Variants of acceptance specifications for modular system design

Guillaume Verdier

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *29 mars 2016* **par :**

GUILLAUME VERDIER

Variants of Acceptance Specifications for Modular System Design

(Variantes de spécifications à ensembles d'acceptation pour la conception modulaire de systèmes)

JURY

ROLF HENNICKER	Professeur des universités	LMU, Munich
GWEN SALAÜN	Maître de conférences, HDR	LIG/INRIA, Grenoble
ALAIN GRIFFAULT	Maître de conférences	LaBRI, Bordeaux
CHRISTIAN PERCEBOIS	Professeur des universités	IRIT, Toulouse
SILVANO DAL ZILIO	Chargé de recherche	LAAS/CNRS, Toulouse
JAN-GEORG SMAUS	Professeur des universités	IRIT, Toulouse
JEAN-BAPTISTE RACLET	Maître de conférences	IRIT, Toulouse

École doctorale et spécialité :
MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance

Unité de Recherche :
Institut de Recherche en Informatique de Toulouse

Directeurs de Thèse :
Jean-Baptiste Raclet et *Jan-Georg Smaus*

Rapporteurs :
Rolf Hennicker et *Gwen Salaün*

Contents

Chapter 1

Introduction

Context. Software programs are taking a more and more important place in our lives. Some of these programs, like the control systems of power plants, aircraft or medical devices for instance, are critical: a failure or malfunction could cause loss of human lives, damages to equipment or environmental harm. Formal methods aim at offering means to design and verify such systems in order to guarantee that they will work as expected. As time passes, these systems grow in scope and size, yielding new challenges. It becomes necessary to develop these systems in a modular fashion to be able to distribute the implementation task to engineering teams. Moreover, being able to reuse some trustworthy parts of the systems and extend them to answer new needs in functionalities is increasingly required. As a consequence, formal methods also have to evolve in order to accommodate both the design and the verification of these larger modular systems and thus address their scalability challenge.

Overview. There are different approaches to ensure that a system verifies a given property. One method is to first design and implement the system, and then to check if the implementation satisfies the property, as advocated by development processes such as the V-model or the waterfall model in which verification is a late phase. For example, one can use model-checking [BK08] to exhaustively check the executions of the system and obtain either the guarantee that for any possible execution, the property will be satisfied, or a counter-example exhibiting a case where the property is violated. If the property is not satisfied, one has to identify the cause of the failure, fix it, and then re-iterate the verification step until the system satisfies the property.

An alternative method that will be followed in this thesis is to rely on techniques leading to *correct-by-construction* systems [HS07]. More precisely, in this approach, the different steps of the design flow are controlled or assisted in such a way that expected properties checked at a certain step are preserved in the next steps and ultimately verified by the implementation.

Consider the example of an iterative system design depicted in Figure 1.1. The top layer represents the first step of a modular design in which the system is seen as the collaboration of three subsystems specified by S_1, S_2 and S_3. It shows a number of current challenges.

Concurrent design. By supporting stepwise refinement, S_1 may be replaced by a more detailed version of it formed by two sub-specifications S_{11} and S_{12}. This new design step must however be checked as being legal, that is as preserving the properties of S_1. If this is the case, S_{11} and S_{12} can be independently implemented by different design teams or suppliers and then composed in a bottom-up fashion to obtain a correct-by-construction realization of S_1.

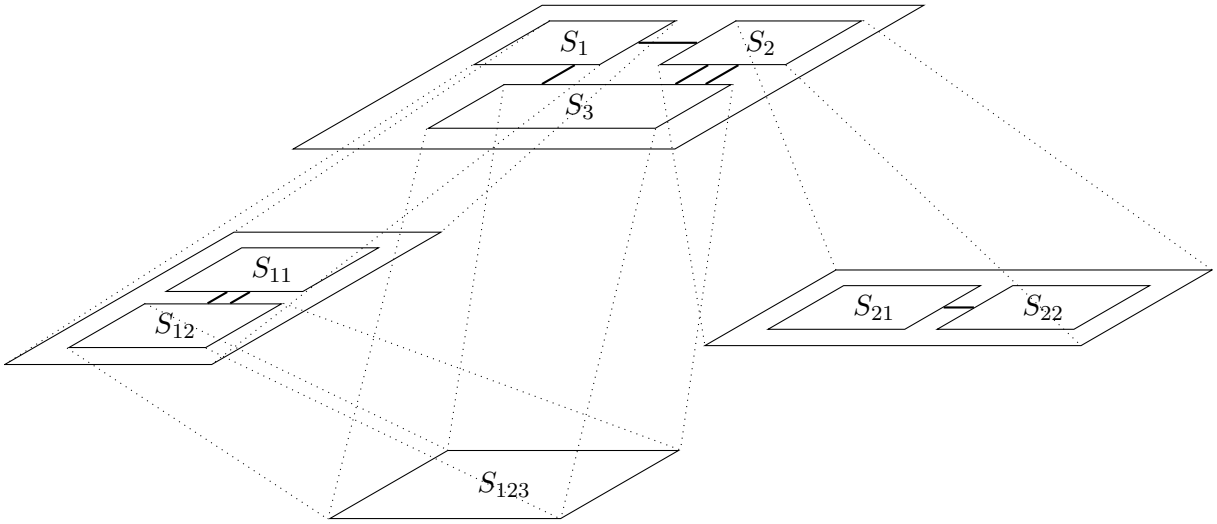


Figure 1.1: The incremental design of a modular system

Subsystem reuse. Next S_2 may be simplified as a preexisting subsystem S_{21}, said *off-the-shelf*, may be offering a similar goal modulo some adaptations represented by the specification S_{22}.

Specification merging. Also, in a next design step, different parts of the system design may be considered as similar enough to share a common implementation which can lead to merge different specifications, for example here, S_{12} and S_3 into S_{123}. As a consequence, designs must not be seen as trees but rather as directed acyclic graphs. The need for a merging operation on specifications also clearly appears in the viewpoint design practice in which different specifications are associated to a same system, each of them focusing on a different aspect (function, safety, timing, resource use, etc.) [RT14].

Reasoning about a system design then requires the definition of a formal model of the system together with a rich algebra on specifications with different operations. They have been first identified in [RBB+09, RBB+11] with their expected properties: refinement, composition via product, decomposition via quotient, and merge via conjunction, while supporting concepts such as independent implementability and property preservation from stepwise refinement. Several instantiations of this theory have later been studied, not exhaustively, in [BDF+13, CCJK12, BLL+14, LV13, BHL14, BDH+15, BFLV15] and in various contexts: with time [DLL+10b, BLPR12, KSL13], with quantities [BJL+12, BFJ+13, FKLT15], and with probabilities [CDL+11, DKL+13]. The work developed in this thesis also proposes different contributions which follow this algebraic approach.

Specifications can then be seen as abstract or early descriptions of the system under design. At least three levels of descriptions are usually considered [CMP06] for them:

Signature level. Typically, names of offered functions are given together with the types of their arguments, types of the return values, and exceptions possibly raised.

Behavioral level. The set of finite or infinite sequences of actions possibly occurring in the system is described hence allowing to address problems like deadlock-freeness or termination.

Semantic level. The provided descriptions allow here to state what the system actually *does*. Ontologies belong to this family of specification formalisms.

The formalisms considered in this thesis fit in the second category. Many theories may be used to express behavioral specifications: logics, in particular temporal logics, process algebras, or automata. Among the numerous contributions in the field of behavioral compositional theories, let us mention the works based on input-complete specifications (such as I/O automata [LT89], FOCUS [BDD+92], or reactive modules [AH99]) or non-input-complete specifications (such as interface automata [dAH01], interfaces with ports [BHL14], or modal interfaces [RBB+11, LNW07a, BFLV15]). In the following, the specification formalisms that we use in our contributions are all derived from a type of automata called *modal specifications*. A modal specification is an automaton with two kinds of transitions allowing to express mandatory and optional behaviors. Refining a modal specification amounts to deciding whether some optional parts should be removed or made mandatory. One can then reduce the variability of a specification by iteratively refining it until no optional parts remain, obtaining an implementation of the specification.

Contributions. This thesis contains two main theoretical contributions, based on an extension of modal specifications called acceptance specifications. The first one is the identification of a subclass of acceptance specifications, called convex-closed acceptance specifications, which allows us to define much more efficient operations while maintaining a high level of expressiveness. The second one is the definition of a new formalism, called marked acceptance specifications, that allows expressing some reachability properties. This could be used for example to ensure that a system is terminating or to express a liveness property for a reactive system. Standard operations are defined on this new formalism and guarantee the preservation of the reachability properties as well as independent implementability. This thesis also describes some more practical results. All the theoretical results on convex-closed acceptance specifications have been proved using the Coq proof assistant. The tool MAccS has been developed to implement the formalisms and operations presented in this thesis. It allowed us to test them easily on some examples, as well as run some experimentations and benchmarks.

Outline. Chapter 2 presents the state of the art; in particular, we will define modal specifications and give an overview of their numerous variants and extensions. Chapter 3 gives a more detailed definition of acceptance specifications and introduces the convex optimization, followed by an overview of the Coq mechanization. The marked extension of acceptance specifications is introduced in Chapter 4. The tool MAccS and experimental results are presented in Chapter 5. Finally, Chapter 6 concludes this thesis and offers some perspectives for future work.

Chapter 2

Modal Specifications

In this chapter, we present the state of the art. In the first section, we define modal specifications and give an overview of several of their extensions. We introduce the notion of specification theory in Section 2.2 and present the different operations it includes. Finally, we discuss the usage of nondeterministic specifications.

2.1 Overview and Variants

Remark. The same formalism is referenced in the literature using three different names: modal specifications, modal transition systems, and modal automata. For the sake of consistency, we will refer to them as "modal specifications" (sometimes abbreviated MS) in the following section, even when the referenced articles use another name. Similarly, we will use the term "acceptance specifications" (AS), even though some authors call them "acceptance automata."

Modal specifications were first introduced in [LT88]. They offer a formalism based on automata to specify some systems by expressing some mandatory and optional transitions. These specifications can then be *refined* by deciding if some optional parts should be removed or made mandatory. This allows to incrementally design a system by refining it step by step until no variability remains.

Consider for example the modal specification depicted in Figure 2.1. It is an automaton with four states labeled 0, 1, 2, and 3, an initial state 0, and some transitions between these states. Observe that contrary to classical automata, there are two kinds of transitions: straight lines are mandatory transitions and dashed lines are the optional ones. This specification describes the behavior of a server which receives some requests and sends a response which may be directly computed or fetched through a query to another server.

We can also see a modal specification as a characterization of a family—finite or not—of systems, called its *models* or *implementations*, represented by automata corresponding to all the possible combinations of implementation choices made by refinement. Some models of the

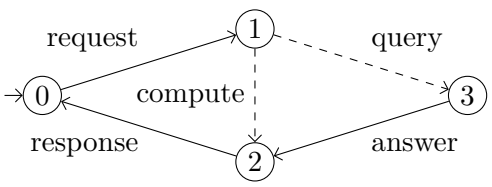
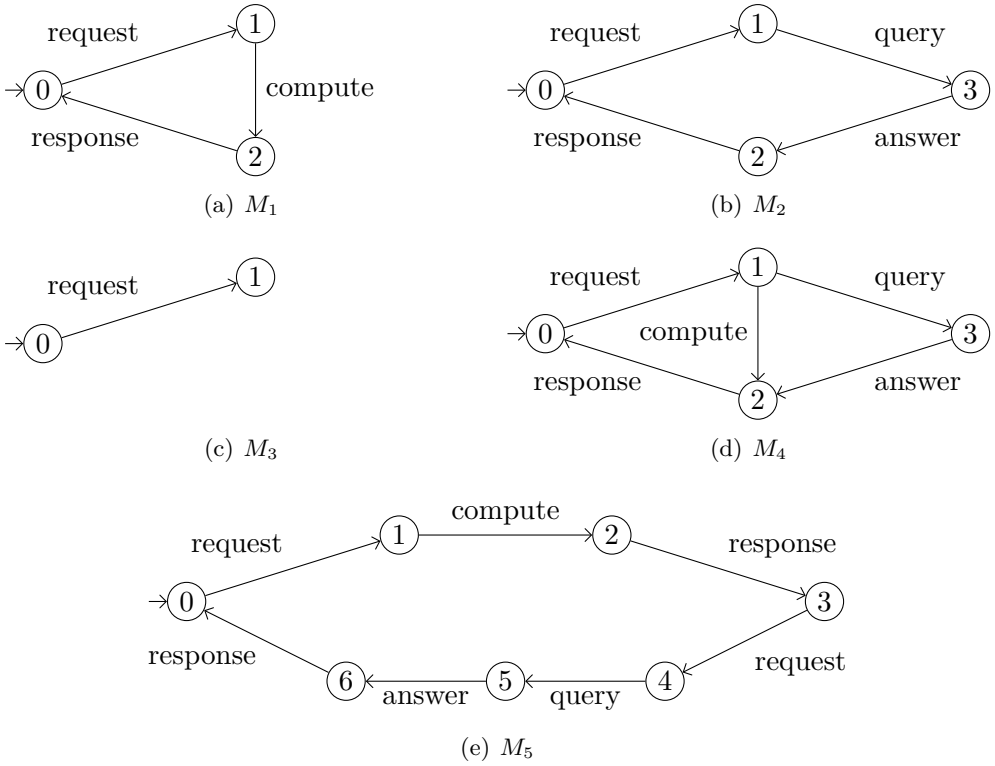


Figure 2.1: A modal specification



(a) M_1 (b) M_2

(c) M_3 (d) M_4

(e) M_5

Figure 2.2: Some models of the modal specification of Figure 2.1

example specification presented previously are depicted in Figure 2.2. From the initial state 0 of the specification, there is one mandatory transition, labeled *request*, so all the models have it. Afterwards, in state 1, there are two optional transitions, which may be realized or not. In M_1, we chose to realize the transition *compute* but not the *query*, while we did the opposite in M_2. In M_3, we decided to realize none and thus do nothing from state 1. Last, in M_4, we implemented both transitions. Then, the transitions *response* from state 2 and *answer* from state 3 are both mandatory, so they are realized in all the models where these states are reached. Finally, M_5 shows that the models of a modal specification have to observe the requirements expressed by the two types of transitions, but not the structure of the specification itself: they can unfold it in order to duplicate some states and make different implementation choices. Therefore, M_5 alternates between computing the result and sending a query to get it. Observe that due to the possibility of unfolding the underlying automaton, the specification has an infinite number of models. For instance, we could build an infinite set consisting of the models realizing the transition *compute* n times (for any natural number n), then the transition *query* once (M_5 is an example of a such model for $n = 1$).

Modal specifications may be based on deterministic or nondeterministic automata. Since the contributions of this thesis are related to deterministic structures, we will now formally define deterministic automata and deterministic modal specifications, as well as the satisfaction relation between a specification and one of its models. We will discuss the choice of using deterministic specifications in Section 2.3.

Definition 1 (Automaton)**.** *A deterministic automaton over an alphabet* Σ *is a tuple* (R, r^0, λ) *where* R *is the set of states,* $r^0 \in R$ *is the initial state, and* $\lambda : R \times \Sigma \to R$ *is the partial labeled transition map. We define the set of fireable actions from a state* r*, denoted* $\mathrm{ready}(r)$*, as the set of*

actions a such that $\lambda(r, a)$ is defined.

Definition 2 (Modal Specification)**.** *A deterministic modal specification over an alphabet Σ is a tuple $(Q, q^0, \delta, \text{may}, \text{must})$ where Q is the set of states, $q^0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to Q$ is the partial labeled transition map, and $\text{may}, \text{must} : Q \to 2^\Sigma$ are the sets of optional and mandatory transitions.*

We also define a special empty modal specification $S_\perp$, which has no models.

Definition 3 (Satisfaction)**.** *An automaton M is a model of a modal specification S, denoted $M \models S$, if and only if there exists a simulation relation $\pi \subseteq R \times Q$ such that $(r^0, q^0) \in \pi$ and for any $(r, q) \in \pi$:*

- $\text{must}(q) \subseteq \text{ready}(r) \subseteq \text{may}(q)$*;*
- *for any $a \in \text{ready}(r)$, we have $(\lambda(r, a), \delta(q, a)) \in \pi$.*

The set of models of S is denoted $[\![S]\!]$.

For example, let us look back at the specification in Figure 2.1. The initial state q^0 is 0 and for any state q, the transitions in $\text{may}(q) \setminus \text{must}(q)$ are depicted with dashed lines while the transitions in $\text{may}(q) \cap \text{must}(q)$ are straight lines. Consider the model M_5 of this specification in Figure 2.2(e): we can see that the simulation relation is $\{(0,0), (1,1), (2,2), (3,0), (4,1), (5,3), (6,2)\}$.

According to the definition of modal specifications, we could have some specifications with more transitions in must than in may. For example, consider the following specification : $(\{0\}, 0, \{(0, a) \mapsto 0, (0, b) \mapsto 0\}, \{0 \mapsto \{a\}\}, \{0 \mapsto \{a, b\}\})$. It consists of a single state 0 with two transitions to itself labeled a and b. The transition a is in both may(0) and must(0) while b only belongs to must(0). If we want to build a model of this specification, the must set tells us that we have to realize the two transitions by a and b, but the may set only allows a. Thus, it is impossible to build a model of this specification.

Definition 4 (Inconsistency)**.** *Given a modal specification S, a state q of S is said to be inconsistent if* $\text{must}(q) \not\subseteq \text{may}(q)$ *or* $\text{ready}(q) \neq \text{may}(q)$*.*

A modal specification is said inconsistent if it has an inconsistent state. The specification $S_\perp$ is consistent.

Theorem 1 (Pruning)**.** *Given an inconsistent modal specification S, there exists a consistent modal specification, called* normal form *of S and denoted $\rho(S)$, with the same models as S.*

We can construct $\rho(S)$ by recursion: remove the inconsistent states and all the transitions leading to them, and repeat the process if it has generated some new inconsistencies. Since inconsistent states have incompatible constraints and can not be realized by the models of the specification, removing them does not change its set of models. A more detailed construction along with a proof of correctness are given in [Rac08].

Remark. If the initial state of S is inconsistent (or if an inconsistent state is reachable from the initial state by taking only must transitions), then $\rho(S) = S_\perp$.

In consequence, we can now assume, without loss of generality, that all the modal specifications are consistent; whenever a specification may not be consistent, we can simply apply ρ in order to get an equivalent consistent specification. The advantage of having a separate pruning operation ρ instead of requiring directly in the definition of modal specifications that $\text{may}(q) \subseteq \text{must}(q)$ is that some operations may temporarily generate an inconsistent specification and then use ρ to remove the inconsistencies, rather than building a consistent specification in a single step.

We also define a refinement relation between modal specifications:

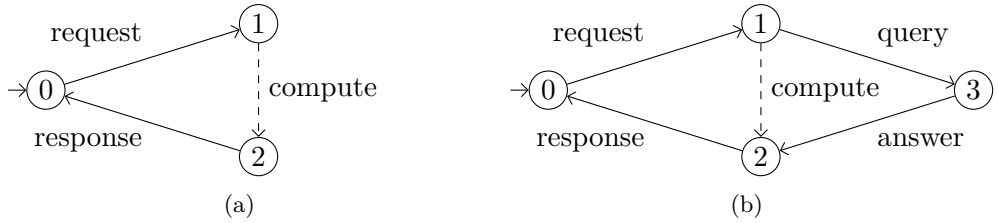


Figure 2.3: Some refinements of the modal specification of Figure 2.1

Definition 5 (Modal refinement)**.** *Given two modal specifications S_1 and S_2, S_1 is a refinement of S_2, denoted $S_1 \leq S_2$, if and only if there exists a simulation relation $\pi \subseteq Q_1 \times Q_2$ such that $(q_1^0, q_2^0) \in \pi$ and for any $(q_1, q_2) \in \pi$:*

- $\text{may}(q_1) \subseteq \text{may}(q_2)$*;*
- $\text{must}(q_2) \subseteq \text{must}(q_1)$*;*
- *for any $a \in \text{may}(q_1)$, we have $(\delta(q_1, a), \delta(q_2, a)) \in \pi$.*

Moreover, for any specification S, $S_\perp \leq S$.

This definition of refinement is equivalent to *thorough refinement*, i.e. sets of models inclusion (see [Rac08] for the proof). Note that while most definitions and theorems of this section can be adapted to nondeterministic modal specifications, it is not the case for this one. We give the counter-example in Section 2.3.

Theorem 2. *Given two modal specifications S_1 and S_2, $S_1 \leq S_2$ if and only if $[\![S_1]\!] \subseteq [\![S_2]\!]$.*

We depicted in Figure 2.3 two possible refinements of the modal specification of Figure 2.1. In the left one, we removed a transition, *query*, from the set may. In the right one, we extended the set must by adding the transition *query* to it.

Variants. Since the introduction of modal specifications in 1988, many variants have been developed, that we will review now.

Mixed specifications [DGG97] are similar to modal specifications without the consistency assumption. Thus, the case of transitions belonging to the must set but not to the may set is handled explicitly, while in modal specifications it is assumed that a pruning step has been applied beforehand if needed.

Intuitively, the must transitions of modal specifications express a conjunction: all the transitions in the must set have to be realized by the implementations. Several variants of modal specifications have been devised in order to express other kinds of constraints.

Disjunctive modal specifications [LX90] allow expressing a disjunction of must transitions: at least one of the transitions has to be realized. For example, we show in Figure 2.4 a disjunctive variant of the modal specification of Figure 2.1 with a disjunctive-must (d-must) for the transitions *compute* and *query* from state 1. This disjunctive modal specification will have essentially the same models as the modal specification, except that at least one of the transitions *compute* and *query* has to be realized, thus forbidding models like M_3 (Figure 2.2(c)). We will now give a formal definition of disjunctive modal specifications and their satisfaction relation. Note that we give the definition of the deterministic version of disjunctive modal specifications.

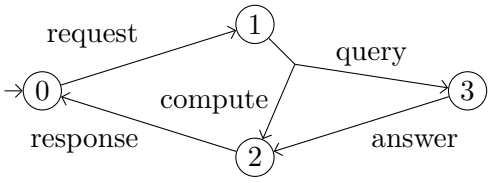


Figure 2.4: A disjunctive modal specification

Definition 6 (Disjunctive Modal Specification)**.** *A deterministic disjunctive modal specification over an alphabet* Σ *is a tuple* $(Q, q^0, \delta, \text{may}, \text{d-must})$ *where* Q *is the set of states,* $q^0 \in Q$ *is the initial state,* $\delta : Q \times \Sigma \to Q$ *is the partial labeled transition map,* $\text{may} : Q \to 2^\Sigma$ *is the set of optional transitions, and* $\text{d-must} : Q \to 2^{2^\Sigma}$ *is a set of disjunctions of mandatory transitions.*

Definition 7 (Satisfaction)**.** *An automaton* M *is a model of a disjunctive modal specification* S*, denoted* $M \models S$*, if and only if there exists a simulation relation* $\pi \subseteq R \times Q$ *such that* $(r^0, q^0) \in \pi$ *and for any* $(r, q) \in \pi$*:*

- $\text{ready}(r) \subseteq \text{may}(q)$*;*
- *for any* $\text{must} \in \text{d-must}(q)$*,* $\text{ready}(r) \cap \text{must} \neq \emptyset$*;*
- *for any* $a \in \text{ready}(r)$*, we have* $(\lambda(r, a), \delta(q, a)) \in \pi$*.*

One-selecting modal specifications [FS08] offer an exclusive disjunction instead of the inclusive disjunction of disjunctive modal specifications. Thus, if we consider the specification of Figure 2.4 to be a one-selecting modal specification, it would also forbid models like M_4 (Figure 2.2(d)) which realizes both transitions *compute* and *query* simultaneously (on the other hand, the model M_5 is fine since these two transitions are realized in different states). Moreover, one-selecting modal specifications also offer exclusive disjunctions of may transitions.

Acceptance specifications [Rac08] are an even more expressive extension of modal specifications since they allow expressing arbitrary constraints on the transitions, not just conjunctions or disjunctions. This formalism is the basis of the contributions of this thesis, so we will present it in details in Chapter 3.

Another approach is to use a boolean formula to express the constraints on the transitions instead of sets of may/must/d-must/... transitions. Modal specifications with obligations [BK10] accept arbitrary positive boolean formulas and boolean modal specifications [BKL$^+$11] extend them with negation. If the formulas are in conjunctive normal form without negation, the specification is a disjunctive modal specification, and if the formulas are only conjunctions of actions, the specification is a modal specification. Parametric modal specifications [BKL$^+$11] add boolean parameters to these specifications.

Definition 8 (Positive Boolean Formula)**.** *A positive boolean formula over an alphabet* Σ *is given by the following grammar:*

$$\varphi ::= a \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \top \mid \bot$$

with $a \in \Sigma$*. We denote the set of all positive boolean formulas as* $\mathcal{B}^+$*.*

Given a formula φ*, the set of actions satisfying the formula, denoted* $[\![\varphi]\!]$*, is defined as:*

$$[\![a]\!] = \{X \mid a \in X\} \quad [\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!] \quad [\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!] \quad [\![\top]\!] = 2^\Sigma \quad [\![\bot]\!] = \emptyset$$

Definition 9 (Modal Specification with Obligations)**.** *A deterministic modal specification with obligations over an alphabet Σ is a tuple (Q, q^0, δ, Ω) where Q is the set of states, $q^0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to Q$ is the partial labeled transition map, and $\Omega : Q \to \mathcal{B}^+$ is the set of obligations.*

Definition 10 (Satisfaction)**.** *An automaton M is a model of a modal specification with obligations S, denoted $M \models S$, if and only if there exists a simulation relation $\pi \subseteq R \times Q$ such that $(r^0, q^0) \in \pi$ and for any $(r, q) \in \pi$:*

- $\text{ready}(r) \in [\![\Omega(q)]\!]$ *or* $\text{ready}(r) = [\![\Omega(q)]\!] = \emptyset$*;*
- *for any $a \in \text{ready}(r)$, we have $(\lambda(r, a), \delta(q, a)) \in \pi$.*

We now give the definition of boolean modal specifications which is very close to the definition of modal specifications with obligations, but with more expressive formulas:

Definition 11 (Boolean Formula)**.** *A boolean formula over an alphabet Σ is given by the following grammar:*

$$\varphi ::= a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \top$$

with $a \in \Sigma$. We denote the set of all boolean formulas as $\mathcal{B}$.

Given a formula φ, the set of actions satisfying the formula, denoted $[\![\varphi]\!]$, is defined as:

$$[\![a]\!] = \{X \mid a \in X\} \quad [\![\neg\varphi]\!] = 2^\Sigma \setminus [\![\varphi]\!] \quad [\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!] \quad [\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!] \quad [\![\top]\!] = 2^\Sigma$$

Definition 12 (Boolean Modal Specification)**.** *A deterministic boolean modal specification over an alphabet Σ is a tuple (Q, q^0, δ, Ω) where Q is the set of states, $q^0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to Q$ is the partial labeled transition map, and $\Omega : Q \to \mathcal{B}$ is the set of obligations.*

Definition 13 (Satisfaction)**.** *An automaton M is a model of a boolean modal specification S, denoted $M \models S$, if and only if there exists a simulation relation $\pi \subseteq R \times Q$ such that $(r^0, q^0) \in \pi$ and for any $(r, q) \in \pi$:*

- $\text{ready}(r) \in [\![\Omega(q)]\!]$ *or* $\text{ready}(r) = [\![\Omega(q)]\!] = \emptyset$*;*
- *for any $a \in \text{ready}(r)$, we have $(\lambda(r, a), \delta(q, a)) \in \pi$.*

We illustrate the relations between these different specification formalisms in Figure 2.5. Note that this is for deterministic specification formalisms. For nondeterministic specifications, [BDF+13] shows that disjunctive modal specifications are equivalent to acceptance specifications, hence most formalisms of Figure 2.5 are equivalent in the nondeterministic case, save the parametric extension. The relations between the specification formalisms indicated in Figure 2.5 and convex and acceptance specifications will be justified in Chapter 3.

Logic equivalences. There are some equivalences between specification formalisms and logics that is, there are constructions to convert an automata-based specification into a logic formula having the same models and vice versa. Modal specifications have been linked to Hennessy-Milner logic (HML) [HM80]: any modal specification has an equivalent HML formula [Lar89] and any consistent and *prime* HML formula is equivalent to a modal specification [BL92]. Moreover, nondeterministic disjunctive modal specifications are equivalent to HML formulas with greatest fixed points [BDF+13].

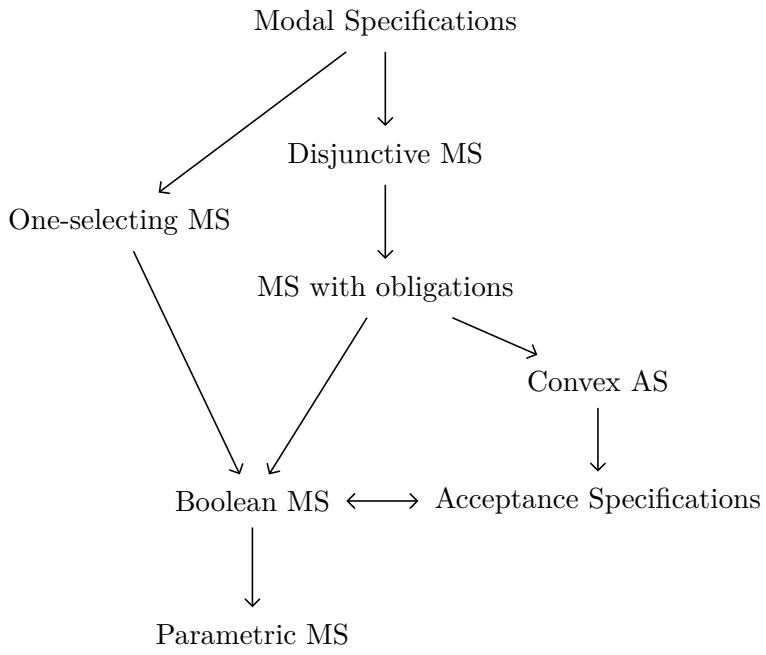


Figure 2.5: Relations between deterministic specification formalisms

Applications. As hinted in the introduction of this thesis, modal specifications have been intensively used as a specification formalism for modular system design via the definition of specification theories. They have also been used in different contexts that we briefly mention now.

In [BG00], Kripke structures with modalities are introduced to represent incomplete state spaces. A 3-valued answer is then provided to the model-checking question; the answer *unknown* corresponds to the situation where the witness paths have a *may* modality. Other uses of modalities in model-checking have been presented in [CDEG03, HJS01].

Modalities have also been used for software product line modeling [AtBFG10]. The optional behavior encoded by the may modality corresponds to possible features of a product from the family specified by the modal specification.

Modalities have been applied to contract-based design [GR09, BDH$^+$12, NITS14]. In essence, a contract is a component specification that can be viewed as a pair $(\mathcal{A}, \mathcal{G})$ of two specification requirements, where $\mathcal{A}$ is an assumption on the environment where the component executes and $\mathcal{G}$ is a guarantee on the behavior of the component (given that the assumption is correctly met). This paradigm offers great improvements in system design [BCN$^+$12]: it eases component integration while enabling compositional design and verification and providing a legal binding between the different suppliers of a development chain.

Extensions. Modal specifications have been extended with input/output actions and interface compatibility notions, based on the approach of interface automata [dAH01]. It was done for both deterministic specifications [LNW07a, RBB$^+$09, RBB$^+$11] and nondeterministic ones [LV12, BFLV15, CCJK12]. Modal specifications with data [BHB10, BHW11, BLL$^+$14] enrich the interfaces with data variables.

Many timed extensions have been proposed for modal specifications, such as timed modal specifications [ČGL93], modal event-clock specifications [BLPR09, BLPR12], timed I/O modal specifications [DLL$^+$10b], and time-parametric modal specifications [KSL13].

Weighted modal specifications [BFJ+13] and label-structured modal specifications [BJL+12] extend modal specifications with quantitative properties. A probabilistic extension has been defined in [JL91].

Petri nets decorated with modalities on transitions have been considered in [EHH12, HHM13].

Marked modal specifications [CR12] add reachability properties by means of marked states. We will talk about this formalism and our extension, marked acceptance specifications, in Chapter 4.

2.2 A Modal Specification Theory

We have presented in the previous section the formalism of modal specifications and its semantics via the definitions of the refinement and satisfaction relations. Now, we define some operations on modal specifications to build a modal specification theory as it is done in [RBB+11]. As already briefly advocated in the introduction of this thesis, this algebra enables modular system design and allows addressing a number of challenges. In what follows, we will motivate precisely each of these operations.

Note also that defining specification theories is the stepping stone for the construction of contract-based theories as advocated in [BDH+12]. In this paper, it is shown that given a specification theory with refinement, product, conjunction and quotient for a given formalism $\mathcal{S}$, it is possible to derive for free a contract theory for pairs $(\mathcal{A}, \mathcal{G})$ of specifications from $\mathcal{S}$ with refinement and product.

2.2.1 Conjunction

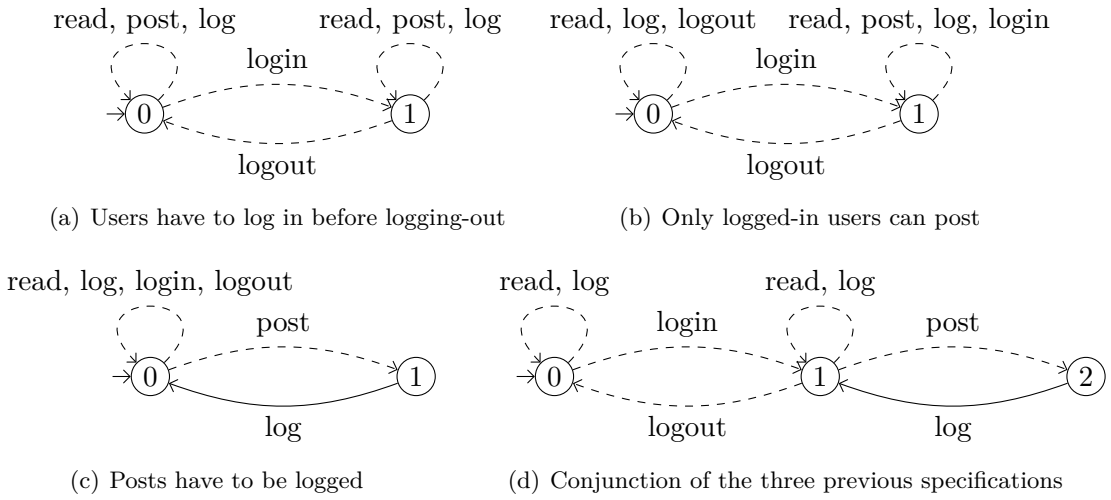
When specifying a system, it may be easier for a team of system designers to describe the different aspects of a system (function, safety, timing, resource use, etc.) in distinct specifications. This discipline is often referred to as viewpoint design (see [RT14] for a survey). Natural questions arising then are: are these viewpoints consistent that is, do they contradict one another? How can one be sure that all aspects are eventually implemented? These questions call for the support of a conjunction operation on specifications characterizing the common implementations of a set of viewpoints described in some specifications. In particular, inconsistency of viewpoints can be tested by checking if a conjunction has an empty set of models.

Conjunction of modal specifications, also called *merge*, has been initially studied in [UC04] when silent actions are involved. It has also been considered for labeled transition systems [LV06], for Moore interfaces [HN12], and for interface automata [DHJP08]. In this last paper, it is also argued that supporting conjunction allows merging specifications considered to be similar enough to share a common implementation, hence alleviating the implementation task.

Consider now for example the specifications in Figure 2.6. The goal is to specify the behavior of a simple forum-like server where users may log in, log out, and read and post messages. Moreover, the server may log some information. We want to express three requirements and write a modal specification for each one:

1. Figure 2.6(a): users are initially anonymous; they may log in and log out afterwards;
2. Figure 2.6(b): only logged-in users are allowed to post a message;
3. Figure 2.6(c): the server has to note in the log file when someone posts a message.

Then, we can use the conjunction operation to merge altogether these three viewpoints of the system in order to obtain a single specification, depicted in Figure 2.6(d). If an automaton is



(a) Users have to log in before logging-out

(b) Only logged-in users can post

(c) Posts have to be logged

(d) Conjunction of the three previous specifications

Figure 2.6: Several requirements of a system and their conjunction

an implementation of this specification, we will have the guarantee that it also implements each requirement, and vice versa. Moreover, refinement is preserved by conjunction, so if we refine the specifications, their conjunction will refine the first conjunction.

Definition 14 (Conjunction)**.** *Given two modal specifications S_1 and S_2, their conjunction $S_1 \wedge S_2$ is the normal form of $S_1 \,\&\, S_2 = (Q_1 \times Q_2, (q_1^0, q_2^0), \delta, \text{may}, \text{must})$ where $\delta((q_1, q_2), a)$ is defined as $(\delta(q_1, a), \delta(q_2, a))$ when both are defined,* $\text{may}((q_1, q_2)) = \text{may}(q_1) \cap \text{may}(q_2)$*, and* $\text{must}((q_1, q_2)) = \text{must}(q_1) \cup \text{must}(q_2)$.

The conjunction of two modal specifications characterizes precisely the intersection of their sets of models:

Theorem 3. *Given two modal specifications S_1 and S_2, $[\![S_1 \wedge S_2]\!] = [\![S_1]\!] \cap [\![S_2]\!]$.*

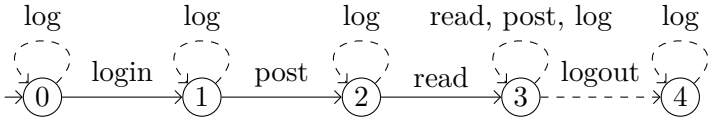
As a consequence, the conjunction operation is commutative and associative. Moreover, since the modal refinement is a thorough refinement, the conjunction is monotonic w.r.t. refinement:

Corollary 1. *Given four modal specifications S_1, S_1', S_2, and S_2' such that $S_1' \leq S_1$ and $S_2' \leq S_2$, $S_1' \wedge S_2' \leq S_1 \wedge S_2$.*

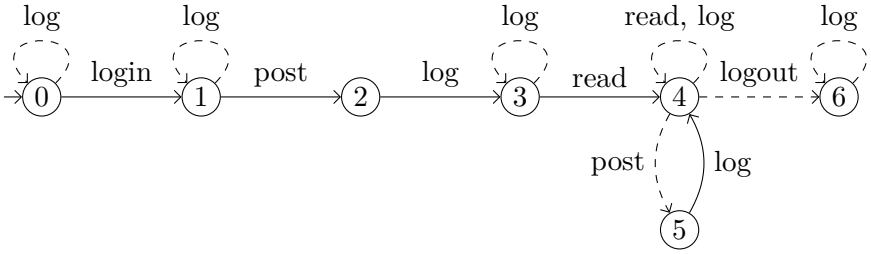
2.2.2 Product

We also want to be able to compose modal specifications by computing their product, which results in a specification where their common transitions have been synchronized. This enables a bottom-up approach to system design: we can start from basic components and compose them together in order to obtain a more complex system.

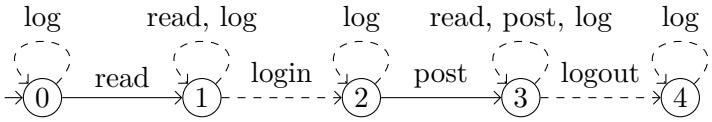
We described in the previous section (Figure 2.6) a server for a message board. We could specify the behavior of some users of this service. For instance, in Figure 2.7(a), we describe a user who wants to ask something: she logs in, posts a message, and then reads the responses, possibly posting other messages. In Figure 2.7(c), we specify another type of user who first browses the board and reads some message, and then may decide to log in and participate in a discussion. We can then compose the specification of a user with the specification of the server (Figure 2.6(d)), as depicted in Figures 2.7(b) and 2.7(d).



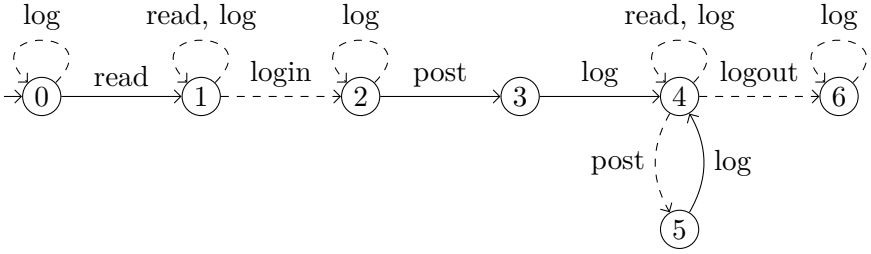
(a) A user asking a question



(b) Product with the server



(c) A user browsing the messages



(d) Product with the server

Figure 2.7: Some users of the message board

Definition 15 (Product)**.** *Given two modal specifications S_1 and S_2, their product is the modal specification $S_1 \otimes S_2 = (Q_1 \times Q_2, (q_1^0, q_2^0), \delta, \text{may}, \text{must})$ where $\delta((q_1, q_2), a)$ is defined as $(\delta(q_1, a), \delta(q_2, a))$ when both are defined,* $\text{may}((q_1, q_2)) = \text{may}(q_1) \cap \text{may}(q_2)$*, and* $\text{must}((q_1, q_2)) = \text{must}(q_1) \cap \text{must}(q_2)$*.*

The product of modal specifications generalizes the product of models by characterizing the set of the products of models of S_1 and S_2:

Theorem 4. *Given two modal specifications S_1 and S_2, and two automata $M_1 \models S_1$ and $M_2 \models S_2$, $M_1 \times M_2 \models S_1 \otimes S_2$.*

Moreover, the product is the most precise characterization of the products of models of S_1 and S_2:

Theorem 5. *Given three modal specifications S_1, S_2 and S, if for any $M_1 \models S_1$ and $M_2 \models S_2$, $M_1 \times M_2 \models S$, then $S_1 \otimes S_2 \leq S$.*

The product operation is commutative and associative. It is also monotonic w.r.t. refinement:

Theorem 6. *Given four modal specifications S_1, S_1', S_2, and S_2' such that $S_1' \leq S_1$ and $S_2' \leq S_2$, $S_1' \otimes S_2' \leq S_1 \otimes S_2$.*

As a result, given an initial design $S_1 \otimes S_2$, the two specifications S_1 and S_2 can be independently refined, potentially by different design teams or suppliers, and then composed in a bottom-up fashion to obtain a correct-by-construction realization of the initial design.

2.2.3 Quotient

The product presented earlier enables a bottom-up approach: one may specify various systems and then compose them together. On the other hand, one may prefer a top-down approach: given the specification of a desired system $\mathcal{G}$ and the specification of some pre-existing trustworthy component $\mathcal{C}$ (from a library for instance), what is the specification of the system $\mathcal{S}$ that we should realize so that its product with $\mathcal{C}$ refines $\mathcal{G}$? This is given by the quotient $\mathcal{G}/\mathcal{C}$ that we consider now in the modal case by following the approach initially developed in [Rac08].

Definition 16 (Quotient)**.** *Given two modal specifications S_1 and S_2, their quotient S_1/S_2 is the normal form of $S_1 /\!\!/ S_2 = ((Q_1 \times Q_2) \cup \{q_\top\}, (q_1^0, q_2^0), \delta, \text{may}, \text{must})$ with:*

$$\delta((q_1, q_2), a) = \begin{cases} (\delta(q_1, a), \delta(q_2, a)) & \text{when both are defined} \\ q_\top & \text{otherwise} \end{cases}$$

$$\begin{cases} a \in \text{may}((q_1, q_2)) \cap \text{must}((q_1, q_2)) & \text{if } a \in \text{must}(q_1) \cap \text{must}(q_2) \\ a \in \text{must}((q_1, q_2)) \setminus \text{may}((q_1, q_2)) & \text{if } a \in \text{must}(q_1) \setminus \text{must}(q_2) \\ a \in \text{may}((q_1, q_2)) \setminus \text{must}((q_1, q_2)) & \text{if } a \in \text{may}(q_1) \setminus \text{must}(q_1) \\ a \in \text{may}((q_1, q_2)) \setminus \text{must}((q_1, q_2)) & \text{if } a \notin \text{may}(q_1) \cup \text{may}(q_2) \\ a \notin \text{may}((q_1, q_2)) \cup \text{must}((q_1, q_2)) & \text{if } a \in \text{may}(q_2) \setminus \text{may}(q_1) \end{cases}$$

This quotient operation is dual of the product:

Theorem 7. *Given three modal specifications S, S_1 and S_2, $S \leq S_1/S_2$ if and only if $S \otimes S_2 \leq S_1$.*

We can also characterize it directly w.r.t. the sets of models of its operands:

Theorem 8. *Given two modal specifications S_1 and S_2, and an automaton M, $M \models S_1/S_2$ if and only if for all $M_2 \models S_2$, $M \times M_2 \models S_1$.*

Observe that we quantify universally on the models of S_2. It is because this reused system must be seen as a black-box: its implementation is unknown, its reuse is enabled only from the description provided by its specification.

The quotient operation is also crucial for contract satisfaction [BCN+12]. As briefly explained in the paragraph *Applications* at the end of Section 2.1, a contract is a pair of specifications $(\mathcal{A}, \mathcal{G})$ where $\mathcal{A}$ describes some assumptions on the environment of a system M; this system M has to guarantee the satisfaction of $\mathcal{G}$ when put in a correct environment satisfying $\mathcal{A}$. More formally, if $E \models \mathcal{A}$ then we must have $M \times E \models G$ which exactly corresponds to check whether $M \models \mathcal{G}/\mathcal{A}$.

Different problems very similar to synthesizing a quotient exist in the literature. We can first mention the problem of *controller synthesis* [RW89] considered in the discrete-event systems community. The goal there is to synthesize a subsystem called a *controller* which aims at enforcing a given specification on a given system. In this context, the system to be controlled is in most cases a deterministic finite automaton [RW89, CL08] whose transitions can be labeled by actions declared uncontrollable, that is the controller cannot forbid them, or unobservable, that is the controller cannot *see* their occurrence. Quotient as considered in this section is quite different from monolithic controller synthesis. Indeed, we compute quotient of two specifications while monolithic controller synthesis can be interpreted as the quotient of a specification, the control objective, by the system to be controlled. It is more relevant to link quotient with distributed controller synthesis. This was advocated in [AVW03] in which quotient of Mu-calculus formulas S_1/S_2 is investigated in order to test the existence of a subcontroller enforcing locally S_2 and globally S_1. Their remarkable theoretical contribution is however unusable in practice because of its complexity cost.

Quotient is also close to computing a protocol converter or an adaptor [YS97, CPS08, MPS12] in order to correct some mismatches between a set of interacting subsystems and thus enforcing a compatibility criterion (deadlock freeness, for instance). The problem has been intensively studied in the service community (see [BBG+04, CMP06] for surveys). There again, a clear difference is that the description of the system to be adapted is a fixed labeled transition system while our quotient handles specifications, e.g. families, possibly infinite, of systems.

More abstractly, all these previous problems are seen in [VYB+11, VPY+15] as solving equations of the form:

$$\mathcal{C} \| \mathcal{X} \sim \mathcal{G}$$

where the goal is to synthesize the unknown subsystem $\mathcal{X}$ that when composed via the operation $\|$ with the given context $\mathcal{C}$ produces a system which is conform for $\sim$ to the given objective represented by $\mathcal{G}$. Language equation solving is considered for regular and infinite languages. Actions from the alphabet can either be inputs if they stem from the system environment or outputs when they originate from the system. Composition may correspond to synchronous product with internalization of synchronized actions (see [VYB+11] for a survey).

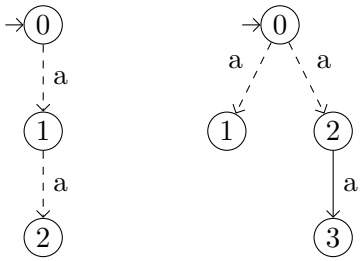
Links between all these problems have been clearly highlighted in [VYB+11, GMW12].

2.3 Nondeterminism

There are both deterministic and nondeterministic versions of modal specifications. The advantage of nondeterministic specifications is rather clear: they are a strict superset of deterministic

specifications, and thus more expressive. However, nondeterminism has some drawbacks.

The first problem was mentioned when we defined the refinement relation on modal specifications and proved that it is equivalent to thorough refinement (Theorem 2). This result does not hold for nondeterministic specifications, as shown in [LNW07b]. Indeed, consider the two nondeterministic specifications depicted in Figure 2.8. There are three implementation choices allowed by the specification S: realizing no transition from the initial state, a single transition by a, or two consecutive transitions by a. In each case, the corresponding choice may be made by implementations of T. However, S does not refine T: starting from the pair of initial states $(0, 0)$, there is a may transition by a which may go to the pair $(1, 1)$ or to the pair $(1, 2)$. There is a may transition by a from state 1 of S, but in the first case, it is forbidden by state 1 of T and in the second case, it is in the must set of state 2 of T. Thus, S does not refine T, while its set of models is included in the set of models of T. According to [LNW07b], thorough refinement is decidable, so it is possible to check it directly rather than using modal refinement, but it is co-NP hard, making it unusable for large specifications.



(a) S (b) T

Figure 2.8: $[\![S]\!] \subseteq [\![T]\!]$ but $S \not\leq T$

The second problem with nondeterministic specifications is that operations are more difficult to define and have a higher complexity—we already saw it for thorough refinement. Consider for instance the quotient operation. We gave a definition of the quotient of deterministic modal specifications in Definition 16, based on the one in [Rac08]. The state space of this quotient is $(Q_1 \times Q_2) \cup \{q_\top\}$. As far as we know, the first definition of the quotient for nondeterministic modal specifications was given in [BDF+13]. The state space of this quotient is $2^{Q_1 \times Q_2}$, i.e., there is an exponential blow-up for the number of states. The authors add: "we conjecture that the exponential blow-up of the construction is in general unavoidable." Moreover, the quotient of nondeterministic modal specifications is not homogeneous: the result is a nondeterministic *disjunctive* modal specification.

In consequence, although nondeterministic specifications are more expressive, deterministic specifications offer some interesting properties, like a homogeneous quotient and the equivalence between thorough refinement and modal refinement, and the operations on these specifications are simpler to define and much more efficient on large systems.

Chapter 3

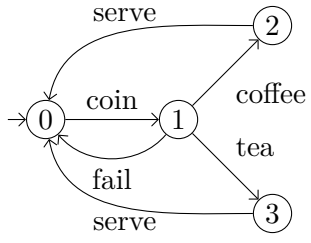
Acceptance Specifications and Convex Optimization

We now give a more detailed definition of acceptance specifications and show that this formalism is more expressive than other variants of modal specifications such as disjunctive modal specifications or modal specifications with obligations. Then, we define the operations of conjunction, product, and quotient on acceptance specifications. In Section 3.3, we introduce the first main contribution of this thesis: the definition of a subclass of acceptance specifications, convex-closed acceptance specifications, which allows defining more efficient operations, in particular for the quotient, while being still more expressive than disjunctive modal specifications or modal specifications with obligations. Finally, we give an overview of the Coq mechanization of the theorems given in this last section.

3.1 Semantics

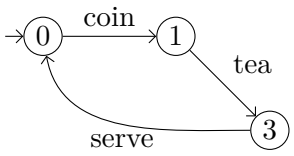
Acceptance trees have been introduced in [Hen85] as a way to represent nondeterministic trees with an underlying deterministic structure. A variant of acceptance trees adapted to automata has been considered in [Rac08] as a specification formalism, called acceptance specifications, which generalizes modal specifications. Instead of expressing two kinds of constraints on transitions—that they are allowed or required—acceptance specifications can express arbitrary constraints on which sets of transitions may be realized by the implementations. Note that the results presented in this section and the next one (Section 3.2) are essentially based on [Rac08].

Consider the example of acceptance specification depicted in Figure 3.1. It specifies the behavior of a coffee machine which waits for someone to put a coin and then offers coffee, tea, or both, or indicates a failure. Observe that there is no more two kinds of transitions, but that a set

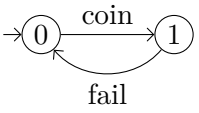


$$\begin{aligned} \mathrm{Acc}(0) &= \{\{\mathrm{coin}\}\} \\ \mathrm{Acc}(1) &= \{\{\mathrm{coffee}\}, \{\mathrm{tea}\}, \{\mathrm{coffee}, \mathrm{tea}\}, \{\mathrm{fail}\}\} \\ \mathrm{Acc}(2) &= \{\{\mathrm{serve}\}\} \\ \mathrm{Acc}(3) &= \{\{\mathrm{serve}\}\} \end{aligned}$$

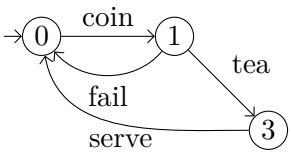
Figure 3.1: A specification of a coffee machine



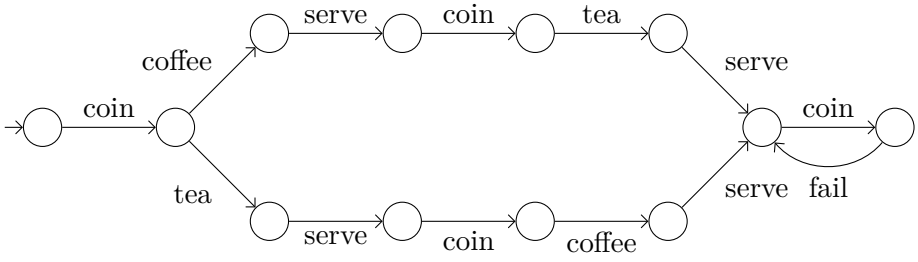
(a) A machine serving only tea



(b) A broken machine



(c) An invalid model



(d) Machine serving one cup of tea and one cup of coffee

Figure 3.2: Some models of the acceptance specification of Figure 3.1

is associated to each state. For states 0, 2 and 3, there is a unique singleton in the acceptance set, which is equivalent to a single must transition. For state 1 on the other hand, the acceptance set has four elements which means that when implementing the specification, one has to choose one of these elements and realize all the transitions it contains.

For example, when implementing the model in Figure 3.2(a), we selected the set of transitions {tea}, while we chose the set {fail} when implementing the model in Figure 3.2(b). On the other hand, the automaton in Figure 3.2(c) is not a model of the specification: from state 1, it has two transitions, $\{\text{tea}, \text{fail}\}$, and this set does not belong to the acceptance set of the corresponding state in the specification.

It is still possible to unfold a specification when implementing it in order to make different implementation choices in different states which correspond to the same state in the specification. For instance, the automaton of Figure 3.2(d) is a model of the specification which serves exactly one cup of tea and one cup of coffee, in an arbitrary order: if coffee is ordered first, it will then offer only tea and fail afterwards, while if tea is asked first, it will offer coffee before failing. The state 1 of the specification is implemented four times in the model, each implementation realizing a different element of the acceptance set.

The formal definition of acceptance specifications is similar to the definition of modal specifications with the may/must sets replaced by an acceptance set:

Definition 17 (Acceptance Specification)**.** *An acceptance specification over an alphabet Σ is a tuple $S = (Q, q^0, \delta, \text{Acc})$ where Q is a finite set of states, $q^0 \in Q$ is the unique initial state, $\delta : Q \times \Sigma \to Q$ is the partial labeled transition map, and $\text{Acc} : Q \to 2^{2^\Sigma}$ associates to each state a set of ready sets called its acceptance set.*

We also define a special empty acceptance specification $S_\perp$, which has no models.

The satisfaction relation between an automaton and an acceptance specification is defined as follows:

Definition 18 (Satisfaction)**.** *An automaton M* satisfies *an acceptance specification S, denoted $M \models S$, if and only if there exists a simulation relation $\pi \subseteq R \times Q$ such that $(r^0, q^0) \in \pi$ and, for all $(r, q) \in \pi$:*

- $\text{ready}(r) \in \text{Acc}(q)$ *and*
- *for any* $a \in \text{ready}(r)$*, we have* $(\lambda(r,a), \delta(q,a)) \in \pi$*.*

Observe that this definition is similar to Definition 3 of satisfaction for modal specifications with the may/must inclusions replaced by acceptance set membership.

