without being able to tell exactly from which point in time this measure started.

As in the previous section, LPOs are related to (timed) words through linearization. Again, the mapping is made explicit, to take into account that some events may be mapped onto the same index of the word.

A finite timed word $(e_1, \tau_1) \ldots (e_n, \tau_n) \in \mathbb{T}_*(\Sigma)$ is a linearization, under the interpretation of some state function ζ, of an LPO (with conditions) if there is a partial function $f : L \nrightarrow [n]$ such that

1. some location is mapped on the last index of the finite word. Thus, there is no "trailing garbage" in the word:

$$\exists l \in L : f(l) = n.$$

2. Let $\nu_0 = \{x \mapsto 0 | x \in \mathbb{C}_F\}$, i.e. all clocks are initially set to 0 and $\nu_i = (\nu_{i-1} + \tau_i)[Res_i := 0]$, with $Res_i = \{x \in \lambda(l) | f(l) = i \ \wedge \lambda(l) \subset \mathbb{C}_F\}$.

3. location ordering is respected in the linearization:

$$\forall l, l' \in dom(f) : l \leq l' \implies f(l) \leq f(l').$$

4. only unrestricted events may be skipped:

$$\forall i \in [n] : i \notin ran(f) \implies e_i \notin \Sigma_R.$$

5. events in the finite word match the labeling of event-labeled locations, according to the indexes returned by f:

$$\forall l \in dom(f) : \lambda(l) \in \Sigma \implies \lambda(l) = e_{f(l)}.$$

6. event-labeled locations must be strictly ordered by f:

$$\forall l, l' \in dom(f) : (\lambda(l) \in \Sigma \wedge \lambda(l') \in \Sigma \wedge l < l') \implies f(l) < f(l').$$

7. premature abortion is only possible if a cold condition is violated

$$dom(f) \subseteq L \implies (\exists l \in L : f(l) = n \wedge \lambda(l) \in \Phi_c \wedge \zeta(e_O \dots e_n) \not\models \lambda(l)).$$

8. hot conditions must be matched:

$$\forall l \in dom(f) : \lambda(l) \in \Phi_h \implies \zeta(e_1 \dots e_{f(l)}), \nu_{f(l)} \models \lambda(l).$$

9. non-terminal conditions must be matched:

$$\forall l \in dom(f) : f(l) < n \wedge \lambda(l) \in \Phi \implies \zeta(e_1 \dots e_{f(l)}), \nu_{f(l)} \models \lambda(l).$$

10. eventual conditions could not be matched sooner:

$$\forall l \in dom(f) : \lambda(l) \in \Phi^E \implies \nexists i : 1 \leq i \leq n : \begin{cases} f(l) > i \\ \zeta(e_1 \dots e_i) \models \lambda(l) \\ \forall l' \leq l : f(l') \leq i. \end{cases}$$

11. ASAP conditions are verified and resets are performed as early as can be:

$$\forall l \in dom(f) : \lambda(l) \in \Phi^A \cup 2^{\mathbb{C}_F} \implies f(l) = \max(\{0\} \cup \{f(l') | l' < l\}).$$

We can, as usual, reuse all previous definitions about LPOs to obtain the semantics of timed LSCs. First, LPOs and CLPOs are, as usual:

Definition 5.13 ($\models \subseteq \mathbb{T}_\infty(\Sigma) \times (\Sigma^* \to \mathcal{I}) \times$ TLPO) Timed LPOs are evaluated against timed words and interpretations.

- for every $\gamma \in \mathbb{T}_*(\Sigma)$,

$$\gamma, \zeta \models \mathcal{L} \iff \gamma, \zeta \text{ is a linearizations of } \mathcal{L},$$

- for every $\gamma \in \mathbb{T}_\omega(\Sigma)$,

$$\gamma, \zeta \models \mathcal{L} \iff \exists w \in \mathbb{T}_*, \gamma' \in \mathbb{T}_\omega : \gamma = w\gamma' \text{ and } w, \zeta \models \mathcal{L}.$$

■

CLPOs can be extended to real-time, by reusing the previous definition: a CLPO $\mathcal{C}$ corresponds to a set of LPOs, viz. *lpo_expand*($\mathcal{C}$). We skip the details and immediately define the semantics of real-time ULSC.

Definition 5.14 ($\models \mathbb{T}_\omega(\Sigma) \times (\Sigma^* \to \mathcal{I}) \times$ ULSC) An infinite timed word γ and a state function ζ satisfy a ULSC $\Box(P, M)$ iff, for all $u, v \in \mathbb{T}_*(\Sigma)$ and $\gamma' \in \mathbb{T}_\omega(\Sigma)$,

$$uv\gamma' = \gamma \text{ and } v, \zeta_u \models P \implies \gamma', \zeta_{uv} \models M.$$

■

Finally, remark that the safety-liveness theorem (Th. 3.22) is seamlessly adapted to the timed case: since clock constraints are ASAP conditions, they introduce new safety conditions on events. Liveness is not modified by the introduction of time.

5.3.4 Timed Intra-Agent Models

Since we have introduced time in our inter-agent specifications, we need to put time into intra-agent specifications. In order to do so, we follow the approach of timed games [105, 43]. A concurrent timed execution proceeds as follows: at every step all agents are asked what actions they want to perform next and *when* they want to perform any of these actions. Let the time of the current step be t and the smallest chosen delay be ϵ. Then, all agents are put to sleep until $t + \epsilon$. At that time, one of the actions chosen to be performed after delay ϵ is non-deterministically taken and the agents gather to play another round. Agents may only act in non-Zeno ways. That is, they may not choose smaller and smaller delays between actions, to finally stop time.

With this additional assumption, it is noteworthy that the untimed case corresponds exactly to the previous case: agents cannot make assumptions about the relative speed of their environment and only propose to perform actions. However, there is a fairness constraint: if an agent keeps asking for performing an action at some absolute time, it will eventually do it.

Definition 5.15 (Real-time Strategy) A real-time strategy for agent $a \in Ag$ is a function $f_a : \mathbb{T}_*(\Sigma) \to (2^{\Sigma_a^s} \times \mathbb{R}_{>0})$ that returns to every finite play what the choice of agent a, i.e. what actions a wants to perform next and when one of these actions should be performed. An infinite timed sequence $(e_0, \tau_0)(e_1, \tau_1)\ldots$ is played according to such a strategy f_a iff for every $i \geq 0$, if $e_i \in \Sigma_a^s$, then $f_a((e_0, \tau_0)\ldots(e_{i-1}, \tau_{i-1})) = (A, \tau_i)$, with $e_i \in A$. The outcomes of a real-time strategy f_a, denoted $Out(f_a)$, are the timed sequences played according to this strategy. ■

Note that outcomes may be non-Zeno. In order to ensure progress, we introduce the technical notion of a *blame*. In an infinite timed sequence, $\gamma = (e_0, \tau_0)(e_1, \tau_1)\ldots$, agent a is blamed at position i, written *blame*(a, γ, i) if $e_i \in \Sigma_a^s$.

Definition 5.16 (Progressive strategy) A strategy f_a is *progressive* if, for every outcome γ, either this outcome is non-Zeno ($\gamma \in \mathbb{T}_\omega(\Sigma)$) or there are only finitely many indices i such that $blame(a, \gamma, i)$. ■

Intuitively, if the time sequence converges, agent a cannot be held responsible, because it does not choose the smallest delay. Alternatively, agent a can be seen as a victim: he is prevented from acting by the abusive behaviour of its environment. We repeat that the *untimed* version of a real-time strategy corresponds to the intra-agent model presented in Sec. 4.2.3.

5.3.5 Analysis Problems

The problems described in Section 4 are still relevant for real-time systems. Here, we will focus on two of these problems: verification of closed systems and realizability checking. These two problems can be solved by automata-theoretic techniques, resorting to the theory of timed automata [5]. First, we present how linearizations of timed LPOs can be recognized by deterministic timed automata [5]. Second, we will show how these automata can be reused to recognize all the models of a timed ULSC specification.

We start by defining the two problems. We focus on closed centralized verification. The agent system is given as a unique timed automaton, recognizing all the runs of this system.

Definition 5.17 (TCCAV) The problem of timed centralized closed agent verification (TCCAV) is to verify, given a timed automaton $\mathcal{A}$ and a timed ULSC specification $\mathcal{S}$, whether $\mathscr{L}(\mathcal{A}) \models \mathcal{S}$. ■

The second problem is realizability. Here, the problem is to decide whether there is some real-time strategy that is correct with respect to some ULSC specification.

Definition 5.18 (TCOAD) Timed Open Agent Design (TCOAD) is the problem of deciding whether there is some progressive real-time strategy f_{Sys} that is a correct implementation a real-time inter-agent specification $\langle S, \mathcal{S} \rangle$. ■

The two problems will be solved using automata-theoretic techniques. Thus, we start by introducing timed automata. Then, we will explain how TLPO linearizations can be recognized by automata and how these automata can be transformed to provide solutions to TCCAV and TCOAD. The solutions are not particularly original and will mostly use well-known results, either from the theory of timed automata [5] or controller synthesis for real-time systems [17, 16, 116, 105].

Definition 5.19 (Timed Automaton) A *timed automaton* (TA) is a tuple

$$\langle Q, q_0, \Delta, \mathcal{X}, \Omega \rangle,$$

where

- Q is a finite set of locations,
- $q_0 \in Q$ is an initial location,
- $\Delta \subseteq Q \times \Sigma \times 2^{\mathcal{X}} \times \Theta(\mathcal{X}) \times Q$ is a transition relation. A transition (q, e, R, ϕ, q') means that, when the control location resides in q and event e occurs, if the constraint ϕ is true (after having reset all clocks in R), the control location moves to q'.
- $\mathcal{X}$ is a finite set of clocks,
- Ω is some acceptance (Büchi, Streett, ...) condition, relative to Q. See section 1.1.2.

A TA is *deterministic* if its transition relation is functional on the source state, the triggering event and the clock constraint. In details, for every two transitions $(q_1, e_1, R_1, \phi_1, q_1')$,$(q_2, e_2, R_2, \phi_2, q_2') \in \Delta$ if source states are the same $(q_1 = q_2)$, events are the same $(e_1 = e_2)$ and constraints are compatible, $(\exists \nu : \nu[R_1 := 0] \models \phi_1$ and $\nu[R_2 := 0] \models \phi_2)$, then target states are the same $(q_1' = q_2')$.

A state of a TA is a couple (l, ν), where $\nu : \mathcal{X} \to \mathbb{R}_{\geq 0}$ is a clock valuation. Let us denote by $S(\mathcal{A})$ the set of states of some TA $\mathcal{A}$. A TA defines a relation $\to \subseteq S(\mathcal{A}) \times \Sigma \times \mathbb{R}_{>0} \times S(\mathcal{A})$ according to the following rule.

$$(q, \nu) \xrightarrow{(e,t)} (q', \nu') \iff \exists (q, e, R, \phi, q') \in \Delta : \nu' = (\nu + t)[R := 0] \text{ and } \nu' \models \phi.$$

It is easy to check that deterministic TA give rise to a deterministic $\to$ relation i.e., functional on its third first arguments.

A TA accepts a timed word $(e_0, \tau_0)(e_1, \tau_1) \ldots \in \mathbb{T}_\omega(\Sigma)$ iff there is a sequence of states $s_0 s_1 \ldots$ such that

1. $s_0 = (q_0, \{x \mapsto 0 | x \in \mathcal{X}\})$,
2. $\forall i \geq 0 : s_i \xrightarrow{(e_i, \tau_i)} s_{i+1}$,
3. $q_0 q_1 \ldots$ satisfies the acceptance condition Ω.

The language of a TA is $\mathscr{L}(\mathcal{A}) = \{\gamma | \gamma \text{ is accepted by } \mathcal{A}\}$. ■

As previously, states are *cuts* in the LPO. Automata clocks are formal clocks $(\mathcal{X} = \mathbb{C}_F)$. For the sake of simplicity, we skip the treatment of fluents and focus on time constraints. In an LPO, we define the *ASAP-closure* of a location l, denoted by $cl(l)$ as the set of all locations greater than l and labeled by ASAP events i.e., clock resets or ASAP conditions. Formally,

$$cl(l) = \{l' | l' > l \text{ and } (\nexists l'' : \lambda(l'') \in \Sigma \text{ and } l < l'' \leq l')\}.$$

The DTA accepting the linearizations of a TLPO $\mathcal{L}$ is

$$\mathcal{A}_{\mathcal{L}} = \langle 2^L \cup \{\text{ko}, \text{ok}\}, \emptyset, \Delta, \textit{Büchi}(\{\text{ok}\})\rangle,$$

where ko and ok are both sink states (i.e. there is a self-loop on them). The former represents failure (i.e. the automaton rejects) and the second represents success (the automaton accepts). There are five types of transitions. *normal transitions* describe legal occurrences of restricted events. *premature success transitions* prescribe that violating a cold condition causes a premature termination. *unrestricted event transitions* skip unrestricted events, *failure transitions* describe the illegal occurrences of restricted events and *termination transitions* indicate that when all locations have been reached, the chart succeeds. Formally,

normal transitions are of the form

$$\left(C_1, e, R, \left(\bigwedge_{\phi \in \Lambda_h} \phi\right) \wedge \left(\bigwedge_{\psi \in \Lambda_c} \psi\right), C_2\right),$$

where $C_1 \subset L$ and there is some $l \in L$ such that

- $\lambda(l) = e$,
- $l \notin C_1$,
- $C_2 = C_1 \cup cl(l)$,
- $\forall l' < l : l' \in C_1$,
- $R = \{l' \in cl(l) | \lambda(l') \in 2^{\mathbb{C}_F}\}$,
- $\Lambda_h = \{l' \in cl(l) | \lambda(l') \in \Phi_h^A\}$,
- $\Lambda_c = \{l' \in cl(l) | \lambda(l') \in \Phi_c\}$.

premature success transitions are of the form

$$\left(C_1, e, R, \left(\bigwedge_{\phi \in \Lambda_h} \phi\right) \wedge \left(\bigvee_{\psi \in \Lambda_c} \neg\psi\right), \text{ok}\right),$$

and are defined if there is some $l \in L$ such that

- $\lambda(l) \in \Sigma$,
- $\forall l' < l : l' \in C_1$,
- $R = \{l' \in cl(l) | \lambda(l') \in 2^{\mathbb{C}_F}\}$,
- $\Lambda_h = \{l' \in cl(l) | \lambda(l') \in \Phi_h^A\}$,
- $\Lambda_c = \{l' \in cl(l) | \lambda(l') \in \Phi_c\}$.

unrestricted event transitions are transitions of the form $(q, e, \emptyset, \top, q)$, where $e \notin \Sigma_R$. They skip unrestricted events.

failure transitions are transitions that complete the automaton's transition relation and lead to ko.

termination transitions are of the form

$$(L, e, \emptyset, \top, \text{ok}).$$

They are taken when all locations in the LPO have been reached.

This automaton can be turned into a non-deterministic automaton recognizing the *complement* of a ULSC language. For some ULSC $S = \Box(P, M)$, we call it $\mathcal{A}_{\neg S}$. To verify that a run $\gamma \in \mathbb{T}_\omega(\Sigma)$ violates a ULSC, this automaton waits in the initial state, thus skipping some finite prefix of γ. Nondeterministically, it decides to check that the ULSC will not be verified, after this prefix. It needs to verify that the prechart is matched, namely ok is reached in $\mathcal{A}_P$, but, immediately after, the main chart is not matched i.e., ok is *never* reached in $\mathcal{A}_M$.

Theorem 5.20 TCCAV is PSPACE-complete. ■

Proof 5.20

Hardness comes from the PSPACE-hardness of emptiness checking for timed automata. Membership follows from the fact there are at most $|L|$ clocks used in $\mathcal{A}_{\neg S}$. Thus, the state-space of the region automaton for $\mathcal{A}_{\neg S}$ contains at most an exponential number of regions, viz. $2^{2\cdot|L|}$. The synchronous product of $\mathcal{A}$, the timed automaton representing the intra-agent specification, with $\mathcal{A}_{\neg S}$ does not increase their state space: it remains simply exponential. Hence, emptiness can be tested in PSPACE, by guessing a simple path in the automaton, just as in the finite state case. □

The TA presented above, i.e. $\mathcal{A}_L$ can be used as a basis to recognize all models of a ULSC. At every step, at new copy of $\mathcal{A}_L$ is launched. However, the clocks of this copy, that were taken from $\mathbb{C}_F$, are renamed, using fresh clocks, in order to avoid conflicts. For renaming, we use a distinct set of *actual clocks*, $\mathbb{C}_A$. An interesting fact is that the number of actual clocks needed to avoid conflicts is upper-bounded by $|L|^2$. This is due to the fact that live copies do not survive for more than $|L|$ steps. A second fact is that this TA is deterministic. Let us call $\mathcal{A}_\mathcal{S}$ the DTA recognizing all models of a real-time LSC specification.

Theorem 5.21 TCOAD is EXPTIME-complete. ■

Proof 5.21

Hardness comes from the fact that this problem extends COAD, which is EXPTIME-complete, see Th. 4.31. Membership comes from the fact that the problem can be reduced to solving a two-player, 3-colour, parity game, on the region automaton. Since the region automaton of a a timed automaton with n locations and k clocks has $\mathcal{O}(n \cdot 2^k)$ states and, in our case $\mathcal{A}_\mathcal{S}$ has a polynomial number of clocks, we get a simply-exponentially large region automaton. □

Next, we provide some details on the reduction from TCOAD. Assume that we have at our disposal a DTA with one Streett pair. We start by building

the region automaton from it. Assume, without loss of generality, that all constraints in $\mathcal{A}$ use only natural numbers. Let us denote the largest constant in clock constraints involving clock x by k_x. Two clock valuations are equivalent, denoted $\nu{\sim}\nu'$ iff all the following constraints hold:

- $\forall x \in \mathcal{X} : (\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor)$ or $(\nu(x) > k_x \wedge \nu'(x) > k_x)$;
- $\forall x, y \in \mathcal{X} : \nu(x) \leq k_x \wedge \nu'(y) \leq k_y : \mathit{fract}(\nu(x)) \leq \mathit{fract}(\nu(y)) \iff \mathit{fract}(\nu'(x)) \leq \mathit{fract}(\nu(y))$;
- $\forall x \in \mathcal{X} : \nu(x) \leq k_x : \mathit{fract}(\nu(x)) = 0 \iff \mathit{fract}(\nu'(x)) = 0$.

Let us denote by $[\nu]$ the equivalence class of ν, i.e. $\{\nu' \in \mathbb{V} | \nu'{\sim}\nu\}$. Such an equivalence class is a *clock region*. Remark that there are only finitely many clock regions.

A clock region α' is a time-successor of a clock region α' iff for each $\nu \in \alpha$, there is some $t \in \mathbb{R}_{>0}$ such that $\nu + t \in \alpha'$. A clock region α' is a next time-successor of a region α iff $\alpha' \neq \alpha$, α' is a time-successor of α and for every time-successors α'' of α such that $\alpha'' \notin \{\alpha, \alpha'\}$, α'' is also a time-successor of α'. The essential property of clock regions is that, for every clock constraint ϕ appearing in $\mathcal{A}$ and every clock region $[\nu]$, all valuations in a region agree on the truth value of the constraint[3]:

$$\nu \models \phi \iff \forall \nu' \in [\nu] : \nu' \models \phi.$$

Thus, we allow ourselves to talk about the truth value of a clock constraint, under the interpretation of a *clock region*.

Let κ be a symbol not in Σ. The region automaton of $\mathcal{A}$ is the Streett Automaton

$$R(\mathcal{A}) = \langle R(Q), R(q_0), R(\Delta), \{R(E), R(F)\}\rangle,$$

where

- $R(Q) = \{(q, \alpha) | q \in Q \text{ and } \alpha \text{ is a clock region}\}$,
- $R(q_0) = (q_0, [\{x \mapsto 0 | x \in \mathcal{X}\}])$,
- The transition relation follows the rules:
  - for $e \in \Sigma$, $((q, \alpha), e, (q', \alpha'))$ iff $\exists (q, e, R, \phi, q') \in \Delta$ such that $\alpha' = \alpha[R := 0]$ and $\alpha' \models \phi$.
  - $((q, \alpha), \kappa, (q, \alpha'))$ iff α' is a next time-successor of α.
- $R(E) = \{(q, \alpha) | q \in E\}$,
- $R(F) = \{(q, \alpha) | q \in F\}$.

[3]This property is relative to a fixed automaton $\mathcal{A}$, because clock regions depend on the largest constants, k_x appearing in clock constraints *of* $\mathcal{A}$

Consider some timed run

$$w = (q_0, \nu_0) \xrightarrow{(e_0,\tau_0)} (q_1, \nu_1) \dots (q_i, \nu_i) \xrightarrow{(e_i,\tau_i)} (q_{i+1}, \nu_{i+1}) \dots .$$

Its *region trace* is the sequence

$$[w] = (s_0, \alpha_0)(s_1, \alpha_1) \dots ,$$

such that

- $\alpha_0 = [\nu_0]$,
- α_{i+1} is the next time-successor of α_i,
- there is a sequence of indices $I = i_0 < \ldots < i_j < \ldots$ such that
  - $\forall j \notin \{i_0, i_1, \ldots\} : s_j = s_{j-1}$,
  - $\forall j \in \{i_0, i_1, \ldots\} : s_j = q_{i_j}$.

By induction, one can prove the following:

Proposition 5.22 Let $\mathcal{A}$ be a timed automaton. A timed run $w \in \mathbb{T}_*(\Sigma^*)$ leads to state (q, ν) iff $[w]$ leads to state $(q, [\nu])$ in $R(\mathcal{A})$. ■

With this construction at hand, it is simple to build a game graph for solving this problem:

$$G_{\mathcal{S}} = \langle V, V_0, \Delta, \Omega \rangle,$$

with

- $V = (\{Sys\} \times R(Q)) \cup (\{Env\} \times \Sigma^s_{Sys} \cup \{\kappa\} \times R(Q))$,
- $V_0 = \{Sys\} \times \Sigma \cup \{\kappa\} \times R(Q)$,
- $(v, v') \in \Delta$ iff
  - $v = (Sys, q)$, $v' = (Env, a, q)$ and $a \in \Sigma^s_{Sys}$, or
  - $v = (Env, x, q)$, $v' = (Sys, q')$ and
    - $(q, x, q') \in R(\Delta)$, or
    - $\exists b \in \Sigma^s_{Env} : (q, b, q') \in R(\Delta)$.
- $\Omega = \mathit{Streett}(\{(E, F)\})$, with $E = \{Sys\} \times R(E) \cup (\{Env\} \times \Sigma^s_{Sys} \cup \{\kappa\} \times R(E))$ and $F = \{Sys\} \times R(F) \cup (\{Env\} \times \Sigma^s_{Sys} \cup \{\kappa\} \times R(F))$

Note that we do not take progressiveness into account. However, it is simple to incorporate, in the acceptance condition of $\mathcal{A}$, an additional condition to cope with this problem. In E, it is asked that player 0 is blamed infinitely often, i.e. one additional bit is needed in the Büchi synchronous product, which is set to 1 every time player 0 makes a move. In F, we also add $|\mathbb{C}_A|$ bits to the synchronous product, in order to ensure that every clock x is either reset or larger than its largest constant k_x infinitely often.

This game graph is very similar to the one presented in section 4.36, to solve COAD. However, the reader must remark that there is one main difference. Here, in every turn, *Sys* starts by picking one move, which is to let time pass (κ) or to perform a discrete action ($a \in \Sigma^s_{Sys}$). Then, *Env* can choose to let this move happen or to preempt this move, by performing one of its own discrete actions $b \in \Sigma^s_{Env}$. This corresponds to the intuition presented above: both players choose a certain duration, that they want to spend idling, and a discrete action to perform afterwards. Thus, player *Sys* may never assume that its opponent will let him play, because *Env* can always choose a smaller (or equal) delay and be the first to move.

5.3.6 Conclusion

We have shown, in this section, that LSCs can be extended with real-time, in a straightforward way. In order to do so, we add clocks and clock constraints to the language. This idea is not ours: Harel and Marelly have already extended LSCs with the same concept [75, 74], being themselves very much inspired by the classical work on timed automata [5]. On the contrary, Message Sequence Charts make use of timers, instead of clock resets and clock constraints. Arrows, decorated with time constraints, can link two locations, say l_1 and l_2 in the diagram. This has the expected meaning that the time elapsing between l_1 and l_2 must satisfy the decorating constraint. Timers are but syntactic sugar on an approach with explicit clock resets and clock constraints.

One deficiency of our timed ULSCs is that they do not allow clocks to be shared between the prechart and the main chart. It is possible to modify our solution to cope with shared clocks, but this requires some heavy modifications to the definitions and to the algorithms.

Extending LSCs with time did not cause much trouble to the definitions: only timed LPOs must be defined, which is not very different from LPOs with conditions. All other concepts remain the same as before. Algorithms for verification and synthesis are easily adapted, too. The complexity of synthesis does not increase, while verification of closed centralized systems become PSPACE-complete instead of coNP-complete.

We close this chapter with another extension to plain LSCs, that makes it possible to describe parameterized systems, in which the number of agents is not a priori fixed.

5.4 Symbolic Instances

Symbolic LSCs (SymLSC) are LSCs with symbolic instances: one instance is a placeholder for many possible instances. It possible to describe the behaviour of unbounded families of agents. In Symbolic LSCs, we need to introduce *agent roles*. The idea is that every agent can play several roles. Thus, to every agent, we associate a set of roles. In logical terms, Symbolic LSCs are to LSCs what first-order logic is to propositional logic.

Definition 5.23 (Roles - Population) *Role* is a set of *roles*. A *population* is

a partial function, with *finite domain*, $Pop : Ag \nrightarrow 2^{Role}$, mapping every agent to the roles he plays. Let $\mathbb{P}$ denote the set of all populations. ■

We assume here that agents may not change roles during system execution. We also drop the hypothesis that Ag is finite and only require it to be countable.

We assume that we are given a countable set of first-order variables, Var. Var and Ag are distinct. An interpretation of $V \subset Var$ is a function $\theta : V \to Ag$. Message terms are also extended to include first-order variables. We write $\Sigma(V)$, for $V \subset Var$, to denote the set $(Ag \cup V) \times \mathcal{M} \times (Ag \cup V)$. Ground events are events from $\Sigma(\emptyset)$. Applying a V-interpretation to an event in $\Sigma(V)$ yields a ground event, in which all occurrences of $v \in V$ are replaced by $\theta(v)$. Let $\mathbb{I}$ represent the set of all interpretations.

In the same vein, we extend LPOs, and transform them in *Quantified Labeled Partial Order* (QLPO). A QLPO is an LPO over $\Sigma'(V)$, or an expression of the form $\forall x : R : Q$ or $\exists x : R : Q$, where $x \in Var$, $R \in Role$ and Q is a QLPO, in which x is a free variable. We use the usual definition of *free* and *bound* variable. A variable is bound if it occurs within the scope of a quantifier. It is free if it is not bound.

Applying an interpretation of variables to an LPO simply replaces all occurrences of variables by their interpretations in event terms ($\Sigma(V)$). If all free variables of an LPO are interpreted in θ, this yields a ground LPO, as well.