Acceptance specifications are very expressive and are in particular more expressive than modal specifications and many variants such as disjunctive modal specifications and modal specifications with obligations. We give the constructions transforming these specifications into acceptance specifications:

Theorem 9. *Given a modal specification* S*, there exists an acceptance specification* S_{Acc} *such that* $[\![S]\!] = [\![S_{\text{Acc}}]\!]$*.*

Proof. If $S = (Q, q^0, \delta, \text{may}, \text{must})$, let $S_{\text{Acc}} = (Q, q^0, \delta, \text{Acc})$ where:

$$\text{Acc}(q) = \{X \mid \text{must}(q) \subseteq X \subseteq \text{may}(q)\}$$

We now prove that these two specifications have the same models.

($\Rightarrow$) Let M be a model of S. There is a simulation relation $\pi \subseteq R \times Q$. We prove that M is a model of S_{Acc} using the same simulation relation. We thus know by hypothesis that $(r^0, q^0) \in \pi$. For any $(r,q) \in \pi$ and $a \in \text{ready}(r)$:

- $\text{ready}(r) \in \text{Acc}(q)$: we know that $\text{must}(q) \subseteq \text{ready}(r) \subseteq \text{may}(q)$; thus $\text{ready}(r) \in \text{Acc}(q)$ by definition of Acc;
- $(\lambda(r,a), \delta(q,a)) \in \pi$ by hypothesis: S and S_{Acc} have the same transition map.

($\Leftarrow$) Let M be a model of S_{Acc}. There is a simulation relation $\pi \subseteq R \times Q$. We prove that M is a model of S using the same simulation relation. We thus know by hypothesis that $(r^0, q^0) \in \pi$. For any $(r,q) \in \pi$ and $a \in \text{ready}(r)$:

- $\text{must}(q) \subseteq \text{ready}(r) \subseteq \text{may}(q)$: we know that $\text{ready}(r) \in \text{Acc}(q)$; thus $\text{must}(q) \subseteq \text{ready}(r) \subseteq \text{may}(q)$ by definition of Acc;
- $(\lambda(r,a), \delta(q,a)) \in \pi$ by hypothesis: S and S_{Acc} have the same transition map. □

Theorem 10. *Given a disjunctive modal specification* S*, there exists an acceptance specification* S_{Acc} *such that* $[\![S]\!] = [\![S_{\text{Acc}}]\!]$*.*

Proof. If $S = (Q, q^0, \delta, \text{may}, \text{d-must})$, let $S_{\text{Acc}} = (Q, q^0, \delta, \text{Acc})$ where:

$$\text{Acc}(q) = \{X \mid X \subseteq \text{may}(q) \wedge \forall\, \text{must} \in \text{d-must}(q), X \cap \text{must} \neq \emptyset\}$$

We now prove that these two specifications have the same models.

($\Rightarrow$) Let M be a model of S. There is a simulation relation $\pi \subseteq R \times Q$. We prove that M is a model of S_{Acc} using the same simulation relation. We thus know by hypothesis that $(r^0, q^0) \in \pi$. For any $(r,q) \in \pi$ and $a \in \text{ready}(r)$:

- $\text{ready}(r) \in \text{Acc}(q)$: we know that $\text{ready}(r) \subseteq \text{may}(q)$ and for any $\text{must} \in \text{d-must}(q)$, $\text{ready}(r) \cap \text{must} \neq \emptyset$; thus $\text{ready}(r) \in \text{Acc}(q)$ by definition of Acc;
- $(\lambda(r,a), \delta(q,a)) \in \pi$ by hypothesis: S and S_{Acc} have the same transition map.

($\Leftarrow$) Let M be a model of S_{Acc}. There is a simulation relation $\pi \subseteq R \times Q$. We prove that M is a model of S using the same simulation relation. We thus know by hypothesis that $(r^0, q^0) \in \pi$. For any $(r, q) \in \pi$ and $a \in \text{ready}(r)$:

- $\text{ready}(r) \in \text{may}(q)$: we know that $\text{ready}(r) \in \text{Acc}(q)$ and by definition of Acc, $\text{ready}(r) \in \text{may}(q)$;
- $\forall \text{must} \in \text{d-must}(q), \text{ready}(r) \cap \text{must} \neq \emptyset$: we know that $\text{ready}(r) \in \text{Acc}(q)$ and conclude by definition of Acc;
- $(\lambda(r, a), \delta(q, a)) \in \pi$ by hypothesis: S and S_{Acc} have the same transition map. □

Theorem 11. *Given a modal specification with obligations S, there exists an acceptance specification S_{Acc} such that $[\![S]\!] = [\![S_{\text{Acc}}]\!]$.*

Proof. If $S = (Q, q^0, \delta, \Omega)$, let $S_{\text{Acc}} = (Q, q^0, \delta, \text{Acc})$ where:

$$\text{Acc}(q) = \begin{cases} \{X \mid X \in [\![\Omega(q)]\!] \wedge X \subseteq \text{ready}(q)\} & \text{if } [\![\Omega(q)]\!] \neq \emptyset \\ \{\emptyset\} & \text{if } [\![\Omega(q)]\!] = \emptyset \end{cases}$$

We now prove that these two specifications have the same models.

($\Rightarrow$) Let M be a model of S. There is a simulation relation $\pi \subseteq R \times Q$. We prove that M is a model of S_{Acc} using the same simulation relation. We thus know by hypothesis that $(r^0, q^0) \in \pi$. For any $(r, q) \in \pi$ and $a \in \text{ready}(r)$:

- $\text{ready}(r) \in \text{Acc}(q)$: if $[\![\Omega(q)]\!] = \emptyset$, $\text{ready}(r) = \emptyset$ by hypothesis and then $\text{ready}(r) \in \text{Acc}(q)$. Otherwise, $\text{ready}(r) \in [\![\Omega(q)]\!]$ by hypothesis. Thus $\text{ready}(r) \in \text{Acc}(q)$ by definition of Acc;
- $(\lambda(r, a), \delta(q, a)) \in \pi$ by hypothesis: S and S_{Acc} have the same transition map.

($\Leftarrow$) Let M be a model of S_{Acc}. There is a simulation relation $\pi \subseteq R \times Q$. We prove that M is a model of S using the same simulation relation. We thus know by hypothesis that $(r^0, q^0) \in \pi$. For any $(r, q) \in \pi$ and $a \in \text{ready}(r)$:

- $\text{ready}(r) \in [\![\Omega(q)]\!]$ or $\text{ready}(r) = [\![\Omega(q)]\!] = \emptyset$: by hypothesis, $\text{ready}(r) \in \text{Acc}(q)$. Either $[\![\Omega(q)]\!] = \emptyset$ and then $\text{ready}(r) \in \text{Acc}(q)$ implies $\text{ready}(r) = \emptyset$, or $\text{ready}(r) \in [\![\Omega(q)]\!]$ by definition of Acc;
- $(\lambda(r, a), \delta(q, a)) \in \pi$ by hypothesis: S and S_{Acc} have the same transition map. □

Note that these transformations to acceptance specifications have an exponential blow-up, since they enumerate all the allowed sets of actions (for instance all the sets between $\text{must}(q)$ and $\text{may}(q)$ for the modal case). We will address this inefficiency in Section 3.3.

There are some acceptance specifications that may not be represented by modal specifications, disjunctive modal specifications or modal specifications with obligations, such as the one of Figure 3.1. Indeed, in state 1 of this specification, there is a disjunction between the actions coffee and tea (i.e., there can be one, the other, or both), and an exclusive disjunction between these actions and the action fail. None of these three formalisms allow to express such constraints. Thus, acceptance specifications are strictly more expressive than these formalisms.

However, boolean modal specifications are expressive enough to be equivalent to acceptance specifications:

Theorem 12. *Given a boolean modal specification S, there exists an acceptance specification S_{Acc} such that $[\![S]\!] = [\![S_{\text{Acc}}]\!]$.*

Proof. The construction of the acceptance specification and the proof are the same as for modal specifications with obligations (Theorem 11) as the proof does not use any information specific to the logic used (i.e., it works for any logic as long as there is a function $[\![.]\!]$ generating a set of sets of actions and a similar definition of satisfaction). □

Theorem 13. *Given an acceptance specification S, there exists a boolean modal specification $S_{\mathcal{B}}$ such that $[\![S]\!] = [\![S_{\mathcal{B}}]\!]$.*

Proof. If $S = (Q, q^0, \delta, \text{Acc})$, let $S_{\mathcal{B}} = (Q, q^0, \delta, \Omega)$ where:

$$\Omega(q) = \bigoplus_{X \in \text{Acc}(q)} \left(\bigwedge_{a \in X} a \wedge \bigwedge_{a \notin X} \neg a \right)$$

with $\oplus$ the exclusive disjunction operation (i.e., $\varphi \oplus \psi = (\varphi \vee \psi) \wedge \neg(\varphi \wedge \psi)$).

We now prove that these two specifications have the same models.

($\Rightarrow$) Let M be a model of S. There is a simulation relation $\pi \subseteq R \times Q$. We prove that M is a model of $S_{\mathcal{B}}$ using the same simulation relation. We thus know by hypothesis that $(r^0, q^0) \in \pi$. For any $(r, q) \in \pi$ and $a \in \text{ready}(r)$:

- $\text{ready}(r) \in [\![\Omega(q)]\!]$: by hypothesis, $\text{ready}(r) \in \text{Acc}(q)$, thus $\text{ready}(r)$ satisfies $\Omega(q)$ for the element of the exclusive disjunction where $X = \text{ready}(q)$;
- $(\lambda(r, a), \delta(q, a)) \in \pi$ by hypothesis: S and $S_{\mathcal{B}}$ have the same transition map.

($\Leftarrow$) Let M be a model of $S_{\mathcal{B}}$. There is a simulation relation $\pi \subseteq R \times Q$. We prove that M is a model of S using the same simulation relation. We thus know by hypothesis that $(r^0, q^0) \in \pi$. For any $(r, q) \in \pi$ and $a \in \text{ready}(r)$:

- $\text{ready}(r) \in \text{Acc}(q)$: by hypothesis, $\text{ready}(r) \in [\![\Omega(q)]\!]$, so there is an $X \in \text{Acc}(q)$ such that the elements of $\text{ready}(r)$ are in X ($\bigwedge_{a \in X} a$) and the elements not in $\text{ready}(r)$ are not in X ($\bigwedge_{a \notin X} a$), thus $X = \text{ready}(r)$ and then $\text{ready}(r) \in \text{Acc}(q)$;
- $(\lambda(r, a), \delta(q, a)) \in \pi$ by hypothesis: S and $S_{\mathcal{B}}$ have the same transition map. □

We now define the refinement relation between two acceptance specifications. It is similar to the definition of refinement between modal specifications (Definition 5); inclusion of the acceptance sets replaces the inclusions of may and must sets.

Definition 19 (Refinement)**.** *Given two acceptance specifications S_1 and S_2, S_1 is a refinement of S_2, denoted $S_1 \leq S_2$, if and only if there exists a simulation relation $\pi \subseteq Q_1 \times Q_2$ such that $(q_1^0, q_2^0) \in \pi$ and for all pairs $(q_1, q_2) \in \pi$:*

- $\text{Acc}_1(q_1) \subseteq \text{Acc}_2(q_2)$ *and*
- *for any $a \in \text{ready}(q_1)$, we have: $(\delta_1(q_1, a), \delta_2(q_2, a)) \in \pi$.*

Moreover, for any specification S, $S_\perp \leq S$.

The refinement of acceptance specifications is also a thorough refinement: it is equivalent to the inclusion of the sets of models.

Theorem 14. *Given two acceptance specifications S_1 and S_2, $S_1 \leq S_2$ if and only if $[\![S_1]\!] \subseteq [\![S_2]\!]$.*

Proof. ($\Rightarrow$) Suppose that $S_1 \leq S_2$ and $M \models S_1$ thanks respectively to the simulation relations π and π_1. Define π_2 such that $(r, q_2) \in \pi_2$ if and only if there exists a state q_1 in S_1 such that $(r, q_1) \in \pi_1$ and $(q_1, q_2) \in \pi$. We prove that $M \models S_2$ thanks to π_2:

- if $(r, q_1) \in \pi_1$ then $\text{ready}(r) \in \text{Acc}_1(q_1)$ by Definition 18; moreover, if $(q_1, q_2) \in \pi$ then $\text{Acc}_1(q_1) \subseteq \text{Acc}_2(q_2)$ by Definition 19. As a result, $\text{ready}(r) \in \text{Acc}_2(q_2)$;
- for any $a \in \text{ready}(r)$, if $(r, q_1) \in \pi_1$ then $(\lambda(r, a), \delta_1(q_1, a)) \in \pi_1$ by Definition 18; moreover, if $(q_1, q_2) \in \pi$ then $(\delta_1(q_1, a), \delta_2(q_2, a)) \in \pi$ by Definition 19. As a result, we have: $(\lambda(r, a), \delta_2(q_2, a)) \in \pi_2$.

($\Leftarrow$) Suppose that $[\![S_1]\!] \subseteq [\![S_2]\!]$. Define π such that $(q_1^0, q_2^0) \in \pi$ and for all $(q_1, q_2) \in \pi$, if $\delta_1(q_1, a)$ and $\delta_2(q_2, a)$ are defined then $(\delta_1(q_1, a), \delta_2(q_2, a)) \in \pi$. We prove that $S_1 \leq S_2$ thanks to π.

Observe first that if $\delta_1(q_1, a)$ is defined then $\delta_2(q_2, a)$ is also defined; this is a direct consequence to the fact that when $\delta_1(q_1, a)$ is defined, the transition can be included in some models which are also models of S_2 and thus $\delta_2(q_2, a)$ is defined. Then, for any $(q_1, q_2) \in \pi$:

- for all $X \in \text{Acc}_1(q_1)$, there exists an $M \models S_1$ such that $(r, q_1) \in \pi_1$ and $\text{ready}(r) = X$. As $[\![S_1]\!] \subseteq [\![S_2]\!]$, M is also a model of S_2 and necessarily $\text{ready}(r) \in \text{Acc}_2(q_2)$. Consequently, $\text{Acc}_1(q_1) \subseteq \text{Acc}_2(q_2)$;
- by definition of π, for any $a \in \text{ready}(q_1)$, we have $(\delta_1(q_1, a), \delta_2(q_2, a))$.

As a result, according to Definition 19, we have $S_1 \leq S_2$. □

We saw that modal specifications could have inconsistent states, which allowed us to give simpler definitions to some operations and then apply a pruning operation in order to ensure a well-formedness property on modal specifications. Similarly, there may be some inconsistencies in acceptance specifications:

- Acc-*consistency.* A state q is Acc-consistent when $\text{Acc}(q) \neq \emptyset$.
- δ, Acc-*consistency.* A state q is δ, Acc-consistent when, for any action $a \in \Sigma$, $\delta(q, a)$ is defined if and only if there exists an $X \in \text{Acc}(q)$ such that $a \in X$, i.e., $\text{ready}(q) = \bigcup \text{Acc}(q)$.

Remark. It is easy to confuse $\text{Acc}(q) = \emptyset$ and $\text{Acc}(q) = \{\emptyset\}$, although these two acceptance sets have very different meanings. Assume that we have a model M of an acceptance specification S with a simulation relation π, and a state q of S.

If $\text{Acc}(q) = \emptyset$, q cannot belong to any pair of π since Definition 18 requires $\text{ready}(r) \in \text{Acc}(q)$, which is impossible when $\text{Acc}(q) = \emptyset$.

On the other hand, if $\text{Acc}(q) = \{\emptyset\}$, there may be a pair $(r, q) \in \pi$, which implies $\text{ready}(r) = \emptyset$, i.e., that there are no outgoing transitions from r.

Definition 20 (Normal form)**.** *An acceptance specification is in* normal form *if it is* Acc-*consistent and* δ, Acc-*consistent in every state q. Moreover, $S_\perp$ is in normal form.*

We demonstrate in Algorithm 1 how to remove the inconsistent states from an acceptance specification and we prove that the resulting specification is in normal form and has the same models as S:

Algorithm 1 $\rho(S\text{: AS})\text{: AS}$

1: **if** $\exists q, \mathrm{Acc}(q) = \emptyset$ **then**
2: **if** $q = q^0$ **then**
3: **return** $S_\perp$
4: **else**
5: $\delta' = \{(q', a) \mapsto \delta(q', a) \mid \delta(q', a) \text{ defined} \wedge \delta(q', a) \neq q\}$
6: $\mathrm{Acc}' = \{q' \mapsto \{X \mid X \in \mathrm{Acc}(q') \wedge \forall a \in X, \delta(q', a) \neq q\}\}$
7: **return** $\rho((Q \setminus \{q\}, q^0, \delta', \mathrm{Acc}'))$
8: **end if**
9: **end if**
10: **if** $\exists q, \mathrm{ready}(q) \neq \bigcup \mathrm{Acc}(q)$ **then**
11: $\delta' = \{(q', a) \mapsto \delta(q', a) \mid \delta(q', a) \text{ defined} \wedge a \in \bigcup \mathrm{Acc}(q')\}$
12: $\mathrm{Acc}' = \{q' \mapsto \{X \mid X \in \mathrm{Acc}(q') \wedge \forall a \in X, \delta(q, a) \text{ defined}\}\}$
13: **return** $\rho((Q, q^0, \delta', \mathrm{Acc}'))$
14: **end if**
15: **return** S

Theorem 15. *For any acceptance specification S, $\rho(S)$ is in normal form and is equivalent to S.*

Proof. (normal form) The base case of the recursive definition of ρ is that there is no state q such that $\mathrm{Acc}(q) = \emptyset$ or $\mathrm{ready}(q) \neq \bigcup \mathrm{Acc}(q)$. This implies that if ρ terminates, the returned specification is Acc-consistent and δ, Acc-consistent, hence in normal form. Each time the function ρ is recursively called, its parameter has fewer states, fewer transitions or smaller acceptance sets. Considering that acceptance specifications are finite, ρ is terminating.

(equivalence) By induction:

- In the base case (line 15), the specification S itself is returned.
- For the first recursive call (line 7), we remove from S the state q and the transitions from other states towards q. Since the acceptance set of q is empty, no model of S can implement q, so the specification passed to the recursive call has the same models as S.
- For the second recursive call (line 13), we removed some transitions which were not allowed by the corresponding acceptance set, and thus could not be realized by any model (the condition $\mathrm{ready}(r) \in \mathrm{Acc}(q)$ would not be satisfiable), as well as elements of the acceptance set containing actions for which δ is not defined, which could not be realized in any model either. Thus, the specification passed to the recursive call also has the same models as S. □

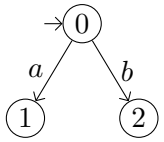
As a result of Theorem 15, from now on and without loss of generality, we assume that acceptance specifications are in normal form.

3.2 An Acceptance Specification Theory

We now show how the operations defined on modal specifications—namely conjunction, product, and quotient—can be extended to acceptance specifications.

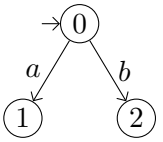
3.2.1 Conjunction

The conjunction of acceptance specifications is similar to the conjunction of modal specifications; computing the acceptance sets simply consists in keeping the common elements of the acceptance



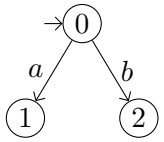
$\mathrm{Acc}(0) = \{\{a\}, \{a, b\}\}$
$\mathrm{Acc}(1) = \mathrm{Acc}(2) = \{\emptyset\}$

(a) S_1



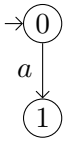
$\mathrm{Acc}(0) = \{\{a\}, \{b\}\}$
$\mathrm{Acc}(1) = \mathrm{Acc}(2) = \{\emptyset\}$

(b) S_2



$\mathrm{Acc}(0) = \{\{a\}\}$
$\mathrm{Acc}(1) = \mathrm{Acc}(2) = \{\emptyset\}$

(c) S_1 & S_2



$\mathrm{Acc}(0) = \{\{a\}\}$
$\mathrm{Acc}(1) = \mathrm{Acc}(2) = \{\emptyset\}$

(d) $S_1 \wedge S_2$

Figure 3.3: S_1 & S_2 may have inconsistencies

sets of the two operands, i.e., their intersection.

Definition 21 (Conjunction)**.** *Given two acceptance specifications S_1 and S_2, the conjunction of S_1 and S_2, denoted $S_1 \wedge S_2$, is the normal form of S_1 & $S_2 = (Q_1 \times Q_2, (q_1^0, q_2^0), \delta, \mathrm{Acc})$ with $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ when both $\delta_1(q_1, a)$ and $\delta_2(q_2, a)$ are defined, and $\mathrm{Acc}((q_1, q_2)) = \mathrm{Acc}_1(q_1) \cap \mathrm{Acc}_2(q_2)$.*

Remark. Computing the normal form is required as S_1 & S_2 may have inconsistencies, as depicted in Figure 3.3: the acceptance set of the initial state only contains an a while there are transitions by both a and b. Applying the cleaning operation removes the transition by b and gives us the conjunction in normal form.

Theorem 16. *Given two acceptance specifications S_1 and S_2, $[\![S_1 \wedge S_2]\!] = [\![S_1]\!] \cap [\![S_2]\!]$.*

Proof. ($\supseteq$) Assume that $M \models S_i$ thanks to π_i for $i = 1, 2$ and define π such that $(r, (q_1, q_2)) \in \pi$ if and only if $(r, q_i) \in \pi_i$. We show that $M \models S_1 \wedge S_2$ using π as simulation relation:

- $\mathrm{ready}(r) \in \mathrm{Acc}_i(q_i)$ as $(r, q_i) \in \pi_i$ and thus $\mathrm{ready}(r) \in \mathrm{Acc}((q_1, q_2))$ by definition of &;
- for any a and r' such that $\lambda(r, a) = r'$, $(r', \delta((q_1, q_2), a)) \in \pi$ is trivial as we know that $(r', \delta_i(q_i, a)) \in \pi_i$ and $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$.

($\subseteq$) Assume that $M \models S_1 \wedge S_2$ thanks to π and define π_i for $i = 1, 2$ such that $(r, q_i) \in \pi_i$ if and only if $(r, (q_1, q_2)) \in \pi$. We show that $M \models S_i$ using π_i as simulation relation:

- $\mathrm{ready}(r) \in \mathrm{Acc}_1(q_1) \cap \mathrm{Acc}_2(q_2)$ by definition of & and thus $\mathrm{ready}(r) \in \mathrm{Acc}_i(q_i)$;
- for any a and r' such that $\lambda(r, a) = r'$, $(r', \delta_i(q_i, a)) \in \pi_i$ is trivial as we know that $(r', \delta((q_1, q_2), a)) \in \pi$ and $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$. □

Corollary 2. *For any acceptance specifications S_1, S_2 and S, $S_1 \wedge S_2$ is the greatest lower bound of S_1 and S_2 for the refinement relation: $S \leq S_1$ and $S \leq S_2$ if and only if $S \leq S_1 \wedge S_2$.*

Proof. If $S \leq S_i$ for $i \in \{1, 2\}$ then, by Theorem 14, $[\![S]\!] \subseteq [\![S_i]\!]$. As a result, $[\![S]\!] \subseteq [\![S_1]\!] \cap [\![S_2]\!]$. By Theorem 16, this is equivalent to $[\![S]\!] \subseteq [\![S_1 \wedge S_2]\!]$. We deduce from Theorem 14 that $S \leq S_1 \wedge S_2$. □

3.2.2 Product

The product of acceptance specifications is built similarly to the product of modal specifications; the acceptance sets are made of the intersections of the elements of the acceptance sets of the operands, which matches the definition of automata product (the ready sets of the product are the intersection of the ready sets of the automata).

Definition 22 (Product)**.** *Given two acceptance specifications S_1 and S_2, their* product $S_1 \otimes S_2$ *is* $(Q_1 \times Q_2, (q_1^0, q_2^0), \delta, \mathrm{Acc})$ *with* $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ *when both* $\delta_1(q_1, a)$ *and* $\delta_2(q_2, a)$ *are defined and* $\mathrm{Acc}((q_1, q_2)) = \{A_1 \cap A_2 \mid A_1 \in \mathrm{Acc}_1(q_1) \wedge A_2 \in \mathrm{Acc}_2(q_2)\}$.

The product preserves normal form, so it is not necessary to prune the computed specification:

Proposition 1. *Given two acceptance specifications S_1 and S_2 (in normal form), the product of S_1 and S_2 is in normal form.*

Proof. (Acc-consistency) As S_1 and S_2 are in normal form, $\mathrm{Acc}_1(q_1)$ and $\mathrm{Acc}_2(q_2)$ are both non-empty. Thus, there exist some $A_1 \in \mathrm{Acc}_1(q_1)$ and $A_2 \in \mathrm{Acc}_2(q_2)$, and then $A_1 \cap A_2 \in \mathrm{Acc}((q_1, q_2))$, which is consequently non-empty.

(δ, *Acc*-consistency) For any action a:

$$\begin{aligned}
\exists A \in \mathrm{Acc}((q_1, q_2)), a \in A &\Leftrightarrow \exists A_1 \in \mathrm{Acc}_1(q_1), \exists A_2 \in \mathrm{Acc}_2(q_2), a \in A_1 \cap A_2 \\
&\Leftrightarrow (\exists A_1 \in \mathrm{Acc}_1(q_1), a \in A_1) \wedge (\exists A_2 \in \mathrm{Acc}_2(q_2), a \in A_2) \\
&\Leftrightarrow a \in \mathrm{ready}(q_1) \wedge a \in \mathrm{ready}(q_2) \\
&\Leftrightarrow \delta_1(q_1, a) \text{ defined and } \delta_2(q_2, a) \text{ defined} \\
&\Leftrightarrow \delta((q_1, q_2), a) \text{ defined}
\end{aligned}$$

□

Theorem 17. *Given two acceptance specifications S_1 and S_2, for any $M_1 \models S_1$ and $M_2 \models S_2$, $M_1 \times M_2 \models S_1 \otimes S_2$.*

Proof. Let π_i be the simulation relation of $M_i \models S_i$ for $i \in \{1, 2\}$ and π the simulation relation such that $((r_1, r_2), (q_1, q_2)) \in \pi$ if and only if (r_1, r_2) is reachable in $M_1 \times M_2$, $(r_1, q_1) \in \pi_1$ and $(r_2, q_2) \in \pi_2$. For any $((r_1, r_2), (q_1, q_2)) \in \pi$:

- $\mathrm{ready}((r_1, r_2)) = \mathrm{ready}(r_1) \cap \mathrm{ready}(r_2) \in \mathrm{Acc}(q_1, q_2)$ by definition of the acceptance set of the product;
- for any a, r_1' and r_2' such that $\lambda((r_1, r_2), a) = (r_1', r_2')$, $((r_1', r_2'), \delta((q_1, q_2), a)) \in \pi$ is trivial as $\lambda((r_1, r_2), a) = (\lambda_1(r_1, a), \lambda_2(r_2, a)) = (r_1', r_2')$.

□

Moreover, $S_1 \otimes S_2$ gives the most precise characterization of the behavior of the product of any models M_1 of S_1 and M_2 of S_2:

Theorem 18. *Given three acceptance specifications S_1, S_2, and S, if for all $M_1 \models S_1$ and $M_2 \models S_2$ we have $M_1 \times M_2 \models S$, then $S_1 \otimes S_2 \leq S$.*

Proof. By contradiction, assume that for any $M_1 \models S_1$ and $M_2 \models S_2$ we have $M_1 \times M_2 \models S$ but $S_1 \otimes S_2 \not\leq S$. Then, there exists an execution common to $\mathrm{Un}(S_1 \otimes S_2)$ and $\mathrm{Un}(S)$ leading to some state (q_1, q_2) in $S_1 \otimes S_2$ and q in S such that $\mathrm{Acc}(q_1, q_2) \not\subseteq \mathrm{Acc}(q)$ that is, there exists $A_1 \in \mathrm{Acc}_1(q_1)$ and $A_2 \in \mathrm{Acc}_2(q_2)$ such that $A_1 \cap A_2 \notin \mathrm{Acc}(q)$. Consider now M_i such that $(r_i, q_i) \in \pi_i$ and $\mathrm{ready}(r_i) = A_i$, for $i = 1, 2$, the product $M_1 \times M_2$ cannot be a model of S as $\mathrm{ready}(r_1, r_2) = A_1 \cap A_2 \notin \mathrm{Acc}(q)$ which contradicts the assumption made at the beginning of the proof. □

It is still possible to refine the operands of the product and have the guarantee that the product will be refined by the product of the refined specifications:

Theorem 19. *For any acceptance specifications S_1, S_1' and S_2, if $S_1' \leq S_1$ then $S_1' \otimes S_2 \leq S_1 \otimes S_2$.*

Proof. Let π_1 be the simulation relation of $S_1' \leq S_1$ and π the simulation relation such that $((q_1', q_2), (q_1, q_2)) \in \pi$ if and only if (q_1', q_2) is reachable in $S_1' \otimes S_2$ and $(q_1', q_1) \in \pi_1$. For any $((q_1', q_2), (q_1, q_2)) \in \pi$:

- Let A be an element of $\mathrm{Acc}((q_1', q_2))$. By definition of the acceptance set of the product, there exists $A_1' \in \mathrm{Acc}_1'(q_1')$ and $A_2 \in \mathrm{Acc}_2(q_2)$ such that $A = A_1' \cap A_2$. As $S_1' \leq S_1$, $A_1' \in \mathrm{Acc}_1(q_1)$ too, so $A = A_1' \cap A_2 \in \mathrm{Acc}((q_1, q_2))$, hence $\mathrm{Acc}((q_1', q_2)) \subseteq \mathrm{Acc}((q_1, q_2))$.
- For any a and q' such that $\delta((q_1', q_2), a) = q'$, $(q', \delta((q_1, q_2), a)) \in \pi$ is trivial as $\delta((q_1', q_2), a) = (\delta_1'(q_1', a), \delta_2(q_2, a))$ and $S_1' \leq S_1$. □

Finally, we prove the classical properties of commutativity and associativity:

Theorem 20. *Given two acceptance specifications S_1 and S_2, $S_1 \otimes S_2 \equiv S_2 \otimes S_1$.*

Proof. Two specifications are equivalent if they refine each other, i.e. $S_1 \otimes S_2 \leq S_2 \otimes S_1$ and $S_2 \otimes S_1 \leq S_1 \otimes S_2$. We will prove directly the equivalence by giving a simulation relation and proving that the acceptance sets are equal, rather than giving two symmetrical simulation relations and proving the inclusion in both directions.

Let π be the simulation relation such that for any pair of states (q_1, q_2) reachable in $S_1 \otimes S_2$, $((q_1, q_2), (q_2, q_1)) \in \pi$. It is clear that $((q_1^0, q_2^0), (q_2^0, q_1^0)) \in \pi$ and for any pair of states (q_1, q_2):

$$\begin{aligned}
\mathrm{Acc}_{S_1 \otimes S_2}((q_1, q_2)) &= \{A_1 \cap A_2 \mid A_1 \in \mathrm{Acc}_1(q_1) \wedge A_2 \in \mathrm{Acc}_2(q_2)\} \\
&= \{A_2 \cap A_1 \mid A_2 \in \mathrm{Acc}_2(q_2) \wedge A_1 \in \mathrm{Acc}_1(q_1)\} \\
&= \mathrm{Acc}_{S_2 \otimes S_1}((q_2, q_1))
\end{aligned}$$

$$(\delta_{S_1 \otimes S_2}((q_1, q_2), a), \delta_{S_2 \otimes S_1}((q_2, q_1), a) = ((\delta_1(q_1, a), \delta_2(q_2, a)), (\delta_2(q_2, a), \delta_1(q_1, a))) \in \pi \qquad \square$$

Theorem 21. *Given three acceptance specifications S_1, S_2 and S_3, $(S_1 \otimes S_2) \otimes S_3 \equiv S_1 \otimes (S_2 \otimes S_3)$.*

Proof. Let π be the simulation relation such that for any states $((q_1, q_2), q_3)$ reachable in $(S_1 \otimes S_2) \otimes S_3$, $(((q_1, q_2), q_3), (q_1, (q_2, q_3))) \in \pi$. It is clear that $(((q_1^0, q_2^0), q_3^0), (q_1^0, (q_2^0, q_3^0))) \in \pi$ and for any states q_1, q_2 and q_3:

$$\begin{aligned}
\mathrm{Acc}_{(S_1 \otimes S_2) \otimes S_3}(((q_1, q_2), q_3)) &= \{A_{1,2} \cap A_3 \mid A_{1,2} \in \mathrm{Acc}_{S_1 \otimes S_2}((q_1, q_2)) \wedge A_3 \in \mathrm{Acc}_3(q_3)\} \\
&= \{(A_1 \cap A_2) \cap A_3 \mid A_i \in \mathrm{Acc}_i(q_i) \text{ for } i \in \{1, 2, 3\}\} \\
&= \{A_1 \cap (A_2 \cap A_3) \mid A_i \in \mathrm{Acc}_i(q_i) \text{ for } i \in \{1, 2, 3\}\} \\
&= \{A_1 \cap A_{2,3} \mid A_1 \in \mathrm{Acc}_1(q_1) \wedge A_{2,3} \in \mathrm{Acc}_{S_2 \otimes S_3}((q_2, q_3))\} \\
&= \mathrm{Acc}_{S_1 \otimes (S_2 \otimes S_3)}((q_1, (q_2, q_3)))
\end{aligned}$$

$$(\delta_{(S_1\otimes S_2)\otimes S_3}(((q_1,q_2),q_3),a),\delta_{S_1\otimes(S_2\otimes S_3)}(q_1,(q_2,q_3)),a) = \\ (((\delta_1(q_1,a),\delta_2(q_2,a)),\delta_3(q_3,a)),((\delta_1(q_1,a),(\delta_2(q_2,a),\delta_3(q_3,a)))) \in \pi \qquad \square$$

3.2.3 Quotient

As for modal specifications, the quotient of acceptance specifications is meant to be the reciprocal function of product. Since the acceptance sets of a product are the intersections of the elements of the acceptance sets of the operands, the acceptance sets of the quotient will be all the sets which intersection with the elements of the denominator belong to the numerator.

Definition 23 (Quotient)**.** *Given two acceptance specifications S_1 and S_2, their quotient is the normal form of $((Q_1 \times Q_2) \cup \{q_\top\}, (q_1^0, q_2^0), \delta, \text{Acc})$ with:*

$$\text{Acc}((q_1,q_2)) = \{X \mid \forall X_2 \in Acc_2(q_2), X \cap X_2 \in Acc_1(q_1)\}$$
$$\text{Acc}(q_\top) = 2^\Sigma$$

and for all $a \in \bigcup Acc((q_1,q_2))$, δ is defined as:

$$\delta((q_1,q_2),a) = \begin{cases} (\delta_1(q_1,a),\delta_2(q_2,a)) & \textit{when both are defined} \\ q_\top & \textit{otherwise} \end{cases}$$
$$\delta(q_\top,a) = q_\top$$

This operation has an exponential blow-up w.r.t. the size of the alphabet: when computing an acceptance set, we have to enumerate all the $X \in 2^\Sigma$ and test if their intersection with all the elements of $\text{Acc}_2(q_2)$ is in $\text{Acc}_1(q_1)$. We will show in the next section how to avoid this blow-up using a particular subset of acceptance sets, while remaining highly expressive.

Theorem 22. *Given three acceptance specifications S, S_1 and S_2, $S \otimes S_2 \leq S_1$ if and only if $S \leq S_1/S_2$.*

Proof. ($\Rightarrow$) Assume that $S \otimes S_2 \leq S_1$ with a simulation relation $\pi_\otimes$. Let π be the simulation relation such that $(q,(q_1,q_2)) \in \pi$ if $((q,q_2),q_1) \in \pi_\otimes$ and $(q,q_\top) \in \pi$. It is clear that $(q^0,(q_1^0,q_2^0)) \in \pi$. For any $(q,(q_1,q_2)) \in \pi$:

- We want to prove that $\text{Acc}(q) \subseteq \text{Acc}_{S_1/S_2}((q_1,q_2))$. As $((q,q_2),q_1) \in \pi_\otimes$, $\text{Acc}_{S\otimes S_2}((q,q_2)) \subseteq \text{Acc}_1(q_1)$. Let $X \in \text{Acc}(q)$. For any $X_2 \in \text{Acc}_2(q_2)$, $X \cap X_2 \in \text{Acc}_{S\otimes S_2}((q,q_2))$ and thus $X \cap X_2 \in \text{Acc}_1(q_1)$. So, by definition of the quotient, $X \in \text{Acc}_{S_1/S_2}((q_1,q_2))$.
- For any a such that $\delta(q,a)$ is defined, if $\delta_1(q_1,a)$ and $\delta_2(q_2,a)$ are defined, $(\delta(q,a),(\delta_1(q_1,a),\delta_2(q_2,a))) \in \pi$ as $((\delta(q,a),\delta_2(q_2,a)),\delta_1(q_1,a)) \in \pi_\otimes$. Otherwise, $(\delta(q,a),q_\top) \in \pi$.

For any q, $(q,q_\top) \in \pi$ and trivially, $\text{Acc}(q) \subseteq \text{Acc}(q_\top)$ and for any a such that $\delta(q,a)$ is defined, $(\delta(q,a),q_\top) \in \pi$.

($\Leftarrow$) Assume that $S \leq S_1/S_2$ with a simulation relation $\pi_/$. Let π be the simulation relation such that $((q,q_2),q_1) \in \pi$ if $(q,(q_1,q_2)) \in \pi_/$. It is clear that $((q^0,q_2^0),q_1^0) \in \pi$. For any $((q,q_2),q_1) \in \pi$:

- For any $X \in \text{Acc}(q)$ and $X_2 \in \text{Acc}_2(q_2)$, $X \cap X_2 \in \text{Acc}_{S\otimes S_2}((q,q_2))$. We know that $\text{Acc}(q) \subseteq \text{Acc}_{S_1/S_2}((q_1,q_2))$ and by definition of the quotient, we deduce that $X \cap X_2 \in \text{Acc}_1(q_1)$. Hence, $\text{Acc}_{S\otimes S_2}((q,q_2)) \subseteq \text{Acc}_1(q_1)$.
- For any a such that $\delta_{S\otimes S_2}((q,q_2))$ is defined, $((\delta(q,a),\delta_2(q_2,a)),\delta_1(q_1,a)) \in \pi$ as $(\delta(q,a),(\delta_1(q_1,a),\delta_2(q_2,a))) \in \pi_/$. $\square$

3.2.4 Dissimilar alphabets

Until now, we only considered specifications defined on a same alphabet Σ. When building large systems from many components, these components are typically not defined on a same alphabet: each one only handles a small set of actions related to the task it must perform. Then, we want to be able to merge or compose these various subsystems to build more complex systems, which requires to adapt the operations defined previously so that they handle correctly the differences in the alphabets of their operands. An approach to solve this, presented for modal specifications in [RBB+11], is to first *extend* each specification so that all the operands of an operation are defined on the same alphabet. This allows to only define some alphabet extension functions and then reuse the operations defined earlier rather than having to reimplement all these operations to handle internally the dissimilar alphabets.

Assume that we have two alphabets Σ and Σ' such that $\Sigma \subseteq \Sigma'$ and an acceptance specification S defined on the alphabet Σ. How can we extend S so that it is defined on the alphabet Σ'? The main idea is to add some self-transitions labeled by the actions in the set $\Sigma' \setminus \Sigma$. Then, these transitions may allow to *synchronize* with other specifications while preserving the behavior of the original specification since these transitions will lead to the same state. We also have to extend the acceptance sets accordingly; otherwise, the generated specification would be inconsistent. There are different ways to add the actions to the acceptance sets. A first method is simply to add the actions to each element of the acceptance sets, i.e:

$$\mathrm{Acc}'(q) = \{X \cup (\Sigma' \setminus \Sigma) \mid X \in \mathrm{Acc}(q)\}$$

This is called *strong extension* as all the models of the new specification are required to realize all the transitions in $\Sigma' \setminus \Sigma$. Another method is to only allow the transitions, which then may or may not be realized by the implementations, i.e.:

$$\mathrm{Acc}'(q) = \{X \cup \sigma \mid X \in \mathrm{Acc}(q) \wedge \sigma \subseteq \Sigma' \setminus \Sigma\}$$

We call this *weak extension*. These two different extensions are actually both useful: we will see later on that in some cases we need a strong extension while we need the weak one in other cases.

We first define the extension of an automaton. Since automata have no modalities, there is a single extension that adds the transitions in $\Sigma' \setminus \Sigma$:

Definition 24. *Given two alphabets Σ and Σ' such that $\Sigma \subseteq \Sigma'$ and an automaton M on the alphabet Σ, we define the* extension $M_{\uparrow\Sigma'}$ *of M to Σ' as the automaton $(R, r^0, \lambda_\uparrow)$ where:*

$$\lambda_\uparrow(r,a) = \begin{cases} \lambda(r,a) & \text{if } a \in \Sigma \text{ and } \lambda(r,a) \text{ is defined} \\ r & \text{if } a \in \Sigma' \setminus \Sigma \end{cases}$$

Definition 25. *Given two alphabets Σ and Σ' such that $\Sigma \subseteq \Sigma'$ and an acceptance specification S on the alphabet Σ, we define the* weak extension $S_{\Uparrow\Sigma'}$ *of S to Σ' as the acceptance specification $(Q, q^0, \delta_{\Uparrow/\uparrow}, \mathrm{Acc}_\Uparrow)$ and the* strong extension $S_{\uparrow\Sigma'}$ *of S to Σ' as the acceptance specification $(Q, q^0, \delta_{\Uparrow/\uparrow}, \mathrm{Acc}_\uparrow)$ where $\delta_{\Uparrow/\uparrow}$ is given by the extension of the underlying automaton (see Definition 24) and:*

$$\mathrm{Acc}_\Uparrow(q) = \{X \cup \sigma \mid X \in \mathrm{Acc}(q) \wedge \sigma \subseteq \Sigma' \setminus \Sigma\}$$

$$\mathrm{Acc}_\uparrow(q) = \{X \cup (\Sigma' \setminus \Sigma) \mid X \in \mathrm{Acc}(q)\}$$

Note that there is an exponential blow-up in the weak extension since it requires to enumerate

all the subsets of $\Sigma' \setminus \Sigma$. We will show in the next section that this blow-up can be completely removed with a particular subclass of acceptance sets.

In order to manipulate acceptance specifications with dissimilar alphabets, we also need to extend the satisfaction and refinement relations with weak and strong alphabet extensions:

Definition 26. *Given two alphabets Σ_S and Σ_M such that $\Sigma_S \subseteq \Sigma_M$, an acceptance specification S over Σ_S, and an automaton M over Σ_M:*

- *M weakly satisfies S, denoted $M \models_w S$ if and only if $M \models S_{\Uparrow \Sigma_M}$;*
- *M strongly satisfies S, denoted $M \models_s S$ if and only if $M \models S_{\uparrow \Sigma_M}$.*

These two extensions of the satisfaction relation are related since the strong satisfaction relation is a subset of the weak one:

Theorem 23. *Given two alphabets Σ_S and Σ_M such that $\Sigma_S \subseteq \Sigma_M$, an acceptance specification S over Σ_S, and an automaton M over Σ_M such that $M \models_s S$, then $M \models_w S$.*

Proof. Assume that $M \models_s S$ with a simulation relation π. $M \models_w S$ with the same simulation relation. For any $(r, q) \in \pi$:

- We know that $\text{ready}(r) \in \text{Acc}_{\uparrow \Sigma_M}(q)$ and thus that there exists an $X \in \text{Acc}(q)$ such that $\text{ready}(r) = X \cup (\Sigma_M \setminus \Sigma_S)$. In consequence, $\text{ready}(r) \in \text{Acc}_{\Uparrow \Sigma_M}$ (with $\sigma = \Sigma_M \setminus \Sigma_S$).
- For any $a \in \text{ready}(r)$, $(\lambda(r, a), \delta(q, a)) \in \pi$ by hypothesis. □

Moreover, extending an automaton preserves the satisfaction relation:

Theorem 24. *Given two alphabets Σ and Σ' such that $\Sigma \subseteq \Sigma'$, an automaton M over Σ, and an acceptance specification S over Σ, the following statements are equivalent:*

$$M \models S \quad \Leftrightarrow \quad M_{\uparrow \Sigma'} \models_s S \quad \Leftrightarrow \quad M_{\uparrow \Sigma'} \models_w S$$

Proof. ($M \models S \Rightarrow M_{\uparrow \Sigma'} \models_s S$) Assume that $M \models S$ with a satisfaction relation π. We prove that $M_{\uparrow \Sigma'} \models S_{\uparrow \Sigma'}$ using the same relation. For any $(r, q) \in \pi$:

- We know that $\text{ready}(r) \in \text{Acc}(q)$. By definition of the extension of automata, $\text{ready}_{\uparrow \Sigma'}(r) = \text{ready}(r) \cup (\Sigma' \setminus \Sigma)$. Since $\text{Acc}_{\uparrow \Sigma'}(q) = \{X \cup (\Sigma' \setminus \Sigma') \mid X \in \text{Acc}(q)$, we conclude that $\text{ready}_{\uparrow \Sigma'}(r) \in \text{Acc}_{\uparrow \Sigma'}(q)$.
- For any $a \in \text{ready}_{\uparrow \Sigma'}(r)$, there are two cases:
  - $a \in \Sigma$: then $\lambda_{\uparrow \Sigma'}(r, a) = \lambda(r, a)$, $\delta_{\uparrow \Sigma'}(q, a) = \delta(q, a)$ and we know by definition of π that $(\lambda(r, a), \delta(q, a)) \in \pi$.
  - $a \in \Sigma' \setminus \Sigma$: then $\lambda_{\uparrow \Sigma'}(r, a) = r$, $\delta \uparrow \Sigma'(q, a) = q$ and we know by hypothesis that $(r, q) \in \pi$.

($M_{\uparrow \Sigma'} \models_s S \Rightarrow M_{\uparrow \Sigma'} \models_w S$) Using the previous result and Theorem 23, we find $M_{\uparrow \Sigma'} \models_w S$.

($M_{\uparrow \Sigma'} \models_w S \Rightarrow M \models S$) Assume that $M_{\uparrow \Sigma'} \models_w S$ with a satisfaction relation π. We prove that $M \models S$ using the same relation. For any $(r, q) \in \pi$:

- We know that $\text{ready}_{\uparrow\Sigma'}(r) \in \text{Acc}_{\Uparrow\Sigma'}(q)$. By definition, this is equivalent to $\text{ready}(r) \cup (\Sigma' \setminus \Sigma) \in \{X \cup \sigma \mid X \in \text{Acc}(q) \wedge \sigma \subseteq \Sigma' \setminus \Sigma\}$. Thus there exist an $X \in \text{Acc}(q)$ and a $\sigma \subseteq \Sigma' \setminus \Sigma$ such that $\text{ready}(r) \cup (\Sigma' \setminus \Sigma) = X \cup \sigma$. Since $\text{ready}(r)$ and X are subsets of Σ and σ contains no elements of Σ, $\text{ready}(r) = X$ and thus $\text{ready}(r) \in \text{Acc}(q)$.
- For any $a \in \text{ready}(r)$, we know that $(\lambda(r,a), \delta(r,a)) \in \pi$ because $\lambda_{\uparrow\Sigma'}(r,a) = \lambda(r,a)$ and $\delta_{\uparrow\Sigma'}(q,a) = \delta(q,a)$. □

We define weak and strong refinements similarly, and prove that strong refinement implies weak refinement:

Definition 27. *Given two alphabets Σ_1 and Σ_2 such that $\Sigma_1 \subseteq \Sigma_2$ and two acceptance specifications S_1 and S_2 over respectively Σ_1 and Σ_2:*

- *S_2 weakly refines S_1, denoted $S_2 \leq_w S_1$ if and only if $S_2 \leq S_{1\Uparrow\Sigma_2}$;*
- *S_2 strongly refines S_1, denoted $S_2 \leq_s S_1$ if and only if $S_2 \leq S_{1\uparrow\Sigma_2}$.*

Theorem 25. *Given two alphabets Σ_1 and Σ_2 such that $\Sigma_1 \subseteq \Sigma_2$ and two acceptance specifications S_1 and S_2 over respectively Σ_1 and Σ_2 such that $S_2 \leq_s S_1$, then $S_2 \leq_w S_1$.*

Proof. Assume that $S_2 \leq_s S_1$ with a simulation relation π. $S_2 \leq_w S_1$ with the same simulation relation. For any $(q_2, q_1) \in \pi$:

- We know that $\text{Acc}_2(q_2) \subseteq \text{Acc}_{1\uparrow S_2}(q_1)$ and thus that for any $X_2 \in \text{Acc}_2(q_2)$, there exists an $X_1 \in \text{Acc}_1(q_1)$ such that $X_2 = X_1 \cup (\Sigma_2 \setminus \Sigma_1)$. Then, $X_2 \in \text{Acc}_{1\Uparrow\Sigma_2}(q_1)$ (with $\sigma = \Sigma_2 \setminus \Sigma_1$) and so $\text{Acc}_2(q_2) \subseteq \text{Acc}_{1\Uparrow S_2}(q_1)$.
- For any $a \in \text{ready}_2(q_2)$, $(\delta_2(q_2,a), \delta_1(q_1,a)) \in \pi$ by hypothesis. □

Moreover, weak and strong refinement are thorough refinements:

Theorem 26. *Given two alphabets Σ_1 and Σ_2 such that $\Sigma_1 \subseteq \Sigma_2$ and two acceptance specifications S_1 and S_2 over respectively Σ_1 and Σ_2, $S_2 \leq_w S_1$ if and only if for any alphabet Σ such that $\Sigma_2 \subseteq \Sigma$ and for any automaton M over Σ such that $M \models_w S_2$, $M \models_w S_1$.*

Proof. ($\Rightarrow$) Assume that $S_2 \leq S_{1\Uparrow\Sigma_2}$ with a simulation relation $\pi_\leq$. Let Σ be a superset of Σ_2 and M a model of $S_{2\Uparrow\Sigma}$ with a simulation relation π_2. We prove that $M \models S_{1\Uparrow\Sigma}$ using a simulation relation π_1 defined as: $(r, q_1) \in \pi_1$ if and only if there exists a q_2 such that $(r, q_2) \in \pi_2$ and $(q_2, q_1) \in \pi_\leq$. For any $(r, q_1) \in \pi_1$:

- We know that $\text{ready}(r) \in \text{Acc}_{2\Uparrow\Sigma}(q_2)$, so there exist an $X_2 \in \text{Acc}_2(q_2)$ and a $\sigma_2 \subseteq \Sigma \setminus \Sigma_2$ such that $\text{ready}(r) = X_2 \cup \sigma_2$. Moreover, $\text{Acc}_2(q_2) \subseteq \text{Acc}_{1\Uparrow\Sigma_2}(q_1)$, so there exist an $X_1 \in \text{Acc}_1(q_1)$ and a $\sigma_1 \subseteq \Sigma_2 \setminus \Sigma_1$ such that $X_2 = X_1 \cup \sigma_1$. Consequently, $\text{ready}(r) \in \text{Acc}_{1\Uparrow\Sigma}(q_1)$ (with $\sigma = \sigma_1 \cup \sigma_2$).
- For any $a \in \text{ready}(r)$, there are three possibilities:
  - $a \in \Sigma_1$: $\delta_{1\Uparrow\Sigma}(q_1,a) = \delta_1(q_1,a)$, $\delta_{2\Uparrow\Sigma}(q_2,a) = \delta_2(q_2,a)$; by definition, $(\lambda(r,a), \delta_2(q_2,a)) \in \pi_2$ and $(\delta_2(q_2,a), \delta_1(q_1,a)) \in \pi_\leq$, so $(\lambda(r,a), \delta_1(q_1,a)) \in \pi_1$;
  - $a \in \Sigma_2 \setminus \Sigma_1$: $\delta_{1\Uparrow\Sigma}(q_1,a) = q_1$, $\delta_{2\Uparrow\Sigma}(q_2,a) = \delta_2(q_2,a)$; by definition, $(\lambda(r,a), \delta_2(q_2,a)) \in \pi_2$ and $(\delta_2(q_2,a), q_1) \in \pi_\leq$, so $(\lambda(r,a), q_1)) \in \pi_1$;

– $a \in \Sigma \setminus \Sigma_2$: $\delta_{1\Uparrow\Sigma}(q_1, a) = q_1$, $\delta_{2\Uparrow\Sigma}(q_2, a) = q_2$; by definition, $(\lambda(r, a), q_2) \in \pi_2$ and $(q_2, q_1) \in \pi_\leq$ by hypothesis, so $(\lambda(r, a), q_1)) \in \pi_1$.

($\Leftarrow$) We know by hypothesis, when $\Sigma = \Sigma_2$, that for any model $M \models S_2$, $M \models S_{1\Uparrow\Sigma_2}$. Since refinement is thorough (Theorem 14), $S_2 \leq S_{1\Uparrow\Sigma_2}$, i.e. $S_2 \leq_w S_1$. □

Theorem 27. *Given two alphabets Σ_1 and Σ_2 such that $\Sigma_1 \subseteq \Sigma_2$ and two acceptance specifications S_1 and S_2 over respectively Σ_1 and Σ_2, $S_2 \leq_s S_1$ if and only if for any alphabet Σ such that $\Sigma_2 \subseteq \Sigma$ and for any automaton M over Σ such that $M \models_s S_2$, $M \models_s S_1$.*

Proof. ($\Rightarrow$) Assume that $S_2 \leq S_{1\uparrow\Sigma_2}$ with a simulation relation $\pi_\leq$. Let Σ be a superset of Σ_2 and M a model of $S_{2\uparrow\Sigma}$ with a simulation relation π_2. We prove that $M \models S_{1\uparrow\Sigma}$ using a simulation relation π_1 defined as: $(r, q_1) \in \pi_1$ if and only if there exists a q_2 such that $(r, q_2) \in \pi_2$ and $(q_2, q_1) \in \pi_\leq$. For any $(r, q_1) \in \pi_1$:

- We know that $\text{ready}(r) \in \text{Acc}_{2\uparrow\Sigma}(q_2)$, so there exists an $X_2 \in \text{Acc}_2(q_2)$ such that $\text{ready}(r) = X_2 \cup (\Sigma \setminus \Sigma_2)$. Moreover, $\text{Acc}_2(q_2) \subseteq \text{Acc}_{1\uparrow\Sigma_2}(q_1)$, so there exists an $X_1 \in \text{Acc}_1(q_1)$ such that $X_2 = X_1 \cup (\Sigma_2 \setminus \Sigma_1)$. Consequently, $\text{ready}(r) \in \text{Acc}_{1\uparrow\Sigma}(q_1)$.
- For any $a \in \text{ready}(r)$, there are three possibilities:
  - $a \in \Sigma_1$: $\delta_{1\uparrow\Sigma}(q_1, a) = \delta_1(q_1, a)$, $\delta_{2\uparrow\Sigma}(q_2, a) = \delta_2(q_2, a)$; by definition, $(\lambda(r, a), \delta_2(q_2, a)) \in \pi_2$ and $(\delta_2(q_2, a), \delta_1(q_1, a)) \in \pi_\leq$, so $(\lambda(r, a), \delta_1(q_1, a)) \in \pi_1$;
  - $a \in \Sigma_2 \setminus \Sigma_1$: $\delta_{1\uparrow\Sigma}(q_1, a) = q_1$, $\delta_{2\uparrow\Sigma}(q_2, a) = \delta_2(q_2, a)$; by definition, $(\lambda(r, a), \delta_2(q_2, a)) \in \pi_2$ and $(\delta_2(q_2, a), q_1) \in \pi_\leq$, so $(\lambda(r, a), q_1)) \in \pi_1$;
  - $a \in \Sigma \setminus \Sigma_2$: $\delta_{1\uparrow\Sigma}(q_1, a) = q_1$, $\delta_{2\uparrow\Sigma}(q_2, a) = q_2$; by definition, $(\lambda(r, a), q_2) \in \pi_2$ and $(q_2, q_1) \in \pi_\leq$ by hypothesis, so $(\lambda(r, a), q_1)) \in \pi_1$.