Definition 5.24 ($\models \subseteq \mathbb{P} \times \mathbb{I} \times \Sigma^\infty \times$ QLPO) Let $\gamma \in \Sigma^\infty$, Pop is a population and θ is a first-order variable interpretation.

- $Pop, \theta, \gamma \models Q$, with $Q \in$ LPO iff $\gamma \models \theta(Q)$.
- $Pop, \theta, \gamma \models \forall x : R : Q$ iff, for every $a \in Ag$,

$$R \in Pop(a) \implies Pop, \theta \cup \{x \mapsto a\}, \gamma \models Q.$$

- $Pop, \theta, \gamma \models \exists x : R : Q$ iff, there is some $a \in Ag$, such that

$$R \in Pop(a) \text{ and } Pop, \theta \cup \{x \mapsto a\}, \gamma \models Q.$$

■

A Symbolic ULSC is a pair $\square(P, M)$ such that

1. P is a A-LPO, with $A \subseteq \Sigma(_P)$. All variables in V_P are free in the prechart P. We do not allow quantifiers in the prechart, as is also done by [119].
2. M is a B-QLPO, with $B \subseteq \Sigma(V)$, and $B \supseteq A$. The sole free variables in M are V_P

Symbolic LSCs are interpreted against populations and infinite words $\gamma \in \Sigma^\omega$. An interpretation satisfies a Symbolic LSC if, whenever the prechart is

matched, the main chart is also matched afterwards. Remark that matching can be done according to several variable interpretations, and we take all of them into account.

Definition 5.25 ($\models \subseteq \mathbb{P} \times \mathbb{I} \times \Sigma^\omega \times$ SymLSC) $Pop, \gamma \models \Box(P, M)$ iff, for every first-order variable interpretation θ, for every decomposition $uv\gamma'$ of γ,

$$Pop, \theta, v \models P \quad \Longrightarrow \quad Pop, \theta, \gamma' \models M.$$

■

A Symbolic LSC specification $\mathcal{S}$ is a finite collection of Symbolic LSCs. As for plain ULSCs, the semantics of a specification is defined through conjunction:

$$Pop, \gamma \models \mathcal{S} \iff \forall S \in \mathcal{S} : Pop, \gamma \models S.$$

Problem 5.26 (SymLSC-SAT) The satisfiability problem for Symbolic LSCs SymLSC-SAT is given a Symbolic LSC specification $\mathcal{S}$ and a finite set *Role*, to decide whether

$$\exists Pop \in \mathbb{P} : \exists \gamma \in \Sigma^\omega : Pop, \gamma \models \mathcal{S}.$$

■

Theorem 5.27 SymLSC-SAT is undecidable. ■

Proof 5.27

We outline how one can reduce the halting problem of a two-counter machine, which is known to be undecidable, to SymLSC-SAT. A two-counter machine (2CM) is a program (i.e. a finite list *prog*), that has two integer counters c_0, c_1 and uses the following statements:

- `init` is an *initialization* statement, that resets c_0 and c_1 to 0. There is only one `init` statement, located at line 0 of *prog*.
- `go to` l_1 `or` l_2, where l_1 and l_2 are line numbers, with $l_1, l_2 > 0$. Executions must jump (nondeterministically) at line l_1 or l_2.
- `halt` is a *halting* statement. There is only one `halt` statement. Its effect is to make the execution back to line zero, i.e. to the `init` statement.
- `inc` i, with $i = 0, 1$. Its effect is to increment counter c_i of one unit and goes on with the statement at the next program line.
- `dec` i decrements c_i and goes on with the statement at the next program line.
- `not` i. The execution goes on if $c_i \neq 0$. Otherwise, the execution stops.
- `zero` i. The execution continues if $c_i = 0$, otherwise, it stops.

The problem of deciding, given *prog*, if there is an execution that will eventually execute `halt` is undecidable. Remark that, if *prog* executes `halt`, then there are two bounds $k_0, k_1 \in \mathbb{N}$ such that $c_i < k_i$ $(i = 0, 1)$, during the whole execution.

By construction, it is easy to see that

1. determining whether there is an infinite execution that goes infinitely often through `init` is undecidable, too. Actually, the same finite execution, from `init` to `halt` can be iterated again and again.

2. if there is such an ever-looping execution, it also uses counter bounds k_0 and k_1.

In order to encode counter values with Symbolic LSCs, we use agent roles. In our case, $Role = \{cntr\}$. Every agent playing role *cntr* can assume three "values": -1 (meaning unused), 0 and 1. The value of counter $c_i (i = 0, 1)$ is the number of agents assuming value i. We also use a concrete instance, named "CPU", which is a "central processing unit". It executes sequentially the 2CM statements as prescribed by *prog* and sets the values of *cntr* agents. Agent *cntr* can receive four messages: "get", "unset", "set0" and "set1". The first one queries the value currently stored (-1, 0 or 1, thus). The three last messages set the value.

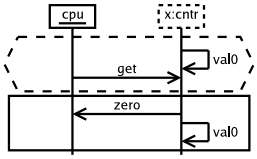


Figure 5.2: Getting x values

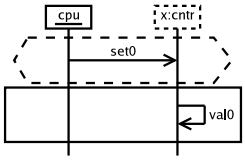


Figure 5.3: Setting x value

The LSC of Fig. 5.4 encodes the semantics of `init`: it sets c_1 and c_0 to 0, by ensuring that there are no *cntr* agents with values 1 or -1. Then, it proceeds to the next statement, which is at line number 1.

The CPU sends to itself the line number of the next statement to execute. If line i is a statement of the form `inc` 1, this line is translated to the LSC of Fig. 5.5. In this LSC, some agent in *cntr* is picked, the value of which is -1 (i.e. it does not belong to any counter), and sets its value to 1. Since all other agents do not take part in this protocol, their value is unchanged. Remark that the execution proceeds at the next line, i.e. $i + 1$.

The same approach is taken to translate the statement `dec` 1. This is illustrated by Fig. 5.6.

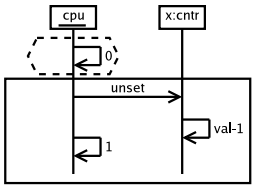


Figure 5.4: `init`

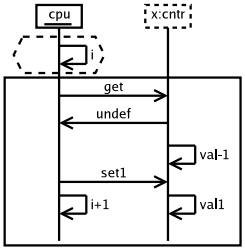


Figure 5.5: `inc 1`

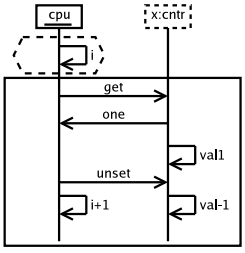


Figure 5.6: `dec 1`

Testing whether $c_0 = 0$ is illustrated by Fig. 5.7. The CPU retrieves the value of *all cntr* and checks that it is indeed either -1 or 1, i.e. nonzero. The encoding of $c_0 \neq 0$ is presented in Fig. 5.8. CPU simply finds one agent the value of which is 0. Thus, $c_0 \neq 0$, clearly.

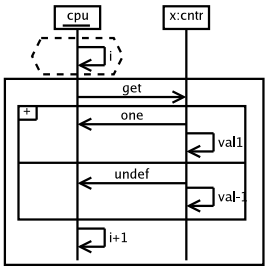


Figure 5.7: `zero 0`

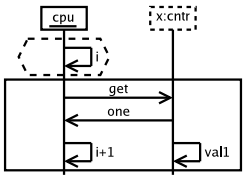


Figure 5.8: `not 0`

Finally, in Fig. 5.9, the LSC imposes that CPU executes `init` infinitely often.

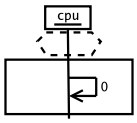


Figure 5.9: `halt` infinitely often

Thus,

1. all models of the specification execute `halt` infinitely often (Fig. 5.9);
2. all models of the specification simulate the 2CM.

□

5.5 Conclusion

In this chapter, we have introduced three extensions to the language of LSCs: conditions, real-time and symbolic instances. Conditions were part of the initial definition of the language, by Damm and Harel [42], while the two latter features have been introduced later on [75, 74, 119]. We formalized these extensions and have shown that our previous "simple" definitions of LSCs were

only slightly modified by their introduction. Our previous algorithms are also straightforwardly adapted to cope with them: conditions, expressed as fluents, do not require any change to verification and synthesis algorithms, whereas timed LSCs obliged us to adapt the translation of LSCs to automata to obtain a translation to timed automata. For symbolic instances, we showed that even the "simplest" problem, namely satisfiability, is undecidable.

The goal pursued in this chapter was to convince the reader that, although our previous results regarding expressiveness, succinctness and computational complexity only considered a restricted subset of LSCs, this subset could be extended to broaden its applicability. In our formalization, we had to depart slightly from the definitions given by Harel and Marelly of these extensions.

Regarding conditions, we added a distinction between ASAP and eventual conditions, in addition to the usual hot/cold condition. Hot conditions must be verified, while violating some cold condition results in the premature but successful termination of the chart. ASAP conditions are verified as soon as they are enabled (i.e. all their successors have been reached), whereas the execution can wait for some eventual condition to become true.

With respect to time, we obliged clock resets to be associated with events. Our formalization forbids also clocks to be shared between the prechart and the main chart. The former requirement is not very restrictive because most real specifications associate clock resets with events. For instance, all examples provided by Harel and Marelly do. The latter requirement can be dropped at the price of a heavier formalization and a semantics which is more difficult to comprehend.

Symbolic LSCs, as presented here, are very different from what was proposed by [74]. The two main differences are that we force quantifiers to be linearly ordered and that variable binding is performed "at once".

immediate binding In our approach, we bind all variables prior to executing the chart. Since the main motivation of Harel and Marelly was executability, they preferred to bind variables on demand. Namely, when some message must be sent to a symbolic variable that is not yet bound, this variable is resolved, according to its quantifier. Here, we try all possible bindings and keep matching ones.

ordered quantifiers In [74], quantifiers are not ordered, because symbolic variables are resolved on demand. This can lead to strange behaviours, because existential and universal quantifiers can be unordered. Thus, in one execution, they will be interpreted as $\forall x : R : \exists y : R' : (\ldots)$ whereas in another one, they will rather be $\exists y : R' : \forall x : R : (\ldots)$. These two sentences have very different meanings and we could not think of any example in which analyst would write such ambiguous statements on purpose. Therefore, we preferred to oblige analysts to linearly order quantifiers.

Chapter 6

Implementation

Contents

Look! It's moving. It's alive. It's alive... It's alive, it's moving, it's alive, it's alive, it's alive, it's alive, IT'S ALIVE!"

Dr. Henry Frankenstein, Frankenstein (1931)

6.1 Introduction

We have implemented a set of algorithms into a tool named REMoRDS (Requirements Engineering and Modeling of Reactive Distributed Systems). REMoRDS is written in Java. Analysts can describe inter-agent specifications using a textual language. The tool supports the following functionalities, as illustrated by the screenshot of Fig.6.1.

Syntax check. Checking that the specification is syntactically correct. This includes interface correctness: agents may only send/receive messages as specified in the structural part of the specification.

Animation. Executes the specification, according to Marelly and Harel's play-out algorithm.

Centralized Realizability Checking. This follows the algorithm presented in Section 4.5, without mercifulness. REMoRDS features an animator to provide feedback on the result.

Incomplete Distributed Synthesis. This is the algorithm presented in Section 4.6.2. If REMoRDS finds a distributed implementation for some agent, it generates an FSP (Finite State Process) description of this implementation that can be used in LTSA (Labeled Transition System Analyzer) [113, 60].

LTL formula generation. For every scenario in the specification, REMoRDS generates an LTL formula that can be inserted in an FSP model and model checked in LTSA.

We did not provide a graphical editor for LSCs, hence, analysts must translate their LSCs manually to our input language, called LPO (Labeled Partial Order). In the remainder of this chapter, we will go through a sample session of distributed reactive system analysis with REMoRDS.

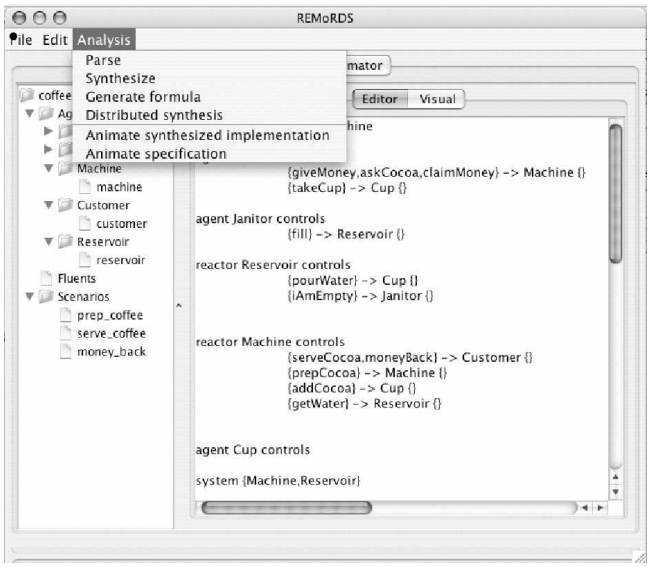


Figure 6.1: REMoRDS: Main functionalities

6.2 Writing a Model

Analysts first have to write a model of the reactive distributed system that they are planning to analyze. This model is made of three parts:

Structural part that presents the various agent classes, what events they send, to what other agents, what classes belong to the system and the list of agents that will be involved in the behavioral part. Every agent is assigned a class. This part corresponds to the structural view described in Sec.4.2.1.

Fluent specification that introduces the list of fluents used in the scenarios, with their initial values and their activating/deactivating events, see Sec.5.2.3.

Inter-Agent specification is made of a list of scenarios. A scenario is simply a labeled partial order, in which some distinguished locations constitute a prechart. A special event, `start` can be used in the prechart, to describe scenarios occurring only at the beginning of the run. `start` is an artificial event that is only fired once, at the beginning of every execution.

REMoRDS features a simple text editor to write models. This makes it possible to directly edit the model within the tool, save it and parse it.

REMoRDS's syntax checker is a standard compiler. It has been built using JavaCup [88] and JFLex [96]. We present in sec 6.2.1 the grammar for LPO, in EBNF (Extended Backus-Naur Form). The reader unfamiliar with BNF is referred to standard textbooks on compilers [3, 180].

Non-terminals are written between less-than (<) and greather-than (>) signs, while terminals are written between single quotes. EBNF meta-symbols are

- `+`, nonempty sequences,
- `*`, all sequences,
- `(`, `)`, for grouping expressions,
- `[`, `]`, optional expression.
- `{`, `..`,`}`, used in range expressions, `{x..y}` represents all terminals between terminals `x` and `y` according to an implicit order, which is clear from the context (natural numbers, characters).
- `<`, `>`, for surrounding nonterminals,
- `::=`, "reduces to" operator,
- `|`, choice between expressions,
- `/*`, `*/`, for surrounding comments. Everything appearing between `/*` and `*/` has no impact on the syntax definition.
- `'`, for surrounding terminals.

6.2.1 Extended Backus-Naur Form of LPO

```
/* Inter-Agent specification */
<iaspec>        ::= 'specification' <id>
                    <structure>
                    (<fluent>)*
                    (<ulsc>)+

<structure>     ::= (<agent>)+ <system> <pop>

/* Declaration of an agent class */
<class>         ::= 'agent' <id> <control>
                |   'reactor' <id> <control>

<control>       ::= 'controls' ( <msgidset> '->' <class-group> )*
<class-group>   ::= <classid> <classidset>
<msgidset>      ::= '{' <idlist> '}'
<classid>       ::= <id>
<classidset>    ::= '{' <idlist> '}'
```

```
/* Which agent classes belong to the system */
<system>        ::= 'system' <classidset>

<pop>           ::= 'population' (<instdecl>)+

<instdecl>      ::= <idlistne> ':' <id>

/* Declaration of fluents appearing in the specification */
<fluent>        ::= 'fluent' <id> '= '
                    '<' <eventsetne> ','
                        <eventsetne>
                    '>' initially <bool>


/* Declaration of a ULSC */
<ulsc>          ::= 'scenario' <id>
                    (<locdecl> | <exclusions>)+
                    <prechart>
                    [<restricts>]
<locdecl>       ::= <id> '(' <loclabel> ')' '>' '{' <idlist> '}'
<exclusions>    ::= <id> '#' <id>
<prechart>      ::= 'with' 'prechart' <idset>
<restricts>     ::= 'also' 'restricts' <eventset>
<loclabel>      ::= <event>
                |   'assert' <boolform>
                |   'start'

<boolform>      ::= <id>
                |   <boolform> '&' <boolform>
                |   <boolform> '|' <boolform>
                |   '~' <boolform>
                |   <bool>
                |   '(' <bool> ')'
<bool>          ::= 'true' | 'false'



/* sdr ! msg ? rcvr means
   that agent sdr sends msg to agent rcvr. */
<event>         ::= <agentid> '!' <id> '?' <agentid>
<agentid>       ::= <id>
<eventlistne>   ::= <event> (',' <event>)*
<eventlist>     ::= [ <eventlistne> ]

<eventset>      ::= '{' <eventlist> '}'
```

```
<eventsetne>    ::= '{' <eventlistne> '}'


/* Utility expressions (lexical) */
<char>          ::= {'A'..'Z'}
                |   {'a'..'z'}
                |   {'0'..'9'}
                |   '.'
                |   '+'
                |   '-'
                |   '_'
                |   '['
                |   ']'
<id>            ::= (<char>)+
<idlistne>      ::=  <id> (','<id>)*
<idlist>        ::= [ <idlistne> ]
```

6.2.2 Additional Rules

The grammar is ambiguous, because of boolean formulae. Priority is needed. Operators priority is `|`, `&`, `~`. Binary operators are set to be left-associative.

An event `s ! m ? r` is defined if there is an agent instance `s` of class `cs` and an agent instance `r` of class `cr` such that `m` is controlled by `cs` and exposed to `cr`, in the structural specification.

Well-formedness rules are provided and checked by REMoRDS.

- Agent classes must all have distinct names,
- Agent instances must all have distinct names,
- All agent classes referred to in the structural specification must be declared. Forward references are allowed.
- Fluents only use defined events,
- Scenario names must all be distinct,
- Location labels are built from declared fluents and defined events.
- In a location expression of the form `id` *label* `> { ` S `}`, all location id's contained in S have been declared higher up in the same scenario.
- In exclusion expressions of the form x `#` y, x and y are location id's declared higher up in the same scenario.

6.2.3 Example

```
specification coffee_machine

agent Customer controls
```

```
{giveMoney,askCocoa,claimMoney} -> Machine {}
{takeCup} -> Cup {}

agent Janitor controls
{fill} -> Reservoir {}

reactor Reservoir controls
{pourWater} -> Cup {}
{iAmEmpty} -> Janitor {}


reactor Machine controls
{serveCocoa,moneyBack} -> Customer {}
{prepCocoa} -> Machine {}
{addCocoa} -> Cup {}
{getWater} -> Reservoir {}


agent Cup controls

system {Machine,Reservoir}

population
machine : Machine
reservoir : Reservoir
janitor : Janitor
customer : Customer
cup : Cup

scenario money_back
1 (customer ! giveMoney ? machine) > {};
2 (customer ! claimMoney ? machine) > {};
3 (machine ! moneyBack ? customer) > {1,2};
with prechart {1,2}
also restricts {machine ! serveCocoa?customer}


scenario serve_coffee
1 (customer! giveMoney ?machine) > {};
2 (customer! askCocoa ?machine) > {};
3 (machine! prepCocoa ?machine) > {1,2};
4 (machine! serveCocoa ?customer) > {3};
with prechart {1,2}
also restricts {machine ! moneyBack ? customer}


scenario prep_coffee
```

```
1 (machine ! prepCocoa ? machine) > {};
2 (machine ! addCocoa ? cup) > {1};
3 (machine ! getWater ? reservoir)  > {2};
4 (reservoir ! pourWater ? cup) > {3};
5 (machine ! serveCocoa ? customer) > {4};
6 (customer ! takeCup ? cup) > {5};
with prechart {1}


scenario fill_up
1 (reservoir ! pourWater ? cup ) > {}
2 (reservoir ! pourWater ? cup ) > {1}
3 (reservoir ! iAmEmpty ? janitor) > {2}
4 (janitor ! fill ? reservoir) > {3}
with prechart {1,2}
```

6.2.4 Comments

The textual representation differs from the graphical and abstract representation in different aspects. First, interfaces are specified at class-level. Then, agent classes are instantiated to some finite population. In the version presented in this thesis, we supposed that agent instances were directly specified, without referring to their class, see Section. 4.2.1.

Second, a sub-class of agents has been introduced, named *reactor*. A reactor is an agent who will not spontaneously act. It will only perform some event when a scenario of the specification requires it to do so. This type was introduced to reduce the number of runs generated for system agents that are not expected to act on their own. For instance, `Reservoir` is a reactor, in Sec. 6.2.3, because it is not expected to pour water or call the janitor to fill it up spontaneously. It will only perform these events in response to proper requests: `getWater` and two `pourWater` in a row.

Third, we do not support choice LPO, for some historical reasons. We started developing REMoRDS in 2003. At that time, we picked another approach to represent choice in LPOs, namely the addition of an *exclusion relation* as in event structures [141]. An exclusion relation $\#$ on an LPO $\langle L, \lambda, \leq, A\rangle$ is a symmetric and irreflexive relation on L such that $\#$ is inherited along $\leq$:

$$l_1 \# l_2 \text{ and } l_3 \geq l_1 \text{ and } l_4 \geq l_2 \implies l_3 \# l_4.$$

We simply sketch how the transition relation on cuts is adapted: $c \xrightarrow{e} c'$ iff there is some e-labeled location l not in c such that $c' = c \cup \{l\} \cup \{l' | l' \# l\}$. Thus, when a location l is reached, we ensure that no excluded location will be reached in the current execution, by removing all excluded locations from the locations that may be executed in the run $(L \setminus c')$. It is easy to see that CLPOs are exponentially more succinct than LPOs with exclusion relation. This is akin to the relation between directed acyclic graphs and trees. The algorithms

presented in Chapter 4 work on CLPOs and their complexity is thus more robust to the input language. This is the reason why CLPOs were presented in this thesis instead of LPOs with exclusion relations.

Fourth, we instantiated our generic definition of ULSCs with conditions to fluents, as explained in Section 5.2. LPO does not support temperatures on any object (message, location, condition) and does not make it possible to distinguish between ASAP and eventual conditions. Therefore, all objects are hot in LPO and conditions are ASAP.

6.3 Animation

When the analyst's model has been written, translated to LPO, loaded in REMoRDS and succesfully parsed, its structure is displayed in the left pane of the editor area, see Fig.6.1. Agents are grouped by classes. Scenarios and fluents are also displayed. The analyst can easily browse the model.

When a scenario is selected in this browser, one can visualize it as a ULSC in the "Visualizer" tab of the editor area, as shown in Fig.6.2. Since there are scenarios in LPO without any equivalent ULSC, the visualizer may display incorrect ULSC. However, in most cases, the analyzer will first have modelled the system of interest with ULSCs on papers and afterwards, have translated it to LPO. In that case, the displayed ULSC is correct, i.e. is equivalent to the one from which the LPO was derived.

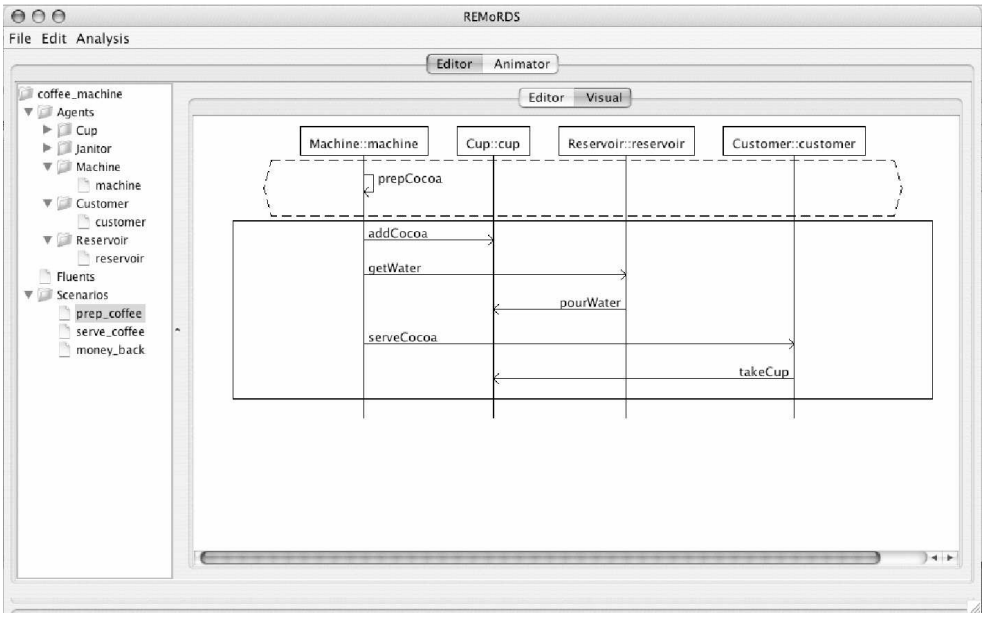


Figure 6.2: REMoRDS: Visualizer

When the analysts is happy with his model, he can start analysing it. A first step in analysis is executing the model. We implemented Harel and Marelly's play-out algorithm in REMoRDS. Our animator is pretty straightforward. Its user interface is illustrated in Fig.6.3. It is made of two main areas: a "history

area" and a "current state area". The history area displays the current trace while the current state area shows the partial executions of all ULSCs.

The history area displays the animation history in two modes. On the left-hand side, the execution tree is displayed. The nodes of this tree are labeled by the states through which the execution passed and the various branches that were followed. The current state is underlined. When a state that was already encountered is visited again, the execution "jumps up" in the tree to that state and all nodes that are labeled by this state are underlined. When a choice is made, a new branch of the tree is created, along which the execution goes on. On the right-hand side, the current trace, i.e. the path from the root of the tree to the current node, is shown as an MSC-like drawing. Two points are noteworthy. First, this MSC does not follow the usual semantics of MSC, as the vertical position of two events determine whether they should be ordered. Second, stuttering, i.e. "do nothing" events are removed, to avoid cluttering the figure.

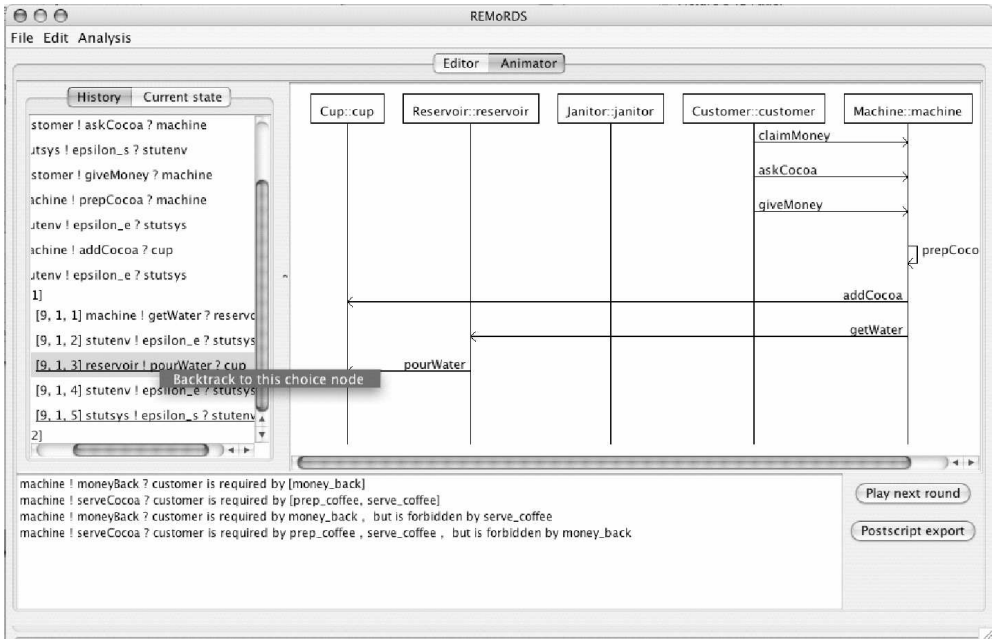


Figure 6.3: REMoRDS: Animator

A lower pane also shows what events are required in the current state. It also lists events that must be delayed in the current state, i.e. those that are both required and forbidden. This simple diagnoser therefore hints at possibly conflicting scenarios.