($\Leftarrow$) We know by hypothesis, when $\Sigma = \Sigma_2$, that for any model $M \models S_2$, $M \models S_{1\uparrow\Sigma_2}$. Since refinement is thorough (Theorem 14), $S_2 \leq S_{1\uparrow\Sigma_2}$, i.e. $S_2 \leq_s S_1$. □

We will now see how to use weak and strong extensions to define conjunction, product, and quotient operations on acceptance specifications with dissimilar alphabets.

Let us first consider conjunction. Since the acceptance set of the conjunction is the intersection of the acceptance sets of its operands, we have to use weak extensions in order to preserve the requirements of each operand (2^Σ is the identity element of intersection). We first prove that weak extension is distributive over conjunction:

Lemma 1. *Given two alphabets Σ and Σ' such that $\Sigma \subseteq \Sigma'$ and two acceptance specifications S_1 and S_2 over Σ, $(S_1 \wedge S_2)_{\Uparrow\Sigma'} \equiv S_{1\Uparrow\Sigma'} \wedge S_{2\Uparrow\Sigma'}$.*

Proof. Let π be the simulation relation such that for any pair of states (q_1, q_2) reachable in $(S_1 \wedge S_2)_{\Uparrow\Sigma'}$, $((q_1, q_2), (q_1, q_2)) \in \pi$. For any $((q_1, q_2), (q_1, q_2)) \in \pi$:

- $$\begin{aligned} &\text{Acc}_{(S_1 \wedge S_2)_{\Uparrow\Sigma'}}((q_1, q_2)) \\ &= \{X \cup \sigma \mid X \in \text{Acc}_{S_1 \wedge S_2}((q_1, q_2)) \wedge \sigma \subseteq \Sigma' \setminus \Sigma\} \\ &= \{X \cup \sigma \mid X \in \text{Acc}_1(q_1) \wedge X \in \text{Acc}_2(q_2) \wedge \sigma \subseteq \Sigma' \setminus \Sigma\} \\ &= \{X \cup \sigma \mid X \in \text{Acc}_1(q_1) \wedge \sigma \subseteq \Sigma' \setminus \Sigma\} \cap \{X \cup \sigma \mid X \in \text{Acc}_2(q_2) \wedge \sigma \subseteq \Sigma' \setminus \Sigma\} \\ &= \text{Acc}_{S_{1\Uparrow\Sigma'} \wedge S_{2\Uparrow\Sigma'}}((q_1, q_2)) \end{aligned}$$

- For any a, q_1' and q_2' such that $\delta_{(S_1 \wedge S_2)_{\Uparrow\Sigma'}}((q_1,q_2),a) = (q_1',q_2')$, $\delta_{S_{1\Uparrow\Sigma'} \wedge S_{2\Uparrow\Sigma'}}((q_1,q_2),a)$ is defined and there are two cases:
  - $a \in \Sigma$: $(q_1',q_2') = \delta_{S_1 \wedge S_2}((q_1,q_2),a) = (\delta_1(q_1,a),\delta_2(q_2,a))$ and $\delta_{S_{1\Uparrow\Sigma'} \wedge S_{2\Uparrow\Sigma'}}((q_1,q_2),a) = (\delta_{S_{1\Uparrow\Sigma'}}(q_1,a),\delta_{S_{2\Uparrow\Sigma'}}(q_2,a)) = (q_1',q_2')$. By definition of π, $((q_1',q_2'),(q_1',q_2')) \in \pi$.
  - $a \in \Sigma' \setminus \Sigma$: $(q_1',q_2') = (q_1,q_2)$ and $\delta_{S_{1\Uparrow\Sigma'} \wedge S_{2\Uparrow\Sigma'}}((q_1,q_2),a) = (\delta_{S_{1\Uparrow\Sigma'}}(q_1,a),\delta_{S_{2\Uparrow\Sigma'}}(q_2,a)) = (q_1,q_2)$. We know by hypothesis that $((q_1,q_2),(q_1,q_2)) \in \pi$. □

Then, we can prove that the extension of the conjunction operation to acceptance specifications with dissimilar alphabets characterizes the intersection of their sets of models:

Theorem 28. *Given three alphabets Σ_1, Σ_2 and Σ such that $\Sigma_1 \cup \Sigma_2 \subseteq \Sigma$, two acceptance specifications S_1 and S_2 over Σ_1 and Σ_2, and an automaton M over Σ, $M \models_w S_{1\Uparrow\Sigma_1\cup\Sigma_2} \wedge S_{2\Uparrow\Sigma_1\cup\Sigma_2}$ if and only if $M \models_w S_1$ and $M \models_w S_2$.*

Proof.

$$
\begin{aligned}
& M \models_w S_{1\Uparrow\Sigma_1\cup\Sigma_2} \wedge S_{2\Uparrow\Sigma_1\cup\Sigma_2} && \\
\Leftrightarrow\ & M \models (S_{1\Uparrow\Sigma_1\cup\Sigma_2} \wedge S_{2\Uparrow\Sigma_1\cup\Sigma_2})_{\Uparrow\Sigma} && \text{by definition of } \models_w \\
\Leftrightarrow\ & M \models S_{1\Uparrow\Sigma} \wedge S_{2\Uparrow\Sigma} && \text{by Lemma 1} \\
\Leftrightarrow\ & M \models S_{1\Uparrow\Sigma} \wedge M \models S_{2\Uparrow\Sigma} && \text{by Theorem 16} \\
\Leftrightarrow\ & M \models_w S_1 \wedge M \models_w S_2 && \text{by definition of } \models_w
\end{aligned}
$$
□

On the other hand, we have to use strong extensions for product: adding the missing transitions to the existing elements of the acceptance sets ensures that their intersection with elements of the acceptance set of the other specification contains both common actions and actions belonging exclusively to one of the alphabets. As for conjunction, we first prove that strong extension is distributive over product.

Lemma 2. *Given two alphabets Σ and Σ' such that $\Sigma \subseteq \Sigma'$ and two acceptance specifications S_1 and S_2 over Σ, $(S_1 \otimes S_2)_{\uparrow\Sigma'} \equiv S_{1\uparrow\Sigma'} \otimes S_{2\uparrow\Sigma'}$.*

Proof. Let π be the simulation relation such that for any pair of states (q_1,q_2) reachable in $(S_1 \otimes S_2)_{\uparrow\Sigma'}$, $((q_1,q_2),(q_1,q_2)) \in \pi$. For any $((q_1,q_2),(q_1,q_2)) \in \pi$:

- $$
  \begin{aligned}
  & \mathrm{Acc}_{(S_1\otimes S_2)_{\uparrow\Sigma'}}((q_1,q_2)) \\
  &= \{A \cup (\Sigma' \setminus \Sigma) \mid A \in \mathrm{Acc}_{S_1\otimes S_2}((q_1,q_2))\} \\
  &= \{(A_1 \cap A_2) \cup (\Sigma' \setminus \Sigma) \mid A_1 \in \mathrm{Acc}_1(q_1) \wedge A_2 \in \mathrm{Acc}_2(q_2)\} \\
  &= \{(A_1 \cup (\Sigma' \setminus \Sigma)) \cap (A_2 \cup (\Sigma' \setminus \Sigma)) \mid A_1 \in \mathrm{Acc}_1(q_1) \wedge A_2 \in \mathrm{Acc}_2(q_2)\} \\
  &= \{A_1 \cap A_2 \mid A_1 \in \mathrm{Acc}_{S_{1\uparrow\Sigma'}}(q_1) \wedge A_2 \in \mathrm{Acc}_{S_{2\uparrow\Sigma'}}(q_2)\} \\
  &= \mathrm{Acc}_{S_{1\uparrow\Sigma'}\otimes S_{2\uparrow\Sigma'}}((q_1,q_2))
  \end{aligned}
  $$
- For any a, q_1' and q_2' such that $\delta_{(S_1\otimes S_2)_{\uparrow\Sigma'}}((q_1,q_2),a) = (q_1',q_2')$, $\delta_{S_{1\uparrow\Sigma'}\otimes S_{2\uparrow\Sigma'}}((q_1,q_2),a)$ is defined and there are two cases:
  - $a \in \Sigma$: $(q_1',q_2') = \delta_{S_1\otimes S_2}((q_1,q_2),a) = (\delta_1(q_1,a),\delta_2(q_2,a))$ and $\delta_{S_{1\uparrow\Sigma'}\otimes S_{2\uparrow\Sigma'}}((q_1,q_2),a) = (\delta_{S_{1\uparrow\Sigma'}}(q_1,a),\delta_{S_{2\uparrow\Sigma'}}(q_2,a)) = (q_1',q_2')$. By definition of π, $((q_1',q_2'),(q_1',q_2')) \in \pi$.
  - $a \in \Sigma' \setminus \Sigma$: $(q_1',q_2') = (q_1,q_2)$ and $\delta_{S_{1\uparrow\Sigma'}\otimes S_{2\uparrow\Sigma'}}((q_1,q_2),a) = (\delta_{S_{1\uparrow\Sigma'}}(q_1,a),\delta_{S_{2\uparrow\Sigma'}}(q_2,a)) = (q_1,q_2)$. We know by hypothesis that $((q_1,q_2),(q_1,q_2)) \in \pi$. □

Then, we extend the two theorems proving that the product is sound (Theorem 17) and optimal (Theorem 18) to the product of acceptance specifications with dissimilar alphabets:

Theorem 29. *Given four alphabets Σ_{M_1}, Σ_{S_1}, Σ_{M_2}, and Σ_{S_2} such that $\Sigma_{S_1} \subseteq \Sigma_{M_1}$ and $\Sigma_{S_2} \subseteq \Sigma_{M_2}$, two acceptance specifications S_1 and S_2 over Σ_{S_1} and Σ_{S_2}, and two automata M_1 and M_2 over Σ_{M_1} and Σ_{M_2} such that $M_1 \models_s S_1$ and $M_2 \models_s S_2$, then $M_{1\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \times M_{2\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \models_s S_{1\uparrow\Sigma_{S_1}\cup\Sigma_{S_2}} \otimes S_{2\uparrow\Sigma_{S_1}\cup\Sigma_{S_2}}$.*

Proof.

$$
\begin{aligned}
& M_1 \models_s S_1 \wedge M_2 \models_s S_2 \\
\Rightarrow\ & M_1 \models S_{1\uparrow\Sigma_{M_1}} \wedge M_2 \models S_{2\uparrow\Sigma_{M_2}} && \text{by definition of } \models_s \\
\Rightarrow\ & M_{1\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \models S_{1\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \wedge M_{2\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \models S_{2\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} && \text{by Theorem 24} \\
\Rightarrow\ & M_{1\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \times M_{2\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \models S_{1\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \otimes S_{2\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} && \text{by Theorem 17} \\
\Rightarrow\ & M_{1\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \times M_{2\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \models (S_{1\uparrow\Sigma_{S_1}\cup\Sigma_{S_2}} \otimes S_{2\uparrow\Sigma_{S_1}\cup\Sigma_{S_2}})_{\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} && \text{by Lemma 2} \\
\Rightarrow\ & M_{1\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \times M_{2\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \models_s S_{1\uparrow\Sigma_{S_1}\cup\Sigma_{S_2}} \otimes S_{2\uparrow\Sigma_{S_1}\cup\Sigma_{S_2}} && \text{by definition of } \models_s
\end{aligned}
$$

□

Lemma 3. *Given three alphabets Σ_1, Σ_2, and Σ such that $\Sigma_2 \subseteq \Sigma_1 \subseteq \Sigma$ and two acceptance specifications S_1 and S_2 over Σ_1 and Σ_2 such that $S_{1\uparrow\Sigma} \leq_s S_2$, then $S_1 \leq_s S_2$.*

Proof. Assume that $S_{1\uparrow\Sigma} \leq S_{2\uparrow\Sigma}$ with a simulation relation π. We prove that $S_1 \leq S_{2\uparrow\Sigma_1}$ using the same relation. For any $(q_1, q_2) \in \pi$:

- We know that $\mathrm{Acc}_{1\uparrow\Sigma}(q_1) \subseteq \mathrm{Acc}_{2\uparrow\Sigma}(q_2)$. Thus, for any $X_1 \in \mathrm{Acc}_1(q_1)$, there exists an $X_2 \in \mathrm{Acc}_2(q_2)$ such that $X_1 \cup (\Sigma \setminus \Sigma_1) = X_2 \cup (\Sigma \setminus \Sigma_2) = X_2 \cup (\Sigma \setminus \Sigma_1) \cup (\Sigma_1 \setminus \Sigma_2)$. Moreover, we know that $X_1 \subseteq \Sigma_1$ and $X_2 \subseteq \Sigma_2$, so we can conclude that $X_1 = X_2 \cup (\Sigma_1 \setminus \Sigma_2)$ and then $\mathrm{Acc}_1(q_1) \subseteq \mathrm{Acc}_{2\uparrow\Sigma_2}(q_2)$.
- For any a such that $\delta_1(q_1, a)$ is defined, $\delta_1(q_1, a) = \delta_{1\uparrow\Sigma}(q_1, a)$ and $\delta_{2\uparrow\Sigma_1}(q_2, a) = \delta_{2\uparrow\Sigma}(q_2, a)$, so $(\delta_1(q_1, a), \delta_{2\uparrow\Sigma_1}(q_2, a)) \in \pi$. □

Theorem 30. *Given three alphabets Σ_{S_1}, Σ_{S_2}, and Σ such that $\Sigma \subseteq \Sigma_{S_1} \cup \Sigma_{S_2}$, and three acceptance specifications S_1, S_2, and S over Σ_{S_1}, Σ_{S_2}, and Σ, if for all alphabets Σ_{M_1} and Σ_{M_2} such that $\Sigma_{S_1} \subseteq \Sigma_{M_1}$ and $\Sigma_{S_2} \subseteq \Sigma_{M_2}$, and for all automata M_1 and M_2 over Σ_{M_1} and Σ_{M_2} such that $M_1 \models_s S_1$ and $M_2 \models_s S_2$ we have $M_{1\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \times M_{2\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \models_s S$, then $S_{1\uparrow\Sigma_{S_1}\cup\Sigma_{S_2}} \otimes S_{2\uparrow\Sigma_{S_1}\cup\Sigma_{S_2}} \leq_s S$.*

Proof.

$$
\begin{aligned}
& \forall M_1 \models_s S_1, \forall M_2 \models_s S_2, M_{1\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \times M_{2\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \models_s S \\
\Rightarrow\ & \text{by definition of } \models_s \text{ and Theorem 24} \\
& \forall M_{1\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \models S_{1\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}}, \forall M_{2\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \models S_{2\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}}, \\
& \quad M_{1\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \times M_{2\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \models S_{\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \\
\Rightarrow\ & \text{by Theorem 18} \\
& S_{1\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \otimes S_{2\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \leq S_{\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \\
\Rightarrow\ & \text{by Lemma 2} \\
& (S_{1\uparrow\Sigma_{S_1}\cup\Sigma_{S_2}} \otimes S_{2\uparrow\Sigma_{S_1}\cup\Sigma_{S_2}})_{\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \leq S_{\uparrow\Sigma_{M_1}\cup\Sigma_{M_2}} \\
\Rightarrow\ & \text{by Lemma 3} \\
& S_{1\uparrow\Sigma_{S_1}\cup\Sigma_{S_2}} \otimes S_{2\uparrow\Sigma_{S_1}\cup\Sigma_{S_2}} \leq_s S
\end{aligned}
$$

□

The last operation to consider is quotient. We have to use both extensions: the weak one for the numerator and the strong one for the denominator.

Lemma 4. *Given two alphabets Σ and Σ' such that $\Sigma \subseteq \Sigma'$ and two acceptance specifications S_1 and S_2 over Σ, $(S_1/S_2)_{\Uparrow\Sigma'} \equiv S_{1\Uparrow\Sigma'}/S_{2\uparrow\Sigma'}$.*

Proof. Let π be the simulation relation such that for any pair of states (q_1, q_2) reachable in $(S_1/S_2)_{\Uparrow\Sigma'}$, $((q_1, q_2), (q_1, q_2)) \in \pi$. For any $((q_1, q_2), (q_1, q_2)) \in \pi$:

- $$\begin{aligned}
&\mathrm{Acc}_{(S_1/S_2)_{\Uparrow\Sigma'}}((q_1, q_2)) \\
&= \{X \cup \sigma \mid X \subseteq \Sigma \wedge \forall X_2 \in \mathrm{Acc}_2(q_2), X \cap X_2 \in \mathrm{Acc}_1(q_1) \wedge \sigma \subseteq \Sigma' \setminus \Sigma\} \\
&= \{X \mid X \subseteq \Sigma' \wedge \forall X_2 \in \mathrm{Acc}_2(q_2), X \cap X_2 \in \mathrm{Acc}_1(q_1)\} \\
&= \{X \mid X \subseteq \Sigma' \wedge \forall X_2 \in \mathrm{Acc}_2(q_2), (X \cap X_2) \cup (X \cap (\Sigma' \setminus \Sigma)) \in \mathrm{Acc}_{S_{1\Uparrow\Sigma'}}(q_1)\} \\
&= \{X \mid X \subseteq \Sigma' \wedge \forall X_2 \in \mathrm{Acc}_2(q_2), X \cap (X_2 \cup (\Sigma' \setminus \Sigma)) \in \mathrm{Acc}_{S_{1\Uparrow\Sigma'}}(q_1)\} \\
&= \mathrm{Acc}_{S_{1\Uparrow\Sigma'}/S_{2\uparrow\Sigma'}}((q_1, q_2))
\end{aligned}$$
- For any a, q_1' and q_2' such that $\delta_{(S_1/S_2)_{\Uparrow\Sigma'}}((q_1, q_2), a) = (q_1', q_2')$, $\delta_{S_{1\Uparrow\Sigma'}/S_{2\uparrow\Sigma'}}((q_1, q_2), a)$ is defined and there are two cases:
  - $a \in \Sigma$: $(q_1', q_2') = \delta_{S_1/S_2}((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ and $\delta_{S_{1\Uparrow\Sigma'}/S_{2\uparrow\Sigma'}}((q_1, q_2), a) = (\delta_{S_{1\Uparrow\Sigma'}}(q_1, a), \delta_{S_{2\uparrow\Sigma'}}(q_2, a)) = (q_1', q_2')$. By definition of π, $((q_1', q_2'), (q_1', q_2')) \in \pi$.
  - $a \in \Sigma' \setminus \Sigma$: $(q_1', q_2') = (q_1, q_2)$ and $\delta_{S_{1\Uparrow\Sigma'}/S_{2\uparrow\Sigma'}}((q_1, q_2), a) = (\delta_{S_{1\Uparrow\Sigma'}}(q_1, a), \delta_{S_{2\uparrow\Sigma'}}(q_2, a)) = (q_1, q_2)$. We know by hypothesis that $((q_1, q_2), (q_1, q_2)) \in \pi$. □

Theorem 31. *Given three alphabets Σ_1, Σ_2, and Σ such that $\Sigma_1 \cup \Sigma_2 \subseteq \Sigma$ and three acceptance specifications S_1, S_2, and S over Σ_1, Σ_2, and Σ, $S \otimes S_{2\uparrow\Sigma} \leq_w S_1$ if and only if $S \leq_w S_{1\Uparrow\Sigma_1\cup\Sigma_2}/S_{2\uparrow\Sigma_1\cup\Sigma_2}$.*

Proof.

$$\begin{aligned}
& S \otimes S_{2\uparrow\Sigma} \leq_w S_1 && \\
\Leftrightarrow\ & S \otimes S_{2\uparrow\Sigma} \leq S_{1\Uparrow\Sigma} && \text{by definition of } \leq_w \\
\Leftrightarrow\ & S \leq S_{1\Uparrow\Sigma}/S_{2\uparrow\Sigma} && \text{by Theorem 22} \\
\Leftrightarrow\ & S \leq (S_{1\Uparrow\Sigma_1\cup\Sigma_2}/S_{2\uparrow\Sigma_1\cup\Sigma_2})_{\Uparrow\Sigma} && \text{by Lemma 4} \\
\Leftrightarrow\ & S \leq_w S_{1\Uparrow\Sigma_1\cup\Sigma_2}/S_{2\uparrow\Sigma_1\cup\Sigma_2} && \text{by definition of } \leq_w
\end{aligned}$$ □

3.3 Convex Acceptance Specifications

We now introduce the first main contribution of this thesis. While acceptance specifications offer a very high expressiveness compared to modal or disjunctive specifications, this may come with a cost in terms of complexity: converting a modal or disjunctive specification into an acceptance specification or computing a quotient have an exponential blow-up w.r.t. the size of the alphabet. In order to mitigate this increased complexity while keeping a high expressiveness, we introduce an optimized subset of acceptance sets called convex-closed acceptance sets. These sets, although less expressive than acceptance sets, are still expressive enough to represent the constraints of modal or disjunctive specifications while avoiding the exponential blow-ups of the operations on acceptance sets.

We first show how these sets are represented, in Section 3.3.1, then we will see how to optimize various operations on acceptance specifications using these convex-closed sets, in Sections 3.3.2, 3.3.3, 3.3.4, and 3.3.5. Since many proofs are quite technical and involve many set operations,

we proved the theorems on convex-closed sets using the Coq proof assistant, as discussed in Section 3.3.6. We conclude with a short discussion in Section 3.3.7 about the possible extension of our results to nondeterministic acceptance specifications.

Note that we focus in this section on operations on acceptance sets and show how to optimize them using the convexity hypothesis. We do not give a definition of operations on convex acceptance specifications as their definition is the same as for acceptance specification; the optimization only resides in the implementation of the operations on acceptance sets (e.g., inclusion for refinement-checking and intersection for conjunction).

We also give the complexity of the operations on both acceptance sets and convex-closed acceptance sets. To do so, we count the number of set operations applied to sets of actions. We consider that acceptance sets are essentially lists of sets without any particular ordering property. Using more complex data structures, such as some kind of balanced tree, may reduce the complexity of some operations in practice (for instance transforming a $O(|\operatorname{Acc}|)$ in a $O(\log(|\operatorname{Acc}|))$). Later on, in Section 5.3, we will present several data structures that can be used to represent acceptance sets and give some experimental results.

3.3.1 Semantics

We first define the sub-class of *convex-closed acceptance set*:

Definition 28 (Convex-closed set)**.** *An acceptance set* Acc *is said to be* convex-closed *if for all* $X, Y \in \operatorname{Acc}$ *and* Z *such that* $X \subseteq Z \subseteq Y$ *we have* $Z \in \operatorname{Acc}$*.*

Then, given a convex-closed acceptance set, we can represent it in an optimized way. Instead of keeping all its elements, it suffices to have the minimum and maximum elements (by inclusion): we know that all the sets in-between them also belong to the set.

Definition 29. *The minimal and maximal elements of an acceptance set* Acc *are:*

$$\begin{cases} \min(\operatorname{Acc}) &= \{X \mid X \in \operatorname{Acc} \wedge \forall Y \in \operatorname{Acc}, Y \subseteq X \rightarrow Y = X\} \\ \max(\operatorname{Acc}) &= \{X \mid X \in \operatorname{Acc} \wedge \forall Y \in \operatorname{Acc}, X \subseteq Y \rightarrow Y = X\} \end{cases}$$

Definition 30 (Interval)**.** *Given two sets* X *and* Y *such that* $X \subseteq Y$*, we call* interval *the acceptance set formed by all the sets* Z *such that* $X \subseteq Z \subseteq Y$*. We denote it* $[X, Y]$*.*

An interval is convex-closed by definition. Then, we can represent any convex-closed acceptance set Acc by its minimal and maximal elements. We denote them Acc^- and Acc^+. Then, we can compute the corresponding acceptance set as a union of intervals made of the elements of Acc^- and Acc^+:

Theorem 32. *For any convex-closed acceptance set* Acc*:*

$$\operatorname{Acc} = \bigcup_{X_m \in \operatorname{Acc}^-} \bigcup_{\substack{Y_M \in \operatorname{Acc}^+ \\ X_m \subseteq Y_M}} [X_m, Y_M]$$

In order to lighten the notations, we will write:

$$\bigcup_{(X_m, Y_M)} f([X_m, Y_M])$$

instead of:

$$\bigcup_{X_m \in \mathrm{Acc}^-} \bigcup_{\substack{Y_M \in \mathrm{Acc}^+ \\ X_m \subseteq Y_M}} f([X_m, Y_M])$$

and likewise for $\bigcap$.

Proof. ($\Rightarrow$) Let X be an element of Acc. There is an $X_m \in \mathrm{Acc}^-$ such that $X_m \subseteq X$ and a $Y_M \in \mathrm{Acc}^+$ such that $X \subseteq Y_M$. By transitivity $X_m \subseteq Y_M$ and by definition $X \in [X_m, Y_M]$, so:

$$X \in \bigcup_{(X_m, Y_M)} [X_m, Y_M]$$

($\Leftarrow$) Assume that :

$$X \in \bigcup_{(X_m, Y_M)} [X_m, Y_M]$$

Then there exist an $X_m \in \mathrm{Acc}^-$ and a $Y_M \in \mathrm{Acc}^+$ such that $X_m \subseteq Y_M$ and $X \in [X_m, Y_M]$. Acc^- and Acc^+ are subsets of Acc, $X_m \subseteq X \subseteq Y_M$ by definition of $[X_m, Y_M]$ and Acc is convex-closed, so $X \in \mathrm{Acc}$. □

This representation allows us to efficiently encode modal specifications: instead of enumerating all the sets between the must and may sets in order to obtain an acceptance set, we can use a convex acceptance set with $\mathrm{Acc}^- = \{\text{must}\}$ and $\mathrm{Acc}^+ = \{\text{may}\}$.

It can also be used to represent many acceptance sets of non-modal specifications, such as the one in Figure 3.1:

$$\mathrm{Acc}^-(0) = \mathrm{Acc}^+(0) = \{\{\text{coin}\}\}$$

$$\mathrm{Acc}^-(1) = \{\{\text{coffee}\}, \{\text{tea}\}, \{\text{fail}\}\} \qquad \mathrm{Acc}^+(1) = \{\{\text{coffee}, \text{tea}\}, \{\text{fail}\}\}$$

$$\mathrm{Acc}^-(2) = \mathrm{Acc}^+(2) = \mathrm{Acc}^-(3) = \mathrm{Acc}^+(3) = \{\{\text{serve}\}\}$$

However, note that the if a convex-closed acceptance set is described by two sets A^- and A^+, these sets do not have to contain only minimal and maximal elements:

Theorem 33. *Given two arbitrary sets A^- and A^+:*

$$\bigcup_{X_m \in A^-} \bigcup_{\substack{Y_M \in A^+ \\ X_m \subseteq Y_M}} [X_m, Y_M] \quad = \quad \bigcup_{X_m \in \min(A^-)} \bigcup_{\substack{Y_M \in \max(A^+) \\ X_m \subseteq Y_M}} [X_m, Y_M]$$

Proof. ($\Rightarrow$) Let X be an element of:

$$\bigcup_{X_m \in A^-} \bigcup_{\substack{Y_M \in A^+ \\ X_m \subseteq Y_M}} [X_m, Y_M]$$

There exist an $X_m \in A^-$ and a $Y_M \in A^+$ such that $X_m \subseteq X \subseteq Y_M$. According to the definition of min, there exists an $X_m^- \in \min(A^-)$ such that $X_m^- \subseteq X_m$. Similarly, there is a $Y_M^+ \in \max(A^+)$ such that $Y_M \subseteq Y_M^+$. Therefore, $X_m^- \subseteq X \subseteq Y_M^+$ and:

$$X \in \bigcup_{X_m \in \min(A^-)} \bigcup_{\substack{Y_M \in \max(A^+) \\ X_m \subseteq Y_M}} [X_m, Y_M]$$

($\Leftarrow$) Let X be an element of:

$$\bigcup_{X_m \in \min(A^-)} \bigcup_{\substack{Y_M \in \max(A^+) \\ X_m \subseteq Y_M}} [X_m, Y_M]$$

Since $\min(A^-)$ returns a subset of A^- and $\max(A^+)$ a subset of A^+, we can conclude that:

$$X \in \bigcup_{X_m \in A^-} \bigcup_{\substack{Y_M \in A^+ \\ X_m \subseteq Y_M}} [X_m, Y_M]$$

□

As a consequence, if an operation returns a convex-closed acceptance set, it does not need to ensure that the sets of *minimal* and *maximal* elements only contain actual minimal and maximal elements: superfluous values have no influence on the corresponding acceptance set and removing them with min/max is merely an optimization to reduce the size of the sets.

However, observe that even when the sets only contain minimal and maximal elements, they may still have some superfluous elements. For example, take $\mathrm{Acc}^- = \{\{a\}, \{b\}\}$ and $\mathrm{Acc}^+ = \{\{a\}, \{c\}\}$. The element $\{b\} \in \mathrm{Acc}^-$ is useless as there is no $Y_M \in \mathrm{Acc}^+$ such that $\{b\} \subseteq Y_M$. Hence, we can't form any interval with it and the convex-closed set $\mathrm{Acc}^- = \{\{a\}\}$, $\mathrm{Acc}^+ = \{\{a\}, \{c\}\}$ has exactly the same elements. Similarly, the element $\{c\} \in \mathrm{Acc}^+$ can be removed as there is no $X_m \in \mathrm{Acc}^-$ such that $X_m \subseteq \{c\}$. So, this convex-closed set is equivalent to $\mathrm{Acc}^- = \mathrm{Acc}^+ = \{\{a\}\}$.

Definition 31 (Normal form)**.** *Given a convex-closed acceptance set, its normal form is the convex-closed acceptance set:*

$$\begin{cases} \mathrm{Acc}^-_{nf} &= \min(\{X_m \mid X_m \in \mathrm{Acc}^- \wedge \exists Y_M \in \mathrm{Acc}^+, X_m \subseteq Y_M\}) \\ \mathrm{Acc}^+_{nf} &= \max(\{Y_M \mid Y_M \in \mathrm{Acc}^+ \wedge \exists X_m \in \mathrm{Acc}^-, X_m \subseteq Y_M\}) \end{cases}$$

Theorem 34. *A convex-closed acceptance set* Acc *and its normal form* Acc_{nf} *represent the same acceptance set, i.e.* $\bigcup_{(X_m, Y_M)}[X_m, Y_M] = \bigcup_{(X_{mnf}, Y_{Mnf})}[X_{mnf}, Y_{Mnf}]$.

Proof. ($\Rightarrow$) Let $X \in \bigcup_{(X_m, Y_M)}[X_m, Y_M]$. There exist $X_m \in \mathrm{Acc}^-$ and $Y_M \in \mathrm{Acc}^+$ such that $X_m \subseteq X \subseteq Y_M$. As $X_m \subseteq Y_M$, $X_m \in \{X \mid X \in \mathrm{Acc}^- \wedge \exists Y_M \in \mathrm{Acc}^+, X \subseteq Y_M\}$ and $Y_M \in \{Y \mid Y \in \mathrm{Acc}^+ \wedge \exists X_m \in \mathrm{Acc}^-, X_m \subseteq Y\}$. Then, there is a minimal $X_{m\mathrm{nf}} \subseteq X_m$ and a maximal $Y_{M\mathrm{nf}} \supseteq Y_M$, hence $X \in \bigcup_{(X_{m\mathrm{nf}}, Y_{M\mathrm{nf}})}[X_{m\mathrm{nf}}, Y_{M\mathrm{nf}}]$.

($\Leftarrow$) $\mathrm{Acc}^-_{\mathrm{nf}} \subseteq \mathrm{Acc}^-$ and $\mathrm{Acc}^+_{\mathrm{nf}} \subseteq \mathrm{Acc}^+$, so any $X \in \bigcup_{(X_{m\mathrm{nf}}, Y_{M\mathrm{nf}})}[X_{m\mathrm{nf}}, Y_{M\mathrm{nf}}]$ also belongs to $\bigcup_{(X_m, Y_M)}[X_m, Y_M]$. □

Remark. Contrary to the normal form of modal or acceptance specifications that is required for some operations (for example, if there is an inconsistency between the acceptance sets and the transition function, it may be impossible to apply some operations because a $\delta(q, a)$ will not be defined while it should be), the normal form of convex-closed acceptance sets is merely an optimization: it makes the Acc^- and Acc^+ sets smaller by removing their useless elements, but the operations that we define on convex-closed acceptance sets should work well with any set, in normal form or not.

When checking if an acceptance specification refines another, we must verify that the acceptance set of the refinement is included in the acceptance set of the refined, as described in Definition 19. This is easily done on convex-closed acceptance sets using only the minimal and maximal elements:

Theorem 35 (Inclusion of convex-closed sets)**.** *If* Acc_1 *and* Acc_2 *are convex-closed, then:*

$$\mathrm{Acc}_1 \subseteq \mathrm{Acc}_2 \Leftrightarrow \begin{cases} \forall X_{m1} \in \mathrm{Acc}_1^-, \exists X_{m2} \in \mathrm{Acc}_2^-, X_{m2} \subseteq X_{m1} \text{ and} \\ \forall Y_{M1} \in \mathrm{Acc}_1^+, \exists Y_{M2} \in \mathrm{Acc}_2^+, Y_{M1} \subseteq Y_{M2} \end{cases}$$

Proof. ($\Rightarrow$) Suppose $\mathrm{Acc}_1 \subseteq \mathrm{Acc}_2$. Let $X_{m1} \in \mathrm{Acc}_1^-$ and $Y_{M1} \in \mathrm{Acc}_1^+$ then we also have $X_{m1}, Y_{M1} \in \mathrm{Acc}_2$. As a result, there exist $X_{m2} \in \mathrm{Acc}_2^-$ and $Y_{M2} \in \mathrm{Acc}_2^+$ such that $X_{m2} \subseteq X_{m1}$ and $Y_{M1} \subseteq Y_{M2}$.

($\Leftarrow$) Suppose that for all $X_{m1} \in \mathrm{Acc}_1^-$ and $Y_{M1} \in \mathrm{Acc}_1^+$, there exist $X_{m2} \in \mathrm{Acc}_2^-$ and $Y_{M2} \in \mathrm{Acc}_2^-$ with $X_{m2} \subseteq X_{m1}$ and $Y_{M1} \subseteq Y_{M2}$. Given $X \in \mathrm{Acc}_1$, $X \in [X_{m1}, Y_{M1}]$ thus, $X \in [X_{m2}, Y_{M2}]$ and $X \in \mathrm{Acc}_2$. □

Complexity 1. Given two acceptance sets Acc_1 and Acc_2, the complexity of testing if Acc_1 is a subset of Acc_2 is $O(|\,\mathrm{Acc}_1\,| \times |\,\mathrm{Acc}_2\,|)$. The formula given in Theorem 35 has a complexity of $O(|\,\mathrm{Acc}_1^-\,| \times |\,\mathrm{Acc}_2^-\,| + |\,\mathrm{Acc}_1^+\,| \times |\,\mathrm{Acc}_2^+\,|)$. It should be faster in general since the minimal and maximal sets are typically smaller than the full acceptance set. In the worst case, if $\mathrm{Acc}^- = \mathrm{Acc}^+ = \mathrm{Acc}$, both tests have the same complexity, with a factor 2 for the test on the minimal and maximal sets.

We demonstrated in Section 3.1 transformations from various extensions of modal specifications into acceptance specifications. We show that these transformations actually generate convex-closed acceptance sets except for boolean modal specifications. We also define a more efficient way to build the acceptance sets by directly computing their minimal and maximal elements, avoiding the exponential blow-up caused by generating acceptance sets.

Theorem 36. *Given a modal specification* S *and a state* q *of* S*, the acceptance set:*

$$\mathrm{Acc} = \{X \mid \mathrm{must}(q) \subseteq X \subseteq \mathrm{may}(q)\}$$

is convex-closed.

Proof. Let X and Y be two elements of Acc and Z a set such that $X \subseteq Z \subseteq Y$. By definition of Acc, we know that $\mathrm{must}(q) \subseteq X$ and $Y \subseteq \mathrm{may}(q)$. Thus, by transitivity, $\mathrm{must}(q) \subseteq Z \subseteq \mathrm{may}(q)$ and so $Z \in \mathrm{Acc}$. □

Theorem 37. *Given a modal specification* S *and a state* q *of* S*, the convex-closed acceptance set given by:*

$$\begin{cases} \mathrm{Acc}^- = \{\mathrm{must}(q)\} \\ \mathrm{Acc}^+ = \{\mathrm{may}(q)\} \end{cases}$$

is equivalent to the acceptance set:

$$\{X \mid \mathrm{must}(q) \subseteq X \subseteq \mathrm{may}(q)\}$$

Proof. Since Acc^- and Acc^+ are singletons, the convex-closed set they express is the interval $[\mathrm{must}(q), \mathrm{may}(q)]$ which by Definition 30 is equal to $\{X \mid \mathrm{must}(q) \subseteq X \subseteq \mathrm{may}(q)\}$. □

As a consequence, while generating an acceptance specification from a modal specification involves an exponential blow-up (to compute all the sets between $\mathrm{must}(q)$ and $\mathrm{may}(q)$), we can generate a convex acceptance specification without this blow-up by only expressing the minimal and maximal elements of the acceptance sets which are directly given by the may and must sets.

We now prove that the acceptance specifications obtained from disjunctive modal specifications or modal specifications with obligations also have convex-closed acceptance sets.

Theorem 38. *Given a disjunctive modal specification S and a state q of S, the acceptance set:*

$$\text{Acc} = \{X \mid X \subseteq \text{may}(q) \wedge \forall\, \text{must} \in \text{d-must}(q), X \cap \text{must} \neq \emptyset\}$$

is convex-closed.

Proof. Let X and Y be two elements of Acc and Z a set such that $X \subseteq Z \subseteq Y$. By definition of Acc, $Y \subseteq \text{may}(q)$ and by transitivity, $Z \subseteq \text{may}(q)$. By definition of Acc, for all must $\in$ d-must(q), $X \cap \text{must} \neq \emptyset$; as $X \subseteq Z$, we can deduce that for all must $\in$ d-must(q), $Z \cap \text{must} \neq \emptyset$. Thus, $Z \in$ Acc. □

Lemma 5. *Given a positive boolean formula φ and $X \in [\![\varphi]\!]$, for any set Y such that $X \subseteq Y$, $Y \in [\![\varphi]\!]$.*

Proof. By induction on φ:

- if $\varphi = a$, $[\![\varphi]\!] = \{X \mid a \in X\}$. As $X \in [\![\varphi]\!]$, $a \in X$. Thus, $a \in Y$ and then $Y \in [\![\varphi]\!]$.
- if $\varphi = \varphi_1 \wedge \varphi_2$, $[\![\varphi]\!] = [\![\varphi_1]\!] \cap [\![\varphi_2]\!]$. $X \in [\![\varphi_1]\!]$ and so, by induction hypothesis, $Y \in [\![\varphi_1]\!]$. Similarly, $Y \in [\![\varphi_2]\!]$ and in consequence, $Y \in [\![\varphi]\!]$.
- if $\varphi = \varphi_1 \vee \varphi_2$, $[\![\varphi]\!] = [\![\varphi_1]\!] \cup [\![\varphi_2]\!]$. If $X \in [\![\varphi_1]\!]$ (resp. $X \in [\![\varphi_2]\!]$), $Y \in [\![\varphi_1]\!]$ (resp. $Y \in [\![\varphi_2]\!]$) by induction hypothesis. Thus, $Y \in [\![\varphi]\!]$.
- if $\varphi = \top$, $[\![\varphi]\!] = 2^{\Sigma}$, hence $Y \in [\![\varphi]\!]$.
- if $\varphi = \bot$, $[\![\varphi]\!] = \emptyset$. This is in contradiction with the hypothesis $X \in [\![\varphi]\!]$. □

Theorem 39. *Given a modal specification with obligations S and a state q of S, the acceptance set:*

$$\text{Acc} = \{X \mid X \in [\![\Omega(q)]\!] \wedge X \subseteq \text{ready}(q)\}$$

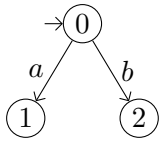
is convex-closed.

Proof. Let X and Y be two elements of Acc and Z a set such that $X \subseteq Z \subseteq Y$. By definition of Acc, $Y \subseteq \text{ready}(q)$ and by transitivity, $Z \subseteq \text{ready}(q)$. By Lemma 5 and as $X \in [\![\Omega(q)]\!]$, $Z \in [\![\Omega(q)]\!]$. Thus, $Z \in$ Acc. □

On the other hand, boolean modal specifications are as expressive as acceptance specifications; therefore the acceptance sets generated may not be convex. Consider for example the boolean modal specification and the equivalent acceptance specification depicted in Figure 3.4. The acceptance set of state 0 is clearly not convex-closed: $\emptyset \in \text{Acc}(0)$, $\{a, b\} \in \text{Acc}(0)$, $\emptyset \subseteq \{b\} \subseteq \{a, b\}$, but $\{b\} \notin \text{Acc}(0)$.

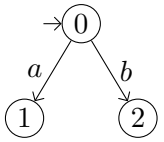
3.3.2 Conjunction

When computing the conjunction of two acceptance specifications (Definition 21), the only operation applied to acceptance sets is intersection. We first prove that convex-closure is preserved by intersection, and thus that we can represent the result of the intersection of two convex-closed acceptance set as a convex-closed acceptance set.



$\Omega(0) = a \vee \neg b$
$\Omega(1) = \Omega(2) = \bot$

(a) $S_{\mathcal{B}}$



$\mathrm{Acc}(0) = \{\emptyset, \{a\}, \{a, b\}\}$
$\mathrm{Acc}(1) = \mathrm{Acc}(2) = \{\emptyset\}$

(b) S_{Acc}

Figure 3.4: A boolean modal specification and the equivalent acceptance specification

Proposition 2. *If* Acc_1 *and* Acc_2 *are convex-closed, then* $\mathrm{Acc}_1 \cap \mathrm{Acc}_2$ *is also convex-closed.*

Proof. Suppose $X, Y \in \mathrm{Acc}_1 \cap \mathrm{Acc}_2$ and Z such that $X \subseteq Z \subseteq Y$. $X, Y \in \mathrm{Acc}_1$ and Acc_1 is convex-closed, so $Z \in \mathrm{Acc}_1$. Similarly, $Z \in \mathrm{Acc}_2$. Thus, $Z \in \mathrm{Acc}_1 \cap \mathrm{Acc}_2$. □

Now, we define an optimized way to compute the intersection of two convex-closed acceptance sets which relies only on their minimal and maximal elements:

Theorem 40 (Conjunction of convex-closed sets)**.** *If* Acc_1 *and* Acc_2 *are convex-closed acceptance sets, the minimum and the maximum elements of* $\mathrm{Acc}_1 \cap \mathrm{Acc}_2$*, are:*

$$\begin{cases} \mathrm{Acc}_{\cap}^{-} &= \min(\{X_{m1} \cup X_{m2} \mid X_{m1} \in \mathrm{Acc}_1^{-} \wedge X_{m2} \in \mathrm{Acc}_2^{-}\}) \\ \mathrm{Acc}_{\cap}^{+} &= \max(\{Y_{M1} \cap Y_{M2} \mid Y_{M1} \in \mathrm{Acc}_1^{+} \wedge Y_{M2} \in \mathrm{Acc}_2^{+}\}) \end{cases}$$

Proof. ($\subseteq$) Let $Z \in \mathrm{Acc}_1 \cap \mathrm{Acc}_2$. For $i \in \{1, 2\}$, $Z \in \mathrm{Acc}_i$, so there exist $X_{mi} \in \mathrm{Acc}_i^{-}$ and $Y_{Mi} \in \mathrm{Acc}_i^{+}$ such that $X_{mi} \subseteq Z \subseteq Y_{Mi}$. Thus $X_{m1} \cup X_{m2} \subseteq Z \subseteq Y_{M1} \cap Y_{M2}$ and there exist $X_{m\cap} \in \mathrm{Acc}_{\cap}^{-}$ and $Y_{M\cap} \in \mathrm{Acc}_{\cap}^{+}$ such that $X_{m\cap} \subseteq X_{m1} \cup X_{m2}$ and $Y_{M1} \cap Y_{M2} \subseteq Y_{M\cap}$.

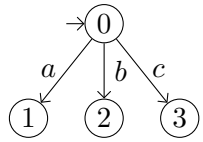
($\supseteq$) Suppose $X_{m\cap} \subseteq Z \subseteq Y_{M\cap}$ with $X_{m\cap} \in \mathrm{Acc}_{\cap}^{-}$ and $Y_{M\cap} \in \mathrm{Acc}_{\cap}^{+}$. By definition, there are some $X_{mi} \in \mathrm{Acc}_i^{-}$ and $Y_{Mi} \in \mathrm{Acc}_i^{+}$, for $i \in \{1, 2\}$, such that $X_{m1} \cup X_{m2} \subseteq Z \subseteq Y_{M1} \cap Y_{M2}$. Thus, for $i \in \{1, 2\}$, $X_{mi} \subseteq Z \subseteq Y_{Mi}$ and, since the Acc_i are convex-closed, $Z \in \mathrm{Acc}_i$, hence $Z \in \mathrm{Acc}_1 \cap \mathrm{Acc}_2$. □

Complexity 2. The conjunction of two acceptance sets Acc_1 and Acc_2 is their intersection (i.e., $\{X \mid X \in \mathrm{Acc}_1 \wedge X \in \mathrm{Acc}_2\}$), which complexity is $O(|\,\mathrm{Acc}_1\,| \times |\,\mathrm{Acc}_2\,|)$. According to Theorem 33, it is not necessary to apply the min and max functions to the sets of "minimal" and "maximal" elements of a convex-closed acceptance set as it does not modify the result. In consequence, when evaluating the complexity of the operations on convex-closed acceptance sets, we ignore the min/max operations. Thus, the complexity of the conjunction operation on convex-closed acceptance sets given in Theorem 40 is $O(|\,\mathrm{Acc}_1^{-}\,| \times |\,\mathrm{Acc}_2^{-}\,| + |\,\mathrm{Acc}_1^{+}\,| \times |\,\mathrm{Acc}_2^{+}\,|)$.

3.3.3 Product

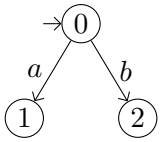
The product operation represents the main limitation of convex-closed sets, since the operation applied to acceptance sets when computing a product does not preserve convex-closure.

Consider the specifications in Figures 3.5(a) and 3.5(b), and their product in Figure 3.5(c). The two acceptance specifications clearly have convex-closed acceptance sets. However, the acceptance set of the initial state of their product is not convex-closed. Take the set $\{a\}$ for example: it does not belong to the acceptance set $\{\emptyset, \{a, b\}\}$ even though $\emptyset \subseteq \{a\} \subseteq \{a, b\}$.



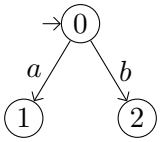
$\mathrm{Acc}(0) = \{\{a, b\}, \{c\}\}$
$\mathrm{Acc}(1) = \mathrm{Acc}(2) = \mathrm{Acc}(3) = \{\emptyset\}$

(a) S_1



$\mathrm{Acc}(0) = \{\{a, b\}\}$
$\mathrm{Acc}(1) = \mathrm{Acc}(2) = \{\emptyset\}$

(b) S_2



$\mathrm{Acc}(0) = \{\emptyset, \{a, b\}\}$
$\mathrm{Acc}(1) = \mathrm{Acc}(2) = \{\emptyset\}$

(c) $S_1 \otimes S_2$

Figure 3.5: Convex-closure is not preserved by product

While we can not define a product of convex-closed sets returning a convex-closed set, we can still use the convexity hypothesis to improve the computation of the acceptance set (that may not be convex-closed) given in Definition 22.

Theorem 41 (Product of convex-closed sets)**.** *If* Acc_1 *and* Acc_2 *are two convex-closed acceptance sets, their product is:*

$$\mathrm{Acc}_1 \otimes \mathrm{Acc}_2 = \bigcup_{(X_{m1}, Y_{M1})} \bigcup_{(X_{m2}, Y_{M2})} [X_{m1} \cap X_{m2}, Y_{M1} \cap Y_{M2}]$$

Proof. ($\subseteq$) Let $Z \in \mathrm{Acc}_1 \otimes \mathrm{Acc}_2$. There are some $Z_1 \in \mathrm{Acc}_1$ and $Z_2 \in \mathrm{Acc}_2$ such that $Z = Z_1 \cap Z_2$. As Acc_1 and Acc_2 are convex-closed, there are some $X_{m1} \in \mathrm{Acc}_1^-$, $Y_{M1} \in \mathrm{Acc}_1^+$, $X_{m2} \in \mathrm{Acc}_2^-$ and $Y_{M2} \in \mathrm{Acc}_2^+$ such that $X_{m1} \subseteq Z_1 \subseteq Y_{M1}$ and $X_{m2} \subseteq Z_2 \subseteq Y_{M2}$. Then, we have $X_{m1} \cap X_{m2} \subseteq Z_1 \cap Z_2 \subseteq Y_{M1} \cap Y_{M2}$ and thus $Z_1 \cap Z_2 \in [X_{m1} \cap X_{m2}, Y_{M1} \cap Y_{m2}]$.

($\supseteq$) Assume:

$$Z \in \bigcup_{(X_{m1}, Y_{M1})} \bigcup_{(X_{m2}, Y_{M2})} [X_{m1} \cap X_{m2}, Y_{M1} \cap Y_{M2}]$$

Then, there exists some X_{m1}, Y_{M1}, X_{m2} and Y_{M2} such that $Z \in [X_{m1} \cap X_{m2}, Y_{M1} \cap Y_{M2}]$ and so $X_{m1} \cap X_{m2} \subseteq Z \subseteq Y_{M1} \cap Y_{M2}$. Let $Z_1 = Z \cup X_{m1}$ and $Z_2 = Z \cup X_{m2}$. Considering that $X_{m1} \cup X_{m2} \subseteq Z$, we can show that $Z_1 \cap Z_2 = Z$ and thus $Z \in \mathrm{Acc}_1 \otimes \mathrm{Acc}_2$. □

This is potentially more efficient than the standard product of acceptance sets as it does not iterate on all the elements of Acc_1 and Acc_2 but only on their minimal and maximal elements.

3.3.4 Quotient

The quotient is probably the operation that will gain the most from using convex-closed sets, since the acceptance set of the quotient given in Definition 23:

$$\mathrm{Acc}_1 \,/\, \mathrm{Acc}_2 = \{X \mid \forall X_2 \in \mathrm{Acc}_2, X \cap X_2 \in \mathrm{Acc}_1\}$$

has an exponential blow-up. Indeed, we have to enumerate all the possible sets X and test if their intersection with elements of Acc_2 is in Acc_1, which gives a complexity of $O(2^{|\Sigma|} \times |\,\mathrm{Acc}_2\,| \times |\,\mathrm{Acc}_1\,|)$.

Proposition 3. *If* Acc_1 *and* Acc_2 *are convex-closed, then* $\mathrm{Acc}_1 \,/\, \mathrm{Acc}_2$ *is also convex-closed.*

Proof. Suppose $X, Y \in \mathrm{Acc}_1 \,/\, \mathrm{Acc}_2$ and Z such that $X \subseteq Z \subseteq Y$. For all $X_2 \in \mathrm{Acc}_2$, $X \cap X_2 \subseteq Z \cap X_2 \subseteq Y \cap X_2$. As, $X, Y \in \mathrm{Acc}_1 \,/\, \mathrm{Acc}_2$, $X \cap X_2, Y \cap X_2 \in \mathrm{Acc}_1$. Moreover, as Acc_1 is convex-closed, $Z \cap Y_2 \in \mathrm{Acc}_1$. As a result, $Z \in \mathrm{Acc}_1 \,/\, \mathrm{Acc}_2$, which is thus convex-closed. □

We now propose an optimized computation of the quotient of two convex-closed acceptance sets which directly generates its minimal and maximal elements from those of its operands.

We first define the quotient of an interval by another:

Proposition 4. *Let* $[X_1, Y_1]$ *and* $[X_2, Y_2]$ *be two intervals:*

$$\frac{[X_1, Y_1]}{[X_2, Y_2]} = \begin{cases} [X_1, Y_1 \cup \overline{Y_2}] & \text{if } X_1 \subseteq X_2 \\ \emptyset & \text{otherwise} \end{cases}$$

Proof. ($\Rightarrow$) Let $Z \in [X_1, Y_1]/[X_2, Y_2]$. By definition of the quotient of acceptance sets, $\forall X, X \in [X_2, Y_2] \rightarrow Z \cap X \in [X_1, Y_1]$. In particular, when $X = X_2$, we get $X_1 \subseteq Z \cap X_2$ which implies that $X_1 \subseteq X_2$: we have to prove that $Z \in [X_1, Y_1 \cup \overline{Y_2}]$. $X_1 \subseteq Z$ is trivial as $X_1 \subseteq Z \cap X_2$. Moreover, when $X = Y_2$, we have $Z \cap Y_2 \subseteq Y_1$ and thus $Z \subseteq Y1 \cup \overline{Y_2}$.

($\Leftarrow$) Assume that $X_1 \subseteq X_2$ and $Z \in [X_1, Y_1 \cup \overline{Y_2}]$ and let $X \in [X_2, Y_2]$. We have $Z \cap X \in [X_1 \cap X_2, (Y_1 \cup \overline{Y_2}) \cap Y_2]$. As $X_1 \subseteq X_2$, $X_1 \cap X_2 = X_1$. Moreover, $(Y_1 \cup \overline{Y_2}) \cap Y_2 = Y_1 \cap Y_2 \subseteq Y_1$. In consequence, $X_1 \subseteq Z \cap X \subseteq Y_1$ and thus $Z \in [X_1, Y_1]/[X_2, Y_2]$. □

Now, we prove that the quotient of two convex-closed acceptance sets is equivalent to the intersection of the quotients of an acceptance set by some intervals.

Lemma 6. *Given two convex-closed acceptance sets* Acc_1 *and* Acc_2*:*

$$\frac{\mathrm{Acc}_1}{\mathrm{Acc}_2} = \bigcap_{(X_{m2}, Y_{M2})} \frac{\mathrm{Acc}_1}{[X_{m2}, Y_{M2}]}$$

Proof.

$$\begin{aligned} Z \in \mathrm{Acc}_1 \,/\, \mathrm{Acc}_2 &\Leftrightarrow \forall\, W \in \mathrm{Acc}_2, W \cap Z \in \mathrm{Acc}_1 \\ &\Leftrightarrow \forall\, W \in \bigcup_{(X_{m2}, Y_{M2})} [X_{m2}, Y_{M2}], W \cap Z \in \mathrm{Acc}_1 \\ &\Leftrightarrow \forall (X_{m2}, Y_{M2}) \in (\mathrm{Acc}_2^-, \mathrm{Acc}_2^+), \forall\, W \in [X_{m2}, Y_{M2}].\ W \cap Z \in \mathrm{Acc}_1 \\ &\Leftrightarrow \forall (X_{m2}, Y_{M2}) \in (\mathrm{Acc}_2^-, \mathrm{Acc}_2^+), Z \in \mathrm{Acc}_1 /[X_{m2}, Y_{M2}] \\ &\Leftrightarrow Z \in \bigcap_{(X_{m2}, Y_{M2})} \mathrm{Acc}_1 /[X_{m2}, Y_{M2}] \end{aligned}$$ □

Then, we show how to translate the quotient of an acceptance set by an interval into a union of quotients of intervals:

Lemma 7. *Given a convex-closed acceptance set* Acc *and an interval* $[X, Y]$*:*

$$\frac{\mathrm{Acc}}{[X, Y]} = \bigcup_{(X_m, Y_M)} \frac{[X_m, Y_M]}{[X, Y]}$$

Proof. ($\subseteq$) Let $Z \in \mathrm{Acc}\,/[X, Y]$. For all $W \in [X, Y]$, $Z \cap W \in [Z \cap X, Z \cap Y]$. Moreover, by definition of the quotient operation, $Z \cap W \in \mathrm{Acc}$. As a result, $[Z \cap X, Z \cap Y] \subseteq \mathrm{Acc}$. Thus there exist $X_m \in \mathrm{Acc}^-$ and $Y_M \in \mathrm{Acc}^+$ such that $[Z \cap X, Z \cap Y] \subseteq [X_m, Y_M]$. As a result, for all $W \in [X, Y]$, $Z \cap W \in [X_m, Y_M]$, that is, $Z \in [X_m, Y_M]/[X, Y]$.

($\supseteq$) Let $Z \in \bigcup ([X_m, Y_M]/[X, Y])$. There exist $X_m \in \mathrm{Acc}^-$ and $Y_M \in \mathrm{Acc}^+$ such that $Z \in [X_m, Y_M]/[X, Y]$. Thus, for all $W \in [X, Y]$, $Z \cap W \in [X_m, Y_M] \subseteq \mathrm{Acc}$. As a result, $Z \in \mathrm{Acc}\,/[X, Y]$. □

We can then combine these three results to transform a quotient of convex-closed acceptance sets into intersections of unions of intervals. We demonstrated in Section 3.3.2 how to compute the minimal and maximal elements of the intersection of two convex-closed acceptance sets. We will generalize this result to the intersection of an arbitrary number of sets and use it to obtain the definition of the quotient. To simplify the notations, we will use $\uplus$ and $\upcap$ to denote the pointwise union and intersection:

$$\begin{aligned}\mathrm{Acc}_1 \uplus \mathrm{Acc}_2 &= \{X \cup Y \mid X \in \mathrm{Acc}_1 \wedge Y \in \mathrm{Acc}_2\}\\ \mathrm{Acc}_1 \upcap \mathrm{Acc}_2 &= \{X \cap Y \mid X \in \mathrm{Acc}_1 \wedge Y \in \mathrm{Acc}_2\}\end{aligned}$$

The intersection of two convex-closed acceptance sets can thus be written:

$$\mathrm{Acc}_1 \cap \mathrm{Acc}_2 = \begin{cases} \mathrm{Acc}^- = \min(\mathrm{Acc}_1^- \uplus \mathrm{Acc}_2^-) \\ \mathrm{Acc}^+ = \min(\mathrm{Acc}_1^+ \upcap \mathrm{Acc}_2^+) \end{cases}$$

Proposition 5. *Given n convex-closed acceptance sets Acc_i, their intersection is:*

$$\bigcap_i \mathrm{Acc}_i = \begin{cases} \mathrm{Acc}^- = \min\left(\bigcup\!\!\!\!\bigcup_i \mathrm{Acc}_i^-\right) \\ \mathrm{Acc}^+ = \max\left(\bigcap\!\!\!\!\bigcap_i \mathrm{Acc}_i^+\right) \end{cases}$$

Proof.

$$\begin{aligned}\bigcap_i \mathrm{Acc}_i &= \mathrm{Acc}_1 \cap \mathrm{Acc}_2 \cap \cdots \cap \mathrm{Acc}_{n-1} \cap \mathrm{Acc}_n \\ &= \begin{cases} \mathrm{Acc}^- = \min\left(\mathrm{Acc}_1^- \uplus \min\left(\mathrm{Acc}_2^- \uplus \ldots \min\left(\mathrm{Acc}_{n-1}^- \uplus \mathrm{Acc}_n^-\right)\ldots\right)\right) \\ \mathrm{Acc}^+ = \max\left(\mathrm{Acc}_1^+ \upcap \max\left(\mathrm{Acc}_2^+ \upcap \ldots \max\left(\mathrm{Acc}_{n-1}^+ \upcap \mathrm{Acc}_n^+\right)\ldots\right)\right) \end{cases} \\ &= \begin{cases} \mathrm{Acc}^- = \min\left(\mathrm{Acc}_1^- \uplus \left(\mathrm{Acc}_2^- \uplus \ldots \left(\mathrm{Acc}_{n-1}^- \uplus \mathrm{Acc}_n^-\right)\ldots\right)\right) \\ \mathrm{Acc}^+ = \max\left(\mathrm{Acc}_1^+ \upcap \left(\mathrm{Acc}_2^+ \upcap \ldots \left(\mathrm{Acc}_{n-1}^+ \upcap \mathrm{Acc}_n^+\right)\ldots\right)\right) \end{cases} \\ &= \begin{cases} \mathrm{Acc}^- = \min\left(\bigcup\!\!\!\!\bigcup_i \mathrm{Acc}_i^-\right) \\ \mathrm{Acc}^+ = \max\left(\bigcap\!\!\!\!\bigcap_i \mathrm{Acc}_i^+\right) \end{cases}\end{aligned}$$

□

Finally, we obtain the definition of the quotient of convex-closed acceptance sets by applying the previous results:

Theorem 42. *If Acc_1 and Acc_2 are two convex-closed acceptance sets, then the minimum and maximum elements of $\mathrm{Acc}_1 \,/\, \mathrm{Acc}_2$ are:*

$$\begin{cases} \mathrm{Acc}_/^- &= \min\Big(\bigcup\!\!\!\!\bigcup_{X_{m2}} \{X_{m1} \mid X_{m1} \in \mathrm{Acc}_1^- \wedge X_{m1} \subseteq X_{m2}\}\Big) \\ \mathrm{Acc}_/^+ &= \max\Big(\bigcap\!\!\!\!\bigcap_{Y_{M2}} \{Y_{M1} \cup \overline{Y_{M2}} \mid Y_{M1} \in \mathrm{Acc}_1^+\}\Big) \end{cases}$$

Proof.

$$\begin{aligned}\frac{\mathrm{Acc}_1}{\mathrm{Acc}_2} &= \bigcap_{(X_{m2},Y_{M2})} \frac{\mathrm{Acc}_1}{[X_{m2},Y_{M2}]} \qquad \text{by Lemma 6} \\ &= \bigcap_{(X_{m2},Y_{M2})} \bigcup_{(X_{m1},Y_{M1})} \frac{[X_{m1},Y_{M1}]}{[X_{m2},Y_{M2}]} \qquad \text{by Lemma 7}\end{aligned}$$

$$
\begin{aligned}
&= \bigcap_{(X_{m2},Y_{M2})} \bigcup_{X_{m1}\subseteq X_{m2}} \bigcup_{Y_{M1}} [X_{m1}, Y_{M1} \cup \overline{Y_{M2}}] && \text{by Proposition 4}\\
&= \bigcap_{(X_{m2},Y_{M2})} \begin{cases} \mathrm{Acc}^- = \bigcup_{X_{m1}\subseteq X_{m2}} \{X_{m1}\} \\ \mathrm{Acc}^+ = \bigcup_{Y_{M1}} \{Y_{M1} \cup \overline{Y_{M2}}\} \end{cases}\\
&= \bigcap_{(X_{m2},Y_{M2})} \begin{cases} \mathrm{Acc}^- = \{X_{m1} \mid X_{m1} \in \mathrm{Acc}_1^- \wedge X_{m1} \subseteq X_{m2}\} \\ \mathrm{Acc}^+ = \{Y_{M1} \cup \overline{Y_{M2}} \mid Y_{M1} \in \mathrm{Acc}_1^+\} \end{cases}\\
&= \begin{cases} \mathrm{Acc}^- = \min\Big(\biguplus_{X_{m2}} \{X_{m1} \mid X_{m1} \in \mathrm{Acc}_1^- \wedge X_{m1} \subseteq X_{m2}\}\Big) \\ \mathrm{Acc}^+ = \max\Big(\bigcap_{Y_{M2}} \{Y_{M1} \cup \overline{Y_{M2}} \mid Y_{M1} \in \mathrm{Acc}_1^+\}\Big) \end{cases} && \text{by Proposition 5}
\end{aligned}
$$

□

Complexity 3. Given two acceptance sets Acc_1 and Acc_2, the complexity of their pointwise union and intersection is $O(|\,\mathrm{Acc}_1\,| \times |\,\mathrm{Acc}_2\,|)$. As a consequence, given n acceptance sets Acc_i, the complexity of their pointwise union and intersection is $O(\prod_{i=1}^{n} |\,\mathrm{Acc}_i\,|)$.

The minimal elements of the quotient are $\biguplus_{X_{m2}} \{X_{m1} \mid X_{m1} \in \mathrm{Acc}_1^- \wedge X_{m1} \subseteq X_{m2}\}$. This computes the pointwise union of $|\,\mathrm{Acc}_2^-\,|$ acceptance sets, each with a size bounded by $|\,\mathrm{Acc}_1^-\,|$ (since we generate subsets of Acc_1^-). This yields a complexity of $O(|\,\mathrm{Acc}_1^-\,|^{|\,\mathrm{Acc}_2^-\,|})$.

The maximal elements of the quotient are $\bigcap_{Y_{M2}} \{Y_{M1} \cup \overline{Y_{M2}} \mid Y_{M1} \in \mathrm{Acc}_1^+\}$. This computes the pointwise intersection of $|\,\mathrm{Acc}_2^+\,|$ acceptance sets of size $|\,\mathrm{Acc}_1^+\,|$, which implies a complexity of $O(|\,\mathrm{Acc}_1^+\,|^{|\,\mathrm{Acc}_2^+\,|})$.

In consequence, the complexity of the quotient operation is $O(|\,\mathrm{Acc}_1^-\,|^{|\,\mathrm{Acc}_2^-\,|} + |\,\mathrm{Acc}_1^+\,|^{|\,\mathrm{Acc}_2^+\,|})$. Observe that it is also exponential, but depends on the size of the minimal and maximal acceptance sets, while the quotient on acceptance sets has a fixed $2^{|\Sigma|}$ exponential, regardless of the size of its parameters.

3.3.5 Dissimilar alphabets

We now consider acceptance sets defined on different alphabets and the extension operations introduced in Section 3.2.4.

Proposition 6. *Given two alphabets Σ and Σ' such that $\Sigma \subseteq \Sigma'$ and a convex-closed acceptance set* Acc*, then* $\mathrm{Acc}_{\Uparrow\Sigma'}$ *and* $\mathrm{Acc}_{\uparrow\Sigma'}$ *are both convex-closed.*

Proof. (weak extension) Let X and Y be two elements of $\mathrm{Acc}_{\Uparrow\Sigma'}$ and Z such that $X \subseteq Z \subseteq Y$. By definition of weak extension, there exist an $X' \in \mathrm{Acc}$ and a $\sigma_X \subseteq \Sigma' \setminus \Sigma$ such that $X = X' \cup \sigma_X$. Similarly, $Y = Y' \cup \sigma_Y$ with $Y' \in \mathrm{Acc}$ and $\sigma_Y \subseteq \Sigma' \setminus \Sigma$. Then, there exist two sets $Z' = Z \cap \Sigma$ and $\sigma_Z = Z \cap (\Sigma' \setminus \Sigma)$ such that $Z = Z' \cup \sigma_Z$. From $X \subseteq Z \subseteq Y$, we deduce $X' \subseteq Z' \subseteq Y'$ and, since Acc is convex-closed, $Z' \in \mathrm{Acc}$. By definition, $\sigma_Z \subseteq \Sigma' \setminus \Sigma$, so $Z \in \mathrm{Acc}_{\Uparrow\Sigma'}$.

(strong extension) Let X and Y be two elements of $\mathrm{Acc}_{\uparrow\Sigma'}$ and Z such that $X \subseteq Z \subseteq Y$. By definition of strong extension, there exists an $X' \in \mathrm{Acc}$ such that $X = X' \cup (\Sigma' \setminus \Sigma)$. Similarly, there is a $Y' \in \mathrm{Acc}$ such that $Y = Y' \cup (\Sigma' \setminus \Sigma)$. Since $X' \cup (\Sigma \setminus \Sigma') \subseteq Z$, there exists a $Z' = Z \cap \Sigma$ such that $Z = Z' \cup (\Sigma \setminus \Sigma')$. Then, $X' \subseteq Z' \subseteq Y'$ and since Acc is convex-closed, $Z' \in \mathrm{Acc}$. In consequence, $Z \in \mathrm{Acc}_{\uparrow\Sigma'}$. □

Since both extension operations preserve convex-closure, we show that these extensions can be computed from the minimal and maximal elements of an acceptance set:

Theorem 43. *Given two alphabets Σ and Σ' such that $\Sigma \subseteq \Sigma'$ and a convex-closed acceptance set* Acc*, the minimum and maximum elements of* $\mathrm{Acc}_{\Uparrow\Sigma'}$ *and* $\mathrm{Acc}_{\uparrow\Sigma'}$ *are:*

$$\begin{cases} \mathrm{Acc}^-_{\Uparrow} &= \mathrm{Acc}^- \\ \mathrm{Acc}^+_{\Uparrow} &= \{X \cup (\Sigma' \setminus \Sigma) \mid X \in \mathrm{Acc}^+\} \end{cases}$$

$$\begin{cases} \mathrm{Acc}^-_{\uparrow} &= \{X \cup (\Sigma' \setminus \Sigma) \mid X \in \mathrm{Acc}^-\} \\ \mathrm{Acc}^+_{\uparrow} &= \{X \cup (\Sigma' \setminus \Sigma) \mid X \in \mathrm{Acc}^+\} \end{cases}$$

Proof. (weak extension)

$$\begin{aligned} &\mathrm{Acc}_{\Uparrow\Sigma'} \\ &= \{X \cup \sigma \mid X \in \mathrm{Acc} \wedge \sigma \subseteq \Sigma' \setminus \Sigma\} \\ &= \{X \cup \sigma \mid \exists X_m \in \mathrm{Acc}^-, \exists Y_M \in \mathrm{Acc}^+, X_m \subseteq X \subseteq Y_M \wedge \sigma \subseteq \Sigma' \setminus \Sigma\} \\ &= \{X' \mid \exists X_m \in \mathrm{Acc}^-, \exists Y_M \in \mathrm{Acc}^+, X_m \subseteq X' \subseteq Y_M \cup (\Sigma' \setminus \Sigma)\} \\ &= \begin{cases} \mathrm{Acc}^-_{\Uparrow} = \mathrm{Acc}^- \\ \mathrm{Acc}^+_{\Uparrow} = \{X \cup (\Sigma' \setminus \Sigma) \mid X \in \mathrm{Acc}^+\} \end{cases} \end{aligned}$$

(strong extension)

$$\begin{aligned} &\mathrm{Acc}_{\uparrow\Sigma'} \\ &= \{X \cup (\Sigma' \setminus \Sigma) \mid X \in \mathrm{Acc}\} \\ &= \{X \cup (\Sigma' \setminus \Sigma) \mid \exists X_m \in \mathrm{Acc}^-, \exists Y_M \in \mathrm{Acc}^+, X_m \subseteq X \subseteq Y_M\} \\ &= \{X' \mid \exists X_m \in \mathrm{Acc}^-, \exists Y_M \in \mathrm{Acc}^+, X_m \cup (\Sigma' \setminus \Sigma) \subseteq X' \subseteq Y_M \cup (\Sigma' \setminus \Sigma)\} \\ &= \begin{cases} \mathrm{Acc}^-_{\uparrow} = \{X \cup (\Sigma' \setminus \Sigma) \mid X \in \mathrm{Acc}^-\} \\ \mathrm{Acc}^+_{\uparrow} = \{X \cup (\Sigma' \setminus \Sigma) \mid X \in \mathrm{Acc}^+\} \end{cases} \end{aligned}$$

□

Complexity 4. Computing the weak extension of an acceptance set on an alphabet Σ to an alphabet Σ' requires to enumerate all the subsets of $\Sigma' \setminus \Sigma$. In consequence, the complexity is $O(|\,\mathrm{Acc}\,| \times 2^{|\Sigma'|-|\Sigma|})$. Using the representation based on the minimum and maximum elements entirely removes this exponential blow-up as it suffices to add $\Sigma' \setminus \Sigma$ to the maximum elements: the complexity is $O(|\,\mathrm{Acc}^+\,|)$.

For strong extension, there is no exponential blow-up in the acceptance case which complexity is $O(|\,\mathrm{Acc}\,|)$. The operation on convex-closed acceptance sets has a complexity of $O(|\,\mathrm{Acc}^-\,| + |\,\mathrm{Acc}^+\,|)$.

3.3.6 Coq mechanization

The proofs of the theorems of the previous sections, in particular the one about quotient, are a bit complex. However, they only involve fairly basic concepts from set theory. So we were interested in using computer-assisted techniques, such as a theorem prover or a proof assistant, to make sure that we made no mistake.

A first attempt was made using Why3. It is a platform that provides a language allowing to write first-order logic properties and functional programs, and to prove some theorems using a variety of theorem provers—such as Alt-Ergo, Simplify, Z3, and many others—and proof assistants, like Coq, Isabelle, or PVS. It comes with a standard library offering, among other things, an axiomatization of (finite) sets.

The first statements, such as the preservation of convexity (Propositions 2 and 3) and the optimized definitions of refinement and conjunction (Theorems 35 and 40), were easily proved

using Alt-Ergo, Z3, and a few lines of Coq. However, proving the correctness of the optimized definition of the quotient (as shown in the previous section by Lemmas 6 and 7, Propositions 4 and 5, and Theorem 42), and actually just writing their statements, appeared to be very difficult when indexed unions and intersections came in play. Moreover, the theorem provers consistently failed to prove the goals and using Coq was difficult (for instance, despite the fact that the sets were known to be finite, reasoning by induction was quite hard).

Thus, we decided to use Coq directly instead. In particular, it allowed us to use a formalization of sets defined in Coq, which was easier to manipulate than the axiomatization generated by Why3.

The standard library of Coq comes not with one but four different formalizations of sets:

ListSet defines some functions to handle a list of values as a set. Although simple to reason about, it lacks any kind of abstraction and offers only a handful of operations.

Ensemble defines a set as a predicate: $\mathsf{Ensemble\ U} := \mathsf{U} \to \mathsf{Prop}$. It is easy to manipulate these sets (testing if $x \in S$ is just $\mathsf{S\ x}$, the union of S and T is $\lambda\ \mathsf{x} \Rightarrow \mathsf{S\ x} \vee \mathsf{T\ x}$, etc.), but they are limited to the realm of propositions (Prop); the operations are thus undecidable and it is not possible to write executable functions with them (or extract OCaml programs).

FSet uses Coq's module system to define an abstract interface for sets and offers several implementations based on lists or trees. It features decidable operations and their specifications.

MSet is an extension and modernization of FSet. It probably offers the most up-to-date and feature-rich set interface in the Coq standard library, although the module system may be slightly heavy to use compared to simple lists or predicates.

We used this last library, MSet, to represent acceptance sets. It was necessary to define some additional operations on sets which are not available in the library, such as the powerset of a set, the indexed variants of standard operations and the pointwise intersection and union, specific to the usage of sets of sets.

Then, we were able to express and prove the theorems of the previous sections following quite closely the paper proofs, with some additional details. One of the main requirements of the Coq proof that is not present with the mathematical proof is to ensure that the operations on sets are Proper, i.e., that they preserve the equality relation on sets. As MSet's sets are abstract, we have no guarantee that the set equality (defined as double inclusion) is equivalent to the structural equality of the underlying data type. Indeed, if sets are represented by unordered lists, $S_1 \cup S_2$ and $S_2 \cup S_1$ will certainly be represented by two different lists, containing the same elements in a different order. In consequence, every time we define a function on sets, we have to prove that if it is called with two sets containing the same elements, it will return equivalent results.

The remaining of this section gives an overview of the Coq mechanization. It assumes that the reader has at least a basic knowledge of dependent type theory and the Coq proof assistant.

Preliminaries

We first had to define a few functions on sets which were not available in MSet.

Since we often have to reason inductively on sets, we first define a module with some utility functions and lemmas used to simplify this kind of reasoning. We would like to be able to define a function starting from a set S and then remove arbitrary elements from S until it is empty. In order to do so, we define a relation, $\mathsf{R\_rm}$, between two sets which difference is exactly one element.

Then, we prove that this relation is well_founded, using the fact that the cardinal of a set is strictly decreasing when removing an element from it. well_founded R, where R is a relation on elements of type T, means that there is no infinitely decreasing chain of elements of T. Thus, if we define a function with a parameter of type T and only call it recursively with a smaller (in the sense of the relation R) parameter, then it will be terminating.

```
Module InductionOn (E : DecidableType) (S : WSetsOn E).
  Definition R_rm (s' s : S.t) := ∃ x, S.In x s ∧ s' = S.remove x s.
  Lemma wf_R_rm : well_founded R_rm.
  Definition Acc_rm_inv : ∀ {s}, Acc R_rm s → ∀ {x}, S.In x s → Acc R_rm (S.remove x s).
```

We also define a wrapper around MSet's function choose : S.t → option E.t returning either an element of a set along with a proof of membership or a proof that the set is empty. This makes it easier to define some recursive functions on sets.

```
  Definition elem_or_empty (s : S.t) : {x | S.In x s} + {S.Empty s}.
End InductionOn.
Module Induction (S : WSets) := InductionOn S.E S.
```

Remark. In simple cases, we can use Function with the cardinal of the set as measure and the functional induction tactic. However, the induction is then tied to one specific function: the induction is not on the set parameter itself, but on the function call. Consequently, if we have a goal like ∀ s, f s = g s, we can do an induction on f s or g s, but not on the set s itself, which would allow us to simplify both functions simultaneously.

MSet offers no function to apply a function to each element of a set. We define a function map that, given a function f and a set S, returns the set $\{f(x) \mid x \in S\}$. Note that contrary to the standard map function on lists, this map is homogeneous: the set returned has the same type as the set parameter. We could easily extend this function in order to return a set of a different type (it would just require to take an additional module parameter for the resulting set), but we never needed it. Observe that thanks to the previous module, the recursive definition of the function on a set is quite simple.

```
Module MapOn (E : DecidableType) (S : WSetsOn E).
  Module SI := InductionOn E S.
  Fixpoint map_rec (f : S.elt → S.elt) (s : S.t) (rec : Acc SI.R_rm s) : S.t :=
    match SI.elem_or_empty s with
    | inleft (exist x Hin) ⇒ S.add (f x) (map_rec f (S.remove x s) (SI.Acc_rm_inv rec Hin))
    | inright _ ⇒ S.empty
    end.
  Definition map f s := map_rec f s (SI.wf_R_rm s).
  Lemma in_map : ∀ f s x, Proper (E.eq ⟹ E.eq) f → S.In x s → ∀ x', E.eq x' (f x) → S.In x' (map f s).
  Lemma map_in : ∀ f s x, Proper (E.eq ⟹ E.eq) f → S.In x (map f s) → ∃ x', S.In x' s ∧ E.eq x (f x').
End MapOn.
Module Map (S : WSets) := MapOn S.E S.
```

We also need a function computing the powerset of a given set. It uses the classical recursive algorithm:

$$2^{\emptyset} = \{\emptyset\} \qquad 2^{\{x\} \cup S} = 2^S \cup \{\{x\} \cup P \mid P \in 2^S\}$$

```
Module PowersetOn (E : DecidableType) (S : WSetsOn E) (A : WSetsOn S).
  Module AM := MapOn S A.
  Module SI := InductionOn E S.
```

```
  Fixpoint powerset_rec (s : S.t) (rec : Acc SI.R_rm s) : A.t :=
    match SI.elem_or_empty s with
    | inleft (exist x Hin) ⇒
      let r := powerset_rec (S.remove x s) (SI.Acc_rm_inv rec Hin) in
      A.union r (AM.map (S.add x) r)
    | inright _ ⇒ A.singleton S.empty
    end.
  Definition powerset s := powerset_rec s (SI.wf_R_rm s).
  Lemma powerset_spec1 : ∀ s s' : S.t, A.In s' (powerset s) → S.Subset s' s.
  Lemma powerset_spec2 : ∀ s s' : S.t, S.Subset s' s → A.In s' (powerset s).
End PowersetOn.
```

Last, we define a generic function for defining indexed functions like, for example, $\bigcup_{i \in I} f(i)$. We define a function indexed which takes the following parameters:

s a set of indices on which we iterate (I in the example);

f a function transforming an index into a value of the return type;

c a function combining two values of the result type ($\cup$ in the example);

e a neutral element for the function **c** (implicitly $\emptyset$ in the example).

```
Module IndexedOn (IE : DecidableType) (I : WSetsOn IE) (TE : DecidableType) (T : WSetsOn TE).
  Module II := InductionOn IE I.
  Fixpoint indexed_rec (e : T.t) (c : T.t → T.t → T.t) (f : I.elt → T.t)
                       (s : I.t) (rec : Acc II.R_rm s) : T.t :=
    match II.elem_or_empty s with
    | inleft (exist x Hin) ⇒ c (f x) (indexed_rec e c f (I.remove x s) (II.Acc_rm_inv rec Hin))
    | inright _ ⇒ e
    end.
  Definition indexed e c f s := indexed_rec e c f s (II.wf_R_rm s).
End IndexedOn.
```

We then specialize this function for the two particular cases of indexed union and intersection.

```
Module IndexedUnionOn (IE : DecidableType) (I : WSetsOn IE)
                      (TE : DecidableType) (T : WSetsOn TE)
                      (Import Idx : IndexedOnI IE I TE T).
  Definition indexed_union := indexed T.empty T.union.
  Lemma indexed_union_spec_1 :
    ∀ f idx i x, Proper (IE.eq ⟹ T.eq) f → I.In i idx → T.In x (f i) → T.In x (indexed_union f idx).
  Lemma indexed_union_spec_2 :
    ∀ f idx s, Proper (IE.eq ⟹ T.eq) f → T.In s (indexed_union f idx) → ∃ i, I.In i idx ∧ T.In s (f i).
End IndexedUnionOn.
```

For intersection, the neutral element is the set of all the values of type T, so we must have a finite type has a parameter. We first define a module type for finite types.

```
Module Type FiniteType (E : DecidableType) (S : WSetsOn E).
  Parameter elements : S.t.
  Parameter finite : ∀ x, S.In x elements.
End FiniteType.

Module IndexedInterOn (IE : DecidableType) (I : WSetsOn IE)
                      (TE : DecidableType) (T : WSetsOn TE)
```

```
                    (TFin : FiniteType TE T) (Import Idx : IndexedOnI IE I TE T).
  Definition indexed_inter := indexed TFin.elements T.inter.
  Lemma indexed_inter_spec_1 :
    ∀ f idx, Proper (IE.eq ⟹ T.eq) f → ∀ x, (∀  i, I.In i idx → T.In x (f i)) → T.In x (indexed_inter f idx).
  Lemma indexed_inter_spec_2 :
    ∀ f idx x, Proper (IE.eq ⟹ T.eq) f → T.In s (indexed_inter f idx) → ∀ i, I.In i idx → T.In x (f i).
End IndexedInterOn.
```

Convexity

We now define a module for the actual mechanization of convex acceptance sets. It is parameterized by the types of the alphabet, sets of actions, acceptance sets, and a proof that the alphabet is finite. We also instantiate the modules defined above for the given sets.

```
Module Convex (Σ_type : OrderedType) (S : SetsOn Σ_type)
              (A : SetsOn S) (Σ_fin : FiniteType Σ_type S).
Module Σ:= Σ_type <+ Σ_fin.
Module AM := MapOn S A.
Module AI := InductionOn S A.
Module P := PowersetOn Σ_type S A.
Module AIdx := IndexedOn S A S A.
Module ASIdx := IndexedOn S A ΣS.
Module Import IUAA := IndexedUnionOn S A S A AIdx.
```

The definition of a convex-closed set is a direct translation of Definition 28:

```
Definition ConvexClosed (s : A.t) : Prop :=
  ∀ x, A.In x s → ∀ y, A.In y s → ∀ z, S.Subset x z → S.Subset z y → A.In z s.
```

The main advantage of convex-closed sets is that they allow to only work on the minimum and maximum elements, so we have to be able to compute these elements. We first give a logical definition of what is a minimal element and then define a function computing the minimal elements of a set, as described in Definition 29.

```
Definition IsMin (s : S.t) (acc : A.t) := A.In s acc ∧ ∀ s', A.In s' acc → S.Subset s' s → S.Equal s' s.
Definition is_min (s : S.t) (acc : A.t) : bool :=
  A.mem s acc && A.for_all (fun s' ⇒ negb (S.subset s' s) || S.equal s' s) acc.
Definition min_elements (acc : A.t) : A.t := A.filter (fun s ⇒ is_min s acc) acc.
Theorem min_elements_spec : ∀ acc m, A.In m (min_elements acc) ↔ IsMin m acc.
```

We also prove that for any element of an acceptance set, there is a minimal element included in it. Note that this is not as trivial to prove as it may seem, since it required to prove that either the element is already minimal or that there a strict subset of it that also belongs to the acceptance set, and then recurse until a minimal element is reached.

```
Lemma min_element : ∀ x acc, A.In x acc → ∃ m, IsMin m acc ∧ S.Subset m x.
```

We define and prove similar functions and theorems for the maximal elements:

```
Definition IsMax (s : S.t) (acc : A.t) := A.In s acc ∧ ∀ s', A.In s' acc → S.Subset s s' → S.Equal s s'.
Definition is_max (s : S.t) (acc : A.t) : bool :=
  A.mem s acc && A.for_all (fun s' ⇒ negb (S.subset s s') || S.equal s s') acc.
Definition max_elements (acc : A.t) : A.t := A.filter (fun s ⇒ is_max s acc) acc.
Theorem max_elements_spec : ∀ acc m, A.In m (max_elements acc) ↔ IsMax m acc.
Lemma max_element : ∀ x acc, A.In x acc → ∃ m, IsMax m acc ∧ S.Subset x m.
```

We now define the notion of interval (Definition 30), which we split in a record containing the bounds of the interval and a well-formedness property. We also define a few utility functions to test membership and equality of intervals.

```
Record Interval := { min : S.t; max : S.t }.
Definition IWF (i : Interval) := S.Subset i.(min) i.(max).
Definition In_I (x : S.t) (i : Interval) := S.Subset i.(min) x ∧ S.Subset x i.(max).
Definition IEq (i1 i2 : Interval) := S.Equal i1.(min) i2.(min) ∧ S.Equal i1.(max) i2.(max).
Instance IEq_Equivalence : Equivalence IEq.
```

We then define a function generating the acceptance set equivalent to an interval $[X, Y]$, i.e., $\{Z \mid X \subseteq Z \subseteq Y\}$. It is not possible to translate this definition directly in Coq as MSet has no set comprehension. We could define it in our case since the alphabet is finite: we could generate 2^{2^Σ} and then use filter to retain only the sets Z verifying $X \subseteq Z \subseteq Y$, but that would be very inefficient. Instead, we use the following formula:

$$\{Z \mid X \subseteq Z \subseteq Y\} = \left\{X \cup P \,\middle|\, P \in 2^{Y \setminus X}\right\}$$

and prove that, indeed, an element belongs to this set if and only if it belongs to the interval (assuming that the interval is well-formed in one case):

```
Definition acc_of_interval (i : Interval) : A.t :=
  AM.map (S.union i.(min)) (P.powerset (S.diff i.(max) i.(min))).
Lemma in_acc_of_interval : ∀ x i, In_I x i → A.In x (acc_of_interval i).
Lemma acc_of_interval_in : ∀ x i, IWF i → A.In x (acc_of_interval i) → In_I x i.
```

Given some minimal and maximal sets, we can compute the corresponding acceptance set:

```
Definition from_min_max (Min Max : A.t) : A.t :=
  indexed_union (fun min ⇒
    indexed_union
      (fun max ⇒ if S.subset min max then acc_of_interval {| min := min; max := max |} else A.empty)
      Max
  ) Min.
Theorem from_min_max_spec :
  ∀ Min Max s,
    A.In s (from_min_max Min Max) ↔
    (∃  min, A.In min Min ∧ ∃ max, A.In max Max ∧ In_I s {| min := min; max := max |}).
```

And we prove Theorem 32:

```
Theorem acc_min_max :
  ∀ acc, ConvexClosed acc → A.Equal acc (from_min_max (min_elements acc) (max_elements acc)).
```

We also prove that given two arbitrary sets, keeping only the minimal and maximal elements does not change the corresponding acceptance set (Theorem 33):

```
Theorem from_min_max_bounds :
  ∀ Min Max,
    A.Equal (from_min_max Min Max) (from_min_max (min_elements Min) (max_elements Max)).
```

Embeddings of other formalisms

We now consider the acceptance sets obtained from other formalisms such as modal specifications. We define the function computing this acceptance set from the may/must sets, prove that it is convex, and exhibit its minimal and maximal elements:

```
Definition modal_to_acceptance (may must : S.t) : A.t :=
  A.filter (fun x ⇒ S.subset must x && S.subset x may) (P.powerset Σ.elements).
Theorem modal_to_acceptance_convex :
  ∀ may must, ConvexClosed (modal_to_acceptance may must).
Theorem modal_to_acceptance_opt :
  ∀ may must,
  A.Equal (modal_to_acceptance may must) (from_min_max (A.singleton must) (A.singleton may)).
```

We also prove that the acceptance sets obtained from a disjunctive modal specification are convex-closed:

```
Definition disjunctive_to_acceptance (may : S.t) (dmust : A.t) : A.t :=
  A.filter
    (fun x ⇒ S.subset x may && A.for_all (fun must ⇒ negb (S.equal (S.inter x must) S.empty)) dmust)
    (P.powerset Σ.elements).
Theorem disjunctive_to_acceptance_convex :
  ∀ may dmust, ConvexClosed (disjunctive_to_acceptance may dmust).
```

We finally consider modal specification with obligations. We first have to define positive boolean formulas (abbreviated PBF in the Coq development) and give their semantics:

```
Inductive PBF := Top | Bot | Var (a : Σ.t) | And (f1 f2 : PBF) | Or (f1 f2 : PBF).
Fixpoint pbf_sem (f : PBF) : A.t :=
  match f with
  | Top ⇒ P.powerset Σ.elements
  | Bot ⇒ A.empty
  | Var a ⇒ A.filter (fun x ⇒ S.mem a x) (P.powerset Σ.elements)
  | And f1 f2 ⇒ A.inter (pbf_sem f1) (pbf_sem f2)
  | Or f1 f2 ⇒ A.union (pbf_sem f1) (pbf_sem f2)
  end.
```

Then, we can compute an acceptance set from a positive boolean formula and prove that it is convex-closed.

```
Definition obligations_to_acceptance (ready : S.t) (f : PBF) : A.t :=
  A.filter (fun x ⇒ A.mem x (pbf_sem f) && S.subset x ready) (P.powerset Σ.elements).
Lemma pbf_sem_subset :
  ∀ f x y, A.In x (pbf_sem f) → S.Subset x y → A.In y (pbf_sem f).
Theorem obligations_to_acceptance_convex :
  ∀ ready f, ConvexClosed (obligations_to_acceptance ready f).
```

Refinement

We now give two definitions of refinement: the standard one on acceptance specifications, which is equivalent to inclusion, and the optimized one on convex-closed acceptance sets. We then prove that they are equivalent, as stated in Theorem 35.

```
Definition refinement := A.subset.
Definition convex_refinement (a1 a2 : A.t) : bool :=
  A.for_all (fun x1 ⇒ A.∃ _ (fun x2 ⇒ S.subset x2 x1) (min_elements a2)) (min_elements a1) &&
  A.for_all (fun y1 ⇒ A.∃ _ (fun y2 ⇒ S.subset y1 y2) (max_elements a2)) (max_elements a1).
Theorem refinement_equivalence :
  ∀ a1 a2, ConvexClosed a1 → ConvexClosed a2 → refinement a1 a2 = convex_refinement a1 a2.
```

Conjunction

We define the conjunction of acceptance sets—it is just their intersection—and prove that convex-closure is preserved by conjunction (Proposition 2).

```
Definition conjunction := A.inter.
Theorem conjunction_convex :
  ∀ a1 a2, ConvexClosed a1 → ConvexClosed a2 → ConvexClosed (conjunction a1 a2).
```

We now want to define the optimized conjunction of convex-closed acceptance sets. To do so, we first define a generic function applying a function to all pairs of elements of two acceptance sets, which we will use in order to define the pointwise union and intersection operations (see page 51):

$$\mathsf{inner}(f, a_1, a_2) = \{f(x_1, x_2) \mid x_1 \in a_1 \wedge x_2 \in a_2\}$$

$$a_1 \uplus a_2 = \mathsf{inner}(\cup, a_1, a_2) \qquad a_1 \owedge a_2 = \mathsf{inner}(\cap, a_1, a_2)$$

```
Fixpoint inner_rec (f : S.t → S.t → S.t) (a1 a2 : A.t) (rec : Acc AI.R_rm a1) : A.t :=
  match AI.elem_or_empty a1 with
  | inleft (exist s1 Hin) ⇒
      A.union (AM.map (f s1) a2) (inner_rec f (A.remove s1 a1) a2 (AI.Acc_rm_inv rec Hin))
  | inright _ ⇒ A.empty
  end.
Definition inner f a1 a2 := inner_rec f a1 a2 (AI.wf_R_rm a1).
Lemma inner_spec_1 :
  ∀ f a1 a2 s, Proper (S.eq ⟹ S.eq ⟹ S.eq) f →
  A.In s (inner f a1 a2) → ∃ s1, A.In s1 a1 ∧ ∃ s2, A.In s2 a2 ∧ S.Equal s (f s1 s2).
Lemma inner_spec_2 :
  ∀ f a1 a2 s1 s2 s, Proper (S.eq ⟹ S.eq ⟹ S.eq) f →
  A.In s1 a1 → A.In s2 a2 → S.Equal s (f s1 s2) → A.In s (inner f a1 a2).
```

We can then define the optimized conjunction and prove Theorem 40:

```
Definition convex_conjunction (acc1 acc2 : A.t) : A.t :=
  from_min_max (min_elements (inner S.union (min_elements acc1) (min_elements acc2)))
               (max_elements (inner S.inter (max_elements acc1) (max_elements acc2))).
Theorem conjunction_equivalence :
  ∀ acc1 acc2, ConvexClosed acc1 → ConvexClosed acc2 →
  A.Equal (convex_conjunction acc1 acc2) (conjunction acc1 acc2).
```

Quotient

Let us now consider the quotient operation. We define it and prove that it preserves convexity (Proposition 3).

```
Definition quotient (a1 a2 : A.t) : A.t :=
  A.filter (fun x ⇒ A.for_all (fun x2 ⇒ A.mem (S.inter x x2) a1) a2) (P.powerset Σ.elements).
Theorem quotient_convex :
  ∀ a1 a2, ConvexClosed a1 → ConvexClosed a2 → ConvexClosed (quotient a1 a2).
```

The definition of the optimized quotient operation on convex-closed acceptance sets involves indexed pointwise union and intersection ($\bigcup\!\!\!\!\cdot$, $\bigcap\!\!\!\!\cdot$). We define these operations using the Indexed module instantiated for acceptance sets AIdx.

```
Definition indexed_inner_union := AIdx.indexed (A.singleton S.empty) (inner S.union).
Definition indexed_inner_inter := AIdx.indexed (A.singleton Σ.elements) (inner S.inter).
```

We now define the optimized quotient operation, as given by Theorem 42:

```
Definition convex_quotient (acc1 acc2 : A.t) : A.t :=
  from_min_max
    (min_elements
      (indexed_inner_union
        (fun min2 ⇒ A.filter (fun min1 ⇒ S.subset min1 min2) (min_elements acc1))
        (min_elements acc2)))
    (max_elements
      (indexed_inner_inter
        (fun max2 ⇒ AM.map (fun max1 ⇒ S.union max1 (S.diff Σ.elements max2))
                           (max_elements acc1))
        (max_elements acc2))).
```

The proof that this definition is equivalent to the quotient on acceptance specifications is fairly long (around 1000 lines of Coq to prove the half-dozen or so of intermediate lemmas), but it follows quite closely the mathematical proof given earlier and allows us to conclude that:

```
Theorem quotient_equivalence :
  ∀ acc1 acc2, ConvexClosed acc1 → ConvexClosed acc2 →
  A.Equal (quotient acc1 acc2) (convex_quotient acc1 acc2).
```

Alphabet extensions

We now study alphabet extensions. We first define a predicate expressing that an acceptance set is defined on a given alphabet Σ':

```
Definition OnAlphabet (acc : A.t) (Σ' : S.t) := ∀ x, A.In x acc → S.Subset x Σ'.
```

We first define the weak extension operation. The mathematical definition given earlier is $\{X \cup \sigma \mid X \in \mathrm{Acc} \wedge \sigma \subseteq \Sigma' \setminus \Sigma\}$. In the Coq development, we express it as $\mathrm{Acc} \uplus 2^{\Sigma' \setminus \Sigma}$. It is strictly equivalent (see the definition of $\uplus$ above) but allows us to reuse lemmas proved on $\uplus$.

```
Definition weak_extension (Σ1 Σ2 : S.t) (acc : A.t) : A.t := inner S.union acc (P.powerset (S.diff Σ2 Σ1)).
```

Then, we can prove that convex-closure is indeed preserved by weak extension (Proposition 6):

```
Theorem weak_extension_convex :
  ∀ Σ1 Σ2 acc, S.Subset Σ1 Σ2 → OnAlphabet acc Σ1 → ConvexClosed acc →
  ConvexClosed (weak_extension Σ1 Σ2 acc).
```

Finally, we define the optimized weak extension operation and prove that it is equivalent to the previous definition following Theorem 43.

```
Definition convex_weak_extension (Σ1 Σ2 : S.t) (acc : A.t) : A.t :=
  from_min_max (min_elements acc)
               (AM.map (S.union (S.diff Σ2 Σ1)) (max_elements acc)).
Theorem weak_extension_equivalence :
  ∀ Σ1 Σ2 acc, S.Subset Σ1 Σ2 → OnAlphabet acc Σ1 → ConvexClosed acc →
  A.Equal (weak_extension Σ1 Σ2 acc) (convex_weak_extension Σ1 Σ2 acc).
```

Similarly, we define strong extension, prove that it preserves convex-closure (Proposition 6), define the optimized extension on convex-closed acceptance sets, and prove that both definitions are equivalent (Theorem 43):

```
Definition strong_extension (Σ1 Σ2 : S.t) (acc : A.t) : A.t := AM.map (S.union (S.diff Σ2 Σ1)) acc.
Theorem strong_extension_convex :
```

```
  ∀ Σ1 Σ2 acc, S.Subset Σ1 Σ2 → OnAlphabet acc Σ1 → ConvexClosed acc →
  ConvexClosed (strong_extension Σ1 Σ2 acc).
Definition convex_strong_extension (Σ1 Σ2 : S.t) (acc : A.t) : A.t :=
  from_min_max (AM.map (S.union (S.diff Σ2 Σ1)) (min_elements acc))
                       (AM.map (S.union (S.diff Σ2 Σ1)) (max_elements acc)).
Theorem strong_extension_equivalence :
  ∀ Σ1 Σ2 acc, S.Subset Σ1 Σ2 → OnAlphabet acc Σ1 → ConvexClosed acc →
  A.Equal (strong_extension Σ1 Σ2 acc) (convex_strong_extension Σ1 Σ2 acc).
```

Product

We finally consider the case of the product operation. Although it does not preserve convex-closure, we can still define an optimized operation on convex-closed acceptance sets, which returns an acceptance set that may not be convex-closed. We thus define the product on acceptance sets, which is the intersection of their elements, and the optimized product on convex-closed sets and prove that they are equivalent, as described in Theorem 41.

```
Definition product (a1 a2 : A.t) : A.t :=
  inner S.inter a1 a2.
Definition convex_product (a1 a2 : A.t) : A.t :=
  union_min_max (fun i1 ⇒
    union_min_max (fun i2 ⇒
      acc_of_interval {| min := S.inter i1.(min) i2.(min); max := S.inter i1.(max) i2.(max) |}
    ) a2
  ) a1.
Theorem product_equivalence :
  ∀ a1 a2, ConvexClosed a1 → ConvexClosed a2 → A.Equal (product a1 a2) (convex_product a1 a2).
End Convex.
```

We will now finish the presentation of this mechanization by instantiating the module we just defined on a specific alphabet. We will use this instantiation to define the counter-example proving that the product does not preserve convex-closure (see Figure 3.5).

We first define a module for the alphabet $\Sigma = \{a, b, c\}$. We have to provide the type of actions and prove that the equality is decidable, then we use Make_UDT (short for *Make Usual Decidable Type*) to generate a module of type DecidableType using Coq's default equality.

```
Module Import Σ_def <: MiniDecidableType.
  Inductive t_ := A | B | C. Definition t := t_.
  Theorem eq_dec : ∀ x y : t, {x = y} + {x ≠ y}.
End Σ_def.
Module Σ_type := Make_UDT Σ_def.
```

Then, we have to extend this module with a comparison operation—we arbitrarily define $A < B < C$. Note that we have to define two functions: lt is, like eq, a logic predicate while compare is an executable function that tells, given two actions, if the first one is lower than, equal to, or greater than the second one.

```
Module Σ_ordered <: OrderedType.
  Include Σ_type.
  Definition lt (a b : t) :=
    match a, b with
    | A, A ⇒ False
    | A, _ ⇒ True
```

```
    | B, C ⇒ True
    | _, _ ⇒ False
    end.
  Instance lt_strorder : StrictOrder lt.
  Definition compare a b :=
    match a, b with
    | A, A ⇒ Eq | B, B ⇒ Eq | C, C ⇒ Eq
    | A, _ ⇒ Lt | B, C ⇒ Lt
    | _, _ ⇒ Gt
    end.
  Theorem compare_spec : ∀ a b, CompareSpec (eq a b) (lt a b) (lt b a) (compare a b).
End Σ_ordered.
```

We then make modules for sets of actions and acceptance sets using the implementation of MSet based on ordered lists. We could easily use another implementation, for instance one based on AVL or red-black trees.

```
Module S := MSetList.Make Σ_ordered.
Module A := MSetList.Make S.
```

Last, we have to prove that the alphabet is finite by providing a set containing all its elements, and then we can instantiate our Convex module.

```
Module Σ_fin <: FiniteType Σ_ordered S.
  Definition elements := S.add A (S.add B (S.singleton C)).
  Theorem finite : ∀ x, S.In x elements.
End Σ_fin.
Module Import C := Convex Σ_ordered S A Σ_fin.
```

We can then prove that there exist two convex-closed acceptance sets which product is not convex-closed:

```
Theorem product_not_convex :
  ∃ acc1, ∃ acc2, ConvexClosed acc1 ∧ ConvexClosed acc2 ∧ ¬ConvexClosed (product acc1 acc2).
Proof.
  ∃ (A.add (S.singleton C) (A.singleton (S.add A (S.singleton B)))).
  ∃ (A.singleton (S.add A (S.singleton B))).
  ...
Qed.
```

3.3.7 Nondeterminism

In the previous sections, we considered only *deterministic* specifications. There is also a nondeterministic specification theory based on acceptance sets [BDF+13, BFK+14], so it could be interesting to see whether using convex-closed sets could also improve the efficiency of operations in the nondeterministic setting based on these initial works.

First, while disjunctive modal specifications, modal specifications with obligations, convex acceptance specifications, and acceptance specifications have a strictly increasing expressiveness in the deterministic case, adding nondeterminism flattens this hierarchy: [BDF+13, BFK+14] proves that with nondeterminism, disjunctive modal specifications are equivalent to acceptance specifications. We can thus conjecture that a nondeterministic specification theory based on convex-closed acceptance sets would be as expressive as acceptance specifications (and disjunctive modal specifications).

In addition, translations between disjunctive modal specifications and acceptance specifications incur an exponential blowup in both directions [BFK$^+$14]. Therefore, some operations, although possible in theory, become intractable. For instance, [BDF$^+$13, BFK$^+$14] defines a quotient on acceptance specifications but not on disjunctive modal specifications; the quotient of two disjunctive modal specifications can be computed by first translating them to acceptance specifications, then performing the quotient operation, and last translating the result back to disjunctive modal specifications—each step having an exponential blow-up. On the other hand, a specification theory based on convex-closed sets may offer the same operations with a lower complexity. We know that in the deterministic case translating a disjunctive modal specification into a convex acceptance specification has no exponential blow-up while there is one when translating to acceptance specifications; we conjecture that there is a similar improvement in the nondeterministic case.

However, adding nondeterminism to convex acceptance sets is not a direct and trivial extension of the results on deterministic specifications. In particular, it requires adapting the notion of convexity on sets since the acceptance sets of nondeterministic specifications contain not just actions, but pairs with an action and the destination state of the corresponding transition. Moreover, operations on nondeterministic specifications have much more complex definitions. For instance, while the acceptance set of a state (q_1, q_2) of the quotient of two deterministic acceptance specifications is given by a single formula $(\{X \mid \forall X_2 \in \mathrm{Acc}_2(q_2), X \cap X_2 \in \mathrm{Acc}_1(q_1)\})$, the quotient given in [BDF$^+$13, BFK$^+$14] has a similar flavor but with a more complex algorithm involving several intermediate definitions (see the objects α, γ, π_a, pt_a, pt in their paper). Ensuring that such an operation preserves convexity and then finding an optimized algorithm using only the minimal and maximal elements of a convex-closed acceptance set to obtain the same result may be challenging.

Chapter 4

Marked Acceptance Specifications

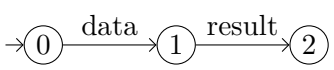
We now present the second main theoretical contribution of this thesis: an extension of acceptance specifications allowing to express reachability properties. We first give the semantics of this new formalism and then show how to extend the operations of conjunction, product, and quotient on acceptance specifications to this new formalism while guaranteeing the preservation of the reachability properties, i.e., the absence of deadlocks and livelocks. This formalism and the operation of quotient, which is the most difficult to define, were published in [VR15b].

4.1 Semantics

The formalisms we have studied until now—modal, acceptance, and convex-closed acceptance specifications—all express local constraints: in each state of the specification, we indicate which transitions or groups of transitions are required, allowed, or forbidden. But we may want to express constraints not just on the transitions from each state, but globally on the paths in each model. Concrete examples abound in practice. For instance, consider Service Oriented Architectures (SOA) formed of several interacting services; it should always be the possible to reach a termination state of a session.

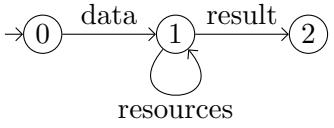
Consider the acceptance specification of a simple server given by Figure 4.1: it receives some data, computes a value from the data, and sends it back. Now, this server may need more resources to do the computation; for instance, if the input data is too large, it may require more memory. A way to express this is to add an optional transition from state 1 to ask for additional resources, as depicted in Figure 4.2. This allows models like the one in Figure 4.3(a) which requests additional resources and then computes the result. However, models are also allowed to request resources, possibly infinitely, and never use them, as shown in Figure 4.3(b). We could try to change the acceptance set of state 1 of the specification, but we could never allow models to request additional resources an arbitrary number of times while requiring that they eventually send a result.

To express this kind of requirements, we need to be able to express constraints not just on the transitions of a given state, but on paths. A way to do so is to extend the specification



$$\mathrm{Acc}(0) = \{\{\text{data}\}\}$$
$$\mathrm{Acc}(1) = \{\{\text{result}\}\}$$
$$\mathrm{Acc}(2) = \{\emptyset\}$$

Figure 4.1: A simple server computing something from some input data



$\text{Acc}(0) = \{\{\text{data}\}\}$
$\text{Acc}(1) = \{\{\text{result}\}, \{\text{resources}\}, \{\text{result}, \text{resources}\}\}$
$\text{Acc}(2) = \{\emptyset\}$

Figure 4.2: The server, allowed to request additional resources

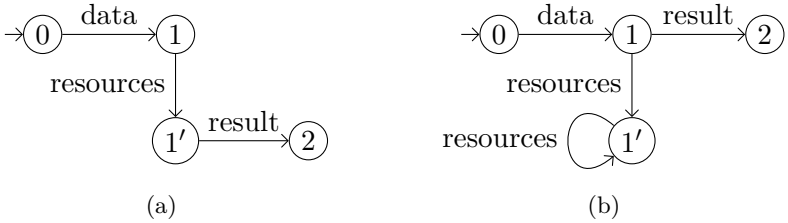


Figure 4.3: Models of the specification in Figure 4.2

formalism with marked states. Then, models are required to have a marked state reachable from any state. An extension of modal specifications with marked states was introduced in [CR12]. We combine acceptance specifications with marked states to form marked acceptance specifications. For our example, we use a marked acceptance specification and mark the last state, as depicted in Figure 4.4 (marked states are circled twice), ensuring that it will eventually be reached in all the models of the specification. Indeed, the automaton of Figure 4.3(a) is a model of this marked acceptance specification since it satisfies the underlying acceptance specification and the marked state 2 is reachable from any state. On the other hand, the automaton of Figure 4.3(b) is not a model of the specification as it is not possible to reach the state 2 from $1'$.

In this example, adding marked states allowed us to express a termination property: the terminating state 2 must always be reachable from any other state. We can also use marked states to express a liveness property. We could for example modify our server to allow it to answer multiple requests, as depicted in Figure 4.5: the initial marked state is a checkpoint which has to be reachable infinitely often.

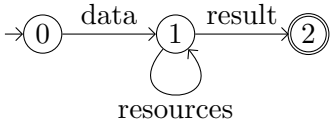
The definition of marked acceptance specifications is a simple extension of acceptance specifications as defined in Chapter 3, Definition 17: we just add a set of *marked states*.

Definition 32 (Marked Acceptance Specification)**.** *A marked acceptance specification over an alphabet Σ is a tuple $S = (Q, q^0, \delta, \text{Acc}, F)$ where Q is a finite set of states, $q^0 \in Q$ is the unique initial state, $\delta : Q \times \Sigma \to Q$ is the partial labeled transition map, $\text{Acc} : Q \to 2^{2^\Sigma}$ associates to each state a set of ready sets called its acceptance set, and $F \subseteq Q$ is a set of marked states.*

The underlying specification of S, denoted $\text{Un}(S)$, is the acceptance specification $(Q, q^0, \delta, \text{Acc})$.

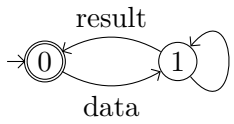
We also define a special empty marked acceptance specification $S_\perp$, which has no models.

The models of marked acceptance specifications are marked automata, i.e., automata with some marked states:



$\text{Acc}(0) = \{\{\text{data}\}\}$
$\text{Acc}(1) = \{\{\text{result}\}, \{\text{resources}\}, \{\text{result}, \text{resources}\}\}$
$\text{Acc}(2) = \{\emptyset\}$

Figure 4.4: The server with a marked state



$\mathrm{Acc}(0) = \{\{\mathrm{data}\}, \emptyset\}$
$\mathrm{Acc}(1) = \{\{\mathrm{result}\}, \{\mathrm{resources}\}, \{\mathrm{result}, \mathrm{resources}\}\}$

Figure 4.5: The server allowed to answer multiple requests

Definition 33 (Marked Automaton)**.** *A deterministic marked automaton over an alphabet* Σ *is a tuple* $M = (R, r^0, \lambda, G)$ *where* R *is a finite set of states,* $r^0 \in R$ *is the unique initial state,* $\lambda : R \times \Sigma \to R$ *is the partial labeled transition map, and* $G \subseteq R$ *is a set of marked states.*

Definition 34. *Given an automaton* M *and a state* r *of* M*, we define* $\mathrm{pre}^*(r)$ *and* $\mathrm{post}^*(r)$ *as the sets of states that are respectively co-reachable and reachable from* r*: they are the smallest sets such that* $r \in \mathrm{pre}^*(r)$*,* $r \in \mathrm{post}^*(r)$*, and for any* r'*,* a *and* r'' *such that* $\lambda(r', a) = r''$*,* $r' \in \mathrm{pre}^*(r)$ *if* $r'' \in \mathrm{pre}^*(r)$ *and* $r'' \in \mathrm{post}^*(r)$ *if* $r' \in \mathrm{post}^*(r)$*.*

We also define $\mathrm{pre}^+(r)$ *as the union of* $\mathrm{pre}^*(r')$ *for all* r' *such that* $\exists a, \lambda(r', a) = r$ *and* $\mathrm{post}^+(r)$ *as the union of* $\mathrm{post}^*(\lambda(r, a))$ *for all* $a \in \mathrm{ready}(r)$*. Let* $\mathrm{Loop}(r) = \mathrm{pre}^+(r) \cap \mathrm{post}^+(r)$*.*

Definition 35 (Terminating automaton)**.** *An automaton is said to be* terminating *if a marked state is reachable from any state, that is if for any state* r *of the automaton,* $\mathrm{post}^*(r) \cap G \neq \emptyset$*.*

Remark. Testing if $\forall r, \mathrm{post}^*(r) \cap G \neq \emptyset$ is equivalent to testing if $\bigcup_{r \in G} \mathrm{pre}^*(r) = R$ and this last formula is typically much more efficient to compute.