In order to exploit this information, the analyst can visualize all the cuts reached in the current state, by going to the "current state area". There, all the cuts linearized by some suffix of the current execution are listed. When a cut is selected, the ULSC is drawn in the right-hand side panel and the events that belong to the cut, i.e. those that have already occurred in the current execution, are highlighted, as illustrated by Fig.6.4.

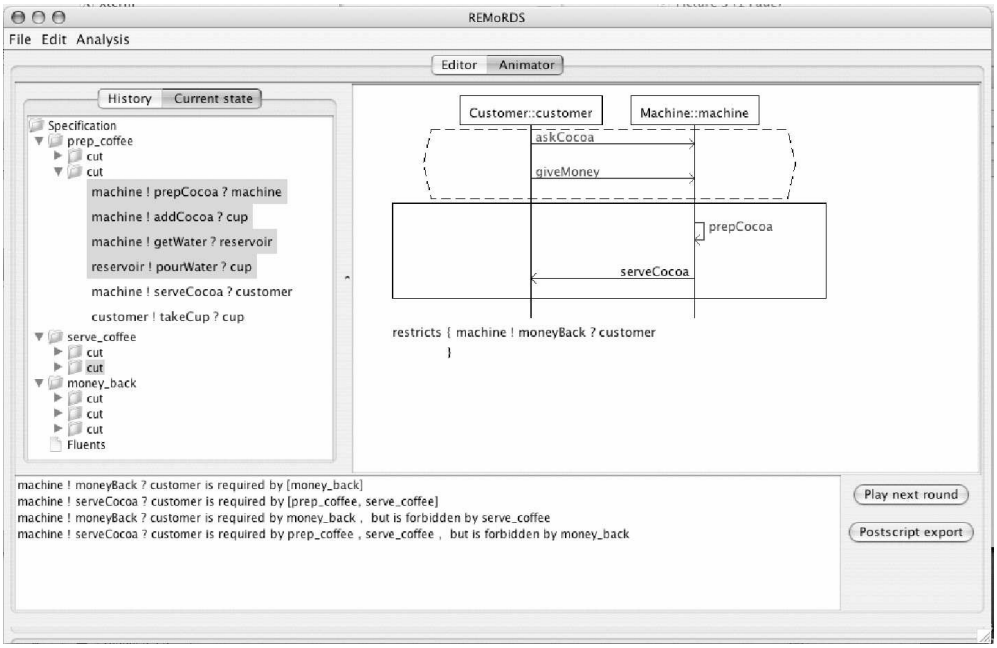


Figure 6.4: REMoRDS: Visualizing a state

If the analyst wants to explore alternative paths, he can double-click any node in the execution tree and jump to this node.

The interaction loop is simple: at every step, REMoRDS lists all environment-controlled events that are not forbidden in the current state. The analyst is then asked to choose from one of them and to tell whether he would like the system to answer immediately to this stimulus or to input another environment-controlled event beforehand. If the system is allowed to answer, he picks some event that is required and not forbidden, performs it and then waits for new input from the analyst.

Clearly, our animator is very primitive. It lacks many features that can be found in the play-out engine, such as linking the animator with a mock-up GUI of the future system [75], or in the AlbertII animation engine, like cooperative animation [83]. However, it is already very helpful to explain counter-examples and experiment with specifications.

6.4 Centralized Realizability Checking

When the analyst has written a syntactically correct model and has played around with it to remove some possible flaws, he can use the "synthesis" algorithm to ensure that his specification is realizable.

The "synthesis" functionality implements the algorithm solving COAD presented in Section 4.23. Recall that this algorithm works in two steps. First, it builds a one-pair Streett automaton from which a three-color parity game graph is constructed. Second, the parity game is solved. We implemented the first part in Java. For the second part, we call an implementation in C of Vöge

and Jurdzinski's algorithm for solving parity games [172]. The result is then read back to extract a strategy.

Since the strategy cannot be displayed, because it is too large to be readable, we have implemented an animator to provide the analyst with feedback. Remember that the 2-player game underlying the problem of implementability is determined. Thus, either the system has a strategy implementing correctly the specification or its environment has a strategy, called here "sabotage plan", against which the system will *never* be able to respect the specification. Thus, whatever the system does against a sabotage plan, the resulting execution will violate the condition of Def. 4.8.

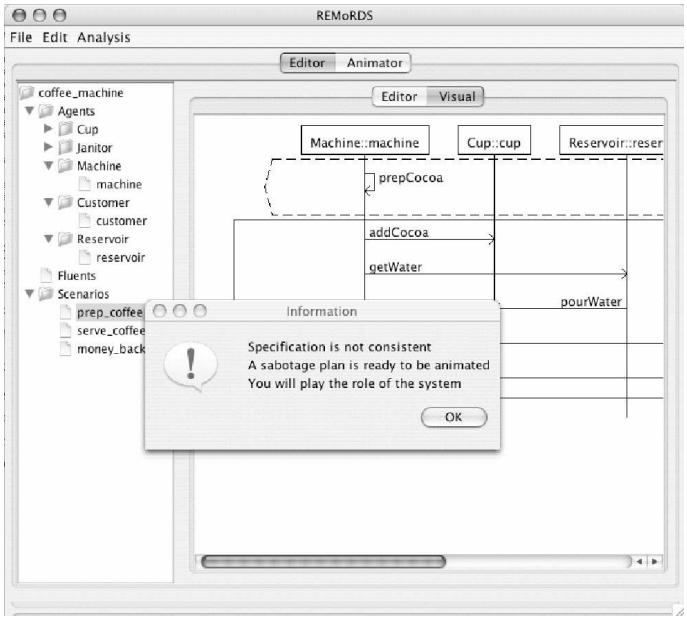


Figure 6.5: REMoRDS: Inconsistent specification

In the former case, the animator assumes the role of the system under development while the analyst plays the environment. The analyst can choose among environment-controlled events and watch the system respond according to the synthesized strategy. If the specification is *not* implementable, a strategy *for the environment* is synthesized, see Fig.6.5. In the animation process, roles are now swapped: the analyst plays the future system and the animator assumes the role of its environment. Then, the animator demonstrates that the specification is not consistent by letting the analyst choose system actions and providing answers driving the system into specification violation.

Our animator has been presented in Sec.6.3. Now, it shall be noted that the result of animation is not a single trace of events but a tree. This is the essential difference between model checking of linear time properties and a game-based approach, like ours. A counter-example is not a single trace of events, but a tree, the branches of which correspond to choices of the human operator. Against every such choice, the animator replies in such a way that the specification will always be met (implementable specification) or that will force the operator

to eventually violate it (sabotage plan). Our animator supports backtracking; by clicking on a tree node, the operator can go back to this choice node, make another choice and creating another sub-tree. Identical nodes, which are roots of isomorphic sub-trees, are identified. This avoids generating very large execution trees.

REMoRDS' synthesizer also works in a progressive fashion. It does not attempt to build the whole game graph immediately but rather constructs an abstract version of it. It explores the game graph up to a user-specified depth and width. Then, all states on the frontier of this zone, which have thus not yet been expanded are considered correct for the system. If a sabotage plan can be found on this smaller graph, the whole implementation is not realizable. Furthermore, the sabotage plan is a correct counter-example. The synthesizer also tries the other combination, by letting the system *lose* on frontier states. Then, if an implementation can be synthesized on the abstract game graph, it is a correct implementation of the whole specification.

In the case of CTAS, we were unable to synthesize an implementation on a pentium III 800Mhz computer with 256Mb RAM. REMoRDS ran out of memory after about 15 minutes and consuming more than 200 Mb of memory. In practice, we have found that REMoRDS can deal with specifications of less than nine scenarios, yielding Streett automata of around 10,000 states and 400,000 transitions. When faced with larger specifications, REMoRDS runs out of memory.

6.5 Incomplete Distributed Synthesis

The incomplete distributed synthesis algorithm has also been implemented in REMoRDS. The analyst needs to specify the name of the agent to be synthesized, in addition to an intra-object specification.

Remember that incomplete distributed synthesis resorts to game solving for checking that the SDI (Standard Distributed Implementation) is correct or can be refined to a correct implementation. Again, we use the algorithm of Vöge and Jurdzinski as a toolbox. Analysts can also be provided with feedback using REMoRDS' animator.

If a correct distributed implementation is built, REMoRDS translates it to *dot* and FSP. The former allows one to visualize small implementations as an automaton graph, automatically layed out using Bell Labs's Graphviz software [50]. The latter makes a process that can readily be inserted in an FSP model and analyzed in LTSA [113]. Thus, a synthesized model of a component can be taken and plugged in the design model of the system.

6.6 LTL Formula Generation

Once the analyst has obtained a satisfactory inter-agent specification, which is implementable, he can turn to the problem of designing an intra-agent model. In order to do so, we rely on the LTSA (Labeled Transition System Analyzer) that

has been developed at Imperial College [113]. The analyst writes his intra-agent specification as an FSP (Finite State Process), with the additional constraint that communication events must be expressed as `sender.event.receiver`. Then, REMoRDS can be used to generate LTL formulae from the inter-agent specification. Those formulae can be input to LTSA and model checked, to ensure that the intra-agent model complies with the initial inter-agent specification.

Fig.6.6 shows a window containing the LTL formula translated from the selected ULSC which pops up. Its content can then be copied-and-pasted to LTSA. Our translation takes advantage of the fact that conditions containing arbitrary boolean combinations of fluents can be used in ULSCs. Of course, one has to ensure that all fluents defined in the LPO file are also defined in the FSP model. REMoRDS uses the formula presented in Th. 3.40. We chose to resort to the theoretically inefficient formula because it gives very good practical results. The polynomial translation is more intricate and is not of a great practical help, for its power is high ($\mathcal{O}(n^5)$). The translation process is rather straightforward and relies on a depth-first traversal of the cut transition system. For performance matters, REMoRDS ensures that it never generates twice the same sub-formula in the same execution. The formula corresponding to a cut is cached and reused later if it needs to appear in another context.

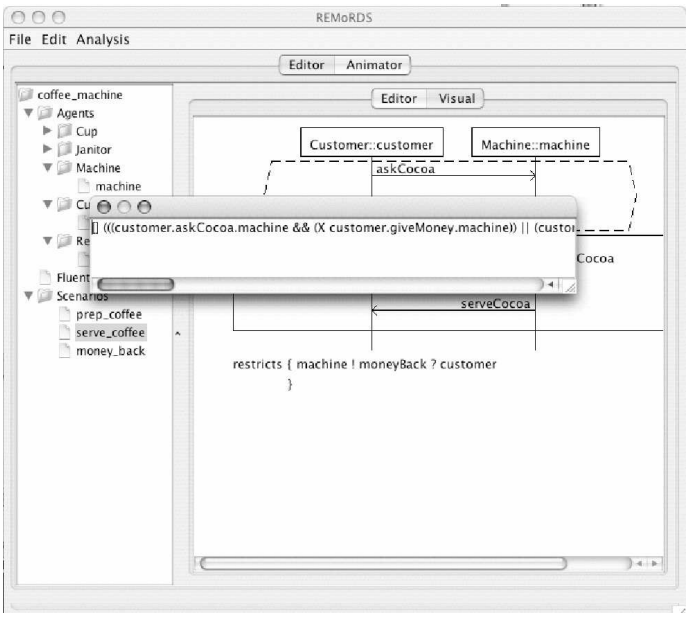


Figure 6.6: REMoRDS: translating ULSC to LTL

It takes less than a second to generate the formula of every scenario of our CTAS case study. Of course, this is but the simplest job in the verification process: an external model checker needs to verify it afterwards, which is highly resource-consuming.

Conclusion

Always look on the bright side of life ...

Monty Python's Life of Brian

This thesis studied a novel approach to engineering reactive distributed systems, described by Harel in [71]. In this approach, instead of manually deriving a design model from informal user requirements, those requirements are first translated to a formal inter-agent specification language, named Live Sequence Chart (LSC). This language is based on scenarios, which have proven their usefulness for communication with non-technical stakeholders [174]. Then, a design model, i.e. an intra-agent specification, is derived from this inter-agent specification. It is postulated that the translation of user requirements to LSC is a straightforward activity. This hypothesis is not irrelevant, as we demonstrated on a real-world case study, taken from NASA's Center TRACON Automation System [12]. Because scenarios are widely used to communicate user requirements [174], we can reasonably expect LSC to be applicable to many projects. The adoption of LSC could be sustained by the fact that LSC belongs to the sequence chart family, a family of languages the root of which is Message Sequence Chart, a widespread scenario-based language, standardized by ITU [166]. This family also contains UML's Interaction Diagrams, that were called Sequence Diagrams [130].