Complexity 5. Computing the set of states reachable or co-reachable from a state or a set of states is a classical graph problem that can be solved for instance with a depth-first or breadth-first search algorithm which has a complexity linear w.r.t. the number of edges in the graph, i.e., $O(|R| \times |\,\mathrm{ready}\,|)$. As noted in the previous remark, deciding if an automaton is terminating amounts to testing if the set of states co-reachable from the marked states is equal to R, so it has the same complexity $O(|R| \times |\,\mathrm{ready}\,|)$.

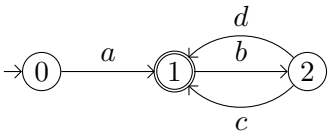
Then, a marked automaton is a model of a marked acceptance specification if and only if the non-marked automaton is a model of the non-marked acceptance specification, the automaton is terminating, and the marked states of the automaton are matched with marked states of the specification.

Definition 36 (Satisfaction)**.** *A terminating automaton* M satisfies *a marked acceptance specification* S*, denoted* $M \models S$*, if and only if there exists a simulation relation* $\pi \subseteq R \times Q$ *such that* $(r^0, q^0) \in \pi$ *and, for all* $(r, q) \in \pi$*:*

- $\mathrm{ready}(r) \in \mathrm{Acc}(q)$*;*
- *if* $r \in G$ *then* $q \in F$*;*
- *for any* $a \in \mathrm{ready}(r)$*, we have* $(\lambda(r, a), \delta(q, a)) \in \pi$*.*

Observe that the second item of Definition 36 is an implication and not an equivalence: a marked state in the specification may be realized by a non-marked state in the automaton. This allows *delaying* the reachability of a marked state.

Complexity 6. Checking satisfaction requires iterating on all the pairs of states (r, q) in the simulation relation π; in the worst case, there are $|R| \times |Q|$ such pairs. Then, for each pair, there are three tests:



$$\begin{aligned}\mathrm{Acc}(0) &= \{\{a\}\}\\ \mathrm{Acc}(1) &= \{\{b\}\}\\ \mathrm{Acc}(2) &= \{\{c\},\{d\},\{c,d\}\}\end{aligned}$$

Figure 4.6: An example of marked acceptance specification

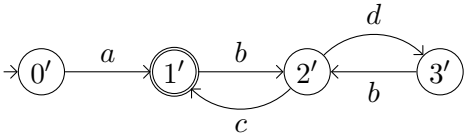


Figure 4.7: A model of the marked acceptance specification of Figure 4.6

- $\mathrm{ready}(r) \in \mathrm{Acc}(q)$ has a complexity of $O(|\,\mathrm{Acc}\,|)$ (naturally, using another type of data structure to represent acceptance sets may change this complexity);
- if $r \in G$ then $q \in F$: $O(1)$ since we count operations on simple sets (i.e., sets of states or actions);
- $\forall a \in \mathrm{ready}(r), (\lambda(r,a), \delta(q,a)) \in \pi$: $O(|\,\mathrm{ready}\,|)$.

This yields a complexity of $O(|R| \times |Q| \times (|\,\mathrm{Acc}\,| + |\,\mathrm{ready}\,|))$.

Figure 4.6 depicts a marked acceptance specification and the automaton of Figure 4.7 is one of its models because of the simulation relation $\pi = \{(0',0),(1',1),(2',2),(3',1)\}$. Note that while the state 1 is marked and $(3',1) \in \pi$, $3'$ is not marked. This is allowed because it is still possible to reach the marked state $1'$.

The refinement relation on acceptance specifications is easily extended to marked acceptance specifications, the same way as the satisfaction relation.

Definition 37 (Refinement)**.** *Given two marked acceptance specifications S_1 and S_2, S_1 is a refinement of S_2, denoted $S_1 \leq S_2$, if and only if there exists a simulation relation $\pi \subseteq Q_1 \times Q_2$ such that $(q_1^0, q_2^0) \in \pi$ and for all pairs $(q_1, q_2) \in \pi$:*

- $\mathrm{Acc}_1(q_1) \subseteq \mathrm{Acc}_2(q_2)$*;*
- *if $q_1 \in F_1$ then $q_2 \in F_2$;*
- *for any $a \in \mathrm{ready}(q_1)$, we have $(\delta_1(q_1,a), \delta_2(q_2,a)) \in \pi$.*

Moreover, for any specification S, $S_\perp \leq S$.

Complexity 7. Refinement checking follows the same pattern as satisfaction checking. Its complexity is thus $O(|Q_1| \times |Q_2| \times (|\,\mathrm{Acc}_1\,| \times |\,\mathrm{Acc}_2\,| + |\,\mathrm{ready}\,|))$.

As for acceptance specifications, the refinement relation on marked acceptance specifications is a thorough refinement: it is equivalent to the inclusion of the sets of models.

Theorem 44. *Given two marked acceptance specifications S_1 and S_2, $S_1 \leq S_2$ if and only if $[\![S_1]\!] \subseteq [\![S_2]\!]$.*

Proof. ($\Rightarrow$) Suppose that $S_1 \leq S_2$ and $M \models S_1$ thanks respectively to the simulation relations π and π_1. Define π_2 such that $(r, q_2) \in \pi_2$ if and only if there exists a state q_1 in S_1 such that $(r, q_1) \in \pi_1$ and $(q_1, q_2) \in \pi$. We prove that $M \models S_2$ thanks to π_2:

- if $(r, q_1) \in \pi_1$ then $\text{ready}(r) \in \text{Acc}_1(q_1)$ by Definition 36; moreover, if $(q_1, q_2) \in \pi$ then $\text{Acc}_1(q_1) \subseteq \text{Acc}_2(q_2)$ by Definition 37. As a result, $\text{ready}(r) \in \text{Acc}_2(q_2)$;
- if $(r, q_1) \in \pi_1$ then $r \in G$ implies $q_2 \in F_2$ by Definition 36; moreover, if $(q_1, q_2) \in \pi$ then $q_1 \in F_1$ implies $q_2 \in F_2$ by Definition 37. As a result, $r \in G$ implies $q_2 \in F_2$;
- for any $a \in \text{ready}(r)$, if $(r, q_1) \in \pi_1$ then $(\lambda(r, a), \delta_1(q_1, a)) \in \pi_1$ by Definition 36; moreover, if $(q_1, q_2) \in \pi$ then $(\delta_1(q_1, a), \delta_2(q_2, a)) \in \pi$ by Definition 37. As a result, we have: $(\lambda(r, a), \delta_2(q_2, a)) \in \pi_2$.

($\Leftarrow$) Suppose that $[\![S_1]\!] \subseteq [\![S_2]\!]$. Define π such that $(q_1^0, q_2^0) \in \pi$ and for all $(q_1, q_2) \in \pi$, if $\delta_1(q_1, a)$ and $\delta_2(q_2, a)$ are defined then $(\delta_1(q_1, a), \delta_2(q_2, a)) \in \pi$. We prove that $S_1 \leq S_2$ thanks to π.

Observe first that if $\delta_1(q_1, a)$ is defined then $\delta_2(q_2, a)$ is also defined; this is a direct consequence to the fact that when $\delta_1(q_1, a)$ is defined, the transition can be included in some models which are also models of S_2 and thus $\delta_2(q_2, a)$ is defined.

- for all $X \in \text{Acc}_1(q_1)$, there exists some $M \models S_1$ such that $(r, q_1) \in \pi_1$ and $\text{ready}(r) = X$. As $[\![S_1]\!] \subseteq [\![S_2]\!]$, M is also a model of S_2 and necessarily $\text{ready}(r) \in \text{Acc}_2(q_2)$. As a result, $\text{Acc}_1(q_1) \subseteq \text{Acc}_2(q_2)$;
- suppose that $q_1 \in F_1$ and consider $M \models S_1$ such that $(r, q_1) \in \pi_1$ and $r \in G$. As $[\![S_1]\!] \subseteq [\![S_2]\!]$, M is also a model of S_2 and necessarily $q_2 \in F_2$. As a result, $q_1 \in F_1$ implies $q_2 \in F_2$;
- by definition of π, for any $a \in \text{ready}(q_1)$, we have $(\delta_1(q_1, a), \delta_2(q_2, a))$.

As a result, according to Definition 37, we have $S_1 \leq S_2$. □

Acceptance specifications have a normal form which ensures that their transitions and acceptance sets are consistent with each other. We extend the definition of normal form with additional requirements on marked states:

- *attractability.* A marked acceptance specification is attracted in q when $\text{post}^*(q) \cap F \neq \emptyset$.
- F, Acc-*consistency.* A state q is F, Acc-consistent when $\emptyset \in \text{Acc}(q)$ implies $q \in F$.

We now extend the algorithm computing the normal form of an acceptance specification (Algorithm 1) to marked acceptance specifications and prove that it is correct:

Theorem 45. *For any marked acceptance specification S, $\rho(S)$ is in normal form and is equivalent to S.*

Proof. (normal form) The base case of the recursive definition of ρ is that there is no state q such that $\text{Acc}(q) = \emptyset$, $\text{post}^*(q) \cap F = \emptyset$, $\emptyset \in \text{Acc}(q) \wedge q \notin F$ or $\text{ready}(q) \neq \bigcup \text{Acc}(q)$. This implies that if ρ terminates, the returned specification is Acc-consistent, δ, Acc-consistent, attracted, and F, Acc-consistent, hence in normal form. Each time the function ρ is recursively called, its parameter has fewer states, fewer transitions or smaller acceptance sets. Considering that acceptance specifications are finite, ρ is terminating.

(equivalence) By induction:

Algorithm 2 $\rho(S\text{: MAS})\text{: MAS}$

1: **if** $\exists q, \mathrm{Acc}(q) = \emptyset \vee \mathrm{post}^*(q) \cap F = \emptyset$ **then**
2: **if** $q = q^0$ **then**
3: **return** $S_\perp$
4: **else**
5: $\delta' = \{(q', a) \mapsto \delta(q', a) \mid \delta(q', a) \text{ defined} \wedge \delta(q', a) \neq q\}$
6: $\mathrm{Acc}' = \{q' \mapsto \{X \mid X \in \mathrm{Acc}(q') \wedge \forall a \in X, \delta(q', a) \neq q\}\}$
7: **return** $\rho((Q \setminus \{q\}, q^0, \delta', \mathrm{Acc}', F \setminus \{q\}))$
8: **end if**
9: **end if**
10: **if** $\exists q, \mathrm{ready}(q) \neq \bigcup \mathrm{Acc}(q)$ **then**
11: $\delta' = \{(q', a) \mapsto \delta(q', a) \mid \delta(q', a) \text{ defined} \wedge a \in \bigcup \mathrm{Acc}(q')\}$
12: $\mathrm{Acc}' = \{q' \mapsto \{X \mid X \in \mathrm{Acc}(q') \wedge \forall a \in X, \delta(q', a) \text{ defined}\}\}$
13: **return** $\rho((Q, q^0, \delta', \mathrm{Acc}', F))$
14: **end if**
15: **if** $\exists q, \emptyset \in \mathrm{Acc}(q) \wedge q \notin F$ **then**
16: $\mathrm{Acc}' = \{q \mapsto \mathrm{Acc}(q) \setminus \{\emptyset\}\} \cup \{q' \mapsto \mathrm{Acc}(q') \mid q \neq q'\}$
17: **return** $\rho((Q, q^0, \delta, \mathrm{Acc}', F))$
18: **end if**
19: **return** S

- In the base case (line 19), the specification S itself is returned.
- For the first recursive call (line 7), we remove from S the state q and the transitions from other states towards q.
  - If the acceptance set of q is empty, no model of S can implement q (the condition $\mathrm{ready}(r) \in \mathrm{Acc}(q)$ implies that $\mathrm{Acc}(q)$ must not be empty).
  - If there is no marked state reachable from q, no model of S can implement q as it would not be terminating.

  In consequence, the specification passed to the recursive call has the same models as S.
- For the second recursive call (line 13), we removed some transitions which were not allowed by the corresponding acceptance set, and thus could not be realized by any model (the condition $\mathrm{ready}(r) \in \mathrm{Acc}(q)$ would not be satisfiable), as well as elements of the acceptance set containing actions for which δ is not defined, which could not be realized in any model either. Thus, the specification passed to the recursive call also has the same models as S.
- For the third recursive call (line 17), if a model of S implements q, there must be at least one transition from q to another state in order to ensure termination, as q is not marked. Consequently, the ready set of the state implementing q must not be empty and removing $\emptyset$ from $\mathrm{Acc}(q)$ does not change the set of models. □

Complexity 8. We first analyze the complexity of each step of the algorithm. Afterward, we will estimate the number of recursive calls to obtain the complexity of the algorithm.

- The first test (line 1) iterates on the states of S and tests for each state q if $\mathrm{Acc}(q) = \emptyset$ ($O(1)$) and if $\mathrm{post}^*(q) \cap F = \emptyset$ ($O(|Q| \times |\,\mathrm{ready}\,|)$ as explained previously); thus the complexity of this test is $O(|Q|^2 \times |\,\mathrm{ready}\,|)$.

- If $q = q^0$, the algorithm ends; otherwise, computing δ' requires iterating on all the transitions of the automaton to remove those going to q, which is $O(|Q| \times |\operatorname{ready}|)$, and similarly, computing Acc' has a complexity of $O(|Q| \times |\operatorname{Acc}|)$.
- The complexity of the second test (line 10) is $O(|Q| \times |\operatorname{Acc}|)$.
- Then, computing δ' requires iterating on all the states of S and for each state q' computing $\bigcup \operatorname{Acc}(q')$ and testing for each $a \in \operatorname{ready}(q')$ if a belongs to this set, which implies a complexity of $O(|Q| \times (|\operatorname{Acc}| + |\operatorname{ready}|))$. For Acc', we iterate on the states of S and on the elements X of the corresponding acceptance set; then, testing if $\forall a \in X, \delta(q', a)$ is defined is equivalent to testing if $X \subseteq \operatorname{ready}(q')$ which is $O(1)$, yielding a complexity of $O(|Q| \times |\operatorname{Acc}|)$.
- The last part of the algorithm has a complexity of $O(|Q|)$.

The final element to determine the complexity of the algorithm is the number of recursive calls:

- For the first case (lines 1–9), a state is removed before the recursive call; since we never add states in the algorithm, there are at most $O(|Q|)$ recursive calls at line 7.
- In the second case (lines 10–14) we ensure that the transition function and the acceptance sets are consistent with each other. Whenever we modify these values in other parts of the algorithm (lines 5, 6 and 16), this consistency property is preserved. Thus, in the worst case, all the states are initially $\delta, \operatorname{Acc}$-inconsistent and there are $O(|Q|)$ recursive calls at line 13.
- Finally, the lase case (lines 15–18) removes F, Acc-inconsistencies. As in the previous case, the other parts of the algorithm cannot introduce such inconsistency, so there are $O(|Q|)$ recursive calls at line 17.

By combining these results, we find that the complexity of Algorithm 2 is:

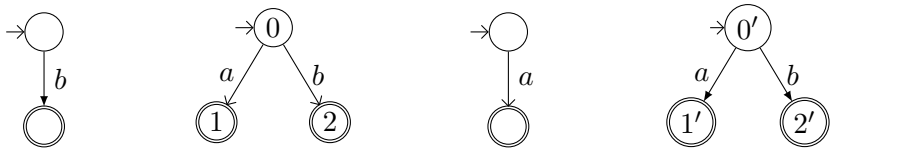
$$O(|Q|^3 \times |\operatorname{ready}| + |Q|^2 \times |\operatorname{Acc}|)$$

4.2 Conjunction

The conjunction operation on marked acceptance specifications is a direct extension of the conjunction operation on acceptance specifications where the set of marked states of the conjunction is the cartesian product of the sets of marked states of the specifications.

Definition 38 (Conjunction)**.** *Given two marked acceptance specifications S_1 and S_2, their conjunction, denoted $S_1 \wedge S_2$, is the normal form of S_1 & $S_2 = (Q_1 \times Q_2, (q_1^0, q_2^0), \delta, \operatorname{Acc}, F_1 \times F_2)$ with $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ when both $\delta_1(q_1, a)$ and $\delta_2(q_2, a)$ are defined, and $\operatorname{Acc}((q_1, q_2)) = \operatorname{Acc}_1(q_1) \cap \operatorname{Acc}_2(q_2)$.*

Complexity 9. The conjunction of two marked acceptance specifications has at most $|Q_1| \times |Q_2|$ states. For each state, we compute the intersection of their acceptance sets which complexity is assumed to be $O(|\operatorname{Acc}_1| \times |\operatorname{Acc}_2|)$. The transition function is built from the transitions present in both specifications, so the complexity is $O(\min(|\delta_1|, |\delta_2|))$. Thus, the complexity of computing S_1 & S_2 is $O(|Q_1| \times |Q_2| \times (|\operatorname{Acc}_1| \times |\operatorname{Acc}_2| + \min(|\delta_1|, |\delta_2|)))$. Then, we have to apply ρ in order to guarantee that the result is in normal form; the complexity of this step is $O((|Q_1| \times |Q_2|)^3 \times \min(|\operatorname{ready}_1|, |\operatorname{ready}_2|) + (|Q_1| \times |Q_2|)^2 \times \min(|\operatorname{Acc}_1|, |\operatorname{Acc}_2|))$.



$\mathrm{Acc}(0) = \{\{b\}, \{a, b\}\}$
$\mathrm{Acc}(1) = \mathrm{Acc}(2) = \{\emptyset\}$

$\mathrm{Acc}(0') = \{\{a\}, \{a, b\}\}$
$\mathrm{Acc}(1') = \mathrm{Acc}(2') = \{\emptyset\}$

(a) M_1 (b) S_1 (c) M_2 (d) S_2

Figure 4.8: Reachability is not compositional

As for acceptance specifications, the conjunction of two marked acceptance specifications characterizes precisely the intersection of the sets of models of its operands:

Theorem 46. *Given two marked acceptance specifications S_1 and S_2, $[\![S_1 \wedge S_2]\!] = [\![S_1]\!] \cap [\![S_2]\!]$.*

Proof. ($\supseteq$) Assume that $M \models S_i$ thanks to π_i for $i = 1, 2$ and define π such that $(r, (q_1, q_2)) \in \pi$ if and only if $(r, q_i) \in \pi_i$. We show that $M \models S_1 \wedge S_2$ using π as simulation relation:

- $\mathrm{ready}(r) \in \mathrm{Acc}_i(q_i)$ as $(r, q_i) \in \pi_i$ and thus $\mathrm{ready}(r) \in \mathrm{Acc}((q_1, q_2))$ by definition of $\wedge$;
- $r \in G$ implies that $(q_1, q_2) \in F_1 \times F_2$ as $r \in G$ implies that $q_i \in F_i$;
- for any a and r' such that $\lambda(r, a) = r'$, $(r', \delta((q_1, q_2), a)) \in \pi$ is trivial as we know that $(r', \delta_i(q_i, a)) \in \pi_i$ and $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$.

($\subseteq$) Assume that $M \models S_1 \wedge S_2$ thanks to π and define π_i for $i = 1, 2$ such that $(r, q_i) \in \pi_i$ if and only if $(r, (q_1, q_2)) \in \pi$. We show that $M \models S_i$ using π_i as simulation relation:

- $\mathrm{ready}(r) \in \mathrm{Acc}_1(q_1) \cap \mathrm{Acc}_2(q_2)$ by definition of $\wedge$ and thus $\mathrm{ready}(r) \in \mathrm{Acc}_i(q_i)$;
- $r \in G$ implies that $q_i \in F_i$ as $r \in G$ implies that $(q_1, q_2) \in F_1 \times F_2$;
- for any a and r' such that $\lambda(r, a) = r'$, $(r', \delta_i(q_i, a)) \in \pi_i$ is trivial as we know that $(r', \delta((q_1, q_2), a)) \in \pi$ and $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$. □

Corollary 3. *For any marked acceptance specifications S_1, S_2 and S, $S_1 \wedge S_2$ is the greatest lower bound of S_1 and S_2 for the refinement relation: $S \leq S_1$ and $S \leq S_2$ if and only if $S \leq S_1 \wedge S_2$.*

Proof. This proposition is a consequence of Theorem 46: if $S \leq S_i$ for $i \in \{1, 2\}$ then, by Theorem 44: $[\![S]\!] \subseteq [\![S_i]\!]$. As a result, we have: $[\![S]\!] \subseteq [\![S_1]\!] \cap [\![S_2]\!]$. By Theorem 46, this is equivalent to: $[\![S]\!] \subseteq [\![S_1 \wedge S_2]\!]$. Last, we deduce from Theorem 44 that $S \leq S_1 \wedge S_2$. □

4.3 Product

We gave a definition of the product of acceptance specifications and would like to extend it to marked acceptance specifications. However, the reachability constraints are not preserved by the product in general. Indeed, Figure 4.8 shows a simple counter-example: $M_1 \models S_1$ and $M_2 \models S_2$; however the product $M_1 \times M_2$ is a single non-marked state, hence the reachability of a marked state is not possible.

This leads us to first consider the following problem: given two marked acceptance specifications, can they be implemented concurrently, i.e., such that the product of any model of the first specification with any model of the second one will be terminating?

An automaton is not terminating if it has a deadlock—a non-marked state with no outgoing transitions—or a livelock—a group of connected non-marked states with no transitions towards the other states. We will first consider the case of deadlock-free products in the following section and then the case of livelock-free products in the next one. We will then define a criterion on marked acceptance specifications, called *compatible reachability*, which is a prerequisite for the product of marked acceptance specifications.

4.3.1 Deadlock-free specifications

In this section, we propose a criterion checking if two marked acceptance specifications have some models which product has a deadlock. We first define the notion of deadlock in an automaton:

Definition 39 (Deadlock)**.** *There is a* deadlock *in an automaton M if there is a state r of M such that r is not marked and* $\text{ready}(r) = \emptyset$.

We now want to identify if two marked acceptance specifications have some models such that their product has a deadlock. We first define compatible acceptance sets:

Definition 40 (Compatible acceptance sets)**.** *Two acceptance sets* Acc_1 *and* Acc_2 *are said to be* compatible*, denoted* $\text{Compat}(\text{Acc}_1, \text{Acc}_2)$*, if and only if for all $X_1 \in \text{Acc}_1$ and $X_2 \in \text{Acc}_2$, $X_1 \cap X_2 \neq \emptyset$.*

We then identify deadlock-free pairs of states, that are pairs of states of two marked acceptance specifications from which no deadlocks may be generated in the product of any two respective implementations:

Definition 41 (Deadlock-free pair of states)**.** *Given two marked acceptance specifications S_1 and S_2, and two states q_1 of S_1 and q_2 of S_2, the pair (q_1, q_2) is said to be deadlock-free, denoted* $\text{DeadFree}(q_1, q_2)$*, if* $\text{Acc}_1(q_1) = \text{Acc}_2(q_2) = \{\emptyset\}$ *or* $\text{Compat}(\text{Acc}_1(q_1), \text{Acc}_2(q_2))$.

Consider for instance the two marked acceptance specifications depicted in Figure 4.8. The pair formed by their initial states is not deadlock-free as $\text{Acc}(0) \neq \{\emptyset\}$, $\text{Acc}(0') \neq \{\emptyset\}$, and $\text{Compat}(\text{Acc}_1(0), \text{Acc}_2(0'))$ is false: $\{b\} \in \text{Acc}_1(0)$, $\{a\} \in \text{Acc}_2(0')$ and $\{b\} \cap \{a\} = \emptyset$.

We then lift the definition of deadlock-free pairs of states to all the relevant pairs of states of the specifications and prove that if the specifications are deadlock-free according to this definition, there will be no deadlocks in the product of any of their models.

Definition 42 (Deadlock-free specifications)**.** *Two marked acceptance specifications S_1 and S_2 are* deadlock-free *if all the reachable pairs of states in* $\text{Un}(S_1) \times \text{Un}(S_2)$ *are deadlock-free.*

Complexity 10. There are at most $|Q_1| \times |Q_2|$ reachable pairs of states in $\text{Un}(S_1) \times \text{Un}(S_2)$. Testing if a pair of states is deadlock-free requires checking $\text{Compat}(\text{Acc}_1, \text{Acc}_2)$ which has a complexity of $O(|\,\text{Acc}_1\,| \times |\,\text{Acc}_2\,|)$. Thus, the complexity of checking deadlock-freeness of two marked acceptance specifications is $O(|Q_1| \times |Q_2| \times |\,\text{Acc}_1\,| \times |\,\text{Acc}_2\,|)$.

Theorem 47. *Two marked acceptance specifications S_1 and S_2 are deadlock-free if and only if for any $M_1 \models S_1$ and $M_2 \models S_2$, $M_1 \times M_2$ is deadlock-free.*

Proof. ($\Rightarrow$) Suppose that (r_1, r_2) is a deadlock in $M_1 \times M_2$. Then (r_1, r_2) is not marked and $\text{ready}((r_1, r_2)) = \emptyset$. Now $\text{ready}((r_1, r_2)) = \text{ready}(r_1) \cap \text{ready}(r_2)$ and moreover, $(r_1, q_1) \in \pi_1$ and $(r_2, q_2) \in \pi_2$ implies $\text{ready}(r_1) \in \text{Acc}_1(q_1)$ and $\text{ready}(r_2) \in \text{Acc}_2(q_2)$. As a result, for $X_1 = \text{ready}(r_1) \in \text{Acc}_1(q_1)$, $X_2 = \text{ready}(r_2) \in \text{Acc}_2(q_2)$, we have: $X_1 \cap X_2 = \emptyset$ and thus $\neg\,\text{Compat}(\text{Acc}_1(q_1), \text{Acc}_2(q_2))$. Moreover, (r_1, r_2) is not marked so (q_1, q_2) is not marked and $\emptyset \notin \text{Acc}_1(q_1)$ and $\emptyset \notin \text{Acc}_2(q_2)$. In consequence, we have $\neg\,\text{DeadFree}(q_1, q_2)$ and S_1 and S_2 are not deadlock-free.

($\Leftarrow$) Suppose that S_1 and S_2 are not deadlock-free: there exist two states q_1 and q_2 such that $\text{DeadFree}(q_1, q_2)$ is false. Then there exist $X_1 \in \text{Acc}_1(q_1)$ and $X_2 \in \text{Acc}_2(q_2)$ which verify $X_1 \cap X_2 = \emptyset$. For any $M_1 \models S_1$ and $M_2 \models S_2$ with $(r_1, q_1) \in \pi_1$ and $(r_2, q_2) \in \pi_2$ such that $\text{ready}(r_1) = X_1$ and $\text{ready}(r_2) = X_2$, we have $\text{ready}((r_1, r_2)) = X_1 \cap X_2 = \emptyset$ in $M_1 \times M_2$. Moreover, $\text{Acc}_1(q_1) \neq \{\emptyset\}$ (or $\text{Acc}_2(q_2) \neq \{\emptyset\}$), so there exists a model of S_1 (resp. S_2) such that a state r implementing q_1 (resp. q_2) is not marked and has at least one transition leading to another marked state, so (r_1, r_2) is not marked. As a result, (r_1, r_2) is a deadlock and $M_1 \times M_2$ is not deadlock-free. □

4.3.2 Livelock-free specifications

In this section, we propose a criterion checking if two marked acceptance specifications have some models which product has a livelock. This criterion is based on the identification of cycles shared between the specifications along with a typing of the transitions leaving these cycles. We then check if it is always possible to leave a cycle, no matter what implementation choices are made.

Before considering the common cycles, a first step consists in unfolding the specifications such that possible synchronizations become unambiguous.

Unfolding

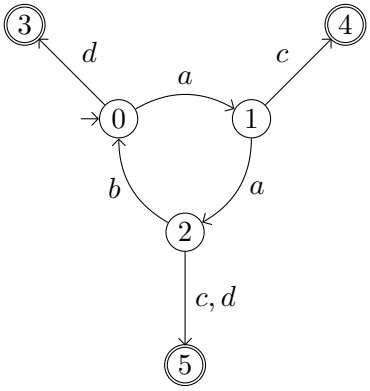
Consider the marked acceptance specifications S_1 and S_2 in Figures 4.9(a) and 4.9(b). The pair of initial states is $(0, 0')$. Both initial states have a transition by a, leading to states 1 and $1'$ from which a transition by a leads to states 2 and $0'$. So, in state $0'$ of S_2, the corresponding state in S_1 may be either 0 or 2. By computing the *unfolding* of S_2 in relation to S_1, we obtain a marked acceptance specification equivalent to S_2 (with the same models) but such that any of its states is only related to at most one state of S_1. This will greatly simplify some operations, such as detecting the livelocks in the cycles and removing potential livelocks.

Given two marked acceptance specifications S_1 and S_2, we define the *partners* of a state q_1 as $Q_2(q_1) = \{q_2 \mid (q_1, q_2) \text{ is reachable in } \text{Un}(S_1) \times \text{Un}(S_2)\}$; the set $Q_1(q_2)$ is defined symmetrically. As a shorthand, if we know that a state q_1 has exactly one partner, we will also use $Q_2(q_1)$ to denote this partner.

We now show that, if some states of S_2 have several partners, it is possible to transform S_2 so that each of its states has at most one partner, while preserving its set of models.

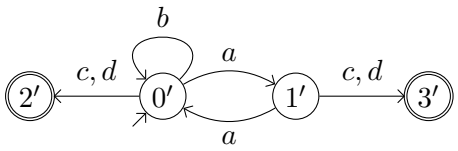
Definition 43 (Unfolding)**.** *Given two marked acceptance specifications S_1 and S_2, the unfolding of S_2 in relation to S_1 is the specification $((Q_1 \cup \{q^?\}) \times Q_2, (q_1^0, q_2^0), \delta_u, \text{Acc}_u, (Q_1 \cup \{q^?\}) \times F_2)$ where:*

- *$q^?$ is a fresh state ($q_1^?$ denotes a state in $Q_1 \cup \{q^?\}$);*



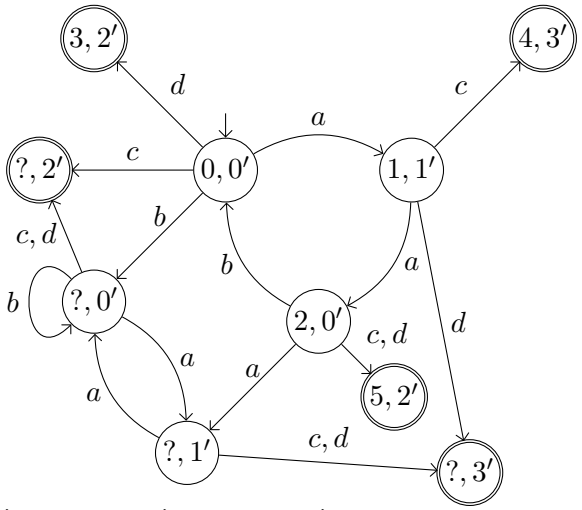
$\mathrm{Acc}(0) = \{\{a\}, \{a, d\}\}$
$\mathrm{Acc}(1) = \{\{a\}, \{a, c\}\}$
$\mathrm{Acc}(2) = \{\{b\}, \{b, c\}, \{b, d\}, \{b, c, d\}\}$
$\mathrm{Acc}(3) = \mathrm{Acc}(4) = \mathrm{Acc}(5) = \{\emptyset\}$

(a) S_1



$\mathrm{Acc}(0') = \{\{a, b, c, d\}\}$
$\mathrm{Acc}(1') = \{\{a, c\}, \{a, d\}, \{a, c, d\}\}$
$\mathrm{Acc}(2') = \mathrm{Acc}(3') = \{\emptyset\}$

(b) S_2



$\mathrm{Acc}(?, 0') = \mathrm{Acc}(0, 0') = \mathrm{Acc}(2, 0') = \{\{a, b, c, d\}\}$
$\mathrm{Acc}(?, 1') = \mathrm{Acc}(1, 1') = \{\{a, c\}, \{a, d\}, \{a, c, d\}\}$
$\mathrm{Acc}(?, 2') = \mathrm{Acc}(3, 2') = \mathrm{Acc}(5, 2') = \mathrm{Acc}(4, 3') = \mathrm{Acc}(?, 3') = \{\emptyset\}$

(c) Unfolding of S_2

Figure 4.9: Example of unfolding

- $\delta_u((q_1^?, q_2), a)$ *is defined if and only if* $\delta_2(q_2, a)$ *is defined and then:*

$$\begin{array}{ll} \delta_u((q_1, q_2), a) = & \begin{cases} (\delta_1(q_1, a), \delta_2(q_2, a)) & \text{if } \delta_1(q_1, a) \text{ is defined} \\ (q^?, \delta_2(q_2, a)) & \text{otherwise} \end{cases} \\ \delta_u((q^?, q_2), a) = & (q^?, \delta_2(q_2, a)) \end{array}$$

- $\mathrm{Acc}_u((q_1^?, q_2)) = \mathrm{Acc}_2(q_2)$.

Consider the marked acceptance specifications S_1 and S_2 in Figures 4.9(a) and 4.9(b). Some states of S_2 have several partners: $Q_1(0') = \{0, 2\}$ and $Q_1(2') = \{3, 5\}$. The unfolding of S_2 is shown in Figure 4.9(c). All its states have at most one partner: $Q_2((0, 0')) = \{0\}$, $Q_2((3, 2')) = \{3\}$, $Q_2((?, 1')) = \emptyset$, ...

Complexity 11. For each state of the unfolding, computing its transition function has a complexity of $O(|\,\mathrm{ready}_2\,|)$ (since the transition function is defined only for the actions on which δ_2 is defined) and the acceptance set is obtained immediately. Thus the complexity of the unfolding operation is $O(|Q_1| \times |Q_2| \times |\,\mathrm{ready}_2\,|)$.

We prove that unfolding a specification preserves its set of models:

Lemma 8. *Given two marked acceptance specifications* S_1 *and* S_2*, and* S_u *the unfolding of* S_2 *in relation to* S_1*,* $S_u \equiv S_2$.

Proof. ($\Rightarrow$) Let M be a model of S_u. Let π_u be the simulation relation between the states of M and the states of S_u and let π_2 be the simulation relation such that $(r, q_2) \in \pi_2$ if and only if there exists a $q_1^?$ such that $(r, (q_1^?, q_2)) \in \pi_u$. $(r^0, q_2^0) \in \pi_2$ and for any $(r, q_2) \in \pi_2$:

- $\mathrm{ready}(r) \in \mathrm{Acc}_2(q_2)$ as $\mathrm{ready}(r) \in \mathrm{Acc}_u((q_1^?, q_2)) = \mathrm{Acc}_2(q_2)$;
- if $r \in G$, $q_2 \in F_2$ as $(q_1^?, q_2) \in (Q_1 \cup \{q^?\}) \times F_2$;
- for any $a \in \mathrm{ready}(r)$, $(\lambda(r, a), \delta_2(q_2, a)) \in \pi_2$ as $(\lambda(r, a), \delta_u((q_1^?, q_2), a)) \in \pi_u$.

Thus M is a model of S_2.

($\Leftarrow$) Let M be a model of S_2. Let π_2 be the simulation relation between the states of M and the states of S_2 and let π_u be the simulation relation such that $(r, (q_1^?, q_2)) \in \pi_u$ if and only if $(r, q_2) \in \pi_2$ and $(q_1^?, q_2)$ is reachable in S_u. $(r^0, (q_1^0, q_2^0)) \in \pi_u$ and for any $(r, (q_1^?, q_2)) \in \pi_2$:

- $\mathrm{ready}(r) \in \mathrm{Acc}_u((q_1^?, q_2))$ as $\mathrm{ready}(r) \in \mathrm{Acc}_2(q_2) = \mathrm{Acc}_u((q_1^?, q_2))$;
- if $r \in G$, $(q_1^?, q_2) \in (Q_1 \cup \{q^?\}) \times F_2$ as $q_2 \in F_2$;
- for any $a \in \mathrm{ready}(r)$, $(\lambda(r, a), \delta_u((q_1^?, q_2), a)) \in \pi_u$ as $(\lambda(r, a), \delta_2(q_2, a)) \in \pi_2$.

Thus M is a model of S_u. □

Lemma 9. *Given two marked acceptance specifications* S_1 *and* S_2*, and* S_u *the unfolding of* S_2 *in relation to* S_1*, for any* $(q_1, (q_1^?, q_2))$ *reachable in* $\mathrm{Un}(S_1) \times \mathrm{Un}(S_u)$*,* $q_1 = q_1^?$.

Proof. If a state is reachable in $\mathrm{Un}(S_1) \times \mathrm{Un}(S_u)$, there is a path from the initial state to it. By induction on this path:

- if it is empty, we are in the initial state $(q_1^0, (q_1^0, q_2^0))$;

- otherwise, we are in a state $(q_1, (q_1, q_2))$ and there is a transition by an action a to another state $(\delta_1(q_1, a), \delta_u((q_1, q_2), a))$. As $\delta_1(q_1, a)$ is defined, $\delta_u((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$, so the destination state is $(\delta_1(q_1, a), (\delta_1(q_1, a), \delta_2(q_2, a)))$. □

Lemma 10. *Given two marked acceptance specifications S_1 and S_2, and S_u the unfolding of S_2 in relation to S_1, for any state q_u of S_u, $|Q_1(q_u)| \leq 1$.*

Proof. Suppose that $|Q_1(q_u)| > 1$. Then, there exist at least two different states q_1 and q'_1 such that (q_1, q_u) and (q'_1, q_u) are reachable in $\mathrm{Un}(S_1) \times \mathrm{Un}(S_2)$. By Definition 43, there exists some $q_1^? \in Q_1 \cup \{q_?\}$ and $q_2 \in Q_2$ such that $q_u = (q_1^?, q_2)$. By Lemma 9, $q_1 = q_1^?$ and $q'_1 = q_1^?$, so $q_1 = q'_1$. But we know by hypothesis that they are different, so $|Q_1(q_u)| \leq 1$. □

We say that two marked acceptance specifications S_1 and S_2 have *single partners* if for all $q_1 \in Q_1$, we have $|Q_2(q_1)| \leq 1$ and for all $q_2 \in Q_2$, we also have $|Q_1(q_2)| \leq 1$.

Finally, we prove that we may unfold two marked acceptance specifications such that they have single partners, while preserving their sets of models.

Theorem 48. *Given two marked acceptance specifications S_1 and S_2, there exist some marked acceptance specifications S'_1 and S'_2, called* unfoldings *of S_1 and S_2, with single partners and which are respectively equivalent to S_1 and S_2.*

Proof. Let S'_1 be the unfolding of S_1 in relation to S_2 and S'_2 the unfolding of S_2 in relation to S'_1.

By Lemma 8, we know that S'_1 has the same models as S_1 and S'_2 as S_2.

By Lemma 10, we know that for any q'_1 in S'_1, $|Q_2(q'_1)| \leq 1$ and that for any q'_2 in S'_2, $|Q'_1(q'_2)| \leq 1$. It remains to prove that $|Q'_2(q'_1)| \leq 1$.

Let q'_1 be a state of S'_1. If $|Q_2(q'_1)| = 0$, then $|Q'_2(q'_1)| = 0$ as S_2 and S'_2 have the same models. Otherwise, there exists a q_2 such that $Q_2(q'_1) = \{q_2\}$. There exist then n states (with $n > 0$) q'_{2_i} of the form $(q_{1_i}^?, q_2)$. But each q'_{2_i} is in relation with at most one state q'_{1_i} of S'_1, as $|Q'_1(q'_{2_i})| \leq 1$, and all these q'_{1_i} are different (as the q'_{2_i} are different). So there is at most one q'_{2_i} in relation with q'_1 and thus $|Q'_2(q'_1)| \leq 1$. □

Cycles.

In order to detect livelocks, we need to study the cycles that may be present in the models of a marked acceptance specification. Intuitively, a cycle is characterized by its states and the transitions between them. For example, let us look back at the marked acceptance specification S in Figure 4.6. It has three possible cycles: $\{1 \mapsto \{b\}, 2 \mapsto \{c\}\}$, $\{1 \mapsto \{b\}, 2 \mapsto \{d\}\}$ and $\{1 \mapsto \{b\}, 2 \mapsto \{c, d\}\}$. The model of S in Figure 4.7 implements this last cycle.

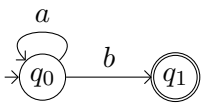
We first formally define what is a path in an automaton and then use it to define cycles.

Definition 44 (Path)**.** *Given a marked acceptance specification S, a sequence of n actions $a_1, \ldots, a_n$ is a* path *from a state q_i to a state q_f if and only if $n = 0$ and $q_i = q_f$ or there exist $n-1$ states $q_2, \ldots, q_n$, called intermediate states, such that $\delta(q_i, a_1) = q_2$, $\forall k \in \{2, \ldots, n-1\}, \delta(q_k, a_k) = q_{k+1}$, and $\delta(q_n, a_n) = q_f$.*

A path is said to be without loops if it is empty or if all the states q_i, q_f and q_k are different.

A path without loops may also be represented by a partial function $p : Q \to \Sigma$ such that $p(q_i) = a_1$ and $\forall k \in \{2, \ldots, n-1\}, p(q_k) = a_k$.

Theorem 49. *Given a marked acceptance specification S, if there is a path from a state q_i to a state q_f, then there is a path without loops from q_i to q_f.*



$\text{Acc}(q_0) = \{\{a\}, \{b\}\}$
$\text{Acc}(q_1) = \{\emptyset\}$

Figure 4.10: A marked acceptance specification with no implementable cycles

Proof. If there is a loop in the path p, there are two states q_j and q_k such that $j < k$ and $q_j = q_k$. Then, let p' be the same path without the states between q_j (included) and q_k (excluded). This path p' goes from q_i to q_f but has strictly fewer states than p. Repeat until there is no more loop in the path. □

Definition 45 (Cycle)**.** *Given a marked acceptance specification S, the partial map $\mathcal{C} : Q \to 2^\Sigma$ represents a* cycle *in S if and only if* $\text{dom}(\mathcal{C}) \neq \emptyset$ *and for any* $q \in \text{dom}(\mathcal{C})$*:*

- $\mathcal{C}(q) \neq \emptyset$*;*
- *there exists an* $X \in \text{Acc}(q)$ *such that* $\mathcal{C}(q) \subseteq X$*;*
- *there exists a non-empty path p from q to any* $q' \in \text{dom}(\mathcal{C})$ *such that* $\text{dom}(p) \subseteq \text{dom}(\mathcal{C})$ *and* $\forall q_p \in \text{dom}(p), p(q_p) \in \mathcal{C}(q_p)$*;*
- $\forall a \in \mathcal{C}(q), \delta(q, a) \in \text{dom}(\mathcal{C})$.

Definition 46 (Cycle implementation)**.** *A model M of a marked acceptance specification S* implements *a cycle $\mathcal{C}$ of S if and only if there exists a set $\mathcal{R}$ of states of M such that:*

- *each* $q \in \text{dom}(\mathcal{C})$ *is implemented by at least one state of* $\mathcal{R}$*;*
- *for each* $r \in \mathcal{R}$ *and for each q such that* $(r, q) \in \pi$*:*
  - $q \in \text{dom}(\mathcal{C})$*;*
  - $\mathcal{C}(q) \subseteq \text{ready}(r)$*;*
  - $\forall a \in \mathcal{C}(q), \lambda(r, a) \in \mathcal{R}$*;*
  - $\forall a \in \text{ready}(r) \backslash \mathcal{C}(q), \lambda(r, a) \notin \mathcal{R}$.

A cycle is said to be *implementable* if there exists a model M of S implementing the cycle.

Algorithm 3 defines the operation Cycle$_\models$-rec which recursively computes the set of cycles in a specification passing by a given state. However, some of these cycles may not be implementable. Consider for example the marked acceptance specification depicted in Figure 4.10, there is a cycle $\mathcal{C} = \{0 \mapsto \{a\}\}$ but it is not implementable; indeed, any model of the specification must eventually realize the transition by b and then it can not simultaneously realize a to make a cycle. Intuitively, a cycle is only implementable if including it still allows to reach a marked state. This means that either the cycle contains a marked state or it is possible to realize a transition that will leave the cycle, in addition to the transitions needed to implement it.

Definition 47 (Implementable cycle)**.** *Given a state q of a marked acceptance specification S, the set of implementable cycles of S passing by q,* $\text{Cycle}_\models(S, q)$*, is* $\{\mathcal{C} \in \text{Cycle}_\models\text{-}rec(S, \{q_p \mapsto \{p(q_p)\} \mid q_p \in \text{dom}(p)\}) \mid p$ *non-empty path from q to* $q \wedge (\text{dom}(\mathcal{C}) \cap F \neq \emptyset \vee \exists q_\mathcal{C} \in \text{dom}(\mathcal{C}), \exists X \in \text{Acc}(q_\mathcal{C}), \mathcal{C}(q_\mathcal{C}) \subset X)\}$.

The set of implementable cycles of S is $\text{Cycle}_\models(S) = \bigcup_{q \in Q} \text{Cycle}_\models(S, q)$.

Algorithm 3 Cycle$_{\models}$-rec (S: MAS, $\mathcal{C}$: Cycle): Set Cycle

```
1: res ← {cycle}
2: for all q ∈ dom(C) do
3:   for all A ∈ Acc(q) do
4:     if C(q) ⊂ A then
5:       for all a ∈ A\C(q) do
6:         for all path p from δ(q, a) to a q' ∈ dom(C), such that dom(p) ∩ dom(C) = ∅ do
7:           C' ← C ∪ {q ↦ C(q) ∪ {a}} ∪ {q_p ↦ {p(q_p)} | q_p ∈ dom(p)}
8:           res ← res ∪ Cycle⊨-rec(S, C')
9:         end for
10:      end for
11:    end if
12:  end for
13: end for
14: return res
```

Complexity 12. We first have to determine the complexity of Cycle$_{\models}$-rec. This is rather difficult as it is a recursive function with several nested loops. We will give a rough estimate of the worst-case complexity:

- The first loop (line 2) has $O(|Q|)$ iterations since a cycle may contain an arbitrary number of states.
- The second loop (line 3) has $O(|\operatorname{Acc}|)$ iterations.
- The third loop (line 5) iterates on a subset of an element of the acceptance set, which size is thus bounded by the size of the ready set of the state, which gives a complexity of $O(|\operatorname{ready}|)$.
- The fourth loop (line 6) is the most complicated one. In the worst case, we estimate that there would be $O(|\operatorname{ready}|^{|Q|})$ paths from a given state. This seems to be a gross overestimation, in particular considering that there are additional constraints on the paths selected, but we have no finer result.
- Last, Cycle$_{\models}$-rec is recursively called (line 8). In the worst case, the algorithm adds exactly one state at each recursive call and finishes when the cycle contains all the states of the specification, meaning that there would be $|Q|$ recursive calls.

Thus, we estimate that the worst-case complexity of the algorithm is:

$$O\left(\left(|Q| \times |\operatorname{Acc}| \times |\operatorname{ready}|^{|Q|+1}\right)^{|Q|}\right)$$

Then, Cycle$_{\models}$ calls Cycle$_{\models}$-rec for all the paths from each state to itself, which gives a complexity for Cycle$_{\models}$:

$$O\left(|Q| \times |\operatorname{ready}|^{|Q|} \times \left(|Q| \times |\operatorname{Acc}| \times |\operatorname{ready}|^{|Q|+1}\right)^{|Q|}\right)$$

The previous definition based on Algorithm 3 allows one to characterize when a cycle can be implemented:

Theorem 50. *Given a marked acceptance specification S, a model M of S implements a cycle $\mathcal{C}$ if and only if $\mathcal{C} \in \text{Cycle}_{\models}(S)$.*

Proof. ($\Rightarrow$) Let $\mathcal{C}$ be a cycle in S and M a model of S implementing $\mathcal{C}$, with $\mathcal{R}$ the set of states of M implementing the states of $\mathcal{C}$. Let r be an element of $\mathcal{R}$ and q a state it implements. We can make a non-empty path p from q to q by taking an arbitrary action in $\mathcal{C}(q)$, going to the next state by this action and repeating until we get back to q. Then, we have to prove that $\mathcal{C}$ is in the set of cycles returned by $\text{Cycle}_{\models}$-rec. For any $q' \in \text{dom}(p)$ and for any $a \in \mathcal{C}(q')$, a step of the loop at line 5 will have this a and one of the paths of the loop at line 6 will match a path of $\mathcal{C}$. Any state $q' \in \text{dom}(\mathcal{C}) \setminus \text{dom}(p)$ will be added similarly by a path generated by the loop at line 6 from a successor of a state in p to another one. Finally, a cycle returned by $\text{Cycle}_{\models}$-rec only belongs to $\text{Cycle}_{\models}$ if $\text{dom}(\mathcal{C}) \cap F \neq \emptyset \vee \exists q_{\mathcal{C}} \in \text{dom}(\mathcal{C}), \exists X \in \text{Acc}(q_{\mathcal{C}}), \mathcal{C}(q_{\mathcal{C}}) \subset X$. M is terminating, thus there is a marked state reachable from any state of $\mathcal{C}$. Either this marked state is in the cycle, thus $\text{dom}(\mathcal{C}) \cap F \neq \emptyset$, or there is a transition leaving the cycle in order to reach the marked state, giving us a $q_{\mathcal{C}}$ and an $X \in \text{Acc}(q_{\mathcal{C}})$. So $\mathcal{C} \in \text{Cycle}_{\models}(S, q)$ and then $\mathcal{C} \in \text{Cycle}_{\models}(S)$.

($\Leftarrow$) Let $\mathcal{C}$ be a cycle in $\text{Cycle}_{\models}(S)$. There exists a state q such that $\mathcal{C} \in \text{Cycle}_{\models}(S, q)$. Let us build an automaton M implementing $\mathcal{C}$ and prove that it is a model of S. We can make an automaton we an arbitrary path from its initial state to a state r realizing q. From this, we add a state r_i for each $q_i \in \text{dom}(\mathcal{C})$ and select an arbitrary $X_i \in \text{Acc}(q_i)$ such that $\mathcal{C}(q_i) \subseteq X_i$ as ready(r_i). Then, we add the required states and transitions outside the implementation of the cycle to ensure that the automaton is well-formed and satisfies the constraints of S. We know that this automaton is terminating since there is either a marked state in $\mathcal{C}$ or a state in it from which we can reach a marked state. □

Livelock-freeness.

Given two marked acceptance specifications with single partners, we can now examine their cycles in order to check if there is a possible livelock in the product of some of their models. To do so, we distinguish two kinds of transitions: those, denoted $\mathcal{A}$, which are always realized when the cycle is implemented and those, denoted $\mathcal{O}$, which may (or may not) be realized when the cycle is implemented. These two values are computed, for a given cycle, by Algorithm 4.

Algorithm 4 critical (S: MAS, $\mathcal{C}$: Cycle): Map Q (Set (Set Σ)) $\times$ Map Q (Set (Set Σ))

```
1: A: Map Q (Set (Set Σ))) = ∅, O: Map Q (Set (Set Σ)) = ∅
2: for all (q, A) ∈ C do
3:    if A ∉ Acc(q) then
4:       A[q] ← {X\A | X ∈ Acc(q) ∧ A ⊂ X}
5:    else if ∃X ∈ Acc(q), A ⊂ X then
6:       O[q] ← {X\A | X ∈ Acc(q) ∧ A ⊂ X}
7:    end if
8: end for
9: return A, O
```

Consider the marked acceptance specification S_1 of Figure 4.9(a). It has a single cycle $\mathcal{C}_1 = \{0 \mapsto \{a\}, 1 \mapsto \{a\}, 2 \mapsto \{b\}\}$. For the state 0, $\mathcal{C}_1(0) = \{a\}$ belongs to Acc(0), which means that an implementation of S_1 may only realize the transition a. An implementation may also realize another transition, d, along with a (as $\{a, d\} \in \text{Acc}(0)$). Thus, when implementing $\mathcal{C}_1$,

there may be a transition, labeled d, leaving the cycle, so $\mathcal{O}(0) = \{d\}$. Similarly, we compute that $\mathcal{O}(1) = \{c\}$ and $\mathcal{O}(2) = \{\{c\}, \{d\}, \{c, d\}\}$.

Consider now the unfolding of the specification S_2 depicted in Figure 4.9(c) and the cycle $\mathcal{C}_2 = \{(0, 0') \mapsto \{a\}, (1, 1') \mapsto \{a\}, (2, 0') \mapsto \{b\}\}$. There is a single element in $\mathrm{Acc}(0, 0')$: $\{a, b, c, d\}$. So any model of the specification implementing the cycle $\mathcal{C}_2$ will have to realize the transitions b, c and d in addition to a. Thus, $\mathcal{A}(0, 0') = \{\{b, c, d\}\}$. Similarly, for state $(1, 1')$, it is impossible to only realize the transition a. According to $\mathrm{Acc}(1, 1') = \{\{a, c\}, \{a, d\}, \{a, c, d\}\}$, there may be either c, d or both in addition to a, which means that $\mathcal{A}(1, 1') = \{\{c\}, \{d\}, \{c, d\}\}$. We compute similarly that $\mathcal{A}(2, 0') = \{\{a, c, d\}\}$.

Definition 48. *Given two marked acceptance specifications S_1 and S_2 with single partners and a cycle $\mathcal{C}_1$ in S_1 such that all its states have a partner, $\mathcal{C}_1$ is livelock-free in relation to S_2, denoted* $\mathrm{LiveFree}(\mathcal{C}_1, S_2)$*, if and only if, when the cycle $\mathcal{C}_2 = \{Q_2(q) \mapsto \mathcal{C}_1(q) \mid q \in \mathrm{dom}(\mathcal{C}_1)\}$ is in* $\mathrm{Cycle}_{\models}(S_2)$*:*

1. $\mathcal{A}_{\mathcal{C}_1} \neq \emptyset$, $\mathcal{A}_{\mathcal{C}_2} \neq \emptyset$ *and there exists* $q'_1 \in \mathrm{dom}(\mathcal{A}_{\mathcal{C}_1})$ *such that* $Q_2(q'_1) \in \mathrm{dom}(\mathcal{A}_{\mathcal{C}_2})$ *and* $\mathrm{Compat}(\mathcal{A}_{\mathcal{C}_1}(q'_1), \mathcal{A}_{\mathcal{C}_2}(Q_2(q'_1)))$ *or*

2. $\mathcal{A}_{\mathcal{C}_1} \neq \emptyset$, $\mathcal{A}_{\mathcal{C}_2} = \emptyset$, $\mathrm{dom}(\mathcal{C}_2) \cap F_2 = \emptyset$ *and* $\forall q'_2 \in \mathrm{dom}(\mathcal{O}_{\mathcal{C}_2})$, $Q_1(q'_2) \in \mathrm{dom}(\mathcal{A}_{\mathcal{C}_1})$ *and* $\mathrm{Compat}(\mathcal{A}_{\mathcal{C}_1}(Q_1(q'_2)), \mathcal{O}_{\mathcal{C}_2}(q'_2))$ *or*

3. $\mathcal{A}_{\mathcal{C}_1} = \emptyset$, $\mathcal{A}_{\mathcal{C}_2} \neq \emptyset$, $\mathrm{dom}(\mathcal{C}_1) \cap F_1 = \emptyset$ *and* $\forall q'_1 \in \mathrm{dom}(\mathcal{O}_{\mathcal{C}_1})$, $Q_2(q'_1) \in \mathrm{dom}(\mathcal{A}_{\mathcal{C}_2})$ *and* $\mathrm{Compat}(\mathcal{O}_{\mathcal{C}_1}(q'_1), \mathcal{A}_{\mathcal{C}_2}(Q_2(q'_1)))$.