Since LSC is a fully formal language, it can be manipulated by computers. Therefore, it is natural to expect the translation from LSC to state machines to be automated. There are nowadays many algorithms deriving state machines from sequence charts models but most of these algorithms are poorly specified. The relationship between the behaviour of the synthesized system and the behaviour specified by its inter-agent model is often left implicit. In most cases, the only relationship is the trivial fact that the system has been derived from the specification using the particular algorithm presented by its authors. We payed special attention to following a neater approach. LSCs were formally defined and we also precisely defined what, in our opinion, is an inter-agent and an intra-agent specification. These models had to render the intuitive properties of our systems of interested, namely distribution and openness. We also spent much effort to come up with a proper notion of "correct implementation". Our definition amounts to saying that "an intra-agent model correctly implements an inter-agent specification, if, whenever this model is deployed in environments respecting the assumptions stated in the inter-agent specification, the specification is met." This approach, that separates responsibilities from the environment from system responsibilities as long been advocated by Jackson [66, 187, 91] and is now widely recognized as a best practice of Requirements Engineering. Using LSC, one can transparently specify hypothesis on the environment and guarantees that must be probided by the future system. Once we had a formal definition of what a correct implementation is, we could also

precisely specify the various problems implied by the transformational approach of Fig.4. In this thesis, we defined three classes of problems:

Analysis problems are related to assessing the soundness of manual transformations of inter-agent specifications. We considered two types of such problems: use case checking and refinement. The former verifies that the combination of all constraints put by the inter-agent specification on the future system behaviour still allow a certain execution. The latter checks that a transformation of the specification still preserves all linear-time properties of the original specification.

Verification problems are related to checking that a certain intra-agent specification complies with a given inter-agent specification. We presented several variants of these problems, depending on whether the system is open or not, and distributed or not.

Synthesis problems are related to the derivation of an intra-agent specification from an inter-agent specification. We considered two types of problems: centralized synthesis, which is to derive a monolythic system made of a single component, and distributed synthesis, which builds a system of several components, communicating through described interfaces.

Specifying more clearly those problems puts us in a more comfortable position. First, we could devise algorithms solving these problems, prove them to be correct, analyse their efficiency and discuss their optimality. Problems themselves become first-class objects of study. Their difficulty can be assessed and their definition can be criticised.

Name	Open	Distributed	Complexity
use case checking	n/a	n/a	PSPACE-complete
refinement	n/a	n/a	PSPACE-complete
verification	no	no	coNP-complete
verification	yes	no	PSPACE-complete
verification	no	yes	PSPACE-complete
verification	yes	yes	PSPACE-complete
synthesis	no	no	PSPACE-complete
synthesis	yes	no	EXPTIME-complete
synthesis	yes	yes	undecidable

Table 6.1: Complexity of problems

We gave algorithmic solutions to all these problems and have shown them to be intractable, according to the theory of computational complexity, see Tab. 6.1. All our algorithms were optimal. The most interesting problem in the above list, namely distributed synthesis, is undecidable. This means that there exists no algorithm which can derive a distributed implementation from an inter-agent specification, if such an implementation exists, or refuse to do so, if it does not exist.

Having a formal specification of our problems gave us two more advantages. First, we could discuss *incomplete solutions*. That is, algorithms that are able to solve *partially* the problem. Using the problem specification, we can clearly tell what one can expect from these partial solutions. For instance, we provided a sound distributed synthesis algorithm. When this algorithm succeeds in building a distributed implementation, the user is guaranteed that a correct solution to the problem has been found. Failure to synthesise an implementation might be a false negative: a solution might exist even though the algorithm did not find one.

Second, we could discuss the *problem definition* itself. We proposed to slightly modify the definition of synthesis to take some quality criterion into account, which we called *mercifulness*. We adapted our synthesis algorithm to deal with mercifulness.

In Chapter 6, we proposed an implementation of several algorithms related to the various problems presented above. This implementation, named REMoRDS, makes it possible to specify a reactive distributed system using LPO, a textual version of LSC, synthesise a centralized implementation from it and generate temporal logic formulae that can be input in LTSA, a model checker for Finite State Processes [113].

We investigated the cost of extending LSC with conditions, real-time and symbolic instances. Conditions can be introduced at no cost: all problems remain in the same complexity class as before. Real-time is introduced almost at no cost: all problems remain in the same class, except verification of centralized and closed systems, which becomes PSPACE-complete. Adding symbolic instances to LSC introduces first-order quantification on agents and makes satisfiability undecidable. As a corollary, one can easily prove that all other problems are also undecidable. Our simple formalization is nicely extended when those features are introduced. We thus provide a neat formalization of conditions, real-time and symbolic instances. Our approach is purely declarative and much shorter than the operational semantics described in the papers that originally introduced them [75, 74, 119, 42].

Our complexity results are mostly negative and may thus sound as bad news. Such results are certainly unexpected for most practitioners, because, as described in Sec. 3.3.2, LSC is a rather inexpressive language. By having sacrificed most of Temporal Logic expressiveness, one would have hoped to get some efficiency improvement in return. The results are pretty sad: even when using LSCs, all problems are still too hard. One would be very much tempted to state that moving "from play-in scenarios to code is an unachievable dream". As a concluding remark, we explain that such a remark needs to be mitigated.

First, this thesis puts forward a few positive results as well. There are two problems which are significantly easier on LSC than on LTL. First, centralized synthesis is EXPTIME-complete on LSC and 2EXPTIME-complete on LTL [135]. Since EXPTIME is strictly included in 2EXPTIME, this represents a great improvement on efficiency. Well, of course, this is a purely theoretic improvement, as EXPTIME is far beyond the reach of practical algorithms. Second, closed centralized verification is only coNP-complete on LSC and is PSPACE-

complete on LTL. This is good news, even practically. Indeed, there is an immediate sub-class of LSC for which verification can be done in polynomial time (resp. linear time): the class of LSC with *polynomially many* (resp. *linearly many*) linearizations. This class might look artificial, but actually most real-world scenario-based specifications tend to specify a total or quasi-total ordering among their events.

Then, the undecidability result obtained does not mean that this line of research shall be abandoned. Such results obliges one not to set too high expectations. The overall transformational approach outlined by Harel (see Fig.4) still makes sense and research in this direction shall be pursued. We showed that it was undecidable whether a specification is consistent, i.e. essentially can be implemented and deployed in adverse environments without running into deadlocks. The same problem is found with all programming languages: it is undecidable whether they contain bugs and yet, millions of programs are compiled and used everyday around the globe. The same idea transposes to model transformation: semantic-preserving transformations from specification-level models to design-level models would be greatly useful to improve current engineering practice. Such transformations would of course also transfer problems from the specification level to the design level, just as usual compilers transform problems in source code into executable bugs. Compilation is thus a first possible track to follow to achieve the scenario-based transformational software engineering dream.

Prototyping with scenarios has also a proven track record. Marelly and Harel's work on play-in and play-out of scenarios [75] has raised great hopes for fast prototyping of systems. This prototype could even be used as a final implementation, thus removing the need for programming the system. Simple scenarios would thus be grafted on an existing application, with simple functions, in order to provide it with a reactive behaviour, this behaviour being enacted by a universal interpreter as the play-out engine.

Index

Bibliography

[1] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 14(4):1–60, October 1992.

[2] Martín Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable specifications of reactive systems. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*, volume 372 of *Lect. Notes in Computer Science*. Springer, 1989.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. 1985.

[4] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[6] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of Message Sequence Charts. In *Proceedings of 22nd International Conference on Software Engineering*, pages 304–313, 2000.

[7] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata: a determinizable class of timed automata. *Theoretical Computer Science*, 211(1-2):253–273, January 1999. Previously published in Proc. of the Sixth Conference on Computer-Aided Verification (CAV), LNCS 818, pp. 11-13, 1994.

[8] Rajeev Alur and Radu Grosu. Shared Variable Interaction Diagrams. In *Proc. of ASE'01, the 16th IEEE International Conference on Automated Software Engineering*, San Diego, USA, November 2001. IEEE.

[9] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM (JACM)*, 49(5):672–713, 2002.

[10] Rajeev Alur, Gerard J. Holzmann, and Doron Peled. An analyser for mesage sequence charts. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 35–48. Springer-Verlag, 1996.

[11] Rajeev Alur and Mihalis Yannakakis. Model Checking of Message Sequence Charts. In *CONCUR'99: Concurrency Theory, Tenth International Conference*, number 1664 in LNCS, pages 114–129. Springer-Verlag, 1999.

[12] NASA AMES. Overview of center tracon control system. `http://ctas.arc.nasa.gov`.

[13] Daniel Amyot and Armin Eberlein. An Evaluation of Scenario Notations for Telecommunication Systems Development. In *Proc. of 9th Int. Conference on Telecommunication Systems (9ICTS)*, Dallas, USA, March 2001.

[14] Daniel Amyot and Armin Eberlein. An evaluation of scenarion notations for telecommunication systems development. *Telecommunications Systems Journal*, 24(1):61–94, September 2003.

[15] Marco Antoniotti. *Synthesis and verification of discrete controllers for robotics and manufacturing devices with temporal logic and the Control-D system*. PhD thesis, New York University, New York, 1995.

[16] Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic controller synthesis for discrete and timed systems. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid System II*, volume 999 of *Lect. Notes in Computer Science*. Springer, 1995.

[17] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. In *Proc. System Structure and Control*. Elsevier, 1998.

[18] Hanêne Ben-Abdallah and Stefan Leue. Syntactic Detection of Process Divergence and Nonlocal Choice in Message Sequence Charts. In E. Brinksma, editor, *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, Third International Workshop, TACAS'97*, number 1217 in LNCS, pages 259–274, Enschede, The Netherlands, 1997. Springer-Verlag.

[19] Armin Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, volume 1579 of *Lect. Notes in Computer Science*. Springer-Verlag, 1999.

[20] Alan W. Biermann and Ramachandran Krishnaswamy. Constructing Programs from Example Computations. *IEEE Transactions on Software Engineering (TSE)*, SE-2(3):141–153, September 1976.

[21] Jürgen Bohn, Werner Damm, Jochen Klose, Adam Moik, and Hartmut Wittke. Modeling and validating train system applications using statemate and live sequence charts. In H. Ehrig, B. J. Krämer, and A. Ertas, editors, *Proceedings of the Conference on Integrated Design and Process Technology (IDPT2002)*. Society for Design and Process Science, 2002.

[22] Bernard Boigelot, Sébastien Jodogne, and Pierre Wolper. On the Use of Weak Automata for Deciding Linear Arithmetic with Integer and Real

Variables. In *Proc.International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, pages 611–625, Sienna, June 2001. Springer-Verlag.

[23] Yves Bontemps. Automated Verification of State-based Specifications Against Scenarios (A Step towards Relating Inter-Object to Intra-Object Specifications). Master's thesis, University of Namur, Namur, Belgium, June 2001.

[24] Yves Bontemps. Realizability of scenario-based specifications. Diplôme d'études approfondies, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique (University of Namur, Computer Science Dept), Namur, Belgium, September 2003.

[25] Yves Bontemps. On the semantics of uml 2.0 interaction diagram. Technical report, University of Namur, Institut d'Informatique, 2004.

[26] Yves Bontemps, Patrick Heymans, and Hillel Kugler. Applying LSCs to the specification of an air traffic control system. In Sebastian Uchitel and Francis Bordeleau, editors, *Proc. of the 2nd Int. Workshop on "Scenarios and State Machines: Models, Algorithms and Tools" (SCESM'03), at the 25th Int. Conf. on Soft. Eng. (ICSE'03)*, Portland, OR, USA, May 2003. IEEE. available at `http://www.info.fundp.ac.be/~ybo`.

[27] Yves Bontemps, Pierre-Yves Schobbens, and Christof Löding. Synthesizing open reactive systems from scenario-based specifications. *Fundamenta Informaticae*, XX:1–31, 2004.

[28] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[29] J.R. Büchi and Lawrence H. Landweber. Solving sequential conditions finite-state strategies. *Trans. Ameri. Math. Soc.*, 138:295–311, 1969.

[30] R.J.A Buhr. Use case maps as architectural entities for complex systems. *IEEE Transactions on Software Engineering*, 24(2):1131–1155, December 1998.

[31] R.J.A Buhr and R.S. Casselman. *Use case maps for object-oriented systems.* Prentice Hall, 1996. `http://citeseer.nj.nec.com/buhr96use.html`.

[32] Annette Bunker and Ganesh Gopalakrishnan. Using Live Sequence Charts for Hardware Protocol Specification and Compliance Verification. In *IEEE International High Level Design Validation and Test Workshop.* IEEE Computer Society Press, November 2001.

[33] Annette Bunker and Ganesh Gopalakrishnan. Verifying a Virtual Component Interface-based PCI Bus Wrapper Using an LSC-Based Specification. Technical Report UUCS-02-004, University of Utah, School of Computing, 2002.

[34] Cristian Calude and Juraj Hromkovič. *Complexity: A Language-Theoretic Point of View*, chapter 1, pages 1–60. Volume 2 of Rozenberg and Salomaa [142], 1997. ISBN 3-540-61486-9.

[35] María Victoria e Cengarle and Alexander Knapp. Uml 2.0 interactions: Semantics and refinement. In Jan Jürjens, Eduardo B. Fernandez, Robert France, and Bernhard Rumpe, editors, *Proc. of 3rd Int. Wsh. Critical Systems Development with UML (CSDUML'04, Proceedings)*, pages 85–99, 2004.

[36] Dexter C. Chandra, Ashok K.and Kozen and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, January 1981.

[37] A. Church. Logic, arithmetic and automata. In *Proc. of Intern. Cong. Math*, pages 23–35, 1963.

[38] Edmund M. Clarke and Bernd-Holger Schlingloff. Model checking. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 24, pages 1635–1790. MIT Press, North-Holland, 2001. ISBN 0-444-50813-9.

[39] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[40] J.M.H Cobben, A. Engels, Sjouke Mauw, and Michel Reniers, A. *Formal Semantics of Message Sequence Charts (ITU-T Recommendation Z.120 Annex B)*. International Telecommunication Union, Eindhoven, The Netherlands, April 1998. `http://www.itu.int`.

[41] Stephen A. Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, Ohio, 1971.

[42] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[43] Luca de Alfaro, Thomas A. Henzinger, and Marieëlle Stoelinga. Timed interfaces. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Proc. of EMSOFT 2002, Second International Workshop on Embedded Software*, volume 2491 of *Lect. Notes in Computer Science*, pages 108–122, Grenoble, France, October 2002. Springer.

[44] Jules Desharnais, Marc Frappier, Ridha Khédri, and Ali Mili. Integration of sequential scenarios. *IEEE Transactions on Software Engineering*, 24(9):695–708, September 1998.

[45] I. Diethelm, L. Geiger, T. Maier, and A. Zündorf. Turning collaboration diagram strips into storycharts. In *Proc. of "Scenarios and State-Machines: models, algorithms and tools" (SCESM) workshop of the 24th Int. Conf. on Software Engineering (ICSE 2002)*, Orlando, FL, May 2002. ACM. `http://www.cs.tut.fi/~tsysta/ICSE/papers/`.

[46] R. Geoff Dromey. *Behavior Trees.* Griffith University, 2003.

[47] R. Geoff Dromey. From requirements to design: Formalizing the key steps. In *Proc. of IEEE International Conference on Software Engineering and Formal Methods*, pages 2–11, Brisbane, Australia, September 2003. IEEE.

[48] R. Geoff Dromey. Genetic design: Amplifying our ability to deal with requirements complexity. In Stefan Leue and Traja Systä, editors, *Proc. of Dagstuhl Seminar on "Scenarios: Models, Transformations and Tools"*, Lect. Notes in Computer Science. Springer, 2005. To appear.

[49] H.-D. Ehrich, M. Kollmann, and R. Pinger. Checking Object System Designs Incrementally. *Journal of Universal Computer Science*, 9(2):106–119, February 2003. `http://www.jucs.org/jucs_9_2/checking_object_system_designs`.

[50] John Ellson, Emden Gansner, Yehuda Koren, Eleftherios Koutsofios, John Mocenigo, Stephen North, Gordon Woodhull, David Dobkin, Vladimir Alexiev, Bruce Lilly, Jeroen Scheerder, Daniel Richard G., and Glen Low. Graphviz - graph visualization software. Software. `http://www.graphviz.org`.

[51] E. Allen Emerson. Temporal and modal logic. In van Leeuwen [170], chapter 16, pages 997–1072. ISBN 0-262-72015-9 (Second Printing, 1998).

[52] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, December 1982.

[53] Andre Engels. Design Decisions on Data and Guards in MSC2000. In S. Graf, C. Jard, and Y. Lahav, editors, *Proc. of SAM2000: 2nd Workshop on SDL and MSC*, pages 33–46, Col de Porte, Grenoble, June 2000. SDL Forum Society on SDL and MSC.

[54] Kousha Etessami. A note on a question of peled and wilke regarding stutter-invariant LTL. *Information Processing Letters*, 75(6):261–263, 2000.

[55] Bernd Finkbeiner and Ingolf Heiko Krüger. Using message sequence charts for component-based formal verification. In *Proc. of OOPSLA 2001 Workshop on Specification and Verification of Component-Based Systems*, Tampa Bay, FL, USA, October 2001.

[56] FUJABA : From UML to Java And BAck Again (University of Padeborn). http://www.uni-paderborn.de/cs/ag-schaefer/Lehre/PG/Fujaba/fujaba.html.

[57] Paul Gastin, Benjamin Lerman, and Marc Zeitoun. Distributed games and distributed control for asynchronous sytems. In *Proc. of Latin American Theoretical Informatics (LATIN'04)*, April 2004.

[58] Thomas Gehrke, Michaela Huhn, Peter Niebert, Arend Rensink, and Heike Wehreim. A Process Algebra Semantics for Message Sequence Charts Including Conditions. In König and Lagerdörf, editors, *Proc. of the 8th GI/ITG Workshop Formale Beschreibungstechniken für verteilte Systeme (FBT '98) (Formal Description Techniques for Distributed Systems)*, pages 185–196. Shaker, 1998.

[59] Blaise Genest, Marius Minea, Anca Muscholl, and Doron Peled. Specifying and verifying partial order properties using template msc. In Igor Walukiewicz, editor, *Proc. of Foundations of Software Science and Computation Structure (FoSSaCS)*, volume 2987 of *Lect. Notes in Computer Science*, pages 195–210, Barcelona, March 2004. Springer.

[60] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. In *Proc. of the 4th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, Helsinki, Finland, September 2003.

[61] Martin Glinz. An Integrated Formal Model of Scenarios Based on Statecharts. In *Proceedings of ESEC'95 - 5th European Software Engineering Using Scenarios*, pages 254–271, Berlin, 1995. Springer-Verlag.

[62] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, 1996. Foreword By-Pierre Wolper.

[63] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. volume 2500 of *Lect. Notes in Computer Science*. Springer, November 2002.

[64] P. Graubmann, E. Rudolph, and J. Grabowski. Towards a petri net based semantics definition of message sequence charts. In O. Faergemand and A. Sarma, editors, *SDL'93: Using Objects*. Elsevier, 1993.

[65] Radu Grosu, Ingolf Krueger, and Thomas Stauner. Hybrid Sequence Charts. In *ISORC'2K, the 3rd IEEE International Symposium of Object-Oriented Real-Time Distributed Systems*, pages 104–111. IEEE, 2000.

[66] Carl Gunter, A., Elsa Gunter, L., Michael Jackson, and Pamela Zave. A reference model for requirements and specification. *IEEE Software*, 17(3):37–43, May/June 2000.

[67] Elsa Gunter, Anca Muscholl, and Doron Peled. Compositional message sequence charts. In *Proc. of TACAS'01 (Tools for Automated Construction and Analysis of Systems)*, volume 2031 of *Lect. Notes in Computer Science*, pages 496–511, 2001.

[68] Aidan Harding, Mark Ryan, and Pierre-Yves Schobbens. A new algorithm for strategy synthesis in ltl games. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS'05)*, Lect. Notes in Computer Science, Edinburgh, April 2005. Springer. To appear.

[69] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-Out of Behavioral Requirements. In *Proc. 4^{th} Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'02), Portland, Oregon*, 2002. To appear. Also available as Tech. Report MCS02-08, The Weizmann Institute of Science.

[70] David Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[71] David Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, January 2001.

[72] David Harel and Hillel Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *International Journal of Foundations of Computer Science*, 13(1):5–51, February 2002. (Preliminary version appeared in, *Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000)*, LNCS 2088, July 2000.).

[73] David Harel and Rami Marelly. Capturing and Analyzing Behavioral Requirements: The Play-In/Play-Out Approach. Technical Report MCS01-15, The Weizmann Institute of Science, Faculty of Mathematics and Computer Science, Rehovot, Israel, September 2001.

[74] David Harel and Rami Marelly. Playing with time: On the specification and execution of time-enriched LSCs. In *Proc. 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02)*, Fort Worth, TX, 2002.

[75] David Harel and Rami Marelly. *Come, let's play! Scenario-based programming using LSCs and the Play-engine.* Springer, 2003. ISBN 3-540-00787-3.

[76] David Harel and Amir Pnueli. On the development of reactive systems. In K.R. Apt, editor, *NATO ASI Series*, volume F-13, pages 477–498, New-York, January 1985. Springer.

[77] David Harel and Michael Politi. *Modeling Reactive Systems with Statecharts: the STATEMATE Approach.* McGraw-Hill, 1998. ISBN 0-070-26205-5.

[78] David Harel, Moshe Y. Vardi, and Orna Kupferman. On the complexity of verifying concurrent transition systems. *Information and Computation*, 173(2), March 2002.

[79] Loïc Hélouët and Claude Jard. Conditions for synthesis of communicating automata from HMSCs. In *Proc. of 5th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, Berlin, April 2000.

[80] Loïc Hélouët and Claude Jard. La manipulation formelle des scénarios (l'exemple des message sequence charts). In *Proc. of MSR'01 (Colloque Francophone sur la Modélisation des Systèmes Réactifs)*, 2001.

[81] Loïc Hélouët, Claude Jard, and Benoît Caillaud. An event structure semantics for message sequence charts. *Mathematical Structures in Computer Science (MSCS)*, 12:377–403, 2002.

[82] J. Henriksen, M. Mukund, K. Kumar, and P. Thiagarajan. On message sequence graphs and finitely generated regular MSC languages. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming (ICALP'2000)*, volume 1853 of *LNCS*, pages 675–686. Springer-Verlag, 2000.

[83] Patrick Heymans. *Animating Albert II Specifications*. PhD thesis, University of Namur, 2001.

[84] S. Heymer. A non-interleaving semantics for msc. In *1st conference on SDL and MSCs (SAM98)*, Berlin, 1998.

[85] S. Heymer. A semantics for mscs based on petri nets components. In *2nd Conference on SDL and MSC (SAM2000)*, Grenoble, June 2000.

[86] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 2001.

[87] P. Hsia, J. Samuel, J. Gao, D. Kund, Y. Toyoshima, and C. Chen. Formal approach to scenario analysis. *IEEE journal*, pages 33–41, March 1994.

[88] Scott Hudson, Frank Flannery, and C. Scott Ananian. Java cup: Lalr parser generator for java(tm). Software, September 1999. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[89] Marc-Philippe Huget. *FIPA Modeling: Interaction Diagrams*. Foundation for Intelligent Physical Agent, Geneva, Switzerland, July 2003. `http://www.auml.org/auml/documents/`.

[90] MSC-96: ITU-T Recommendation Z.120 : Message Sequence Chart (MSC), 1996. `http://www.itu.int/`.

[91] Michael Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. Addison Wesley, 1995.

[92] Marcin Jurdziński. Small progress measures for parity games. In Horst Reichel and Sophie Tison, editors, *17th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2000)*, volume 1770 of *Lect. Notes in Computer Science*, pages 290–301, Lille, France, February 2000. Springer.

[93] J.A.W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, 1968.

[94] J. Katoen and L. Lambert. Pomsets for message sequence charts. In *1st conference on SDL and MSCs (SAM98)*, Berlin, 1998.

[95] Daniel Kirsten. Alternating tree automata and parity games. In Grädel et al. [63], chapter 9, page 385.

[96] Gerwin Klein. Jflex: The fast scanner generator for java. Software, September 1999. `http://jflex.de/`.

[97] Jochen Klose, Thomas Kropf, and Jürgen Ruf. A visual approach to validating system level designs. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 186–191. ACM Press, 2002.

[98] Jochen Klose and Hartmut Wittke. An Automata Based Interpretation of Live Sequence Charts. In T. Margaria and W. Yi, editors, *Proc. of TACAS (Tools and Algorithms for the Construction and Analysis of Systems) 2001*, volume 2031 of *LNCS*, page 512, Genova, Italy, April 2001. Springer-Verlag.

[99] Kai Koskimies, Tatu Männistö, Tarja Systä, and Jyrki Tuomi. SCED: A Tool for Dynamic Modelling of Object Systems. Technical Report Report A-1996-4, Department of Computer Science, University of Tampere, University of Tampere, Department of Computer Science, P.O. Box 607, FIN-33101 Tampere, Finland, July 1996. ISBN 951-44-4003-X, ISSN 0783-6910.

[100] Johannes Koskinen, Erkki Mäkinen, and Tarja Systä. Minimally adequate teacher synthesizes shuttles, too. In Holger Giese and Ingolf Krüger, editors, *Proc. of the 3rd Int. Workshop on "Scenarios and State Machines: Models, Algorithms and Tools" (SCESM'04)*, Edinburgh, May 2004. IEE.

[101] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From mscs to statecharts. Kluwer Academic Publishers, 1999. Franz J. Rammig (ed.).

[102] Hillel Kugler, David Harel, Amir Pnueli, Lu Yuan, and Yves Bontemps. Temporal logic for live sequence charts. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS'05)*, Lect. Notes in Computer Science. Springer, April 2005. To appear.

[103] Orna Kupferman and Moshe Y. Vardi. Synthesizing distributed systems. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, July 2001.

[104] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. Module checking. *Information and Computation*, 164:322–344, 2001.

[105] Salvatore La Torre. Finite automata on timed ω-trees. *Theoretical Computer Science*, 293(3):479–505, February 2003.

[106] Hichem M. Lamouchi and John G. Thistle. Effective control synthesis for DES under partial observations. In *Proc. 39th IEEE Conference on Decision and Control (Session on Discrete Event Systems)*, pages 22–28, Sidney, Australia, December 2000. IEEE.

[107] Leslie Lamport and Nancy Lynch. *Handbook of Theoretical Computer Science*, volume B, chapter 18, Distributed Computing: Models and Methods, pages 1157–1199. Elsevier - MIT Press, Amsterdam, The Netherlands and Cambridge, Massachussets, 1990. ISBN 0-262-72015-9 (Second Priniting, 1998).

[108] Y. Li and W.M. Wonham. On supervisory control of real-time discrete-event systems. *Information Sciences*, 46(3):159–183, 1988.

[109] Christof Löding and Wolfgang Thomas. Alternating automata and logics over infinite words. In *Proc. of the IFIP International Conference on Theoretical Computer Science - Exploring New Frontiers of Theoretical Informatics (IFIP TCS2000)*, volume 1872 of *Lect. Notes in Computer Science*, pages 521–535. Springer, 2000.

[110] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[111] P. Madhusudhan and P.S. Thiagarajan. Distributed controller synthesis for local specifications. In *Proc. of the ICALP'01*, volume 2076 of *Lect. Notes in Computer Science*, pages 396–407. Springer, 2001.