Definition 49 (Livelock-free specifications)**.** *Two marked acceptance specifications S_1 and S_2 with single partners are* livelock-free *if all the implementable cycles of S_1 are livelock-free in relation to S_2.*

Note that this definition only tests the implementable cycles of S_1. It is not necessary to do the symmetrical test (checking that the implementable cycles of S_2 verify LiveFree) because we only compare the cycle of S_1 with the same cycle in S_2 and the three tests of Definition 48 are symmetric.

Complexity 13. We first consider the algorithm computing the $\mathcal{A}$, $\mathcal{O}$ sets. Given a cycle $\mathcal{C}$, it iterates on all the states belonging to the cycle and then on the corresponding acceptance sets, so its complexity is $O(|\mathcal{C}| \times |\,\mathrm{Acc}\,|)$.

Then, given a cycle $\mathcal{C}_1$, we check $\mathrm{LiveFree}(\mathcal{C}_1, S_2)$. We compute the $\mathcal{A}$, $\mathcal{O}$ sets of $\mathcal{C}_1$ and $\mathcal{C}_2$ (which have the same size) and test three cases. They have the same complexity as they follow the same scheme consisting of testing Compat on the elements of the $\mathcal{A}$, $\mathcal{O}$ sets for each state in the cycle; thus the complexity is $O(|\mathcal{C}_1| \times |\,\mathrm{Acc}_1\,| \times |\,\mathrm{Acc}_2\,|)$.

Finally, we have to compute all the implementable cycles of the specifications, which complexity was given earlier, and determine the number of cycles returned, which will tell us how many times LiveFree is called. This is again a tough question, but we can give an upper bound for the worst case: $O(2^{|Q| \times |\,\mathrm{ready}\,|})$. By combining the complexity of these operations, we obtain the following complexity for checking livelock-freeness:

$$O\Big(|Q_1| \times |\,\mathrm{ready}_1\,|^{|Q_1|} \times \Big(|Q_1| \times |\,\mathrm{Acc}_1\,| \times |\,\mathrm{ready}_1\,|^{|Q_1|+1}\Big)^{|Q_1|} + |Q_2| \times |\,\mathrm{ready}_2\,|^{|Q_2|} \times \Big(|Q_2| \times |\,\mathrm{Acc}_2\,| \times |\,\mathrm{ready}_2\,|^{|Q_2|+1}\Big)^{|Q_2|} + 2^{|Q_1|\times|\,\mathrm{ready}_1\,|+|Q_2|\times|\,\mathrm{ready}_2\,|} \times |Q_1| \times |\,\mathrm{Acc}_1\,| \times |\,\mathrm{Acc}_2\,|\Big)$$

The previous definition offers a necessary and sufficient condition to identify marked acceptance specifications which can have two respective models whose product has a livelock:

Theorem 51. *Two marked acceptance specifications S_1 and S_2 with single partners are livelock-free if and only if for any $M_1 \models S_1$ and $M_2 \models S_2$, $M_1 \times M_2$ is livelock-free.*

Proof. ($\Rightarrow$) Assume that there exists $M_1 \models S_1$, $M_2 \models S_2$ such that $M_1 \times M_2$ has a livelock, that is, there exists (r_1, r_2) such that $\mathrm{Loop}((r_1, r_2)) \neq \emptyset$, $\mathrm{Loop}((r_1, r_2)) \cap G = \emptyset$ and there is no transition (r', a, r'') such that $r' \in \mathrm{Loop}((r_1, r_2))$ and $r'' \notin \mathrm{Loop}((r_1, r_2))$.

- If there exists a cycle $\mathcal{C}_1 \in \mathrm{Cycle}_{\models}(S_1)$ which is implemented in M_1 by the states of $\mathrm{Loop}(r_1)$ and $\mathcal{C}_2 = \{Q_2(q) \mapsto \mathcal{C}_1(q) \mid q \in \mathrm{dom}(\mathcal{C}_1)\}$ is implemented in M_2 by the states of $\mathrm{Loop}(r_2)$:
  - if there is no transition leaving $\mathrm{Loop}(r_1)$, then $\mathcal{A}_{\mathcal{C}_1} = \emptyset$ and $\mathrm{dom}(\mathcal{C}_1) \cap F_1 \neq \emptyset$, so the three tests of Definition 48 fail and S_1 and S_2 are not livelock-free; symmetrically S_1 and S_2 are not livelock-free if there is no transition leaving $\mathrm{Loop}(r_2)$;
  - if there are transitions leaving $\mathrm{Loop}(r_1)$ and $\mathrm{Loop}(r_2)$, they are not compatible, i.e. they have different actions or different source states. If in both models, some of these transitions are in $\mathcal{A}$ (they have to be present whenever the cycle is implemented), the test 1 of Definition 48 will detect that they are not compatible. If there are some transitions in $\mathcal{A}_{\mathcal{C}_1}$ but none in $\mathcal{A}_{\mathcal{C}_2}$, test 2 will detect that M_2 may implement a transition that will not be covered by the transitions in $\mathcal{A}_{\mathcal{C}_1}$. Test 3 handles the symmetrical case. Finally, if there are transitions neither in $\mathcal{A}_{\mathcal{C}_1}$ nor $\mathcal{A}_{\mathcal{C}_2}$, it is always possible to generate a livelock and all three tests fail.
- Otherwise, multiple cycles are implemented simultaneously in the model by unfolding them or two slightly different cycles are implemented in M_1 and M_2, and then there will also be a livelock in the models which implement only one of the cycles, which brings us back to the first case.

($\Leftarrow$) Assume that S_1 and S_2 are not livelock-free. Then, there exists a cycle $\mathcal{C}_1$ such that $\neg\,\mathrm{LiveFree}(\mathcal{C}_1, S_2)$. Then, the three conditions of Definition 48 are all false.

- If $\mathcal{A}_{\mathcal{C}_1} \neq \emptyset$ and $\mathcal{A}_{\mathcal{C}_2} \neq \emptyset$, then for any $q_1' \in \mathrm{dom}(\mathcal{A}_{\mathcal{C}_1})$ in S_1, $\mathrm{Compat}(\mathcal{A}_{\mathcal{C}_1}(q_1'), \mathcal{A}_{\mathcal{C}_2}(Q_2(q_1')))$ is false. So there exists a model M_1 of S_1 implementing $\mathcal{C}_1$ and a model M_2 of S_2 implementing $\mathcal{C}_2$ such that there is no transition leaving the cycle in their product, hence there is a livelock in $M_1 \times M_2$.
- If $\mathcal{A}_{\mathcal{C}_1} \neq \emptyset$ and $\mathcal{A}_{\mathcal{C}_2} = \emptyset$, there exists $q_2' \in \mathrm{dom}(\mathcal{O}_{\mathcal{C}_2})$ such that $\mathrm{Compat}(\mathcal{A}_{\mathcal{C}_1}(Q_1(q_2')), \mathcal{O}_{\mathcal{C}_2}(q_2'))$ is false. So for any model M_1 of S_1 implementing $\mathcal{C}_1$, its product with a model M_2 of S_2 implementing $\mathcal{C}_2$ for which the only transition leaving the cycle is from an implementation of q_2' will have a livelock.

- If $\mathcal{A}_{\mathcal{C}_1} = \emptyset$ and $\mathcal{A}_{\mathcal{C}_2} \neq \emptyset$, we are in the case symmetric to the previous one.
- If $\mathcal{A}_{\mathcal{C}_1} = \emptyset$ and $\mathcal{A}_{\mathcal{C}_2} = \emptyset$, either one of the specifications has no transitions leaving the cycle ($\mathcal{O}_{\mathcal{C}_i} = \emptyset$ too), so there are some models such that their product has a livelock, or both $\mathcal{O}_{\mathcal{C}_1}$ and $\mathcal{O}_{\mathcal{C}_2}$ are not empty, and then there exists an $M_1 \models S_1$ implementing $\mathcal{C}_1$ such that the only transition(s) leaving the cycle is (are) from a state r_1 and an $M_2 \models S_2$ implementing $\mathcal{C}_2$ such that the only transition(s) leaving the cycle is (are) from a state r_2 which is never paired with r_1 in $M_1 \times M_2$, hence there is a livelock in $M_1 \times M_2$. □

4.3.3 Compatible reachability

By combining the tests for deadlock-free and livelock-free specifications, we can now define a criterion checking if two marked acceptance specifications have some models which product is not terminating.

Definition 50 (Compatible reachability)**.** *Two marked acceptance specifications S_1 and S_2 have a* compatible reachability*, denoted $S_1 \sim_{\mathcal{T}} S_2$, if and only if they are deadlock-free and their unfoldings are livelock-free.*

Complexity 14. We gave the complexity of checking deadlock-freeness, computing the unfolding of a specification, and checking livelock-freeness in the previous sections. It is clear that this last step is the most complex by far, so the complexity of checking compatible reachability is the same as checking livelock-freness.

Theorem 52. *Given two marked acceptance specifications S_1 and S_2, $S_1 \sim_{\mathcal{T}} S_2$ if and only if for any $M_1 \models S_1$ and $M_2 \models S_2$, $M_1 \times M_2$ is terminating.*

Proof. By definition, $S_1 \sim_{\mathcal{T}} S_2$ if and only if S_1 and S_2 are deadlock-free and livelock-free. By Theorems 47, 48, and 51, this is equivalent to: for any $M_1 \models S_1$ and $M_2 \models S_2$, $M_1 \times M_2$ is deadlock-free and livelock-free, that is, $M_1 \times M_2$ is terminating. □

4.3.4 Product definition

Given two marked acceptance specifications with compatible reachability, we can now compute their product which is a simple extension of the product of non-marked acceptance specifications.

Definition 51 (Product)**.** *Given two marked acceptance specifications S_1 and S_2 with compatible reachability, their* product *$S_1 \otimes S_2$ is the normal form of the marked acceptance specification $(Q_1 \times Q_2, (q_1^0, q_2^0), \delta, \mathrm{Acc}, F_1 \times F_2)$ with $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ when both $\delta_1(q_1, a)$ and $\delta_2(q_2, a)$ are defined and $\mathrm{Acc}(q_1, q_2) = \{A_1 \cap A_2 \mid A_1 \in \mathrm{Acc}_1(q_1) \wedge A_2 \in \mathrm{Acc}_2(q_2)\}$.*

Complexity 15. The product has at most $|Q_1| \times |Q_2|$ states. For each state, the complexity of computing the acceptance set is $O(|\,\mathrm{Acc}_1\,| \times |\,\mathrm{Acc}_2\,|)$ and the complexity of computing the transition function is $O(\min(|\delta_1|, |\delta_2|))$. So the complexity of building this specification is $O(|Q_1| \times |Q_2| \times (|\,\mathrm{Acc}_1\,| \times |\,\mathrm{Acc}_2\,| + \min(|\delta_1|, |\delta_2|)))$. Then, we have to apply ρ in order to guarantee that the result is in normal form; the complexity of this step is $O((|Q_1| \times |Q_2|)^3 \times \min(|\,\mathrm{ready}_1\,|, |\,\mathrm{ready}_2\,|) + (|Q_1| \times |Q_2|)^2 \times |\,\mathrm{Acc}_1\,| \times |\,\mathrm{Acc}_2\,|)$.

Theorem 53. *Given two marked acceptance specifications S_1 and S_2 with compatible reachability, for any $M_1 \models S_1$ and $M_2 \models S_2$, $M_1 \times M_2 \models S_1 \otimes S_2$.*

Proof. By Theorem 52, $M_1 \times M_2$ is terminating.

Let π_i be the simulation relation of $M_i \models S_i$ for $i \in \{1,2\}$ and π the simulation relation such that $((r_1,r_2),(q_1,q_2)) \in \pi$ if and only if (r_1,r_2) is reachable in $M_1 \times M_2$, $(r_1,q_1) \in \pi_1$ and $(r_2,q_2) \in \pi_2$. For any $((r_1,r_2),(q_1,q_2)) \in \pi$:

- $\text{ready}((r_1,r_2)) = \text{ready}(r_1) \cap \text{ready}(r_2) \in \text{Acc}(q_1,q_2)$ by definition of the acceptance set of the product;
- $(r_1,r_2) \in G_1 \times G_2$ implies that $(q_1,q_2) \in F_1 \times F_2$ as $r_1 \in G_1$ implies that $q_1 \in F_1$ and $r_2 \in G_2$ that $q_2 \in F_2$;
- for any a, r_1' and r_2' such that $\lambda((r_1,r_2),a) = (r_1',r_2')$, $((r_1',r_2'),\delta((q_1,q_2),a)) \in \pi$ is trivial as $\lambda((r_1,r_2),a) = (\lambda_1(r_1,a),\lambda_2(r_2,a)) = (r_1',r_2')$. □

Moreover, $S_1 \otimes S_2$ gives the most precise characterization of the behavior of the product of any models M_1 of S_1 and M_2 of S_2:

Theorem 54. *Given two marked acceptance specifications S_1 and S_2, if $S_1 \sim_{\mathcal{T}} S_2$ and if there exists a marked acceptance specification S such that for any $M_1 \models S_1$ and $M_2 \models S_2$ we have $M_1 \times M_2 \models S$, then $S_1 \otimes S_2 \leq S$.*

Proof. By contradiction assume that for any $M_1 \models S_1$ and $M_2 \models S_2$ we have $M_1 \times M_2 \models S$ but $S_1 \otimes S_2 \not\leq S$. Then, there exists an execution common to $\text{Un}(S_1 \otimes S_2)$ and $\text{Un}(S)$ leading to some state (q_1,q_2) in $S_1 \otimes S_2$ and q in S such that $\text{Acc}(q_1,q_2) \not\subseteq \text{Acc}(q)$ that is, there exist $A_1 \in \text{Acc}_1(q_1)$ and $A_2 \in \text{Acc}_2(q_2)$ such that $A_1 \cap A_2 \notin \text{Acc}(q)$. Consider now M_i such that $(r_i,q_i) \in \pi_i$ and $\text{ready}(r_i) = A_i$, for $i = 1,2$, the product $M_1 \times M_2$ cannot be a model of S as $\text{ready}(r_1,r_2) = A_1 \cap A_2 \notin \text{Acc}(q)$ which contradicts the assumption made at the beginning of the proof. □

One important principle in modular and concurrent design of systems is the fact that a property checked on a primary version of some system artifacts remains true on any refined version of them. This is what guarantees that the system parts corresponding to compatible specifications can be designed concurrently. This is respected for compatible reachability and product:

Proposition 7. *For all marked acceptance specifications S_1, S_1' and S_2, if $S_1 \sim_{\mathcal{T}} S_2$ and $S_1' \leq S_1$ then $S_1' \sim_{\mathcal{T}} S_2$ and $S_1' \otimes S_2 \leq S_1 \otimes S_2$.*

Proof. Let M_1 and M_2 be models of S_1' and S_2. As $S_1' \leq S_1$, by Theorem 44, M_1 is also a model of S_1. Moreover, the product $M_1 \times M_2$ is terminating as $S_1 \sim_{\mathcal{T}} S_2$, by Theorem 52. As a result, by Theorem 52, $S_1' \sim_{\mathcal{T}} S_2$.

Let π_1 be the simulation relation of $S_1' \leq S_1$ and π the simulation relation such that $((q_1',q_2),(q_1,q_2)) \in \pi$ if and only if (q_1',q_2) is reachable in $S_1' \otimes S_2$ and $(q_1',q_1) \in \pi$. For any $((q_1',q_2),(q_1,q_2)) \in \pi$:

- Let A be an element of $\text{Acc}((q_1',q_2))$. By definition of the acceptance set of the product, there exist $A_1' \in \text{Acc}_1'(q_1')$ and $A_2 \in \text{Acc}_2(q_2)$ such that $A = A_1' \cap A_2$. As $S_1' \leq S_1$, $A_1' \in \text{Acc}_1(q_1)$ too, so $A = A_1' \cap A_2 \in \text{Acc}((q_1,q_2))$, hence $\text{Acc}((q_1',q_2)) \subseteq \text{Acc}((q_1,q_2))$.
- $(q_1',q_2) \in F_1' \times F_2$ implies $(q_1,q_2) \in F_1 \times F_2$ as $q_1' \in F_1'$ implies $q_1 \in F_1$ by definition of the refinement.
- For any a and q' such that $\delta((q_1',q_2),a) = q'$, $(q',\delta((q_1,q_2),a)) \in \pi$ is trivial as $\delta((q_1',q_2),a) = (\delta_1'(q_1',a),\delta_2(q_2,a))$ and $S_1' \leq S_1$. □

4.4 Quotient

In this section, we study the extension of the quotient of acceptance specifications to marked acceptance specifications, in order to enable the incremental design of reachability properties.

4.4.1 Pre-quotient

We first define an operation called *pre-quotient*. Given two marked acceptance specifications S_1 and S_2, it returns a marked acceptance specification $S_1 /\!\!/ S_2$ such that the product of any of its models with any model of S_2 will be an automaton which satisfies S_1 but does not guarantee the termination condition. Another operation, defined in the next two sections, will then be used to define a quotient guaranteeing termination in Section 4.4.4.

Definition 52 (Pre-quotient)**.** *Given two marked acceptance specifications S_1 and S_2, their* pre-quotient *$S_1 /\!\!/ S_2$ is the marked acceptance specification $(Q_1 \times Q_2, (q_1^0, q_2^0), \delta, \mathrm{Acc}, F)$ with:*

- $\mathrm{Acc}((q_1, q_2)) = \{X \mid (\forall X_2 \in \mathrm{Acc}_2(q_2), X \cap X_2 \in \mathrm{Acc}_1(q_1)) \wedge X \subseteq (\bigcup \mathrm{Acc}_1(q_1)) \cap (\bigcup \mathrm{Acc}_2(q_2))\}$*;*
- *$\delta((q_1, q_2), a)$ is defined if and only if there exists an $X \in \mathrm{Acc}((q_1, q_2))$ such that $a \in X$ and then $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$;*
- *$(q_1, q_2) \in F$ if and only if $q_1 \in F_1$ or $q_2 \notin F_2$.*

Complexity 16. The pre-quotient has at most $|Q_1| \times |Q_2|$ states. In order to compute the acceptance set of a pair of states, there are three steps:

1. enumerate all the $X \in 2^\Sigma$, which complexity is $O(2^{|\Sigma|})$;
2. test if $\forall X_2 \in \mathrm{Acc}_2(q_2), X \cap X_2 \in \mathrm{Acc}_1(q_1)$: $O(|\,\mathrm{Acc}_2\,| \times |\,\mathrm{Acc}_1\,|)$;
3. test if $X \subseteq (\bigcup \mathrm{Acc}_1(q_1)) \cap (\bigcup \mathrm{Acc}_2(q_2))$: assuming that S_1 and S_2 are in normal form, this can also be written as $X \subseteq \mathrm{ready}_1(q_1) \cap \mathrm{ready}_2(q_2)$, which is considered to be $O(1)$.

Then, in order to build $\delta((q_1, q_2))$, we can compute $\bigcup \mathrm{Acc}((q_1, q_2))$ once, which is $O(2^{|\Sigma|})$, and then iterate on the actions in the resulting set to build the pairs of destination states, i.e., $O(|\Sigma|)$.

This gives a final complexity of $O(|Q_1| \times |Q_2| \times 2^{|\Sigma|} \times (|\,\mathrm{Acc}_2\,| \times |\,\mathrm{Acc}_1\,| + |\Sigma|))$.

Theorem 55 (Soundness)**.** *Given two marked acceptance specifications S_1 and S_2 and an automaton $M \models S_1 /\!\!/ S_2$, for any $M_2 \models S_2$ such that $M \times M_2$ is terminating, $M \times M_2 \models S_1$.*

Proof. Let $\pi_{/\!/}$ and π_2 be the simulation relations of $M \models S_1 /\!\!/ S_2$ and $M_2 \models S_2$. Let $\pi \subseteq ((R \times R_2) \times Q_1)$ be the simulation relation such that $((r, r_2), q_1) \in \pi$ if there exists a q_2 such that $(r_2, q_2) \in \pi_2$ and $(r, (q_1, q_2)) \in \pi_{/\!/}$. For any $((r, r_2), q_1) \in \pi$:

- $\mathrm{ready}(r, r_2) \in \mathrm{Acc}_1(q_1)$: by definition of the product of automata, $\mathrm{ready}(r, r_2) = \mathrm{ready}(r) \cap \mathrm{ready}(r_2)$ and by definition of the acceptance set of the pre-quotient, this intersection is in the acceptance set of q_1.
- $(r, r_2) \in G_1 \times G_2$ implies $q_1 \in F_1$: by definition of the pre-quotient, if $r \in G_1$, then $q_1 \in F_1$.
- for any a, if $\lambda((r, r_2), a) = (r', r_2')$, then $\delta_1(q_1, a)$ is defined and $((r', r_2'), \delta_1(q_1, a)) \in \pi$: $\lambda(r, a) = r'$, so $\delta((q_1, q_2), a)$ is defined and equal to some (q_1', q_2') and, by definition of the pre-quotient, $\delta_1(q_1, a) = q_1'$ and $\delta_2(q_2, a) = q_2'$; $(r, (q_1, q_2)) \in \pi_{/\!/}$ so $(r', (q_1', q_2')) \in \pi_{/\!/}$, $(r_2, q_2) \in \pi_2$, so $(r_2', q_2') \in \pi_2$, hence $((r', r_2'), q_1') \in \pi$. □

In general, it is also expected that the specification returned by a quotient is complete, that is, it should characterize *all* the possible automata which product with any model of S_2 is a model of S_1. However, this can lead to a very large specification as the quotient S_1/S_2 should then include all the transitions which are not fireable in S_2 (and thus removed in the product of the models). We propose to return a compact quotient specification without *unnecessary transitions* regarding S_2, i.e., without the transitions that will always be cut by the product with models of S_2. Then, completeness of a quotient S_1/S_2 amounts to guarantee that any automaton which product with any model of S_2 is a model of S_1 is a model of S_1/S_2 after the removal of these useless transitions.

Definition 53 (Unnecessary transition)**.** *Given a marked acceptance specification S and an automaton M, M has no* unnecessary transitions *regarding S, denoted $M \sim_{\mathcal{U}} S$, if and only if there exists a simulation relation $\pi \subseteq R \times Q$ such that $(r^0, q^0) \in \pi$ and for all $(r, q) \in \pi$:*

- $\text{ready}(r) \subseteq \bigcup \text{Acc}(q)$*;*
- *for every a and r' such that $\lambda(r, a) = r'$, $(r', \delta(q, a)) \in \pi$.*

Definition 54. *Given an automaton M and a marked acceptance specification S, $\rho_u(M, S)$ is the automaton $M' = (R \times Q, (r^0, q^0), \lambda', G \times Q)$ with:*

$$\lambda'((r, q), a) = \begin{cases} (\lambda(r, a), \delta(q, a)) \text{ if } a \in \bigcup \text{Acc}(q) \\ \text{undefined otherwise} \end{cases}$$

Complexity 17. The complexity of checking if an automaton has no unnecessary transitions regarding a marked acceptance specification is $O(|R| \times |Q| \times (|\,\text{Acc}\,| + |\,\text{ready}\,|))$.

Building $\rho_u(M, S)$ has a complexity of $O(|R| \times |Q| \times |\,\text{ready}\,|)$ (remember that for marked acceptance specifications in normal form, $\bigcup \text{Acc}(q)$ is equivalent to $\text{ready}(q)$ on the underlying automaton).

Theorem 56. *Given an automaton M and a marked acceptance specification S, $\rho_u(M, S) \sim_{\mathcal{U}} S$. Moreover, for all $M_S \models S$, the automata $M \times M_S$ and $\rho_u(M, S) \times M_S$ are bisimilar.*

Proof. $\rho_u(M, S) \sim_{\mathcal{U}} S$: let π be the simulation relation such that for any state (r, q) of $\rho_u(M, S)$, $((r, q), q) \in \pi$; by definition of ρ_u, $\text{ready}(r, q) \subseteq \bigcup \text{Acc}(q)$.

$M \times M_S$ and $\rho_u(M, S) \times M_S$ are bisimilar: let π be the simulation relation such that for any state (r, r_S) of $M \times M_S$ and $((r, q), r_S)$ of $\rho_u(M, S) \times M_S$, $((r, r_S), ((r, q), r_S)) \in \pi$. Then, $\text{ready}(((r, q), r_S)) = (\text{ready}(r) \cap \bigcup \text{Acc}(q)) \cap \text{ready}(r_S)$. As r_S implements q, $\text{ready}(r_S) \subseteq \bigcup \text{Acc}(q)$, so $\text{ready}(((r, q), r_S)) = \text{ready}(r) \cap \text{ready}(r_S) = \text{ready}((r, r_S))$. □

Theorem 57. *Given two marked acceptance specifications S_1 and S_2 and an automaton M such that $M \sim_{\mathcal{U}} S_2$ and for all $M_2 \models S_2$ we have $M \times M_2 \models S_1$, then $M \models S_1 /\!\!/ S_2$.*

Proof. Let π be a simulation relation such that $(r^0, (q_1^0, q_2^0)) \in \pi$ and for any $(r, (q_1, q_2)) \in \pi$, a and r' such that $\lambda(r, a) = r'$, $(r', \delta((q_1, q_2), a)) \in \pi$. This definition of π is only correct if for any $(r, (q_1, q_2)) \in \pi$ and a such that $\lambda(r, a)$ is defined, $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ is also defined. As $M \sim_{\mathcal{U}} S_2$, $a \in \bigcup \text{Acc}_2(q_2)$, so there exists an $X \in \text{Acc}_2(q_2)$ such that $a \in X$ and then $\delta_2(q_2, a)$ is defined (as S_2 is well-formed). As $\delta_2(q_2, a)$ is defined, there exists an automaton $M_2 \models S_2$ with a state r_2 implementing q_2 such that (r, r_2) is reachable in $M \times M_2$ and $\lambda_2(r_2, a)$ is defined. Then, $\lambda((r, r_2), a)$ is defined and, as $M \times M_2 \models S_1$, it implies that $\delta((q_1, q_2), a)$ is defined.

There are then three points to prove for any $(r, (q_1, q_2)) \in \pi$:

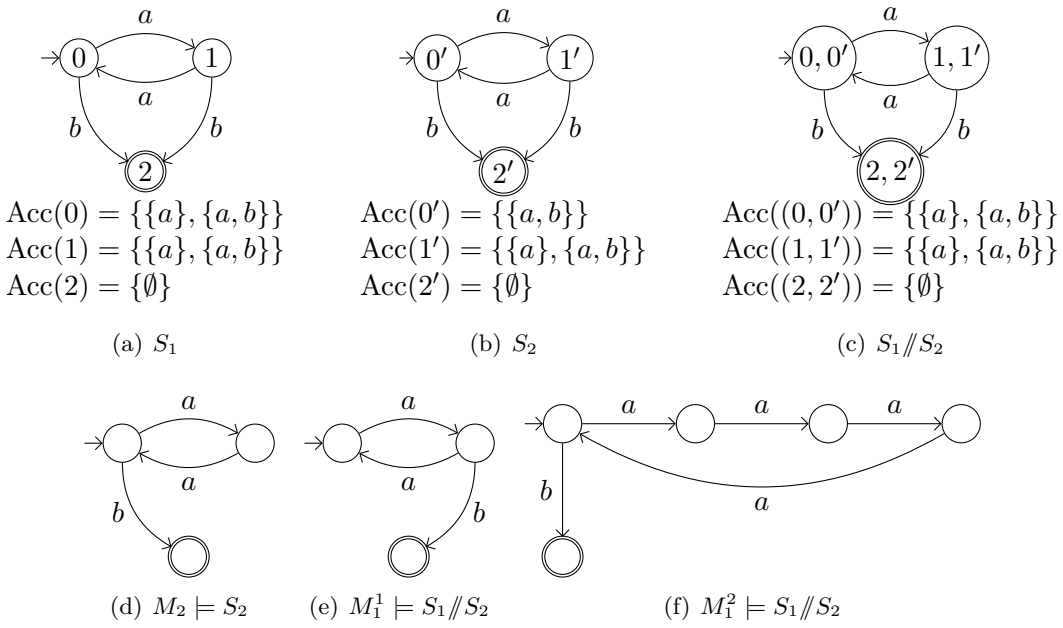
- $\text{ready}(r) \in \text{Acc}((q_1, q_2))$: by definition of the pre-quotient, $\text{ready}(r)$ must verify two properties:
  - $\forall X_2 \in \text{Acc}_2(q_2), \text{ready}(r) \cap X_2 \in \text{Acc}_1(q_1)$:
    Let X_2 be an element of $\text{Acc}_2(q_2)$. There exists an automaton M_2 with a state r_2 such that (r, r_2) is reachable in $M \times M_2$ and $\text{ready}(r_2) = X_2$. Then, as $M \times M_2 \models S_1$ by a simulation relation $\pi_\times$ and $((r, r_2), q_1) \in \pi_\times$, $\text{ready}(r) \cap \text{ready}(r_2) = \text{ready}(r) \cap X_2 \in \text{Acc}_1(q_1)$.
  - $\text{ready}(r) \subseteq \bigcup \text{Acc}_1(q_1) \cap \bigcup \text{Acc}_2(q_2)$:
    By definition of $\sim_{\mathcal{U}}$, $\text{ready}(r) \subseteq \bigcup \text{Acc}_2(q_2)$.
    Assume that $\text{ready}(r) \not\subseteq \bigcup \text{Acc}_1(q_1)$: there is an $a \in \text{ready}(r)$ such that $a \notin \bigcup \text{Acc}_1(q_1)$. As M has no unnecessary transition regarding S_2, there is a model M_2 of S_2 with a state r_2 such that (r, r_2) is reachable in $M \times M_2$ and $a \in \text{ready}(r_2)$. Then, the transition $((r, r_2), a)$ is defined in $M \times M_2$. As $M \times M_2 \models S_1$, the transition (q_1, a) has to be defined, which is in contradiction with the hypothesis that $a \notin \bigcup \text{Acc}_1(q_1)$. Thus, $\text{ready}(r) \subseteq \bigcup \text{Acc}_1(q_1)$.
- $r \in G$ implies $(q_1, q_2) \in F_{/\!/}$, that is $q_1 \in F_1$ or $q_2 \notin F_2$:
  This property is only false if $r \in G$, $q_1 \notin F_1$ and $q_2 \in F_2$. In this case, there exists an automaton $M_2 \models S_2$ with a state r_2 such that (r, r_2) is reachable in $M \times M_2$ and $r_2 \in G_2$. Then, $M \times M_2 \models S_1$ by a simulation relation $\pi_\times$, $((r, r_2), q_1) \in \pi_\times$ and (r, r_2) is marked. By definition of satisfaction, it implies that $q_1 \in F_1$, which is impossible as we already know that $q_1 \notin F_1$. So $r \in G$ implies $(q_1, q_2) \in F_{/\!/}$.
- for any a and r' such that $\lambda(r, a) = r'$, $(r', \delta((q_1, q_2), a)) \in \pi$ is trivial by definition of π. □

Corollary 4 (Completeness)**.** *Given two marked acceptance specifications S_1 and S_2 and an automaton M such that for all $M_2 \models S_2$, we have $M \times M_2 \models S_1$, then $\rho_u(M, S_2) \models S_1 /\!/ S_2$.*

Proof. By Theorem 56, we know that $\rho_u(M, S_2) \sim_{\mathcal{U}} S_2$ and for any $M_2 \models S_2$, $\rho_u(M, S_2) \times M_2$ is bisimilar to $M \times M_2$, which implies that $\rho_u(M, S_2) \times M_2 \models S_1$. Then, Theorem 57 implies that $\rho_u(M, S_2) \models S_1 /\!/ S_2$. □

This pre-quotient operation returns a specification $S_1 /\!/ S_2$ which does not always have a compatible reachability with the divisor S_2. For example, consider the specifications S_1 and S_2 of Figures 4.11(a) and 4.11(b); their pre-quotient is shown in Figure 4.11(c). If we take the models M_2 of S_2 (Figure 4.11(d)) and M_1^1 of $S_1 /\!/ S_2$ (Figure 4.11(e)), their product is not terminating as it has a livelock; hence, the result of the pre-quotient does not have a compatible reachability with the divisor S_2. One may think that the pre-quotient is erroneous and should not allow realizing only the transition a from the state $(0, 0')$ (i.e., that $\text{Acc}((0, 0'))$ should only be $\{\{a, b\}\}$). Indeed, it would forbid the incorrect model, but it would also disallow some valid models such as M_1^2 of Figure 4.11(f), which does not always realize the transition b, but does it once and can thus synchronize with any model of S_2, as they always realize this transition. The construction proposed in the next two sections will allow refining $S_1 /\!/ S_2$ in order to guarantee compatible reachability.

We now consider the following problem: given two marked acceptance specifications S_1 and S_2 that do not have a compatible reachability, can we refine S_1 such that the obtained specification S_1' has a compatible reachability with S_2? Solving this problem allows to automatically assist the



(a) S_1 (b) S_2 (c) $S_1 /\!\!/ S_2$

(d) $M_2 \models S_2$ (e) $M_1^1 \models S_1 /\!\!/ S_2$ (f) $M_1^2 \models S_1 /\!\!/ S_2$

Figure 4.11: Example of pre-quotient

system designer when a step of the design flow leads to incompatible specifications. We will then use the proposed solution in order to refine the result given by the pre-quotient operation and obtain a sound and complete quotient with reachability guarantees, as explained in Section 4.4.4.

4.4.2 Deadlock correction

First, given two non-deadlock-free marked acceptance specifications S_1 and S_2, we propose to refine S_1 such that the obtained marked acceptance specification S_1' is deadlock-free with S_2. For this, we iteratively eliminate all pairs of states (q_1, q_2) such that $\text{DeadFree}(q_1, q_2)$ is false, as described in Algorithm 5.

Algorithm 5 dead_correction (S_1: MAS, S_2: MAS): MAS

1: $S_1' \leftarrow S_1$
2: dead_pairs $\leftarrow \{(q_1, q_2) \mid q_1 \in Q_1 \wedge q_2 \in Q_2 \wedge \neg\,\text{DeadFree}(q_1, q_2)\}$
3: **for all** $(q_1, q_2) \in$ dead_pairs **do**
4: $\quad \text{Acc}_1'(q_1) \leftarrow \{X_1 \mid X_1 \in \text{Acc}_1'(q_1) \wedge \forall X_2 \in \text{Acc}_2(q_2), X_1 \cap X_2 \neq \emptyset\}$
5: **end for**
6: $S_1' \leftarrow \rho(S_1')$
7: **return** S_1'

Note that Algorithm 5 returns $S_\perp$ when for any model M_1 of S_1, there exists a model M_2 of S_2 such that $M_1 \times M_2$ has a deadlock.

Theorem 58 (Deadlock correction)**.** *Given two marked acceptance specifications S_1 and S_2, $M_1 \models S_1$ is such that for any $M_2 \models S_2$, $M_1 \times M_2$ is deadlock-free if and only if $M_1 \models$ dead_correction(S_1, S_2).*

Proof. ($\Rightarrow$) Assume that for any $M_1 \models S_1$ and $M_2 \models S_2$, $M_1 \times M_2$ is deadlock-free. By Theorem 47, S_1 and S_2 are deadlock-free, which implies that there is no pair of states (q_1, q_2) such that

$\neg\,\mathrm{DeadFree}(q_1, q_2)$. Thus, the set dead_pairs is empty and dead_correction$(S_1, S_2) = S_1$, so $M_1 \models \mathrm{dead\_correction}(S_1, S_2)$.

($\Leftarrow$) Assume that there exists an $M_2 \models S_2$ such that $M_1 \times M_2$ has a deadlock pair of states (r_1, r_2). By Theorem 47, this implies that S_1 and S_2 are not deadlock-free and thus that there exists a pair of states (q_1, q_2) (implemented by (r_1, r_2)) reachable in $\mathrm{Un}(S_1) \times \mathrm{Un}(S_2)$ such that $\neg\,\mathrm{DeadFree}(q_1, q_2)$. Then, in dead_correction$(S_1,\ S_2)$, either the acceptance set of q_1 has been reduced so that $\mathrm{Compat}(\mathrm{Acc}'_1(q_1), \mathrm{Acc}_2(q_2))$ is true and $\mathrm{DeadFree}(q_1, q_2)$ or q_1 is not reachable anymore and then (q_1, q_2) is not reachable in $\mathrm{Un}(S_1) \times \mathrm{Un}(\mathrm{dead\_correction}(S_1, S_2))$. Consequently, either $\mathrm{ready}(r_1) \notin \mathrm{Acc}'_1(q_1)$ or $(r_1, q_1) \notin \pi$, and thus M_1 is not a model of dead_correction(S_1, S_2). □

Complexity 18. The first step of the algorithm (line 2) computes all the pairs of states where a deadlock may occur; its complexity is $O(|Q_1| \times |Q_2| \times |\,\mathrm{Acc}_1\,| \times |\,\mathrm{Acc}_2\,|)$.

The second step (lines 3–5) iterates on all these pairs and removes some elements from the corresponding acceptance sets; the complexity of this loop is $O(|Q_1| \times |Q_2| \times |\,\mathrm{Acc}_1\,| \times |\,\mathrm{Acc}_2\,|)$.

Finally, since the previous step may generate specifications that are not in normal form, ρ is applied to clean invalid states, transitions, or acceptance sets.

Combining these steps gives the following complexity for dead_correction:

$$O(|Q_1| \times |Q_2| \times |\,\mathrm{Acc}_1\,| \times |\,\mathrm{Acc}_2\,| + |Q_1|^3 \times |\,\mathrm{ready}_1\,| + |Q_1|^2 \times |\,\mathrm{Acc}_1\,|)$$

4.4.3 Livelock correction

Secondly, given S_1 and S_2 two deadlock-free marked acceptance specifications, we propose to refine the set of models of S_1 such that the obtained specification S'_1 is livelock-free with S_2. In order to avoid potential livelocks between two marked acceptance specifications, we will use two methods: removing some transitions so that states from which it is not possible to guarantee termination will not be reached and forcing some transitions to be eventually realized in order to guarantee that it will be possible to leave cycles without marked states. For this last method, we introduce marked acceptance specifications with priorities that are marked acceptance specifications in which we identify some transitions called *priorities*; in the satisfaction relation, we then add a constraint to eventually realize these transitions.

Definition 55 (Marked acceptance specification with priorities)**.** *A marked acceptance specification with priorities is a tuple* $(Q, q^0, \delta, \mathrm{Acc}, P, F)$ *where* $(Q, q^0, \delta, \mathrm{Acc}, F)$ *is a marked acceptance specification and* $P : 2^{2^{Q \times \Sigma}}$ *is a set of* priorities.

Definition 56 (Satisfaction)**.** *An automaton M implements a marked acceptance specification with priorities S if M implements the underlying marked acceptance specification with a simulation relation π and for all $\mathcal{P} \in P$, either* $\forall (q, a) \in \mathcal{P}, \forall r, (r, q) \notin \pi$ *or* $\exists (q, a) \in \mathcal{P}, \exists r, (r, q) \in \pi \wedge a \in \mathrm{ready}(r)$.

Intuitively, P represents a conjunction of disjunctions: at least one transition from each element of P must be implemented by the models of the specification.

Let S_1 and S_2 be two marked acceptance specifications and q_1 a state of S_1 such that q_1 belongs to a livelock. Then, there exist a cycle $\mathcal{C}_1$ in S_1 and its partner $\mathcal{C}_2$ in S_2 such that the conditions given in Definition 48 are false. Given this cycle, Algorithm 6 ensures that the possible livelock will not happen, either by adding some priorities or removing some transitions. We then iterate over the possible cycles, fixing those that may cause a livelock, as described in Algorithm 7.

Figure 4.12 shows some examples of the application of the different rules defined in Algorithm 7 using the specifications depicted in Figure 4.9.

For the two marked acceptance specifications of Figure 4.11, the correction is just to add a priority for the transition $((0, 0'), b)$ of the pre-quotient: it disallows the invalid models such as M_1^1 of Figure 4.11(e) but is permissive enough to allow valid models like M_1^2 of Figure 4.11(f).

Algorithm 6 live_correction_cycle (S_1: MASp, $\mathcal{C}_1$: Cycle, S_2: MAS, $\mathcal{C}_2$: Cycle): MASp

1: **if** $\mathcal{A}_{\mathcal{C}_2} \neq \emptyset$ **then**
2: $\quad Q_A \leftarrow \{q_1 \mid Q_2(q_1) \in \mathrm{dom}(\mathcal{A}_{\mathcal{C}_2}) \wedge \forall A \in \mathcal{A}_{\mathcal{C}_2}[Q_2(q_1)], A \cap \mathrm{ready}(q_1) \neq \emptyset\}$
3: $\quad$ **if** $Q_A \neq \emptyset$ **then**
4: $\quad\quad P \leftarrow \{\bigcup_{1 \leq i \leq |Q_A|} \{(q_i, a) \mid a \in X_i\} \mid X_i \in \{A \cap \mathrm{ready}(q_i) \mid A \in \mathcal{A}_{\mathcal{C}_2}[Q_2(q_i)]\}\}$
5: $\quad\quad$ **return** $(Q_1, q_1^0, \delta_1, \mathrm{Acc}_1, P_1 \cup P, F_1)$
6: $\quad$ **end if**
7: **else if** $\mathrm{dom}(\mathcal{C}_2) \cap F_2 = \emptyset$ **then**
8: $\quad \mathrm{Acc}' \leftarrow \mathrm{Acc}_1$
9: $\quad$ **for all** $q_1 \in \{Q_1(q_2) \mid q_2 \in \mathrm{dom}(\mathcal{O}_{\mathcal{C}_2})\}$ **do**
10: $\quad\quad \mathrm{Acc}'(q_1) \leftarrow \{X \mid X \in \mathrm{Acc}_1(q_1) \wedge \forall O \in \mathcal{O}_{\mathcal{C}_2}[Q_2(q_1)], X \cap O \neq \emptyset\}$
11: $\quad$ **end for**
12: $\quad$ **return** $\rho((Q_1, q_1^0, \delta_1, \mathrm{Acc}', P_1, F_1))$
13: **end if**
14: $\mathrm{Acc}' \leftarrow \mathrm{Acc}_1$
15: **for all** $q_1 \in Q_1$ **do**
16: $\quad \mathrm{Acc}'(q_1) \leftarrow \{X \mid X \in \mathrm{Acc}_1(q_1) \wedge \forall a \in X, \delta(q_1, a) \notin \mathrm{dom}(\mathcal{C}_1)\}$
17: **end for**
18: **return** $\rho((Q_1, q_1^0, \delta_1, \mathrm{Acc}', P_1, F_1))$

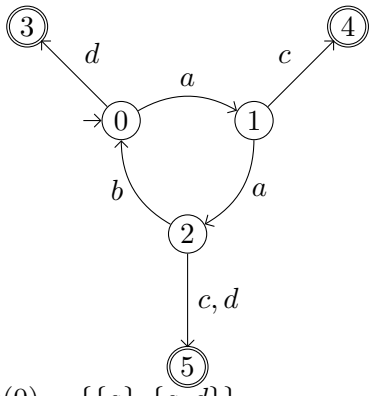
Algorithm 7 live_correction (S_1: MAS, S_2: MAS): MASp

1: $S_1' \leftarrow (Q_1, q_1^0, \delta_1, \mathrm{Acc}_1, \emptyset, F_1)$
2: **for all** $\mathcal{C}_1 \in \mathrm{Cycle}_{\models}(S_1)$ such that $\forall q_1 \in \mathrm{dom}(\mathcal{C}_1), |Q_2(q_1)| = 1$ **do**
3: $\quad$ **if** $\neg\,\mathrm{LiveFree}(\mathcal{C}_1, S_2)$ **then**
4: $\quad\quad \mathcal{C}_2 \leftarrow \{Q_2(q) \mapsto \mathcal{C}_1(q) \mid q \in \mathrm{dom}(\mathcal{C}_1)\}$
5: $\quad\quad S_1' \leftarrow \mathrm{live\_correction\_cycle}(S_1', \mathcal{C}_1, S_2, \mathcal{C}_2)$
6: $\quad$ **end if**
7: **end for**
8: **return** S_1'

Theorem 59 (Livelock correction)**.** *Given two marked acceptance specifications S_1 and S_2, $M_1 \models S_1$ is such that for any $M_2 \models S_2$, $M_1 \times M_2$ is livelock-free if and only if $M_1 \models$ live_correction(S_1, S_2).*

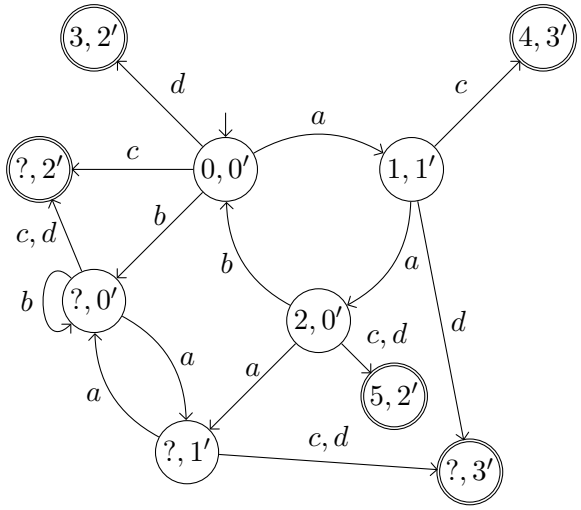
Proof. ($\Rightarrow$) Assume that for any $M_1 \models S_1$ and $M_2 \models S_2$, $M_1 \times M_2$ is livelock-free. By Theorem 51, S_1 and S_2 are livelock-free which means, by Definition 49, that for any implementable cycle $\mathcal{C}_1$ in S_1 such that its states have a partner in S_2, we have $\mathrm{LiveFree}(\mathcal{C}_1, S_2)$. In this case, the test at line 3 of Algorithm 7 is always false and so live_correction(S_1, S_2) returns S_1, of which M_1 is a model by hypothesis.

($\Leftarrow$) Assume that there exists an $M_2 \models S_2$ such that $M_1 \times M_2$ has a livelock.



$\mathrm{Acc}(0) = \{\{a\}, \{a, d\}\}$
$\mathrm{Acc}(1) = \{\{a\}, \{a, c\}\}$
$\mathrm{Acc}(2) = \{\{b\}, \{b, c\}, \{b, d\}, \{b, c, d\}\}$
$\mathrm{Acc}(3) = \mathrm{Acc}(4) = \mathrm{Acc}(5) = \{\emptyset\}$
$P = \{\{(0, d), (2, c), (2, d)\}\}$

(a) live_correction(S_1, S_2): example of compatible reachability correction for the first case (lines 1 to 6 of Algorithm 6)



$\mathrm{Acc}(?, 0') = \mathrm{Acc}(0, 0') = \mathrm{Acc}(2, 0') = \{\{a, b, c, d\}\}$
$\mathrm{Acc}(?, 1') = \{\{a, c\}, \{a, d\}, \{a, c, d\}\}$
$\mathrm{Acc}(1, 1') = \{\{a, c\}, \{a, c, d\}\}$
$\mathrm{Acc}(?, 2') = \mathrm{Acc}(3, 2') = \mathrm{Acc}(5, 2') = \mathrm{Acc}(4, 3') = \mathrm{Acc}(?, 3') = \{\emptyset\}$

(b) live_correction(S_2, S_1): example of compatible reachability correction for the second case (lines 7 to 12 of Algorithm 6)

Figure 4.12: Examples of livelock correction for the specifications of Figure 4.9

- If there exists a cycle $\mathcal{C}_1 \in \text{Cycle}_{\models}(S_1)$ which is implemented in M_1 by the states of the loop in which there is a livelock when combined with M_2, then live_correction_cycle will be called with $\mathcal{C}_1$. There are three cases:

  - If $\mathcal{A}_{\mathcal{C}_2}$ is not empty, some transitions are present in all the models of S_2 implementing $\mathcal{C}_2$, so the models of S_1 should realize (at least) one of these transitions once. If it is possible, some priorities are added, see lines 3 to 5 of Algorithm 6. This addition will only remove the models of S_1 that never realize any transition in $\mathcal{A}_{\mathcal{C}_2}$ and thus that will have a livelock with some models of M_2 (which only realize the transitions of $\mathcal{A}_{\mathcal{C}_2}$).
  - If $\mathcal{A}_{\mathcal{C}_2}$ is empty but there is no marked state in $\mathcal{C}_2$, all the models of S_2 implementing $\mathcal{C}_2$ will eventually realize a transition of $\mathcal{O}_{\mathcal{C}_2}$ in order to reach a marked state (as there is none in the cycle). The only way to avoid a livelock with any model of S_2 is to realize all the transitions that these models may use to reach a marked state, which is done in lines 7 to 12.
  - Otherwise, there will always be a possible livelock with some models of S_2, so the only possibility is to disallow all the models which implement this cycle, which is done in lines 13 to 18.

  So M_1 is not a model of the marked acceptance specification with priorities returned by live_correction_cycle for $\mathcal{C}_1$ and thus it is not a model of live_correction(S_1, S_2).

- Otherwise, multiple cycles are implemented simultaneously and there will also be livelocks in the models which implement only one of the cycles. As argued in the previous item, applying live_correction_cycle for these cycles will generate a specification forbidding the corresponding models, and then M_2 will not be a model of the resulting specification as it only combines the behavior of these models. □

Complexity 19. We first consider the complexity of live_correction_cycle. The algorithm depends on the values of $\mathcal{A}_{\mathcal{C}_2}$ and $\mathcal{O}_{\mathcal{C}_2}$, so a first step is to compute this; their complexity was given earlier. Then, we have three cases. The first one (lines 1–6) has two main steps. First it computes the set Q_A (line 2) with a complexity of $O(|Q_1| \times |\operatorname{Acc}_2|)$. Then, if Q_A is not empty, it generates a set of priorities (line 4); the complexity of this step is $O(|\operatorname{Acc}_2|^{|Q_1|} \times |\operatorname{ready}_1|)$. The second case (lines 7–12) first has a loop removing some elements from some acceptance sets which has a complexity of $O(|Q_1| \times |\operatorname{Acc}_1| \times |\operatorname{Acc}_2|)$ and then applies ρ, which complexity was given earlier. The third case (lines 13–18) has a similar loop which complexity is $O(|Q_1| \times |\operatorname{Acc}_1| \times |\operatorname{ready}_1|)$ and then calls ρ. Thus, the most complex part of this algorithm is the computation of the set of priorities (line 4), so we conclude that its complexity is $O(|\operatorname{Acc}_2|^{|Q_1|} \times |\operatorname{ready}_1|)$.

We now study the complexity of live_correction. It iterates on the implementable cycles of S_1. We explained earlier that an upper bound for the number of implementable cycles in a specification is $2^{|Q_1| \times |\operatorname{ready}_1|}$, so we obtain the following complexity:

$$O\left(2^{|Q_1| \times |\operatorname{ready}_1|} \times |\operatorname{Acc}_2|^{|Q_1|} \times |\operatorname{ready}_1|\right)$$

By applying successively these operations (dead_correction and live_correction), we define the following operation $\rho_{\mathcal{T}}$:

$$\rho_{\mathcal{T}}(S_1, S_2) = \text{live\_correction}(\text{dead\_correction}(S_1, S_2), S_2)$$

This operation has the same complexity as live_correction, since live_correction is more complex than dead_correction (exponential versus polynomial).

Given two marked acceptance specifications S_1 and S_2, $\rho_{\mathcal{T}}(S_1, S_2)$ refines the set of models of S_1 as precisely as possible so that their product with any model of S_2 is terminating.

Theorem 60 (Incompatible reachability correction)**.** *Given two marked acceptance specifications S_1 and S_2, for any $M \models \rho_{\mathcal{T}}(S_1, S_2)$ and $M_2 \models S_2$, $M \times M_2$ is terminating, and an $M_1 \models S_1$ is such that for any $M_2 \models S_2$, $M_1 \times M_2$ is terminating if and only if $M_1 \models \rho_{\mathcal{T}}(S_1, S_2)$.*

Proof. For any $M \models \rho_{\mathcal{T}}(S_1, S2)$ and $M_2 \models S_2$, $M \times M_2$ is terminating if and only if $M \times M_2$ is deadlock-free and livelock-free. By Theorems 47 and 51, this is true if and only if $\rho_{\mathcal{T}}(S_1, S_2)$ and S_2 are deadlock-free and livelock-free, which is true by definition of $\rho_{\mathcal{T}}$ and Theorems 58 and 59. □

4.4.4 Quotient definition

We can now combine the pre-quotient and cleaning operations to define the quotient of two marked acceptance specifications.

Definition 57. *The* quotient *of two marked acceptance specifications S_1 and S_2, denoted S_1/S_2, is $\rho_{\mathcal{T}}(S_1 /\!\!/ S_2, S_2)$.*

Complexity 20. The complexity of the quotient operation is the combination of the complexities of the pre-quotient, which is exponential w.r.t. the size of the alphabet, and the livelock correction algorithm, which is exponential w.r.t. the number of states of the specifications (actually, it depends on the number of states in the cycles of the specifications, but in the worst case, all the states belong to a same cycle):

$$O\left(|Q_1| \times |Q_2| \times 2^{|\Sigma|} \times (|\operatorname{Acc}_2| \times |\operatorname{Acc}_1| + |\Sigma|) + 2^{|Q_1| \times |\operatorname{ready}_1|} \times |\operatorname{Acc}_2|^{|Q_1|} \times |\operatorname{ready}_1|\right)$$

Theorem 61 (Soundness)**.** *Given two marked acceptance specifications S_1 and S_2 and an automaton $M \models S_1/S_2$, for any $M_2 \models S_2$, we have $M \times M_2 \models S_1$.*

Proof. By Theorem 60, we know that for any $M_2 \models S_2$, $M \times M_2$ is terminating. Thus, Theorem 55 implies that $M \times M_2 \models S_1$. □

Theorem 62 (Completeness)**.** *Given two marked acceptance specifications S_1 and S_2 and an automaton M such that for all $M_2 \models S_2$, we have $M \times M_2 \models S_1$, then $\rho_u(M, S_2) \models S_1/S_2$.*

Proof. We know by Corollary 4 that $\rho_u(M, S_2) \models S_1 /\!\!/ S_2$. We then deduce by Theorem 60 that $\rho_u(M, S_2) \models S_1/S_2$. □

As a consequence, incremental design of component-based systems is enabled. Given S_1 and S_2, the system designer can either distribute the implementation tasks S_1/S_2 and S_2 or, alternatively, decide to reuse an off-the-shelf component implementing S_2. The product of the models of S_2 and S_1/S_2 will realize S_1 and will, in particular, satisfy by construction the reachability objectives it includes.

4.5 Related Work

In this chapter, we introduced marked acceptance specifications, a specification formalism for under-specified systems under reachability constraints. We developed a specification theory for them with refinement, product, conjunction, and quotient guaranteeing by construction reachability properties.

Modal specifications enriched with marked states have been first introduced in [DDM10a] for the supervisory control of services. A product of marked modal specifications has been investigated in [CR12]. As the quotient is not considered in these papers, the need for the more expressive framework of marked acceptance specifications was not found as pointed out.

Marked acceptance specifications can also be related to automata-theoretic specifications in which states are annotated with propositional formulas expressing implementation variants and, possibly, an obligation of progress. This is the case of *annotated automata* [WMN05] and *operating guidelines* [MS05, LW11]. While both formalisms have a product operator, they are missing the conjunction and quotient operators.

The reachability considered in this paper can be stated in CTL by `AG(EF(final))` and cannot be captured in LTL. Thus, satisfiability of a marked acceptance specification cannot be based on the LTL model checking for modal specifications studied in [BČK11].

The compatibility criterion associated here to specifications is related to a reachability property. In the controller synthesis community, it is often referred to as non-blockingness [CL08]. Usually for interface automata [dAH01] or modal interfaces [LNW07a, RBB$^+$11, LV12, LV13, BFLV15] the compatibility refers to a safety property: *error* states are not reachable in some environment. Several notions of compatibility are introduced in [BSBM04] for services. In particular, the one called *deadlock-freeness* is equivalent to the compatible reachability presented here.

Chapter 5

Implementation

In addition to the theoretical results presented in the previous chapters, we implemented these new formalisms in a tool called MAccS [VR15a]. We first give an overview of the tool and its possibilities. Then, we present the state of the art for tools that manipulate similar formalisms. Finally, we show some benchmarks illustrating the optimization offered by convex-closed sets w.r.t. acceptance sets, as well as a comparison of different data structures used to represent sets.

5.1 Overview

In the previous chapters, we introduced convex and marked acceptance specifications and defined some operations on these, which we proved correct. But we are also interested in having a concrete implementation of these formalisms to be able to use them in practice, show how they work without having to manually compute the results of the operations, and examine their performance.

We developed a tool called MAccS (abbreviation of Marked Acceptance Specifications, although it now supports additional specification formalisms) which implements our works and some existing specification formalisms (such as modal specifications) for benchmarking purposes. It is written in C++ and comes both as a library for embedding it in applications or doing automatic processing and with a GUI allowing to easily manipulate some specifications and apply operations to them.

The graphs underlying to the automata and specifications are represented using the Boost Graph Library [SLL02]. The GUI is made with the framework Qt and Dot [GN00] is used to generate the layout of the automata and specifications. A screenshot is shown in Figure 5.1.

Automata and specifications can be created interactively using the GUI or written in a simple textual format. An excerpt of the representation in this format of the specification depicted in Figure 5.1 is shown below. It is also possible to import and export automata and specifications from and to the Dot format [GN00].

```
init {{login}}
read {{read,post},{read,post,logout}}
...
end marked {{}}
init -login-> read
read -read-> reply1
reply1 -response-> read
...
```

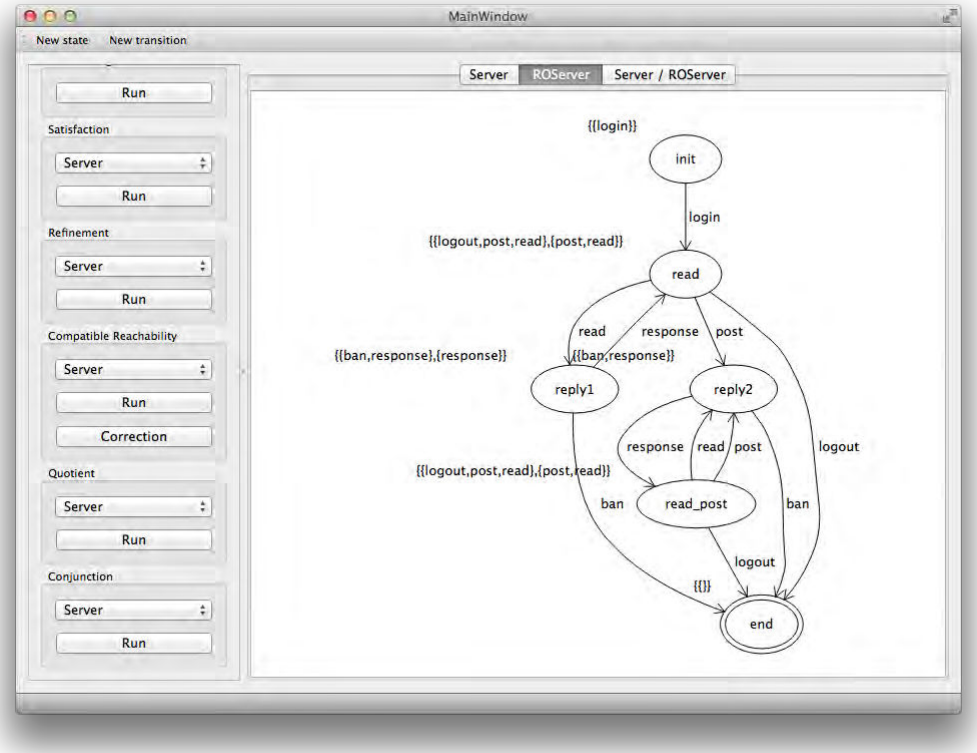


Figure 5.1: Screenshot of MAccS

5.2 State of the Art

A number of tools have been developed to implement various specification formalisms:

TAV [GLZ89, BLS95] is probably the very first tool implementing modal specifications. It can be used to define some specifications, check the refinement relation, and compute their parallel composition.

EPSILON [ČGL93] is an extension of TAV with timed modal specifications.

MTSA [DFCU08] extends a tool on transition systems, LTSA, to nondeterministic modal transition systems. It offers various operations such as refinement checking, parallel composition, and LTL model-checking.

MoTraS [KS13] handles nondeterministic modal transition systems and their disjunctive, boolean, and parametric extensions. The support for modal and disjunctive transition systems is quite extensive (refinement, LTL model-checking, deterministic hull, conjunction, parallel composition). For boolean and parametric transition systems, only two operations are available: refinement checking and the deterministic hull.

MIO Workbench [BMSH10, BML11] uses modal input/output interfaces. It implements the standard operations of a specification theory: refinement, conjunction, product, and quotient.

Mica [Cai11] implements the deterministic modal interface theory described in [RBB+11] and the operations of the associated specification theory.

Tool	Lang.	Theory	Non-Det.	Operations					
				$\leq$	$\wedge$	$\otimes$	/	MC	Det. Hull
TAV	Prolog	MTS	✓	✓		✓		HML	
EPSILON	Prolog	Timed MS	✓	✓		✓			
MTSA	Java	MTS	✓	✓		✓		LTL[a]	
MoTraS	Java	MTS	✓	✓	✓	✓	✓[b]	LTL	✓
		DMTS	✓	✓	✓	✓		LTL	✓
		BMTS, PMTS	✓	✓					✓
MIO W.	Java	MIO	✓	✓	✓[b]	✓	✓[b]		
Mica	OCaml	MIO		✓	✓	✓	✓		n/a
ECDAR PyECDAR	Java Python	Timed I/O		✓	✓	✓	✓		n/a
BALM	C	FSM	✓[c]	✓			✓		
MAccS	C++	MAS, CAS		✓	✓	✓	✓		n/a

The "Non-Det." column indicates if nondeterministic specifications are allowed.
The operations denoted in the table are: refinement ($\leq$), conjunction ($\wedge$), product ($\otimes$), quotient (/), model-checking (we indicate in the column the logic used), and the deterministic hull of nondeterministic specifications.

[a] According to [BČK11], it sometimes produces an incorrect result.
[b] Only for deterministic specifications.
[c] Non-deterministic finite state machines are determinized.

Table 5.1: Overview of the functionalities of related tools

ECDAR [DLL+10a] extends the UPPAAL model checker with a specification theory based on timed input/output automata. These specifications also have may/must modalities, with the constraint that transitions labeled with output actions are *uncontrollable*, i.e., they have to be in the must set.

PyECDAR [LT13] is a Python implementation of the same specification theory as ECDAR.

BALM-II [CPM+12] solves equations and inequations over finite state machines. Given two finite state machines C and A, it can find the most general X that is a solution of the equation $C \bullet X = A$, where $\bullet$ denotes the synchronous composition operator. It also works with inequations, i.e., $C \bullet X \subseteq A$, and with the so-called parallel or asynchronous composition operator. This equation-solving is similar to our quotient operation: given two specifications C and A, the most general X such that $C \otimes X \leq A$ is $X = A/C$.

As far as we know, no tool implements a marked extension of some specification formalism. Regarding acceptance specifications, nondeterministic disjunctive transition systems are equivalent

to nondeterministic acceptance automata which are naturally a superset of deterministic acceptance specifications, so it should be possible to use MoTraS to manipulate these. But we are not aware of any implementation of deterministic acceptance specifications or deterministic convex acceptance specifications, with algorithms optimized to use the hypotheses of determinism or convexity.

5.3 Benchmarks

We now present some experimental results. The first benchmark compares convex acceptance specifications with non-convex ones in order to see if the optimized representation we proposed is indeed more efficient. Then, we will compare several implementations of sets based on trees, hash tables, or bit fields.

5.3.1 Convex versus acceptance specifications

In Section 3.3, we introduced convex-closed acceptance sets and showed how they can be represented efficiently in order to reduce the complexity of standard operations on acceptance specifications, such as refinement, conjunction, and quotient. In this section, we show some experimental results confirming these performance improvements.

We generate random specifications and compute some operations: checking if they refine themselves, their conjunction, etc. There are mainly two variables which may be changed for the benchmarks: the number of states and the size of the alphabet. Some results are shown in Figure 5.2. We observe that increasing the number of states has not much influence on the performance difference between the algorithms (Figure 5.2 only shows such benchmark for the refinement operation, but the results for the other operations are similar). Indeed, using convex-closed acceptance sets only improves the performance of the operations on each acceptance set which depends on the size of the alphabet, not on the number of states. On the other hand, there is a clear performance improvement when the alphabet grows larger. In particular, for the quotient operation, we see that the exponential blow-up disappears.

5.3.2 Representation of sets

A critical element to have efficient operations on acceptance specifications is the representation of acceptance sets. We showed how one can use a specific subset of acceptance sets, convex-closed sets, to do so. But using an adequate data structure to represent these acceptance sets could also greatly improve performances.

We compare three data structures:

std::set is the traditional set implementation of the C++ standard library, based on balanced trees (typically red-black trees).

std::unordered_set is a new set implementation introduced in the C++11 standard based on a hash table.

std::bitset represents a fixed-size sequence of bits. Since we know the alphabet Σ of the specifications, any set of actions can be represented as a subset of Σ and thus as a set of $|\Sigma|$ bits, where the n-th bit indicates if the n-th element of Σ is present in the subset. Moreover, set operations such as the union or intersection are very easily done using bitwise operations.

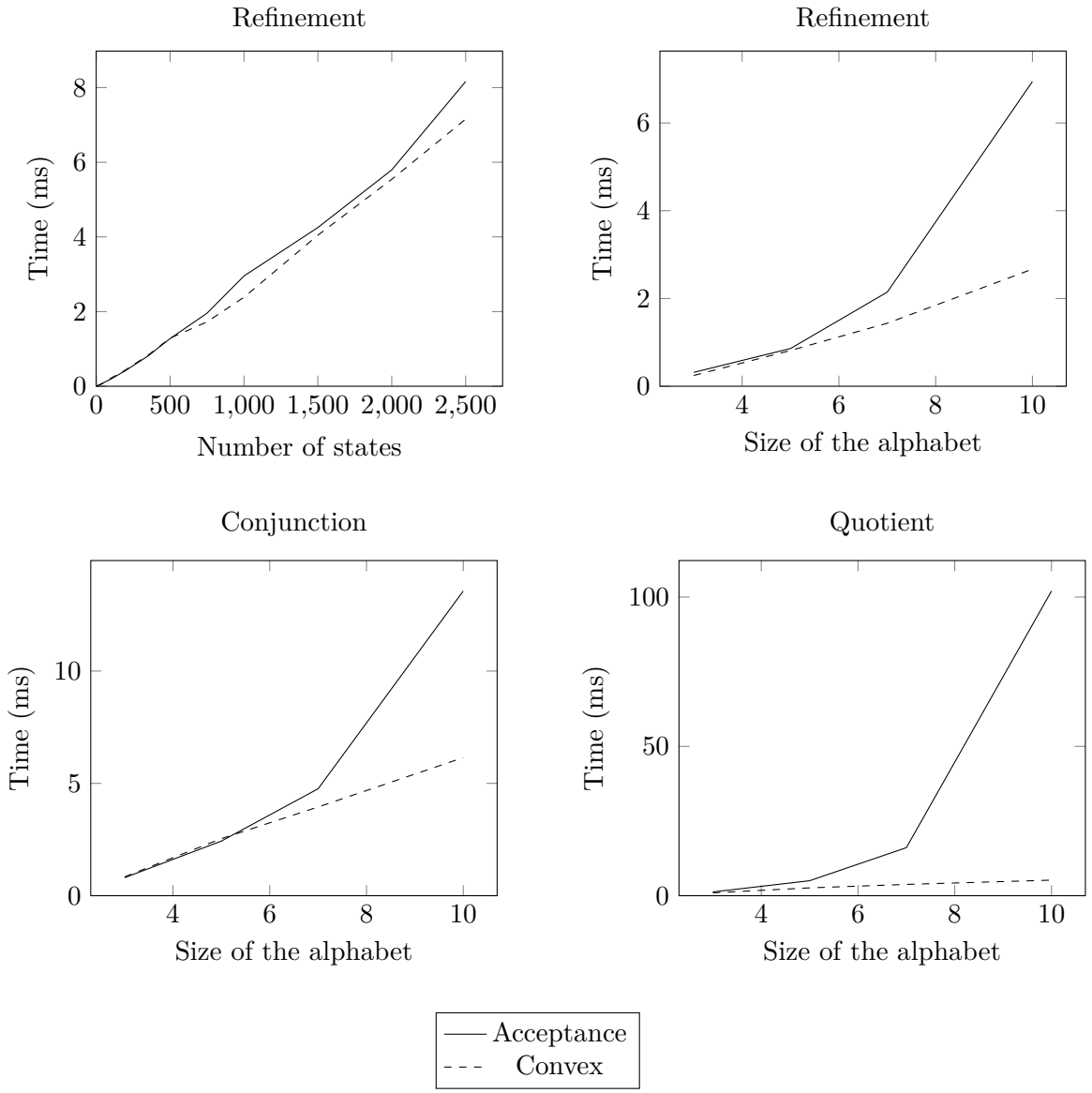


Figure 5.2: Acceptance sets versus convex-closed acceptance sets

Observe that we typically don't have very large sets (it would not make much sense to have an alphabet with thousands of elements), but numberous small sets, so comparing the "big O" complexity of the operations may not be very useful for our use case.

We compare these three implementations of sets on the operations on acceptance specifications. The results are depicted in Figure 5.3. We observe that **std::bitset** is clearly faster than the two other implementations. These two other implementations, **std::set** and **std::unordered_set**, are rather close, although the former seems marginally faster than the latter.

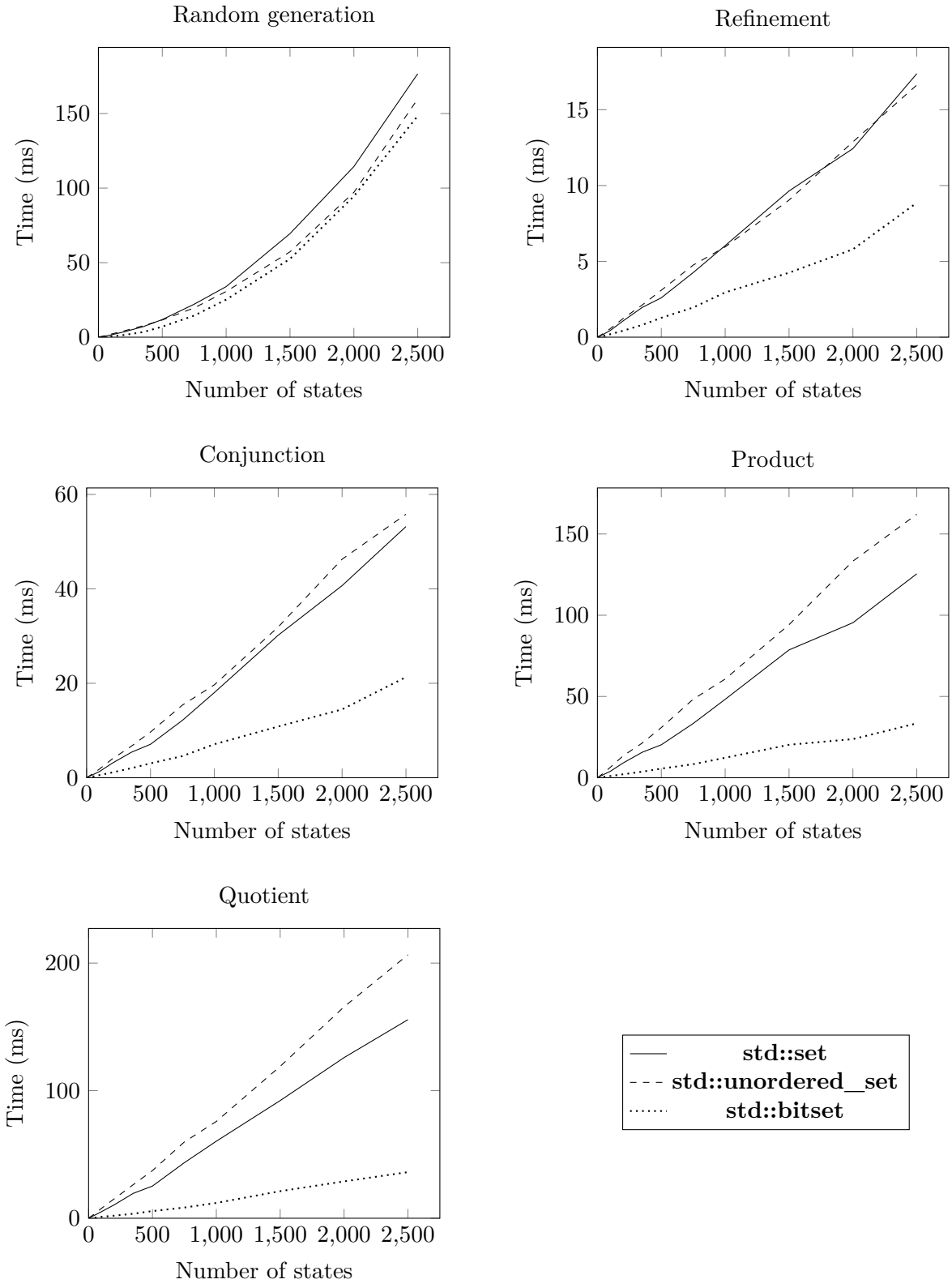


Figure 5.3: Set implementations benchmarks

Chapter 6

Conclusion

6.1 Contributions

In this thesis, we presented two main theoretical results in the form of two new specification formalisms based on acceptance specifications; one offering improved performances in exchange for a slightly reduced expressiveness and the other allowing to express a new type of constraints on paths using marked states. We also implemented these specification formalisms in a tool and presented some experimental results.

Convex acceptance specifications offer more efficient operations than acceptance specifications. While acceptance specifications offer a very expressive specification formalism, some operations, in particular the quotient, have a high complexity. Using convex-closed acceptance sets allows us to define operations with a lower complexity and, in particular, to avoid an exponential blow-up w.r.t. the size of the alphabet in the quotient operation. Moreover, convex acceptance specifications, although less expressive than acceptance specifications, are still more expressive than many other specification formalisms such as modal specifications, disjunctive modal specifications, and modal specifications with obligations. We used the Coq proof assistant to ensure the validity of our results.

Marked acceptance specifications allow expressing reachability properties and come with all the typical operations to make a complete specification theory, i.e., refinement checking, conjunction, product, and quotient. While many specification formalisms only focus on expressing local properties with, for example, modalities, boolean formulas, or acceptance sets, marked states can be used to express constraints on paths by guaranteeing that some states will always be reachable. This can be used to ensure the absence of deadlocks and livelocks and, for instance, that a system is terminating or that a checkpoint will be reachable infinitely often.

MAccS is a tool offering an implementation of these theories. Its graphical interface can be used to design some specifications and apply some operations to them. It can also save and load specifications from a textual representation and be integrated in other pieces of software. We also used it to run some benchmarks: we showed that using convex-closed acceptance sets was indeed much faster in practice than using arbitrary acceptance sets and we demonstrated that our tool could handle specifications with thousands of states in a fraction of a second.

6.2 Future Work

We now discuss some possible orientations for future work. We first describe some possible extensions of the contributions of this thesis and then suggest different directions for new specification theories.

6.2.1 Short term

From a practical point of view, MAccS could use some additional work. In particular, it is currently single-threaded; some parts of the algorithms (such as the computation of the acceptance set of each state for conjunction, product, and quotient) could be parallelized in order to make use of all the cores available in a computer.

We used the Coq proof assistant to check the proofs on convex-closed acceptance sets. It would be interesting to continue this work in order to verify the complete specification theory. However, graphs and automata, like sets, are not inductive structures and are thus difficult to reason about using Coq or similar languages like Isabelle/HOL or Agda. While Coq's standard library offers good foundations for manipulating sets, there is no such standard framework for automata. Entire theses have been dedicated to this subject, for instance [Pic12] for the representation of graphs using coinduction in Coq and [Gio13] for the representation of pointer structures, including graphs, in Isabelle/HOL.

From a more theoretical point of view, several improvements could be made to the theories presented in this thesis. First, we introduced some alphabet extension operations on acceptance specifications in order to handle dissimilar alphabets. Such operations should be extended to marked acceptance specifications. This is an important step to design and build large systems with reachability properties as the various components may have different alphabets.

Secondly, we could extend our specification formalisms with input/output labels: the inputs would represent actions emitted by the environment of the system under design while the outputs would stand for the actions stemming from the system. The question of product in an open setting has been studied for interface automata in [dAH01]. Error states are identified as being the states in which one interface automata may output an action that cannot be matched with a transition labeled by the corresponding input in another interface automata. The presence of error states does not lead to forbid the composition; instead an optimistic approach is advocated: composition is allowed if there exists a third interface automata, called *environment*, closing the system and avoiding the reachability of the error states. Extending optimistic composition to marked acceptance specifications would lead to consider a more *cooperative* environment that would help for the reachability of global marked states.

6.2.2 Mid term

We studied two different problems in this thesis: on one hand we optimized the representation of convex-closed acceptance sets and on the other hand we introduced a more expressive formalism using marked states. It would be interesting to combine these two results to make "marked convex-closed acceptance specifications." However, ensuring that convex-closure is preserved—in particular by the quotient operation and the complex part removing livelocks—may be difficult.

In this thesis, we only considered deterministic specifications, as explained in Section 2.3. There is a large amount of work on nondeterministic specifications and it could thus be interesting to see if our results are preserved when considering nondeterministic specifications. This may be rather challenging since nondeterministic operations are typically much more difficult to define than

their deterministic counterparts. Moreover, the quotient of nondeterministic acceptance automata defined in [BDF$^+$13] has an exponential blow-up for its number of states; as already conjectured in Section 3.3.7, convexity may partially help to improve the situation. Another possibility would be to consider alternative semantics for nondeterministic specifications based on failure traces [BHR84] instead of a simple simulation semantics, as advocated in [BV15].

6.2.3 Long term

Other kind of compatibility properties could be targeted in the context of a specification theory like, for instance, the opacity [Maz04] initially defined in the security community. By definition, a system is said opaque if a given set of traces, called the secret, cannot be inferred from a partial observation. To the best of our knowledge, no compositionality results exist for it. The starting points for a specification theory offering correct-by-construction opaque systems would be [AČZ06] for the refinement and [DDM10b] for the quotient.

A motivation for introducing marked acceptance specification was the need for a specification formalism to model under-specified services together with their possible session termination thanks to the marked states. Now, in a next step, an orchestration of services could be represented by a modal specification whose transitions would be labeled by the identifier of a service modeled via a marked acceptance specification. Each transition would then be interpreted as a call to the corresponding service whose associated returns would occur when final states are reached in the callees. Alternatively, modal visible pushdown automata could be studied directly.

We mentioned the possibility of continuing the mechanization of our results in the Coq proof assistant in order to ensure their validity. When considering extensions of specification theories with parameters or data, in particular over infinite domains, typical decision procedures often become very inefficient and some problems are even undecidable. A way to solve this type of problems is to generate proofs obligations which can then be proved using automatic solvers or proof assistants. For instance, this is the approach used in the Atelier B and research projects such as BWare [DDMM14]. Then, a Coq mechanization could be useful not just to increase the confidence in the results, but also to offer users a way of proving some properties on their specifications.

Version française

Chapitre 1

Introduction

Contexte. Les programmes informatiques prennent une place de plus en plus importante dans nos vies. Certains de ces programmes, comme par exemple les systèmes de contrôle de centrales électriques, d'avions ou de systèmes médicaux, sont critiques : une panne ou un dysfonctionnement pourraient causer la perte de vies humaines ou des dommages matériels ou environnementaux importants. Les méthodes formelles visent à offrir des moyens de concevoir et vérifier de tels systèmes afin de garantir qu'ils fonctionneront comme prévu. Au fil du temps, ces systèmes deviennent de plus en plus évolués et complexes, ce qui est source de nouveaux défis pour leur vérification. Il devient nécessaire de développer ces systèmes de manière modulaire afin de pouvoir distribuer la tâche d'implémentation à différentes équipes d'ingénieurs. De plus, il est important de pouvoir réutiliser des éléments certifiés et les adapter pour répondre à de nouveaux besoins. Aussi les méthodes formelles doivent évoluer afin de s'adapter à la conception et à la vérification de ces systèmes modulaires de taille toujours croissante.

Présentation. Il y a différentes manières de s'assurer qu'un système vérifie une certaine propriété. Une méthode est de commencer par concevoir et implémenter le système, puis de vérifier que l'implémentation satisfait la propriété, comme préconisé par des processus de développement comme les modèles en V ou en cascade. Par exemple, on peut utiliser des outils de *model-checking* [BK08] pour tester exhaustivement toutes les exécutions du système et obtenir soit une garantie que pour toute exécution, la propriété est satisfaite, soit un contre-exemple correspondant à un cas où la propriété est violée. Si la propriété n'est pas satisfaite, il faut en identifier la cause, la corriger et recommencer l'étape de vérification jusqu'à ce que la propriété soit vérifiée par le système.

Une autre méthode, que nous suivrons dans cette thèse, est d'utiliser des techniques permettant d'obtenir un système *correct par construction* [HS07]. En particulier, dans cette approche, les différentes étapes du flot de conception sont contrôlées ou aidées de telle sorte que les propriétés vérifiées à une certaine étape seront préservées dans les étapes suivantes et finalement satisfaites par l'implémentation.

Prenons par exemple la conception itérative d'un système illustrée dans la figure 1.1. La couche supérieure représente la première étape de la conception d'un système modulaire dans laquelle le système est vu comme étant issu de la collaboration de trois sous-systèmes spécifiés par S_1, S_2 et S_3. Cela illustre plusieurs problèmes.

Conception distribuée. La spécification S_1 peut être raffinée et remplacée par une version plus détaillée formée de deux sous-spécifications S_{11} et S_{12}. Il faut cependant s'assurer que cette étape de conception est autorisée, c'est-à-dire qu'elle préserve les propriétés de S_1. Si c'est

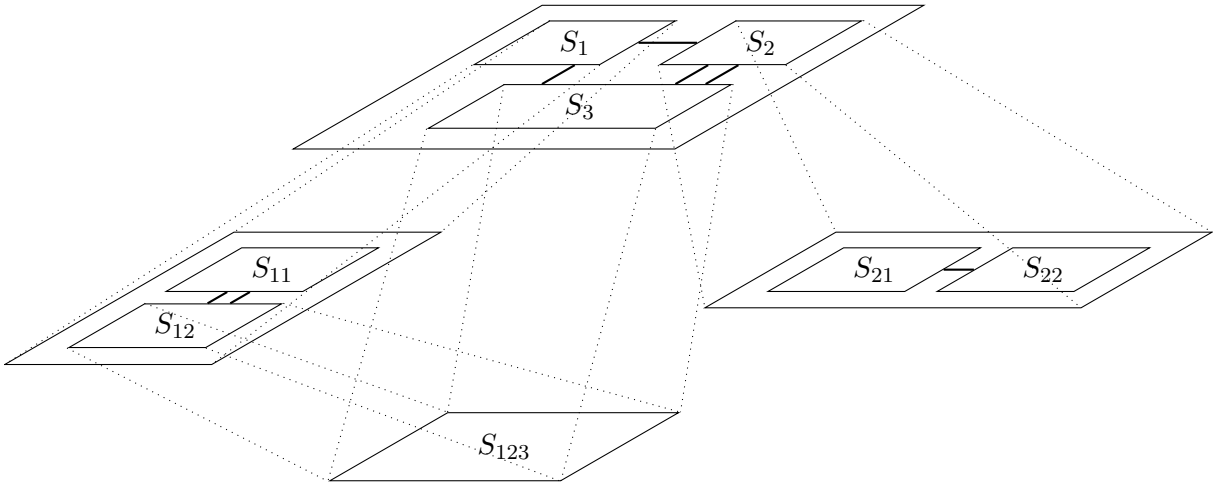


Figure 1.1 : La conception incrémentale d'un système modulaire

bien le cas, S_{11} et S_{12} peuvent être implémentées indépendamment l'une de l'autre par différentes équipes ou sous-traitants et ensuite composées afin d'obtenir une implémentation correcte par construction de S_1.

Réutilisation de composants. Ensuite, S_2 peut être simplifiée en exploitant le fait qu'un composant préexistant S_{21}, dont on dit qu'il est *pris sur l'étagère*, peut offrir un comportement proche que l'on adapte avec la spécification S_{22}.

Fusion de spécifications. Par ailleurs, dans une nouvelle étape de conception, nous pourrions considérer que plusieurs parties du système sont suffisamment proches pour pouvoir être implémentées simultanément par un seul système, ce qui conduit à fusionner plusieurs spécifications, par exemple ici S_{12} et S_3 en S_{123}. Par conséquent, le processus de conception ne doit pas être vu comme arborescent mais plutôt comme un graphe acyclique dirigé. L'utilité d'une opération de fusion sur les spécifications apparaît aussi clairement dans la pratique de la conception par points de vue dans laquelle plusieurs spécifications sont associées à un même système, chacune se concentrant sur un aspect en particulier (fonctionnel, sûreté, temporel, etc.) [RT14].

Raisonner sur la conception de systèmes requiert de définir un modèle formel du système ainsi qu'une algèbre sur les spécifications comportant plusieurs opérations. Elles ont d'abord été identifiées dans [RBB+09, RBB+11], avec les propriétés qu'elles devraient satisfaire : raffinement, composition avec une opération de produit, décomposition avec un quotient et fusion avec une conjonction, tout en offrant des concepts comme l'implémentabilité indépendante et la préservation de propriétés par raffinement. Cette théorie a été appliquée à différents modèles comme, de manière non exhaustive, [BDF+13, CCJK12, BLL+14, LV13, BDH+15, BHL14, BFLV15] ainsi qu'à divers contextes : avec du temps [DLL+10b, BLPR12, KSL13], des propriétés quantitatives [BJL+12, BFJ+13, FKLT15] ou des probabilités [CDL+11, DKL+13]. Les travaux développés dans cette thèse présentent également des contributions qui suivent cette approche algébrique.

Les spécifications peuvent être vues comme des descriptions abstraites de systèmes en cours de conception. Au moins trois niveaux de descriptions sont généralement étudiés [CMP06] :

Niveau signature. Typiquement, les noms des fonctions disponibles sont fournis avec le type de leurs paramètres, le type de leurs valeurs de retour et les exceptions pouvant survenir.

Niveau comportemental. L'ensemble des séquences finies ou infinies d'actions pouvant se produire dans le système est décrit, ce qui permet de s'intéresser à des problèmes comme l'absence d'interblocages ou la terminaison.

Niveau sémantique. Les descriptions fournies permettent d'exprimer ce que *fait* réellement le système. Les ontologies appartiennent à cette famille de formalismes de spécification.

Les formalismes étudiés dans cette thèse appartiennent à la deuxième catégorie. Diverses théories peuvent être utilisées pour exprimer des spécifications comportementales : des logiques, en particulier des logiques temporelles, des algèbres de processus ou des automates. Parmi les nombreuses contributions dans le domaine des théories comportementales compositionelles, notons les travaux basés sur des spécifications *input-complete* (comme les *I/O automata* [LT89], FOCUS [BDD+92] ou les modules réactifs [AH99]) ou des spécifications non *input-complete* (comme les automates d'interface [dAH01], les interfaces avec ports [BHL14] ou les interfaces modales [RBB+11, LNW07a, BFLV15]). Dans ce qui suit, les formalismes de spécification que nous utilisons dans nos différentes contributions sont tous basés sur un type d'automates appelé *spécifications modales*. Une spécification modale est un automate doté de deux types de transitions permettant d'exprimer des comportements obligatoires et optionnels. Raffiner une spécification modale revient à décider si les parties optionnelles devraient être supprimées ou rendues obligatoires. Il est alors possible de réduire la variabilité d'une spécification en la raffinant itérativement jusqu'à ce qu'il ne reste plus de partie optionnelle, ce qui correspond alors à une implémentation de la spécification.

Contributions. Cette thèse contient deux principales contributions théoriques basées sur une extension des spécifications modales, les spécifications à ensembles d'acceptation. La première contribution est l'identification d'une sous-classe des spécifications à ensembles d'acceptation, appelée « spécifications à ensembles d'acceptation convexes », qui permet de définir des opérations bien plus efficaces tout en gardant un niveau d'expressivité élevé. La seconde contribution est la définition d'un nouveau formalisme, appelé « spécifications à ensembles d'acceptation marquées », qui permet d'exprimer des propriétés d'atteignabilité. Ceci peut, par exemple, être utilisé pour s'assurer qu'un système termine ou exprimer une propriété de vivacité dans un système réactif. Les opérations usuelles sont définies sur ce nouveau formalisme et elles garantissent la préservation des propriétés d'atteignabilité. Cette thèse présente également des résultats d'ordre plus pratique. Tous les résultats théoriques sur les spécifications à ensembles d'acceptation convexes ont été prouvés en utilisant l'assistant de preuves Coq. L'outil MAccS a été développé pour implémenter les formalismes et opérations présentés dans cette thèse. Il permet de les tester aisément sur des exemples, ainsi que d'étudier leur efficacité sur des cas concrets.

Plan. Le chapitre 2 présente l'état de l'art. En particulier, nous donnerons la définition des spécifications modales et offrirons un aperçu des nombreuses extensions et variantes de ce formalisme. Le chapitre 3 donne une définition détaillée des spécifications à ensembles d'acceptation et introduit l'optimisation convexe, suivie d'un aperçu de la mécanisation en Coq. L'extension des spécifications à ensembles d'acceptation avec des états marqués est introduite dans le chapitre 4. L'outil MAccS et des résultats expérimentaux sont présentés dans le chapitre 5. Enfin, le chapitre 6 conclut cette thèse et offre des perspectives pour de futurs travaux.

Chapitre 2

Spécifications modales

Dans ce chapitre, nous présentons l'état de l'art. Dans la première section, nous donnerons la définition des spécifications modales ainsi qu'un aperçu de plusieurs extensions. Dans la section 2.2, nous introduirons la notion de théorie de spécification et présenterons les différentes opérations qu'une telle théorie comporte. Enfin, nous discuterons de l'utilisation de spécifications non déterministes.

2.1 Présentation et variantes

Remarque. Le même formalisme est désigné par trois noms différents dans la littérature : spécifications modales, systèmes de transitions modaux et automates modaux. Afin de rester homogène, nous utiliserons le terme de « spécifications modales » (parfois abrégé SM) dans cette section, même si les articles auxquels nous faisons référence utilisent un autre nom. De même, nous parlerons de « spécifications à ensembles d'acceptation » (SEA) bien que certains les appellent « automates à ensembles d'acceptation ».

Les spécifications modales ont été introduites dans [LT88]. Elles offrent un formalisme basé sur des automates qui permet de spécifier des systèmes en exprimant que des transitions sont obligatoires ou optionnelles. Ces spécifications peuvent ensuite être *raffinées* en décidant si des parties optionnelles devraient être supprimées ou rendues obligatoires. Ceci permet de concevoir un système de manière incrémentale en le raffinant pas à pas, jusqu'à ce qu'il ne reste plus que des comportements obligatoires.

Considérons par exemple la spécification modale de la figure 2.1. Il s'agit d'un automate avec quatre états étiquetés 0, 1, 2 et 3, un état initial 0 et des transitions entre ces états. Mais contrairement aux automates classiques, il y a deux types de transitions : les lignes pleines représentent les transitions obligatoires et les lignes en pointillés les transitions optionnelles. Cette spécification décrit le comportement d'un serveur qui reçoit des requêtes et envoie une réponse qui peut soit être calculée directement, soit être obtenue en envoyant une demande à un autre serveur.

On peut aussi voir une spécification modale comme la caractérisation d'une famille — finie ou non — de systèmes, appelés ses *modèles* ou *implémentations*, représentés par des automates correspondant aux différentes combinaisons de choix d'implémentation pouvant être faits par raffinement. Quelques modèles de l'exemple de spécification présenté précédemment sont représentés dans la figure 2.2. À partir de l'état initial 0 de la spécification, il y a une transition obligatoire, étiquetée « requête », aussi tous les modèles ont cette transition. Ensuite, dans l'état 1, il y a deux transitions optionnelles, qui peuvent donc être réalisées ou non. Dans M_1, nous choisissons de

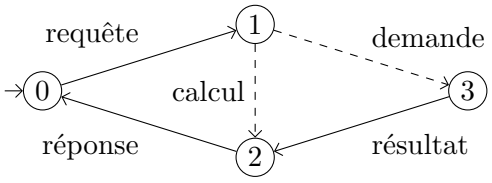
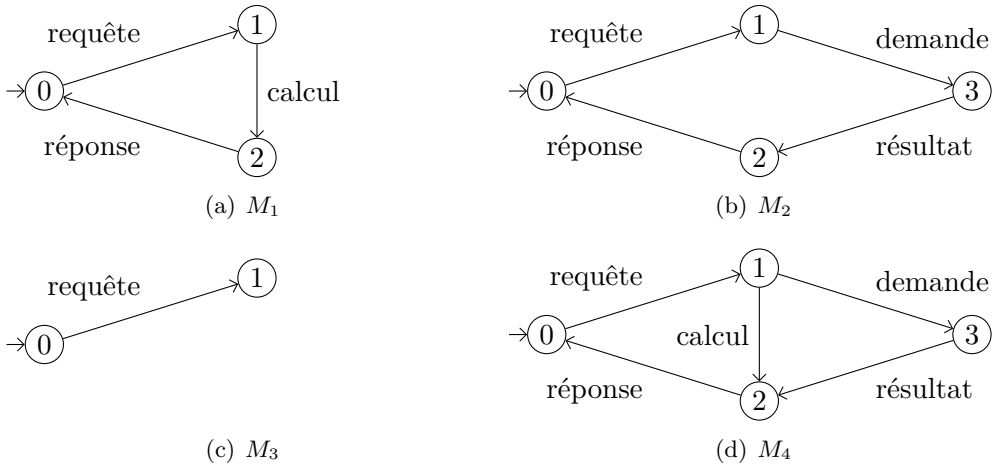
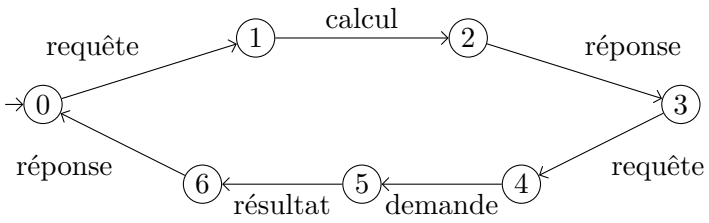


Figure 2.1 : Une spécification modale



(a) M_1 (b) M_2

(c) M_3 (d) M_4



(e) M_5

Figure 2.2 : Des modèles de la spécification modale de la figure 2.1

réaliser la transition « calcul » mais pas la transition « demande », tandis que nous avons fait l'inverse dans M_2. Dans M_3, nous décidons de n'en réaliser aucune et donc de ne rien faire à partir de l'état 1. Enfin, dans M_4, nous implémentons les deux transitions. Ensuite, les transitions « réponse » à partir de l'état 2 et « résultat » à partir de l'état 3 sont toutes deux obligatoires et sont donc réalisées dans tous les modèles où ces états sont atteints. Enfin, M_5 montre que les modèles d'une spécification modale doivent respecter les contraintes exprimées par les deux types de transitions, mais pas la structure de la spécification en elle-même : ils peuvent la déplier afin de dupliquer certains états et faire différents choix d'implémentation. Notons que du fait de cette possibilité de dépliage de l'automate sous-jacent, la spécification a un nombre infini de modèles. Par exemple, nous pouvons construire un ensemble infini contenant les modèles réalisant la transition « calcul » n fois (for tout entier naturel n), puis la transition « demande » une fois (M_5 est un de ces modèles pour $n = 1$).

Les spécifications modales peuvent être basées sur des automates déterministes ou non déterministes. Les contributions de cette thèse étant basées sur des structures déterministes, nous allons maintenant définir formellement la notion d'automate déterministe et de spécification modale déterministe, ainsi que la relation de satisfaction entre une spécification et un de ses modèles. Nous

discuterons du choix d'utiliser des spécifications déterministes dans la section 2.3.

Définition (Automate)**.** *Un automate déterministe sur un alphabet Σ est un triplet (R, r^0, λ) où R est l'ensemble des états, $r^0 \in R$ l'état initial et $\lambda : R \times \Sigma \to R$ la fonction partielle de transition. Nous définissons l'ensemble des actions tirables d'un état r, noté* $\mathrm{ready}(r)$*, comme l'ensemble des actions a telles que $\lambda(r, a)$ est défini.*

Définition (Spécification modale)**.** *Une spécification modale déterministe sur un alphabet Σ est un quintuplet $(Q, q^0, \delta, \mathrm{may}, \mathrm{must})$ où Q est l'ensemble des états, $q^0 \in Q$ l'état initial, $\delta : Q \times \Sigma \to Q$ la fonction partielle de transition et $\mathrm{may}, \mathrm{must} : Q \to 2^\Sigma$ les ensembles de transitions optionnelles et obligatoires.*

Nous définissons également une spécification modale particulière $S_\perp$ qui n'a aucun modèle.

Définition (Satisfaction)**.** *Un automate M est un modèle d'une spécification modale S, noté $M \models S$, si et seulement s'il existe une relation de simulation $\pi \subseteq R \times Q$ telle que $(r^0, q^0) \in \pi$ et pour tout $(r, q) \in \pi$:*

- $\mathrm{must}(q) \subseteq \mathrm{ready}(r) \subseteq \mathrm{may}(q)$ *;*
- *pour tout $a \in \mathrm{ready}(r)$, $(\lambda(r, a), \delta(q, a)) \in \pi$.*

L'ensemble des modèles de S est noté $[\![S]\!]$.

Par exemple, revenons à la spécification de la figure 2.1. L'état initial q^0 est 0 et pour tout état q les transitions dans $\mathrm{may}(q) \setminus \mathrm{must}(q)$ sont représentées par des lignes en pointillés tandis que les transitions dans $\mathrm{may}(q) \cap \mathrm{must}(q)$ sont des lignes pleines. Prenons le modèle M_5 de cette spécification, représenté dans la figure 2.2(e) : la relation de simulation correspondante est $\{(0,0), (1,1), (2,2), (3,0), (4,1), (5,3), (6,2)\}$.

D'après la définition des spécifications modales, il est possible d'avoir des spécifications avec plus de transitions dans must que dans may. Par exemple, prenons la spécification suivante : $(\{0\}, 0, \{(0, a) \mapsto 0, (0, b) \mapsto 0\}, \{0 \mapsto \{a\}\}, \{0 \mapsto \{a, b\}\})$. Elle est composée d'un unique état 0 et de deux transitions vers lui-même étiquetées a et b. La transition a est à la fois dans may(0) et must(0) tandis que b n'appartient qu'à must(0). Essayons de construire un modèle de cette spécification : l'ensemble must requiert que nous réalisions les deux transitions a et b, mais l'ensemble may autorise uniquement a. Aussi, il est impossible de construire un modèle de cette spécification.

Nous définissons une notion de spécification modale *inconsistante* (définition 4) — c'est-à-dire une spécification avec un état q tel que $\mathrm{must}(q) \not\subseteq \mathrm{may}(q)$ ou $\mathrm{ready}(q) \neq \mathrm{may}(q)$ — et montrons que pour toute spécification S inconsistante, il existe une spécification $\rho(S)$ consistante avec le même ensemble de modèles (théorème 1). Aussi, nous pouvons supposer que toute spécification modale est consistante sans perdre de généralité dans nos résultats : si une spécification est inconsistante, il suffit d'appliquer ρ pour obtenir une spécification consistante équivalente. L'avantage d'avoir une opération ρ séparée plutôt que de requérir la consistance directement dans la définition des spécifications modales est que certaines opérations peuvent générer temporairement une spécification inconsistante pour ensuite appliquer ρ afin d'enlever ces inconsistances, plutôt que de devoir générer une spécification consistante d'un seul coup.

Définition (Raffinement modal)**.** *Étant données deux spécifications modales S_1 et S_2, S_1 est un raffinement de S_2, noté $S_1 \leq S_2$, si et seulement s'il existe une relation de simulation $\pi \subseteq Q_1 \times Q_2$ telle que $(q_1^0, q_2^0) \in \pi$ et pour tout $(q_1, q_2) \in \pi$:*

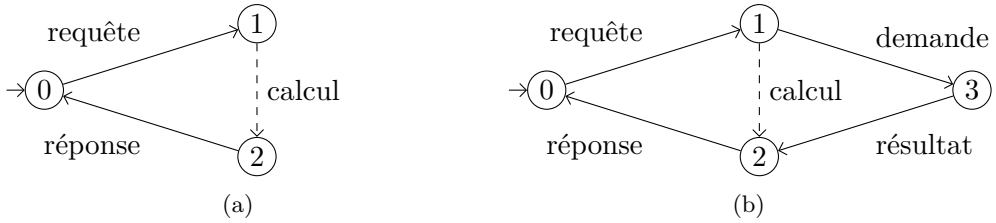


Figure 2.3 : Des raffinements de la spécification modale de la figure 2.1

- $\text{may}(q_1) \subseteq \text{may}(q_2)$;
- $\text{must}(q_2) \subseteq \text{must}(q_1)$;
- *pour tout* $a \in \text{may}(q_1)$, $(\delta(q_1, a), \delta(q_2, a)) \in \pi$.

De plus, pour toute spécification S, $S_\perp \leq S$.

Cette définition du raffinement modal est équivalente au *thorough refinement*, c'est-à-dire à l'inclusion des ensembles de modèles (voir théorème 2, ainsi que [Rac08] pour la preuve). Alors que la plupart des définitions et théorèmes de cette section peuvent être adaptés au cas non déterministe, ce n'est pas le cas de ce théorème. Nous donnerons le contre-exemple dans la section 2.3.

Nous illustrons dans la figure 2.3 deux raffinements possibles de la spécification modale de la figure 2.1. Dans celle de gauche, nous avons enlevé une transition, « demande », de l'ensemble may. Dans celle de droite, nous avons étendu l'ensemble must en lui ajoutant la transition « demande ».

Variantes. Depuis l'introduction des spécifications modales en 1988, de nombreuses variantes ont été développées :

- les *mixed specifications* [DGG97] sont très proches des spécifications modales, mais sans l'hypothèse de consistance : le cas où une transition appartient à l'ensemble must mais pas à l'ensemble may doit être géré explicitement par les différentes opérations ;
- les spécifications modales disjonctives [LX90] permettent d'exprimer, en plus des may et must, des disjonctions de must : au moins une transition dans l'ensemble must disjonctif (d-must) doit être réalisée par les modèles de la spécification ;
- les spécifications modales *one-selecting* [FS08] proposent une disjonction exclusive plutôt que la disjonction inclusive des spécifications modales disjonctives ainsi que des disjonctions exclusives sur les transitions may ;
- les spécifications à ensembles d'acceptation [Rac08] offrent un formalisme encore plus expressif puisqu'il permet d'exprimer des contraintes arbitraires sur les transitions ; ce formalisme étant à la base des contributions de cette thèse, nous le présenteront de manière plus détaillée dans le chapitre 3 ;
- une autre approche consiste à exprimer les contraintes sur les transitions avec des formules logiques plutôt que des ensembles de transitions may/must/d-must/... ; plusieurs formalismes utilisent cette approche :
  - les spécifications modales avec obligations [BK10] utilisent des formules booléennes positives ;

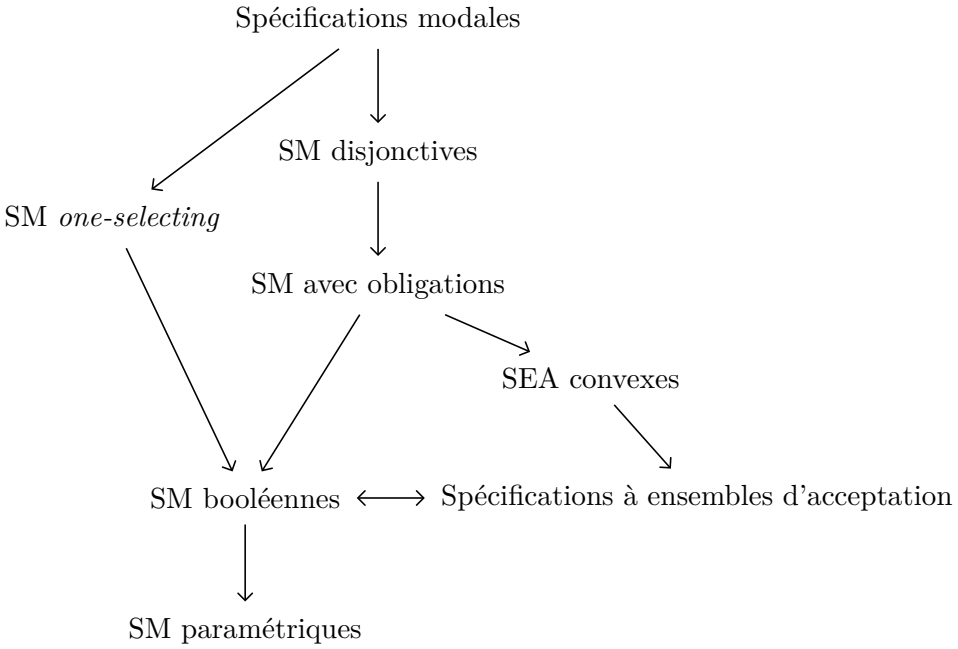


Figure 2.4 : Relations entre les formalismes déterministes

- les spécifications modales booléennes [BKL$^+$11] ajoutent l'opérateur de négation ;
- les spécifications modales paramétriques [BKL$^+$11] ajoutent des paramètres booléens aux spécifications modales booléennes.

Ces diverses extensions des spécifications modales sont plus ou moins expressives. Nous l'illustrons dans la figure 2.4, les spécifications modales (en haut) étant le formalisme le moins expressif. Notons qu'il s'agit de l'expressivité des formalismes déterministes. Dans le cas non déterministe, [BDF$^+$13] montre que les spécifications modales disjonctives sont équivalentes aux spécifications à ensembles d'acceptation. En conséquence, la plupart des formalismes indiqués dans la figure 2.4 sont équivalents dans le cas non déterministe.

Équivalences logiques. Il y a des équivalences entre formalismes de spécification et logiques, c'est-à-dire qu'il existe des constructions permettant de convertir une spécification basée sur un automate en une formule logique caractérisant le même ensemble de modèles et vice versa. Les spécifications modales ont été reliées à la logique de Hennessy-Milner (HML) [HM80] : toute spécification modale a une formule HML équivalente [Lar89] et toute formule HML consistante et première est équivalente à une spécification modale [BL92]. De plus, les spécifications modales disjonctives non déterministes sont équivalentes aux formules HML avec des plus grands points fixes [BDF$^+$13].

Applications. Comme évoqué en introduction, les spécifications modales ont été utilisées comme formalisme de spécification pour la conception modulaire de systèmes via la définition de théories de spécification. Elles ont été utilisées dans différents contextes comme :

- le *model-checking*, avec des structures de Kripke avec modalités dans [BG00], ainsi que dans [CDEG03, HJS01] ;

- la modélisation de lignes de produits logicielles [AtBFG10] ;
- la conception basée sur des contrats [GR09, BDH+12, NITS14].

Extensions. Les spécifications modales ont donné lieu à de nombreuses extensions, notamment :

- avec des notions d'entrées/sorties et de compatibilité d'interfaces en se basant sur l'approche des automates d'interface [dAH01], aussi bien pour les spécifications déterministes [LNW07a, RBB+09, RBB+11] que pour les non déterministes [LV12, BFLV15, CCJK12] ;
- avec des notions de données [BHB10, BHW11, BLL+14] ;
- de diverses manières avec du temps : *timed modal specifications* [ČGL93], *modal event-clock specifications* [BLPR09, BLPR12], *timed I/O modal specifications* [DLL+10b], *time-parametric modal specifications* [KSL13] ;
- avec des propriétés quantitatives : *weighted modal specifications* [BFJ+13] et *label-structured modal specifications* [BJL+12] ;
- avec des probabilités [JL91] ;
- des réseaux de Petri décorés avec des modalités sur les transitions ont été étudiés dans [EHH12, HHM13] ;
- avec des propriétés d'atteignabilité exprimées par des états marqués [CR12] ; nous parlerons de ce formalisme et de notre extension, les spécifications à ensembles d'acceptation marquées, dans le chapitre 4.

2.2 Une théorie de spécification modale

Nous avons présenté dans la première section le formalisme des spécifications modales et sa sémantique avec la définition des relations de satisfaction et de raffinement. Nous définissons maintenant des opérations sur les spécifications modales afin de construire une théorie de spécification modale suivant l'approche de [RBB+11].

De plus, définir une théorie de spécification est la base de la construction de théories basées sur des contrats comme présenté dans [BDH+12]. Dans cet article, il est montré qu'étant donnée une théorie de spécification avec du raffinement et des opérations de produit, conjonction et quotient sur un formalisme $\mathcal{S}$, il est possible d'en déduire une théorie de contrat pour des paires $(\mathcal{A}, \mathcal{G})$ de spécifications de $\mathcal{S}$ avec du raffinement et un produit.

2.2.1 Conjonction

Afin de spécifier un système, il peut être plus facile pour les concepteurs de décrire les différents aspects du système (fonctionnel, sûreté, temporel, consommation de ressources, ...) avec différentes spécifications. Ceci est souvent appelé conception par point de vue (voir [RT14]). Des questions se posent sur ces différents points de vue : sont-ils consistants ou en contradiction les uns avec les autres ? Comment peut-on s'assurer que tous les aspects exprimés seront finalement implémentés ? Une opération de conjonction sur les spécifications permet de caractériser les implémentations communes d'un ensemble de points de vue décrits par des spécifications. En particulier, les

inconsistances entre points de vue peuvent être testées en regardant si leur conjonction a un ensemble de modèles vide.

La définition complète de la conjonction de deux spécifications modales est donnée dans la définition 14. De plus, on peut prouver (théorème 3) que la conjonction caractérise exactement l'intersection des modèles de ses opérandes.

2.2.2 Produit

Nous voulons aussi pouvoir composer des spécifications modales en calculant leur produit, qui renvoie une spécification où leurs actions communes ont été synchronisées. Ceci permet de concevoir des systèmes de bas en haut : nous pouvons partir de composants élémentaires et les composer les uns avec les autres afin d'obtenir un système plus complexe.

La définition complète du produit de deux spécifications modales est donnée dans la définition 15. Le produit de spécifications modales généralise la notion de produit d'automates en caractérisant l'ensemble des produits de modèles des deux spécifications (théorème 4) ; de plus, il en est la caractérisation la plus précise (théorème 5).

2.2.3 Quotient

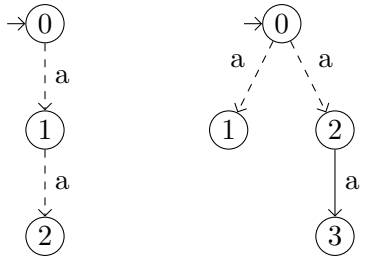
Le produit présenté précédemment offre une approche de bas en haut. D'un autre côté, certains peuvent préférer une approche de haut en bas : étant donné la spécification d'un système souhaité $\mathcal{G}$ et la spécification d'un composant pré-existant $\mathcal{C}$ (venant d'une bibliothèque logicielle par exemple), quelle est la spécification du système $\mathcal{S}$ que nous devrions réaliser de telle sorte que son produit avec $\mathcal{C}$ raffine $\mathcal{G}$? Ceci est donné par le quotient $\mathcal{G}/\mathcal{C}$ que nous considérons ici dans le cas modal (définition 16) en suivant l'approche de [Rac08]. Cette opération est duale du produit (théorème 7).