[112] P. Madhusudhan and PS Thiagarajan. A decidable class of asynchronous distributed controllers. In *Proc. of CONCUR'02*, volume 2421 of *Lect. Notes in Computer Science*, Brno, Czech Republic, 2002. Springer.

[113] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. John Wiley & Sons Ltd., New-York, 1999.

[114] Monika Maidl. The common fragment of CTL and LTL. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pages 643–652, 2000.

[115] Erkki Mäkinen and Tarja Systä. Acta informatica. *Minimally Adequate Teacher Synthesizes Statechart Diagrams*, 38:235–259, 2002.

[116] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems. In E.W. Mayr and C. Puech, editors, *Proc. of STACS'95*, volume 900 of *Lect. Notes in Computer Science*, pages 229–242. Springer, 1995.

[117] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New-York, 1995.

[118] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):68–93, 1984.

[119] Rami Marelly, David Harel, and Hillel Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *Proc. 17th Ann. ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02)*, pages 83–100, Seattle, WA, 2002.

[120] Donald A. Martin. Borel determinacy. *Annals of Mathematics*, 102:363–371, 1975.

[121] Sjouke Mauw and L.M.G Feijs. MSC and Data. In Yair Lahav, Adam Wolisz, Joachim Fischer, and Eckhardt Holz, editors, *Proc. of SAM98 - 1st workshop on SDL and MSC,*, pages 85–96, Humboldt-Universität, 1998. SDL Forum Society on SDL and MSC. number 104 in Informatik-berichte.

[122] Sjouke Mauw and Michel Reniers, A. Operational Semantics for MSC'96. *Computer Networks and ISDN Systems*, 17(31):1785–1799, 1999.

[123] Sjouke Mauw, Michel A. Reniers, and T.A.C. Willemse. *Handbook of Software Engineering and Knowledge Engineering*, volume 1 (Fundamentals), chapter Message Sequence Charts in the Software Engineering Process, pages 437–463. World Scientific Publishing Co. Pte. Ltd., December 2001.

[124] Ken McMillan, L. *Symbolic Model Checking.* Kluwer Academic Publiishers, 1993.

[125] R. McNaughton and S. Pappert. *Counter-free Automata.* MIT Press, 1971.

[126] Henry Muccini. An approach for detecting implied scenarios. In *Proc. ICSE 2002 Workshop on "Scenarios and State Machines: Models, Algorithms, and Tools"*, Orlando, Florida, 2002.

[127] Jishnu Mukerji and Joaquin Miller. Mda guide v 1.0.1 (omg), March 2003.

[128] Anca Muscholl and Doron Peled. Deciding properties of message sequence charts. In Stefan Leue and Tarja Systä, editors, *Proceedings of Dagstuhl Seminar on "Scenarios: Models, Algorithms and Tools"*, Lect. Notes in Computer Science. Springer, 2005. To appear.

[129] Object Management Group (UML Revision Task Force). *OMG UML Specification with Action Semantics(v1.4)*, July 2002. `http://www.uml.org`.

[130] Object Management Group (UML Revision Task Force). *OMG UML Specification (2.0)*, September 2003. `http://www.omg.org/uml`.

[131] Gerardo Padilla and Bengt Jonsson. An Execution Semantics for MSC-2000. In *SDL 2001: Meeting UML (Proceedings of the 10th International SDL Forum)*, number 2078 in LNCS, pages 365–378, Copenhagen, Denmark, June 2001. Springer-Verlag.

[132] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[133] Doron A. Peled, Edmund M. Clarke, and Orna Grumberg. *Model Checking.* MIT Press, Cambridge, Massachusetts, 2000. ISBN 02-620327-08.

[134] Amir Pnueli and Roni Rosner. On the Synthesis of a Reactive Module. In *Proceedings of the sixteenth annual ACM symposium on Principles of programming languages*, pages 179–190, 1989.

[135] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *Automata, Languages and Programming, 16th International Colloquium (ICALP)*, volume 372 of *Lect. Notes in Computer Science*, pages 652–671, Stresa, Italy, July 1989. Springer-Verlag.

[136] Vaughan R. Pratt. Modeling concurrency with partial orders. *Int. J. of Parallel Programming*, 15(1):33–71, February 1986.

[137] Anuj Puri, Stavros Tripakis, and Pravin Varaiya. Problems and examples of decentralized observation and control for discrete event systems. In Benoît Caillaud, Philippe Darondeau, Luciano Lavagno, and Xiaolan Xie, editors, *Synthesis and Control of Discrete Event Systems*. Kluwer Academic Publisher, January 2002. ISBN 0-7923-7639-0.

[138] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, 77, 1:81–98, 1989.

[139] Jean-François Raskin. *Logics, Automata and Classical Theories for Deciding Real Time*. PhD thesis, University of Namur, Namur, Belgium, June 1999.

[140] Roni Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, April 1992.

[141] Abhik Roychoudhury and P.S. Thiagarajan. Communicating transaction processes. In Felice Balarin and Johan Lilius, editors, *Proc. of the 3rd Int. Conf. on Applications of Concurrency to System Design (ACSD'03)*, pages 157–166, Guimarães, Portugal, June. IEEE Computer Science Press.

[142] Grzegorz Rozenberg and Arto Salomaa, editors. Springer, Berlin, New-York, 1997. ISBN 3-540-61486-9.

[143] Johannes Ryser and Martin Glinz. SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test. Technical Report 2000/3, Institut für Informatik - Universität Zurich, Winterthurerstrasse 190, 8057 Zurich, Switzerland, 2000.

[144] Johannes Ryser and Martin Glinz. Dependency Charts as a Means to Model Inter-Scenario Dependencies. In G. Engels, A. Oberweis, and A. Zündorf, editors, *Modellierung 2001, GI-Workshop*, Lecture Notes in Informatics, pages 71–80, Bad Lippsringe, Germany, 2001. GI-Edition.

[145] Shmuel Safra. On the complexity of ω-automata. In IEEE, editor, *29th annual Symposium on Foundations of Computer Science, October 24–26, 1988, White Plains, New York*, pages 319–327, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1988. IEEE Computer Society Press.

[146] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, April 1970.

[147] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):13 pages, 2001.

[148] Bikram Sengupta and Rance Cleaveland. Triggered message sequence charts. In *SIGSOFT2002/FSE-10*, Charleston, SC, USA, November 2002. ACM, ACM Press.

[149] A. Prasad Sistla and Edmund M. Clarke. Complexity of propositional temporal logics. *Journal of the ACM*, 32:733–749, 1985.

[150] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, third edition edition, 1996.

[151] Heikki Tauriainen. On translating linear temporal logic into alternating and nondeterministic automata. Technical Report HUT-TCS-A83, Helsinky University of Technology (HUT), 2003.

[152] John G. Thistle. Supervisory control of discrete event systems. *Mathl. Comput. Modelling*, 23(11/12):25–53, 1996.

[153] John G. Thistle and R. P. Malhamé. Control of ω-automata under state fairness assumptions. *Systems and Controls Letters*, 33:265–274, 1998.

[154] John G. Thistle and W. M. Wonham. Control of infinite behavior of finite automata. *SIAM J. Control and Optimization*, 32(4):1075–1097, July 1994.

[155] John G. Thistle and W. M. Wonham. Supervision of infinite behavior of discrete-event systems. *SIAM J. Control and Optimization*, 32(4):1098–1113, July 1994.

[156] Wolfgang Thomas. Automata on infinite objects. In van Leeuwen [170], chapter 4, pages 134–191. ISBN 0-262-72015-9 (Second Printing, 1998).

[157] Wolfgang Thomas. Languages, automata, and logic. In Rozenberg and Salomaa [142], chapter 7. ISBN 3-540-61486-9.

[158] Wolfgang Thomas. Infinite games and verification. In *Proceedings of the International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *Lect. Notes in Computer Science*, pages 58–64. Springer, 2002.

[159] Wolfgang Thomas. Automata and reactive systems. Lecture Notes, 2002–2003. `http://www-i7.informatik.rwth-aachen.de:81/scripte/ars-english.ps.gz`.

[160] Joseph Tripakis. Undecidable problems of decentralized observation and control on regular languages. *Information Processing Letters*, 90, 2004.

[161] Sebastiàn Uchitel. *Elaboration of Behaviour Models and Scenario-based Specifications using Implied Scenarios*. PhD thesis, Imperial College London, January 2003.

[162] Sebastiàn Uchitel and Jeff Kramer. A Workbench for Synthesizing Behaviour Models from Scenarios. In *Proc. of the 23rd IEEE International Conference on Software Engineering (ICSE'01)*. ACM, 2001.

[163] Sebastiàn Uchitel, Jeff Kramer, and Jeff Magee. Detecting implied scenarios in message sequence chart specifications. In Volker Gruhn, editor, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 74–82, New York, September 10–14 2001. ACM Press.

[164] Sebastiàn Uchitel, Jeff Kramer, and Jeff Magee. Negative scenarios for implied scenario elicitation. In William G. Griswold, editor, *Proceedings of the Tenth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-02)*, volume 27, 6 of *Software Engineering Notes*, pages 109–118, New York, November 18–22 2002. ACM Press.

[165] Sebastiàn Uchitel, Jeff Magee, and Jeff Kramer. From Sequence Diagrams to Behaviour Models. In *WTUML: Workshop on Transformations in UML. Satellite event of the European Joint Conference on Theory and Practice of Software (ETAPS'01)*, Genova, Italy, April 2001.

[166] International Telecommunication Union. MSC-2000: ITU-T Recommendation Z.120 : Message Sequence Chart (MSC), 2000. `http://www.itu.int/`.

[167] International Telecommunication Union. GRL: ITU-T Recommendation Z.151: Goal Requirements Language (GRL), September 2003. `http://www.itu.int/`.

[168] International Telecommunication Union. UCM: ITU-T Recommendation Z.152: Use Case Maps (UCM), September 2003. `http://www.itu.int/`.

[169] International Telecommunication Union. MSC-2004: ITU-T Recommendation Z.120 : Message Sequence Chart (MSC), 2004. Draft version. Still unapproved.

[170] Jan van Leeuwen, editor. MIT Press and Elsevier Science Publishers, Amsterdam, The Netherlands and Cambridge, Massachussets, 1990. ISBN 0-262-72015-9 (Second Printing, 1998).

[171] Moshe Vardi, Y. and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):183–221, April 1986.

[172] Jens Vöge and Marcin Jurdziński. A discrete strategy improvement algorithm for solving parity games (Extended abstract). In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Proceedings*, volume 1855 of *Lect. Notes in Computer Science*, pages 202–215, Chicago, IL, USA, July 2000. Springer.

[173] N. Wallmeier, Patrick Hütten, and Wolfgang Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In Oscar H. Ibarra and Zhe Dang, editors, *Proceedings of the 8th International Conference on the Implementation and Application of Automata (CIAA 2003)*, volume 2759 of *Lect. Notes in Computer Science*. Springer, 11–22.

[174] Klaus Weidenhaupt, Klaus Pohl, Matthias Jarke, and Peter Haumer. Scenario Usage in System Development: A Report on Current practice. *IEEE Software*, 15(2):34–45, March 1998.

[175] Eric W. Weisstein. Borel set. From MathWorld – A Wolfram Web Resource. `http://mathworld.wolfram.com/BorelSet.html`.

[176] Eric W. Weisstein. Hasse diagram. From MathWorld – A Wolfram Web Resource. `http://mathworld.wolfram.com/HasseDiagram.html`.

[177] Jon Whittle and Johan Schumann. Generating Statechart Designs from Scenarios. In *22nd International Conference on Software Engineering (ICSE 2000)*, pages 314–323, Limerick, Ireland, June 2000. ACM.

[178] Jon Whittle and Johan Schumann. Statechart Synthesis from Scenarios: an Air Traffic Control Case Study. In *Proc. of "Scenarios and State-Machines: models, algorithms and tools" workshop at the 24th Int. Conf. on Software Engineering (ICSE 2002)*, Orlando, FL, May, 20th 2002. ACM. `http://www.cs.tut.fi/~tsysta/ICSE/papers/`.

[179] Karl E. Wiegers. *Software Requirements.* Microsoft Press, 2nd edition, February 2003. ISBN 0735618798.

[180] Reinhard Wilhelm and Dieter Maurer. *Les compilateurs. Théorie. Construction. Génération.* MIM-Enseignement. Masson, 1994.

[181] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1-2), 72–99 1983.

[182] Pierre Wolper and Patrice Godefroid. Partial-order methods for temporal verification. In *Proc. of CONCUR'93*, volume 715 of *Lect. Notes in Computer Science*, pages 233–246. Springer, 1993.

[183] Pierre Wolper and Denis Leroy. Reliable hashing without collosion detection. In *Proc. of Computer-Aided Verification (CAV)*, pages 59–70, 1993.

[184] H. Wong-Toi and D. L. Dill. Synthesizing processes and schedulers from temporal specifications. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification '90: Proceedings of a DIMACS Workshop*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 177–186. American Mathematical Society, 1991.

[185] M. Yamanaka, S. Komura, J. Kato, and H. Ichikawa. Deriving protocols from message sequence charts in a communicating processes model. *IEICE Transactions on Information and Systems*, E79-D(11):1533–1544, 1996.

[186] Pamela Zave. A distributed alternative to finite-state-machine specifications. *ACM Transactions on Programming Languages and Systems*, 7(1):10–36, January 1985.

[187] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997.

[188] Tewfik Ziad, Loïc Hélouët, and Jean-Marc Jézéquel. Modeling behaviors in product lines. In *Proc. of REPL'02 (International workshop on Requirements Engineering for Product Lines)*, Essen, Germany, 2002.

[189] Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200:135–183, 1998.