2.3 Non déterminisme

Les spécifications modales ont été définies aussi bien avec des automates déterministes que non déterministes. L'avantage des spécifications non déterministes est assez clair : elles représentent un sur-ensemble strict des spécifications déterministes et sont donc plus expressives. Cependant, le non déterminisme a aussi des inconvénients.

Le premier problème a été mentionné lorsque nous avons défini la relation de raffinement sur les spécifications modales et avons prouvé que cette relation était équivalente au *thorough refinement* (théorème 2). Ce résultat ne tient pas pour les spécifications non déterministes, comme montré dans [LNW07b]. Ainsi, prenons par exemple les deux spécifications non déterministes de la figure 2.5. Il y a trois choix d'implémentation autorisés par la spécification S : ne réaliser aucune transition à partir de l'état initial, réaliser une seule transition par a ou bien deux transitions consécutives par a. Dans chaque cas, le choix correspondant peut être fait par un modèle de T. Cependant, S ne raffine pas T : en partant de la paire d'états initiaux $(0, 0)$, il y a une transition may par a qui peut aller vers la paire $(1, 1)$ ou la paire $(1, 2)$. Il y a une transition may par a à partir de l'état 1 de S, mais dans le premier cas, elle est interdite par l'état 1 de T et dans le second cas, elle est dans l'ensemble must de l'état 2 de T. D'après [LNW07b], le *thorough refinement* est décidable, donc il est possible de le tester directement plutôt que d'utiliser le raffinement modal, mais l'algorithme est co-NP dur, le rendant inutilisable sur des spécifications de grande taille.

Le second problème avec les spécifications non déterministes est que les opérations sont plus difficiles à définir et ont une complexité plus importante — nous l'avons déjà vu pour le *thorough*



(a) S (b) T

Figure 2.5 : $[\![S]\!] \subseteq [\![T]\!]$ mais $S \not\leq T$

refinement. Prenons par exemple l'opération de quotient. Nous en avons donné une définition pour les spécifications modales déterministes dans la définition 16, d'après celle de [Rac08]. L'espace d'état de ce quotient est $(Q_1 \times Q_2) \cup \{q_\top\}$. À notre connaissance, la première définition du quotient sur des spécifications modales non déterministes a été donnée dans [BDF$^+$13]. L'espace d'état de ce quotient est $2^{Q_1 \times Q_2}$, c'est-à-dire qu'il y a une explosion exponentielle pour le nombre d'états. Les auteurs conjecturent que cette explosion exponentielle ne peut pas être évitée en général. De plus, le quotient de spécifications modales non déterministes n'est pas homogène : le résultat est une spécification modale *disjonctive* non déterministe.

En conséquence, bien que les spécifications non déterministes soient plus expressives, les spécifications déterministes offrent des propriétés intéressantes, comme un quotient homogène et l'équivalence entre raffinement modal et *thorough* ; de plus, les opérations sur ces spécifications sont plus simples à définir et plus efficaces sur de grands systèmes.

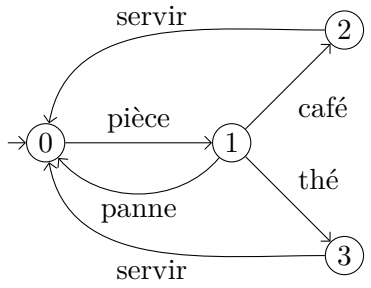
Chapitre 3

Spécifications à ensembles d'acceptation et optimisation convexe

Nous allons maintenant donner une définition plus détaillée des spécifications à ensembles d'acceptation et démontrer que ce formalisme est plus expressif que d'autres extensions des spécifications modales comme les spécifications modales disjonctives ou les spécifications modales avec obligations. Ensuite, nous définissons les opérations de conjonction, produit et quotient sur les spécifications à ensembles d'acceptation. Dans la section 3.3, nous introduisons la première des principales contributions de cette thèse : la définition d'une sous-classe des spécifications à ensembles d'acceptation qui permet de définir des opérations plus efficaces, en particulier pour le quotient, tout en restant plus expressif que les spécifications modales disjonctives ou que les spécifications modales avec obligations. Enfin, nous donnons un aperçu de la mécanisation Coq des différents théorèmes donnés dans cette dernière section.

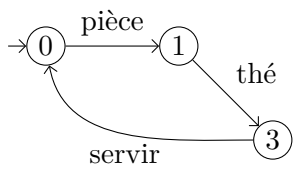
3.1 Sémantique

Les *acceptance trees* ont été introduits dans [Hen85] pour représenter des arbres non déterministes avec une structure déterministe. Une variante de ces arbres adaptée aux automates a été étudiée dans [Rac08] en tant que formalisme de spécification, appelé spécifications à ensembles d'acceptation, qui généralise les spécifications modales. Au lieu d'exprimer deux types de contraintes sur les transitions — qu'elles sont autorisées ou obligatoires — les spécifications à ensembles d'acceptation peuvent exprimer des contraintes arbitraires sur les ensembles de transitions pouvant être réalisés par les implémentations. Notons que les résultats présentés dans cette section et la suivante (section 3.2) sont essentiellement basés sur [Rac08].

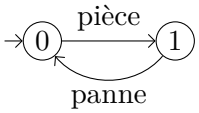


$$\begin{aligned} \mathrm{Acc}(0) &= \{\{\text{pièce}\}\} \\ \mathrm{Acc}(1) &= \{\{\text{café}\}, \{\text{thé}\}, \{\text{café}, \text{thé}\}, \{\text{panne}\}\} \\ \mathrm{Acc}(2) &= \{\{\text{servir}\}\} \\ \mathrm{Acc}(3) &= \{\{\text{servir}\}\} \end{aligned}$$

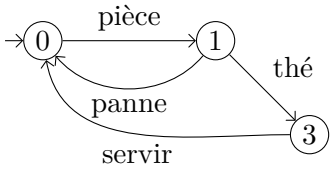
Figure 3.1 : Une spécification d'une machine à café



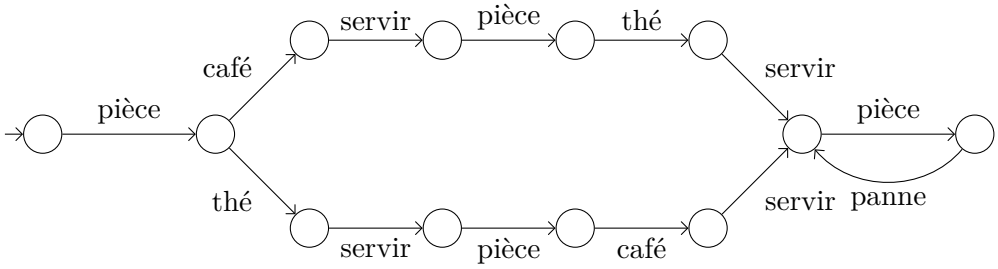
(a) Une machine ne servant que du thé



(b) Une machine cassée



(c) Un modèle incorrect



(d) Machine servant une tasse de thé et une tasse de café

Figure 3.2 : Des modèles de la spécification à ensembles d'acceptation de la figure 3.1

Prenons par exemple la spécification à ensembles d'acceptation de la figure 3.1. Elle spécifie le comportement d'une machine à café qui attend que quelqu'un mette une pièce puis offre du café, du thé, les deux ou indique une erreur. Contrairement aux spécifications modales, il n'y a pas deux types de transitions mais un ensemble associé à chaque état. Pour les états 0, 2 et 3, il y a un unique singleton dans l'ensemble d'acceptation, ce qui est équivalent à une unique transition must. Pour l'état 1 par contre, l'ensemble d'acceptation a quatre éléments ce qui signifie que lorsque l'on implémente la spécification, il faut choisir un de ces quatre ensembles et réaliser toutes les transitions qu'il contient.

Par exemple, lorsque nous avons implémenté le modèle de la figure 3.2(a), nous avons choisi l'ensemble de transitions {thé}, tandis que nous avons pris l'ensemble {panne} quand nous avons implémenté le modèle de la figure 3.2(b). D'un autre côté, l'automate de la figure 3.2(c) n'est pas un modèle de la spécification : depuis l'état 1, il a deux transitions, {thé, panne}, et cet ensemble n'appartient pas à l'ensemble d'acceptation de l'état correspondant dans la spécification.

Il est toujours possible de déplier une spécification quand on l'implémente afin de faire des choix d'implémentation différents dans différents états correspondant au même état de la spécification. Ainsi l'automate de la figure 3.2(d) est un modèle de la spécification qui sert exactement une tasse de thé et une tasse de café, dans n'importe quel ordre : si le café est demandé en premier, il n'offrira ensuite que du thé puis tombera en panne, tandis que si le thé est pris en premier, il n'offrira que du café avant de tomber en panne. L'état 1 de la spécification est implémenté quatre fois dans le modèle, chaque implémentation réalisant un élément différent de l'ensemble d'acceptation correspondant.

La définition formelle des spécifications à ensembles d'acceptation est proche de celle des spécifications modales avec les ensembles may/must remplacés par un ensemble d'acceptation $\mathrm{Acc} : Q \to 2^{2^{\Sigma}}$. La définition complète est donnée dans la définition 17.

Les relations de satisfaction et de raffinement sur les spécifications à ensembles d'acceptation sont également adaptées des définitions sur les spécifications modales ; voir définitions 18 et 19.

Les spécifications à ensembles d'acceptation sont très expressives et en particulier nous prouvons

qu'elles sont plus expressives que :

- les spécifications modales, théorème 9 ;
- les spécifications modales disjonctives, théorème 10 ;
- les spécifications modales avec obligations, théorème 11.

Nous démontrons également que les spécifications modales booléennes sont équivalentes aux spécifications à ensembles d'acceptation (théorèmes 12 et 13).

3.2 Une théorie de spécification avec ensembles d'acceptation

Nous montrons maintenant comment les opérations définies sur les spécifications modales, c'est-à-dire la conjonction, le produit et le quotient, peuvent être étendues aux spécifications à ensembles d'acceptation.

3.2.1 Conjonction

La conjonction de spécifications à ensembles d'acceptation est assez similaire à la conjonction de spécifications modales ; le calcul des ensembles d'acceptation revient à garder les éléments communs aux ensembles d'acceptation des deux opérandes, c'est-à-dire leur intersection :

$$\mathrm{Acc}((q_1, q_2)) = \mathrm{Acc}_1(q_1) \cap \mathrm{Acc}_2(q_2)$$

La définition complète est donnée dans la définition 21 et, comme pour la conjonction de spécifications modales, nous prouvons que l'ensemble des modèles de la conjonction est égal à l'intersection des ensembles des modèles des deux opérandes (théorème 16).

3.2.2 Produit

Le produit de spécifications à ensembles d'acceptation est également calculé de manière similaire au produit de spécifications modales ; les ensembles d'acceptation sont construits à partir des intersections des éléments des ensembles d'acceptation des opérandes, ce qui correspond à la définition du produit d'automates :

$$\mathrm{Acc}((q_1, q_2)) = \{A_1 \cap A_2 \mid A_1 \in \mathrm{Acc}_1(q_1) \wedge A_2 \in \mathrm{Acc}_2(q_2)\}$$

La définition complète est donnée dans la définition 22 et nous prouvons les mêmes théorèmes que sur les spécifications modales (théorèmes 17 et 18 notamment).

3.2.3 Quotient

Comme pour les spécifications modales, le quotient de spécifications à ensembles d'acceptation est censé être la fonction inverse du produit. Puisque les ensembles d'acceptation du produit sont les intersections des éléments des ensembles d'acceptation des opérandes, les ensembles d'acceptation du quotient sont tous les ensembles dont l'intersection avec les éléments du dénominateur appartient au numérateur :

$$\mathrm{Acc}((q_1, q_2)) = \{X \mid \forall X_2 \in \mathrm{Acc}_2(q_2), X \cap X_2 \in \mathrm{Acc}_1(q_1)\}$$

La définition complète est donnée dans la définition 23 et nous prouvons qu'il s'agit bien de l'inverse du produit dans le théorème 22.

3.2.4 Alphabets dissemblables

Jusqu'à présent, nous n'avons considéré que des spécifications définies sur un même alphabet Σ. Lorsque que l'on conçoit un système complexe à partir de nombreux composants, ces composants ne sont typiquement pas définis sur le même alphabet : chacun ne gère qu'un petit nombre d'actions liées à la tâche qu'il effectue. Ensuite, nous voulons pouvoir fusionner ou composer ces différents sous-systèmes afin de construire des systèmes plus complexes, ce qui nécessite de pouvoir adapter les opérations définies précédemment de telle sorte qu'elles puissent gérer correctement les différences entre les alphabets de leurs opérandes. Une manière de résoudre ce problème, présentée pour les spécifications modales dans [RBB+11], est de commencer par *étendre* chaque spécification afin que les opérandes d'une opération soient définis sur le même alphabet. Ceci permet de définir uniquement des fonctions d'extension d'alphabet et ensuite de réutiliser les opérations définies précédemment, plutôt que de devoir réimplémenter toutes ces opérations afin qu'elles gèrent les différences d'alphabet de manière interne.

Supposons que l'on ait deux alphabets Σ et Σ' tels que $\Sigma \subseteq \Sigma'$ ainsi qu'une spécification à ensembles d'acceptation S définie sur Σ. Comment pouvons-nous étendre S afin qu'elle soit définie sur Σ' ? L'idée principale est d'ajouter des boucles de chaque état vers lui-même avec comme étiquettes les actions de $\Sigma' \setminus \Sigma$. Ensuite, ces transitions permettront de se synchroniser avec d'autres spécifications tout en préservant le comportement de la spécification originale, puisque ces transitions restent dans le même état. Nous devons également étendre les ensembles d'acceptation en conséquence, sans quoi les spécifications obtenues par extension seraient inconsistantes. Il y a différentes manières d'ajouter les actions aux ensembles d'acceptation. Une première méthode consiste à simplement ajouter les actions à chaque élément des ensembles d'acceptation :

$$\mathrm{Acc}'(q) = \{X \cup (\Sigma' \setminus \Sigma) \mid X \in \mathrm{Acc}(q)\}$$

Ceci est appelé *extension forte* car tous les modèles de la nouvelle spécification doivent obligatoirement réaliser les transitions de $\Sigma' \setminus \Sigma$. Une autre méthode consiste à seulement autoriser les transitions, qui peuvent alors être réalisées ou non par les implémentations :

$$\mathrm{Acc}'(q) = \{X \cup \sigma \mid X \in \mathrm{Acc}(q) \wedge \sigma \subseteq \Sigma' \setminus \Sigma\}$$

Nous appelons ceci *extension faible*. Ces deux extensions différentes sont toutes deux utiles : selon les cas et les opérations, nous aurons parfois besoin de l'extension forte et parfois de la faible.

Les définitions complètes de ces deux extensions sont données dans la définition 25.

Afin de s'assurer de la correction de ces opérations, il faut également définir une notion d'extension d'alphabet sur les automates (définition 24) ainsi que des relations de satisfaction et raffinement faibles et forts selon l'extension utilisée (définitions 26 et 27).

Enfin, nous montrons comment adapter des opérations sur des spécification définies sur des alphabets dissemblables à l'aide de ces opérations d'extension :

- la conjonction en faisant d'abord l'extension faible des deux opérandes (théorème 28) ;
- le produit en faisant d'abord l'extension forte des deux opérandes (théorèmes 29 et 30) ;
- le quotient en faisant d'abord l'extension faible du numérateur et l'extension forte du dénominateur (théorème 31).

3.3 Spécifications à ensembles d'acceptation convexes

Nous introduisons maintenant la première contribution majeure de cette thèse. Nous avons vu que les spécifications à ensembles d'acceptation sont très expressives par rapport aux spécifications modales ou disjonctives, mais cette expressivité a un coût en termes de complexité : convertir une spécification modale ou disjonctive en spécification à ensembles d'acceptation, ou calculer un quotient, provoque une explosion exponentielle par rapport à la taille de l'alphabet. Afin de limiter cette augmentation de complexité tout en restant très expressif, nous introduisons une sous-classe optimisée des ensembles d'acceptation appelée ensembles d'acceptation convexes-clos. Ces ensembles, bien que moins expressifs que les ensembles d'acceptation, sont encore suffisamment expressifs pour représenter les contraintes exprimées par des spécifications modales ou disjonctives tout en évitant les explosions exponentielles des opérations sur les ensembles d'acceptation.

Nous allons commencer par montrer comment ces ensembles sont représentés, puis nous verrons comment utiliser l'hypothèse de convexité pour optimiser différentes opérations sur les ensembles d'acceptation. Comme beaucoup de preuves sont assez techniques et font appel à diverses opérations ensemblistes, nous avons prouvé les théorèmes sur les ensembles convexes-clos avec l'assistant de preuve Coq.

3.3.1 Sémantique

Nous commençons par définir la sous-classe des *ensembles d'acceptation convexes-clos* :

Définition (Ensemble convexe-clos)**.** *Un ensemble d'acceptation est dit* convexe-clos *si pour tout* $X, Y \in \mathrm{Acc}$ *et* Z *tel que* $X \subseteq Z \subseteq Y$, $Z \in \mathrm{Acc}$.

Alors, étant donné un ensemble d'acceptation convexe-clos, on peut le représenter de manière optimisée. Au lieu de garder tous ses éléments, il suffit d'avoir ses éléments minimaux et maximaux (par inclusion, voir définition 29) : nous savons alors que tous les ensembles compris entre eux appartiennent aussi à l'ensemble (théorème 32).

La relation de raffinement teste l'inclusion des ensembles d'acceptation. Nous prouvons qu'il est possible de décider de l'inclusion d'ensembles convexes-clos à partir de leurs éléments minimaux et maximaux (théorème 35).

Dans la section 3.1, nous avons vu que les spécifications à ensembles d'acceptation étaient plus expressives que divers autres formalismes. Nous réexaminons ces différents formalismes en les comparant maintenant aux spécifications à ensembles d'acceptation convexes. Celles-ci sont plus expressives que :

- les spécifications modales, théorème 37 ;
- les spécifications modales disjonctives, théorème 38 ;
- les spécifications modales avec obligations, théorème 39.

Par contre, les spécifications modales booléennes étant équivalentes aux spécifications à ensembles d'acceptation, elles ne peuvent pas toutes être représentées avec des ensembles d'acceptation convexes-clos.

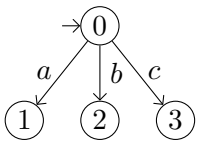
3.3.2 Conjonction

Lorsque l'on calcule la conjonction de spécifications à ensembles d'acceptation, la seule opération appliquée aux ensembles d'acceptation est l'intersection. Nous prouvons d'abord que la convexité

est préservée par intersection (proposition 2) puis que l'on peut calculer les éléments minimaux et maximaux de l'intersection directement à partir des éléments minimaux et maximaux de ses opérandes (théorème 40).

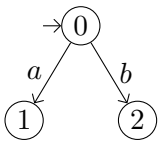
3.3.3 Produit

L'opération de produit représente la principale limitation des ensembles convexes-clos, puisque l'opération appliquée aux ensembles d'acceptation durant le calcul d'un produit ne préserve pas la convexité.



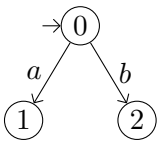
$\mathrm{Acc}(0) = \{\{a, b\}, \{c\}\}$
$\mathrm{Acc}(1) = \mathrm{Acc}(2) = \mathrm{Acc}(3) = \{\emptyset\}$

(a) S_1



$\mathrm{Acc}(0) = \{\{a, b\}\}$
$\mathrm{Acc}(1) = \mathrm{Acc}(2) = \{\emptyset\}$

(b) S_2



$\mathrm{Acc}(0) = \{\emptyset, \{a, b\}\}$
$\mathrm{Acc}(1) = \mathrm{Acc}(2) = \{\emptyset\}$

(c) $S_1 \otimes S_2$

Figure 3.3 : La convexité n'est pas préservée par le produit

Observons les spécifications des figures 3.3(a) et 3.3(b) et leur produit, figure 3.3(c). Les ensembles d'acceptation des deux spécifications sont clairement convexes-clos. Par contre, l'ensemble d'acceptation de l'état initial de leur produit ne l'est pas. Prenons par exemple l'ensemble $\{a\}$: il n'appartient pas à l'ensemble d'acceptation $\{\emptyset, \{a, b\}\}$ alors que $\emptyset \subseteq \{a\} \subseteq \{a, b\}$.

Bien que l'on ne puisse pas définir un produit d'ensembles convexes-clos renvoyant un ensemble convexe-clos, nous pouvons tout de même exploiter l'hypothèse de convexité pour améliorer le calcul de l'ensemble d'acceptation du produit, qui peut ne pas être convexe-clos (théorème 41).

3.3.4 Quotient

Le quotient est probablement l'opération qui gagnera le plus à utiliser des ensembles convexes-clos, puisque l'ensemble d'acceptation calculé par le quotient :

$$\mathrm{Acc}_1 \,/\, \mathrm{Acc}_2 = \{X \mid \forall X_2 \in \mathrm{Acc}_2, X \cap X_2 \in \mathrm{Acc}_1\}$$

a une explosion exponentielle. En effet, il faut énumérer tous les ensembles X d'actions et tester si leur intersection avec les éléments de Acc_2 est dans Acc_1, ce qui donne une complexité de $O(2^{|\Sigma|} \times |\,\mathrm{Acc}_2\,| \times |\,\mathrm{Acc}_1\,|)$.

Nous prouvons que si Acc_1 et Acc_2 sont convexes-clos, l'ensemble d'acceptation obtenu par quotient l'est aussi (proposition 3) puis nous définissons une opération calculant les éléments minimaux et maximaux de l'ensemble d'acceptation du quotient à partir des éléments minimaux et maximaux de Acc_1 et Acc_2 (théorème 42).

3.3.5 Alphabets dissemblables

Enfin, nous étudions les opérations d'extension d'alphabet et prouvons qu'elles préservent la convexité (proposition 6) et que les éléments minimaux et maximaux des ensembles d'acceptation

étendus peuvent être calculés directement à partir des éléments minimaux et maximaux de l'ensemble d'acceptation initial (théorème 43).

3.3.6 Mécanisation Coq

Les preuves des théorèmes des sections précédentes, en particulier celles sur le quotient, sont assez complexes. Cependant, elles ne font appel qu'à des concepts assez simples de théorie des ensembles. Aussi, nous avons souhaité utiliser des techniques de preuve assistée par ordinateur pour s'assurer de la validité de nos résultats. Pour cela, nous avons choisi d'utiliser l'assistant de preuves Coq, avec la bibliothèque MSet pour la représentation des ensembles. Nous avons défini en Coq les notions d'ensemble d'acceptation et d'ensemble d'acceptation convexe-clos, défini les opérations de conjonction, produit, quotient ainsi que les extensions faible et forte sur ces deux formalismes et prouvé les différents théorèmes des sections 3.3.1 à 3.3.5.

3.3.7 Non déterminisme

Dans les sections précédentes, nous n'avons étudié que des spécifications *déterministes*. Il y a aussi une théorie de spécification non déterministe basée sur des ensembles d'acceptation [BDF$^+$13, BFK$^+$14] ; il pourrait donc être intéressant de voir si l'utilisation d'ensembles convexes-clos pourrait également améliorer l'efficacité des opérations dans le cas non déterministe.

Tout d'abord, alors que les spécifications modales disjonctives, les spécifications modales avec obligations, les spécifications à ensembles d'acceptation convexes et les spécifications à ensembles d'acceptation ont une expressivité strictement croissante dans le cas déterministe, l'ajout du non déterminisme met à plat cette hiérarchie : [BDF$^+$13, BFK$^+$14] prouve qu'avec le non déterminisme, les spécifications modales disjonctives sont équivalentes aux spécifications à ensembles d'acceptation. Nous pouvons donc conjecturer qu'une théorie de spécification basée sur des ensembles convexes-clos serait aussi expressive que les spécifications à ensembles d'acceptation (et que les spécifications modales disjonctives).

De plus, les traductions entre spécifications modales disjonctives et spécifications à ensembles d'acceptation causent des explosions exponentielles dans les deux sens [BFK$^+$14]. En conséquence, certaines opérations, bien que possibles en théorie, sont inutilisables en pratique. Par exemple, [BDF$^+$13, BFK$^+$14] définit un quotient sur les spécifications à ensembles d'acceptation, mais pas sur les spécifications modales disjonctives ; le quotient de deux spécifications modales disjonctives peut être calculé en les traduisant en spécifications à ensembles d'acceptation, puis en appliquant le quotient et enfin en traduisant le résultat en spécification modale disjonctive — chaque étape ayant une explosion exponentielle. D'un autre côté, une théorie de spécification basée sur des ensembles convexes-clos pourrait offrir des opérations avec une complexité moindre. Nous savons que dans le cas déterministe, la traduction d'une spécification modale disjonctive en spécification à ensembles d'acceptation convexes n'a pas d'explosion exponentielle alors qu'il y en a une dans la traduction en spécification à ensembles d'acceptation ; nous conjecturons qu'il y a un gain similaire dans le cas non déterministe.

Cependant, l'ajout de non déterminisme à des ensembles d'acceptation convexes-clos n'est pas une extension simple et directe de nos résultats sur les spécifications déterministes. En particulier, il faudrait adapter la notion de convexité sur les ensembles puisque les ensembles d'acceptation des spécifications non déterministes ne contiennent pas juste des actions, mais des paires contenant l'action et l'état destination de la transition correspondante. De plus, les opérations sur les spécifications non déterministes ont des définitions bien plus complexes. Par exemple, alors

que l'ensemble d'acceptation d'un état (q_1, q_2) du quotient de deux spécifications à ensembles d'acceptation déterministes est donné par une seule formule $(\{X \mid \forall X_2 \in \mathrm{Acc}_2(q_2), X \cap X_2 \in \mathrm{Acc}_1(q_1)\})$, le quotient de [BDF$^+$13, BFK$^+$14] part de la même idée, mais avec un algorithme plus complexe qui requiert plusieurs définitions intermédiaires (voir les objets α, γ, π_a, pt_a, pt dans leur article). S'assurer qu'une telle opération préserve la convexité, puis trouver un algorithme optimisé n'utilisant que les éléments minimaux et maximaux d'un ensemble d'acceptation convexe-clos pour obtenir le même résultat semble être une tâche assez difficile.

Chapitre 4

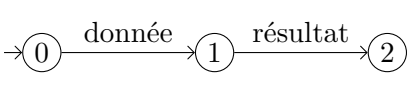
Spécifications à ensembles d'acceptation marquées

Nous présentons maintenant la seconde principale contribution théorique de cette thèse : une extension des spécifications à ensembles d'acceptation permettant d'exprimer des propriétés d'atteignabilité. Nous commençons par donner la sémantique de ce nouveau formalisme puis nous montrons comment étendre les opérations de conjonction, produit et quotient sur les spécifications à ensembles d'acceptation à ce nouveau formalisme tout en préservant les propriétés d'atteignabilité, c'est-à-dire l'absence d'interblocages. Ce formalisme et l'opération de quotient, qui est la plus difficile à définir, ont été présentés dans [VR15b].

4.1 Sémantique

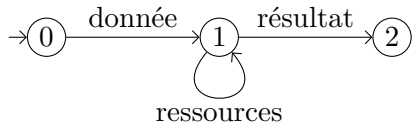
Les formalismes que nous avons étudiés jusqu'à présent — spécifications modales ou à ensembles d'acceptation, convexes ou non — expriment tous des propriétés locales : dans chaque état de la spécification, nous indiquons quelles transitions ou quels groupes de transitions sont requis, autorisés ou interdits. Mais nous pourrions vouloir exprimer des contraintes non pas juste sur les transitions partant de chaque état, mais globalement sur les chemins de chaque modèle.

Prenons par exemple la spécification à ensembles d'acceptation donnée dans la figure 4.1 : il s'agit d'un serveur recevant une donnée, calculant une valeur à partir de cette donnée et la retournant. On peut imaginer que ce serveur puisse avoir besoin de davantage de ressources ; par exemple, si la donnée en entrée est trop grande, il peut avoir besoin de plus de mémoire. Une manière d'exprimer cela est d'ajouter une transition optionnelle à partir de l'état 1 permettant de demander davantage de ressources, comme présenté dans la figure 4.2. Ceci autorise des modèles comme celui de la figure 4.3(a) qui demande des ressources supplémentaires avant de calculer le résultat. Cependant, les modèles peuvent aussi demander des ressources, peut-être même de manière infinie, sans jamais les utiliser, comme illustré dans la figure 4.3(b). Nous pourrions essayer de changer l'ensemble d'acceptation de l'état 1 de la spécification, mais nous ne pourrons jamais



$\mathrm{Acc}(0) = \{\{\text{donnée}\}\}$
$\mathrm{Acc}(1) = \{\{\text{résultat}\}\}$
$\mathrm{Acc}(2) = \{\emptyset\}$

Figure 4.1 : Un serveur simpliste calculant une valeur à partir d'une donnée en entrée



$\mathrm{Acc}(0) = \{\{\text{donnée}\}\}$
$\mathrm{Acc}(1) = \{\{\text{résultat}\}, \{\text{ressources}\}, \{\text{résultat}, \text{ressources}\}\}$
$\mathrm{Acc}(2) = \{\emptyset\}$

Figure 4.2 : Le serveur, autorisé à demander des ressources supplémentaires

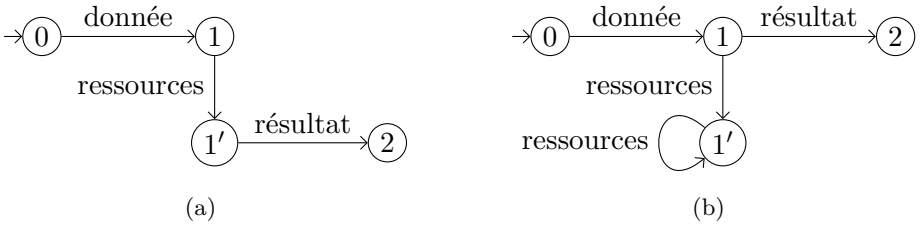


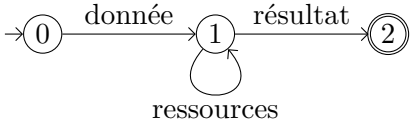
Figure 4.3 : Modèles de la spécification de la figure 4.2

autoriser les modèles à demander des ressources supplémentaires un nombre quelconque de fois tout en les forçant à finir par envoyer un résultat.

Pour exprimer ce type d'exigences, il faut pouvoir exprimer des contraintes non pas sur les transitions d'un état particulier, mais sur les chemins. Une manière de faire cela est d'étendre le formalisme de spécification avec des états marqués. Alors, les modèles doivent avoir un état marqué atteignable à partir de tout état. Une extension des spécifications modales avec des états marqués a été introduite dans [CR12]. Nous combinons les spécifications à ensembles d'acceptation avec des états marqués afin de former des spécifications à ensembles d'acceptation marquées. Pour notre exemple, nous pouvons utiliser une spécification à ensembles d'acceptation marquée et marquer le dernier état, comme indiqué dans la figure 4.4 (les états marqués sont entourés deux fois), ce qui garantit qu'il sera finalement atteint dans tous les modèles de la spécification. Ainsi, l'automate de la figure 4.3(a) est un modèle de cette spécification à ensembles d'acceptation marquée car il satisfait les contraintes de la spécification à ensembles d'acceptation sous-jacente et l'état marqué 2 est atteignable depuis n'importe quel état. D'un autre côté, l'automate de la figure 4.3(b) n'est pas un modèle de la spécification puisqu'il n'est pas possible d'atteindre l'état 2 à partir de $1'$.

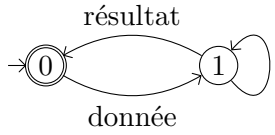
Dans cet exemple, l'ajout de la notion d'état marqué nous a permis d'exprimer une propriété de terminaison : l'état final 2 doit être atteignable à partir de tout autre état. Nous pouvons aussi utiliser les états marqués pour exprimer des propriétés de vivacité. Nous pourrions par exemple modifier notre serveur pour lui permettre de répondre à plusieurs requêtes, comme indiqué dans la figure 4.5 : l'état initial marqué correspond à un point devant être atteignable infiniment souvent.

La définition des spécifications à ensembles d'acceptation marquées est une simple extension des spécifications à ensembles d'acceptation définies dans le chapitre précédent : nous ajoutons seulement un ensemble d'états marqués $F \subseteq Q$ (définition 32). De même, nous étendons les automates avec des états marqués (définition 33). Un automate marqué est dit *terminant* si pour



$\mathrm{Acc}(0) = \{\{\text{donnée}\}\}$
$\mathrm{Acc}(1) = \{\{\text{résultat}\}, \{\text{ressources}\}, \{\text{résultat}, \text{ressources}\}\}$
$\mathrm{Acc}(2) = \{\emptyset\}$

Figure 4.4 : Le serveur avec un état marqué



$\mathrm{Acc}(0) = \{\{\text{donnée}\}, \emptyset\}$
$\mathrm{Acc}(1) = \{\{\text{résultat}\}, \{\text{ressources}\}, \{\text{résultat}, \text{ressources}\}\}$

Figure 4.5 : Le serveur pouvant répondre à plusieurs requêtes

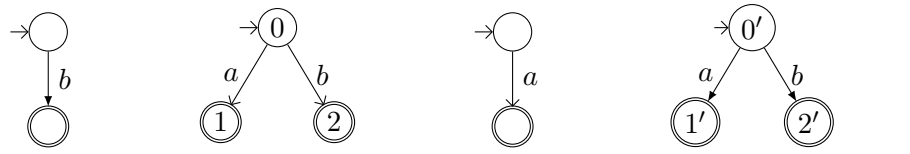
tout état r, il y a un état marqué parmi ses successeurs (définition 35). Alors, un automate marqué est modèle d'une spécification à ensembles d'acceptation marquée si et seulement s'il est terminant, est modèle de la spécification à ensembles d'acceptation sous-jacente et que ses états marqués correspondent à des états marqués dans la spécification (définition 36). De manière similaire, nous étendons la notion de raffinement (définition 37) et prouvons qu'il est *thorough* (théorème 44).

4.2 Conjonction

L'opération de conjonction sur les spécifications à ensembles d'acceptation marquées est une extension directe de l'opération de conjonction sur les spécifications à ensembles d'acceptation avec comme ensemble d'états marqués le produit cartésien des ensembles d'états marqués des spécifications (définition 38). L'ensemble des modèles de cette conjonction est bien l'intersection des ensembles de modèles des spécifications (théorème 46).

4.3 Produit

Nous avons défini le produit de spécifications à ensembles d'acceptation et souhaiterions l'étendre aux spécifications à ensembles d'acceptation marquées. Cependant, les contraintes d'atteignabilité ne sont, de manière générale, pas préservées par le produit. Ainsi, la figure 4.6 montre un contre-exemple : $M_1 \models S_1$ et $M_2 \models S_2$, mais le produit $M_1 \times M_2$ est composé d'un unique état non marqué, ce qui fait qu'il n'est pas possible d'atteindre un état marqué.



$\mathrm{Acc}(0) = \{\{b\}, \{a, b\}\}$
$\mathrm{Acc}(1) = \mathrm{Acc}(2) = \{\emptyset\}$

$\mathrm{Acc}(0') = \{\{a\}, \{a, b\}\}$
$\mathrm{Acc}(1') = \mathrm{Acc}(2') = \{\emptyset\}$

(a) M_1 (b) S_1 (c) M_2 (d) S_2

Figure 4.6 : L'atteignabilité n'est pas préservée par produit

Ceci nous amène à d'abord considérer le problème suivant : étant données deux spécifications à ensembles d'acceptation marquées, peuvent-elles être implémentées de manière concurrente, c'est-à-dire de telle sorte que le produit de n'importe quel modèle de la première spécification avec n'importe quel modèle de la seconde terminera ?

Un automate ne termine pas s'il contient un *deadlock* — un état non marqué sans transition sortante — ou un *livelock* — un groupe d'états connectés non marqués sans aucune transition vers d'autres états. Nous considérerons d'abord le cas des produits sans *deadlocks* dans la prochaine

section, puis le cas des produits sans *livelocks* dans la suivante. Nous définirons ensuite un critère sur les spécifications à ensembles d'acceptation marquées, appelé *atteignabilité compatible*, qui est un prérequis pour le produit de spécifications à ensembles d'acceptation marquées.

4.3.1 Spécifications sans *deadlocks*

Nous définissons un critère permettant de tester à partir de leurs ensembles d'acceptation si deux spécifications à ensembles d'acceptation marquées n'ont pas de modèles dont le produit a un *deadlock* (définition 42) et nous prouvons qu'il est correct (théorème 47).

4.3.2 Spécifications sans *livelocks*

Nous proposons ensuite un critère permettant de tester si deux spécifications à ensembles d'acceptation marquées ont des modèles dont le produit a un *livelock*. Ce test se base sur l'identification de cycles partagés entre les spécifications associé à un typage des transitions sortant de ces cycles. Nous testons ensuite s'il est toujours possible de quitter chaque cycle, indépendamment des choix d'implémentation pouvant être faits.

Ce critère est assez complexe et se compose essentiellement des étapes suivantes :

- un *dépliage* des deux spécifications de telle sorte que dans le produit de leurs automates sous-jacents, chaque état d'une spécification ne soit associé qu'à un seul état de l'autre spécification (voir principalement définition 43 et théorème 48) ;
- une analyse de chaque spécification pour déterminer ses cycles (algorithme 3, définition 47 et théorème 50), avec une difficulté supplémentaire due au fait que l'on ne souhaite garder que les cycles *implémentables*, c'est-à-dire réalisables par une implémentation de la spécification, ce qui n'est pas le cas de tous les cycles à cause des contraintes d'atteignabilité (voir l'exemple figure 4.10) ;
- un typage des transitions sortantes de chaque cycle (algorithme 4) indiquant si l'ensemble d'acceptation requiert que la transition soit toujours réalisée quand le cycle est implémenté ou si elle est optionnelle et peut donc être réalisée ou non selon les choix d'implémentation de chaque modèle ;
- le critère proprement dit (définitions 48 et 49 et théorème 51) qui cherche les cycles des spécifications pouvant se synchroniser et étudie si le typage de leurs transitions sortantes garantit qu'une de ces transitions sera toujours réalisée et donc que le produit des cycles ne générera pas de *livelock*.

4.3.3 Atteignabilité compatible

Nous pouvons ensuite combiner les deux tests pour s'assurer de l'absence de *deadlocks* et de *livelocks* dans les produits des modèles de deux spécifications (définition 50 et théorème 52).

4.3.4 Définition du produit

Enfin, étant données deux spécifications à ensembles d'acceptation marquées avec une atteignabilité compatible, nous pouvons calculer leur produit qui est une simple extension du produit sur les spécifications à ensembles d'acceptation non marquées (définition 51 et théorèmes 53 et 54).

4.4 Quotient

Dans cette section, nous étudions l'extension de l'opération de quotient sur les spécifications à ensembles d'acceptation aux spécifications à ensembles d'acceptation marquées afin de permettre la conception incrémentale avec des propriétés d'atteignabilité.

4.4.1 Pré-quotient

Nous commençons par définir une opération appelée *pré-quotient*. Étant données deux spécifications à ensembles d'acceptation marquées S_1 et S_2, le pré-quotient retourne une spécification à ensembles d'acceptation marquée $S_1 /\!\!/ S_2$ telle que le produit de n'importe lequel de ses modèles avec un modèle de S_2 est un automate qui satisfait S_1 mais sans garantir la condition de terminaison. Une autre opération, définie dans les deux sections suivantes, sera ensuite utilisée pour définir un quotient garantissant la terminaison.

Le pré-quotient est essentiellement une extension du quotient sur les spécifications à ensembles d'acceptation où une paire d'états (q_1, q_2) est marquée si q_1 est marqué ou si q_2 n'est pas marqué (définition 52). Nous prouvons que ce pré-quotient est correct modulo une hypothèse supplémentaire de terminaison (théorème 55). En général, nous voulons également prouver que la spécification retournée par le quotient est complète, c'est-à-dire qu'elle caractérise tous les automates dont le produit avec un modèle de S_2 est un modèle de S_1. Cependant, ceci peut conduire à des spécifications de très grande taille puisque le quotient S_1/S_2 doit alors inclure toutes les transitions qui ne sont pas tirables dans S_2 (et qui sont donc supprimées dans le produit des modèles). Nous proposons de retourner un quotient plus compact sans les transitions *non indispensables* par rapport à S_2, c'est-à-dire sans les transitions qui seront coupées par le produit avec n'importe quel modèle de S_2. Alors, la complétude de ce quotient revient à garantir que n'importe quel automate dont le produit avec n'importe quel modèle de S_2 est un modèle de S_1 est un modèle de S_1/S_2 après la suppression de ses transitions inutiles par rapport à S_2. Nous définissons cette notion de transitions non indispensables (définition 53) et prouvons la complétude du pré-quotient (théorème 57 et corollaire 4).

Ce pré-quotient retourne une spécification $S_1 /\!\!/ S_2$ qui peut ne pas avoir d'atteignabilité compatible avec S_2. Nous étudions dans les deux sections suivantes comment raffiner ce pré-quotient afin de garantir l'atteignabilité compatible. En fait, nous étudions un problème un peu plus général : étant données deux spécifications à ensembles d'acceptation marquées S_1 et S_2, pouvons-nous raffiner S_1 en une spécification S'_1 de telle sorte que S'_1 ait une atteignabilité compatible avec S_2 ?

4.4.2 Correction des *deadlocks*

Nous proposons un algorithme éliminant toutes les paires d'états de S_1 pouvant provoquer un *deadlock* avec S_2 (algorithme 5) et prouvons que la spécification retournée caractérise exactement l'ensemble des modèles de S_1 qui n'ont pas de *deadlock* avec un modèle de S_2 (théorème 58).

4.4.3 Correction des *livelocks*

Ensuite, étant données deux spécifications à ensembles d'acceptation marquées sans *deadlocks* S_1 et S_2, nous raffinons S_1 en une spécification S'_1 sans *livelocks* avec S_2. Afin d'éviter les potentiels *livelocks* entre les modèles de ces deux spécifications, nous utilisons deux méthodes : enlever des transitions afin que les états à partir desquels il n'est pas possible de garantir la terminaison ne soient pas atteints et forcer certaines transitions à être finalement prises afin de garantir qu'il

soit toujours possible de quitter les cycles sans états marqués. Pour cette dernière méthode, nous introduisons les spécifications à ensembles d'acceptation marquées avec priorités qui sont des spécifications à ensembles d'acceptation marquées dans lesquelles nous identifions des transitions avec des *priorités* ; dans la relation de satisfaction, nous ajoutons une contrainte exprimant que ces transitions doivent finir par être implémentées (définitions 55 et 56).

Nous proposons ensuite un algorithme (découpé en deux parties, voir algorithmes 6 et 7) qui raffine S_1 pour éviter les *livelocks* possibles avec S_2 et prouvons qu'il est correct et complet (théorème 59).

Les algorithmes de cette section et de la précédente peuvent enfin être combinés afin de définir une opération qui, étant données deux spécifications à ensembles d'acceptation marquées S_1 et S_2, renvoie un raffinement de S_1 qui caractérise précisément l'ensemble des modèles de S_1 dont le produit avec des modèles de S_2 termine (théorème 60).

4.4.4 Définition du quotient

Nous pouvons maintenant utiliser les opérations définies dans les sections précédentes pour définir le quotient de deux spécifications à ensembles d'acceptation marquées (définition 57) et prouver qu'il est correct (théorème 61) et complet (théorème 62).

Chapitre 5

Implémentation

En plus des résultats théoriques présentés dans les chapitres précédents, nous avons implémenté ces nouveaux formalismes dans un outil appelé MAccS [VR15a]. Nous commençons par donner un aperçu de cet outil, puis nous présentons un état de l'art des outils permettant de manipuler des formalismes similaires et enfin, nous montrons des résultats expérimentaux illustrant le gain de performances offert par les ensembles convexes clos, ainsi qu'une comparaison de l'efficacité de différentes structures de données pour représenter ces ensembles d'acceptation.

5.1 Présentation

L'outil MAccS (abréviation de « Marked Acceptance Specifications », bien qu'il gère désormais d'autres formalismes de spécification en plus des spécifications à ensembles d'acceptation marquées) implémente les théories de spécification décritent dans cette thèse ainsi que des formalismes existants (comme les spécifications modales) pour servir de référence dans des tests de performance. Il est écrit en C++ et est fourni à la fois sous forme de bibliothèque pour l'intégrer dans d'autres programmes et avec une interface graphique permettant de manipuler facilement des spécifications et de leur appliquer diverses opérations.

Les graphes sous-jacents aux automates et aux spécifications sont représentés à l'aide de la bibliothèque de graphes de Boost [SLL02]. L'interface graphique est faite avec Qt et Dot [GN00] est utilisé pour le calcul de la position des états et transitions dans le rendu. Une capture d'écran est montrée dans la figure 5.1.

Les automates et spécifications peuvent être créés interactivement avec l'interface graphique ou écrits dans un format textuel simple. Un extrait de la représentation dans ce format de la spécification de la figure 5.1 est indiqué ci-dessous. Il est aussi possible d'importer et exporter des automates et spécifications depuis et vers le format Dot [GN00].

```
init {{login}}
read {{read,post},{read,post,logout}}
...
end marked {{}}
init -login-> read
read -read-> reply1
reply1 -response-> read
...
```

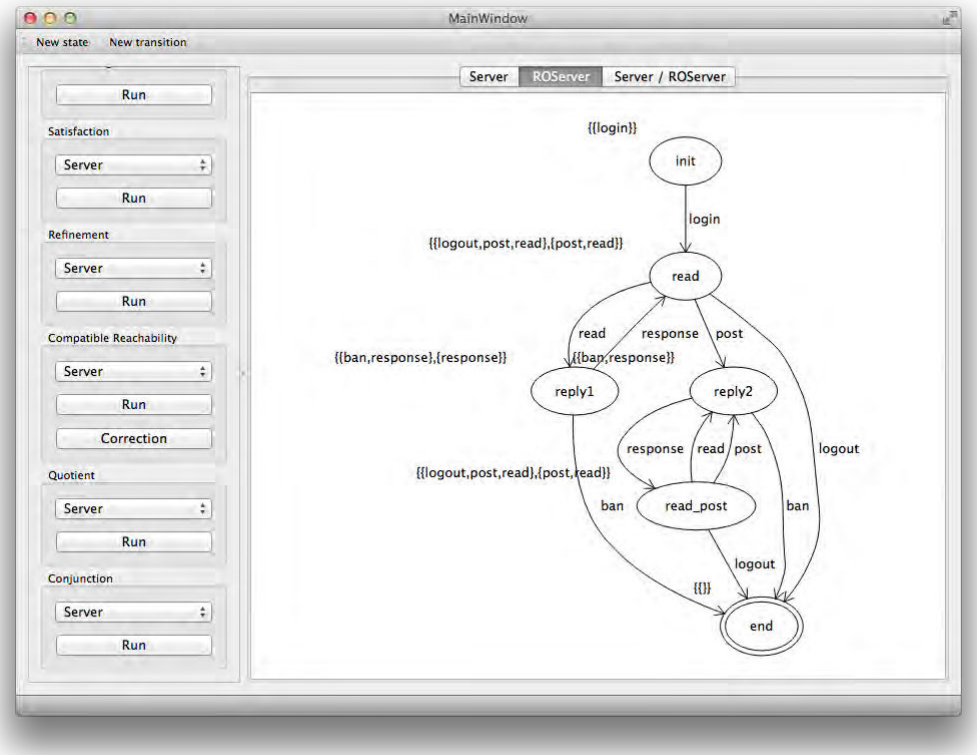


Figure 5.1 : Capture d'écran de MAccS

5.2 État de l'art

Un certain nombre d'outils ont été développés pour implémenter divers formalismes de spécification. La table 5.1 donne un aperçu de ces outils et de leurs fonctionnalités. Une description plus détaillée de chaque outil est donnée dans la version anglaise (section 5.2).

À notre connaissance, aucun outil n'implémente d'extension marquée d'un formalisme de spécification. Concernant les spécifications à ensembles d'acceptation, les systèmes de transitions disjonctifs non déterministes sont équivalents aux automates à ensembles d'acceptation non déterministes qui sont naturellement un sur-ensemble des spécifications à ensembles d'acceptation déterministes, aussi il devrait être possible d'utiliser MoTraS pour les manipuler. Mais nous ne connaissons pas d'implémentation de spécifications à ensembles d'acceptation déterministes ou de spécifications à ensembles d'acceptation convexes déterministes, avec des algorithmes optimisés pour utiliser les hypothèses de déterminisme ou de convexité.

5.3 Performances

Nous présentons maintenant des résultats expérimentaux.

Une première partie compare les spécifications avec des ensembles d'acceptation convexes avec des non convexes pour voir si la représentation optimisée que nous avons proposée est effectivement plus efficace en pratique. Nous générons des spécifications aléatoires et leur appliquons diverses opérations : test de raffinement, conjonction, etc. Il y a principalement deux variables que l'on peut faire varier : le nombre d'états et la taille de l'alphabet. Des résultats sont indiqués dans la figure 5.2. Nous observons qu'augmenter le nombre d'états n'a pas vraiment d'influence sur la différence de temps d'exécution entre les algorithmes. En effet, utiliser des ensembles convexes-clos

Outil	Lang.	Théorie	ND	Opérations					
				$\leq$	$\wedge$	$\otimes$	/	MC	ED
TAV [GLZ89, BLS95]	Prolog	MTS	✓	✓		✓		HML	
EPSILON [ČGL93]	Prolog	Timed MS	✓	✓		✓			
MTSA [DFCU08]	Java	MTS	✓	✓		✓		LTL[a]	
MoTraS [KS13]	Java	MTS	✓	✓	✓	✓	✓[b]	LTL	✓
		DMTS	✓	✓	✓	✓		LTL	✓
		BMTS, PMTS	✓	✓					✓
MIO Workbench [BMSH10, BML11]	Java	MIO	✓	✓	✓[b]	✓	✓[b]		
Mica [Cai11]	OCaml	MIO		✓	✓	✓	✓		n/a
ECDAR [DLL+10a] PyECDAR [LT13]	Java Python	Timed I/O		✓	✓	✓	✓		n/a
BALM [CPM+12]	C	FSM	✓[c]	✓			✓		
MAccS [VR15a]	C++	MAS, CAS		✓	✓	✓	✓		n/a

La colonne « ND » indique si les spécifications non déterministes sont permises.
Les opérations indiquées dans la table sont : raffinement ($\leq$), conjonction ($\wedge$), produit ($\otimes$), quotient (/), *model-checking* (la colonne indique la logique utilisée) et l'enveloppe déterministe pour les spécifications non déterministes.

[a] D'après [BČK11], le résultat retourné est parfois erroné.
[b] Seulement pour les spécifications déterministes.
[c] Les machines à état non déterministes sont déterminisées.

Table 5.1 : Aperçu des fonctionnalités de divers outils

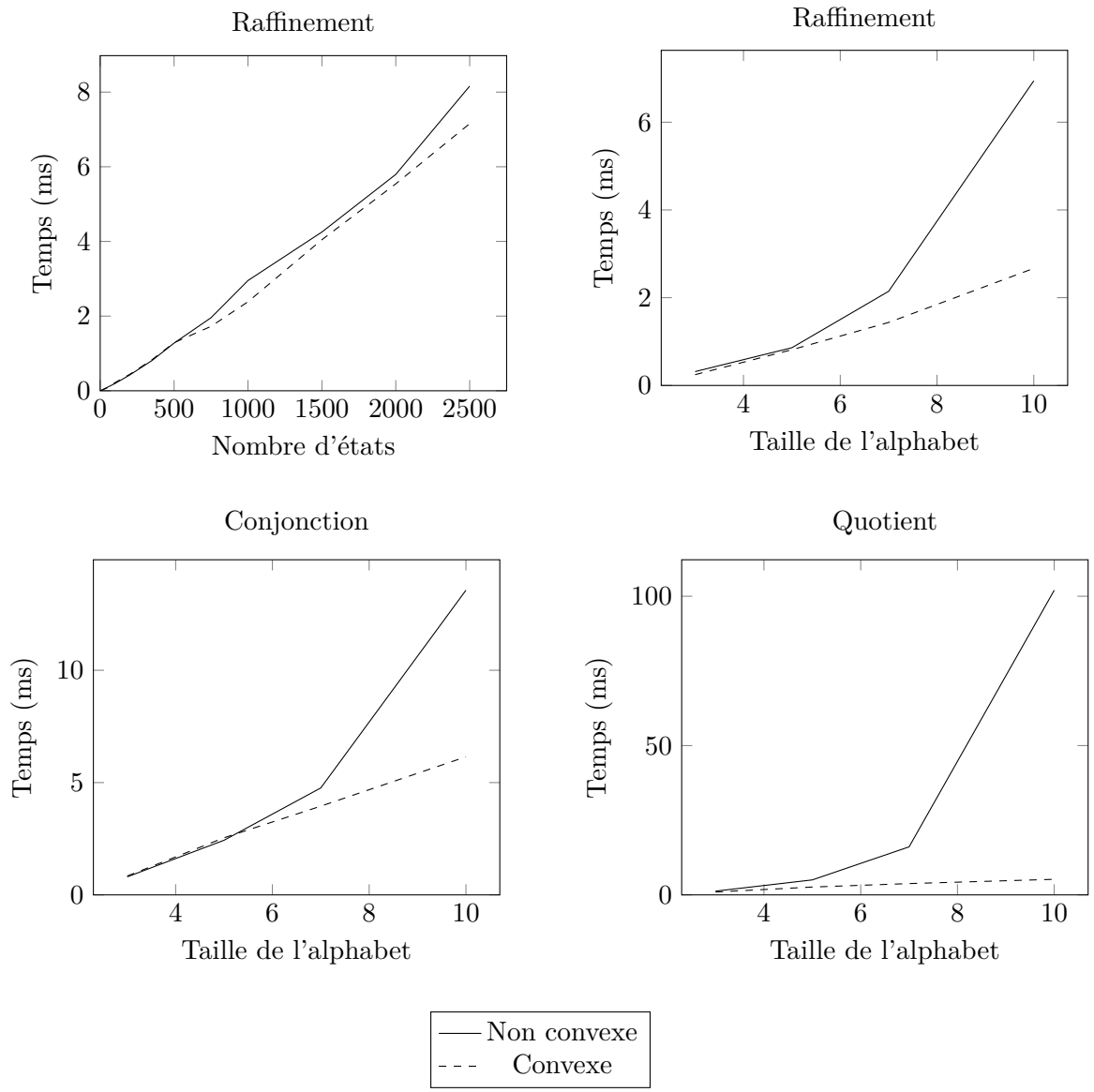


Figure 5.2 : Comparaison entre ensembles d'acceptation convexes et non convexes

améliore l'efficacité des opérations sur chaque ensemble, ce qui dépend de la taille de l'alphabet mais pas du nombre d'états. Par contre, il y a un gain très net quand l'alphabet augmente de taille. En particulier, pour l'opération de quotient, on voit disparaître l'explosion exponentielle.

Un élément fondamental pour avoir des opérations efficaces est la manière dont sont représentés les ensembles d'acceptation. Nous avons montré comment utiliser un sous-ensemble particulier des ensembles d'acceptation, les ensembles convexes-clos, pour gagner en performance. Mais utiliser une structure de données adaptée pour représenter ces ensembles pourrait aussi offrir des gains importants. Nous avons comparé trois structures de données : `std::set`, basé sur des arbres balancés, `std::unordered_set`, introduit dans le standard C++11 et basé sur des tables de hachages, et `std::bitset`, qui représente une séquence de bits (un ensemble d'actions appartenant à Σ est représenté par une séquence de $|\Sigma|$ bits, chaque bit indiquant si une action donnée appartient à l'ensemble ou non). Nous observons dans la figure 5.3 que `std::bitset` est clairement plus efficace que les deux autres implémentations pour représenter nos ensembles d'acceptation.

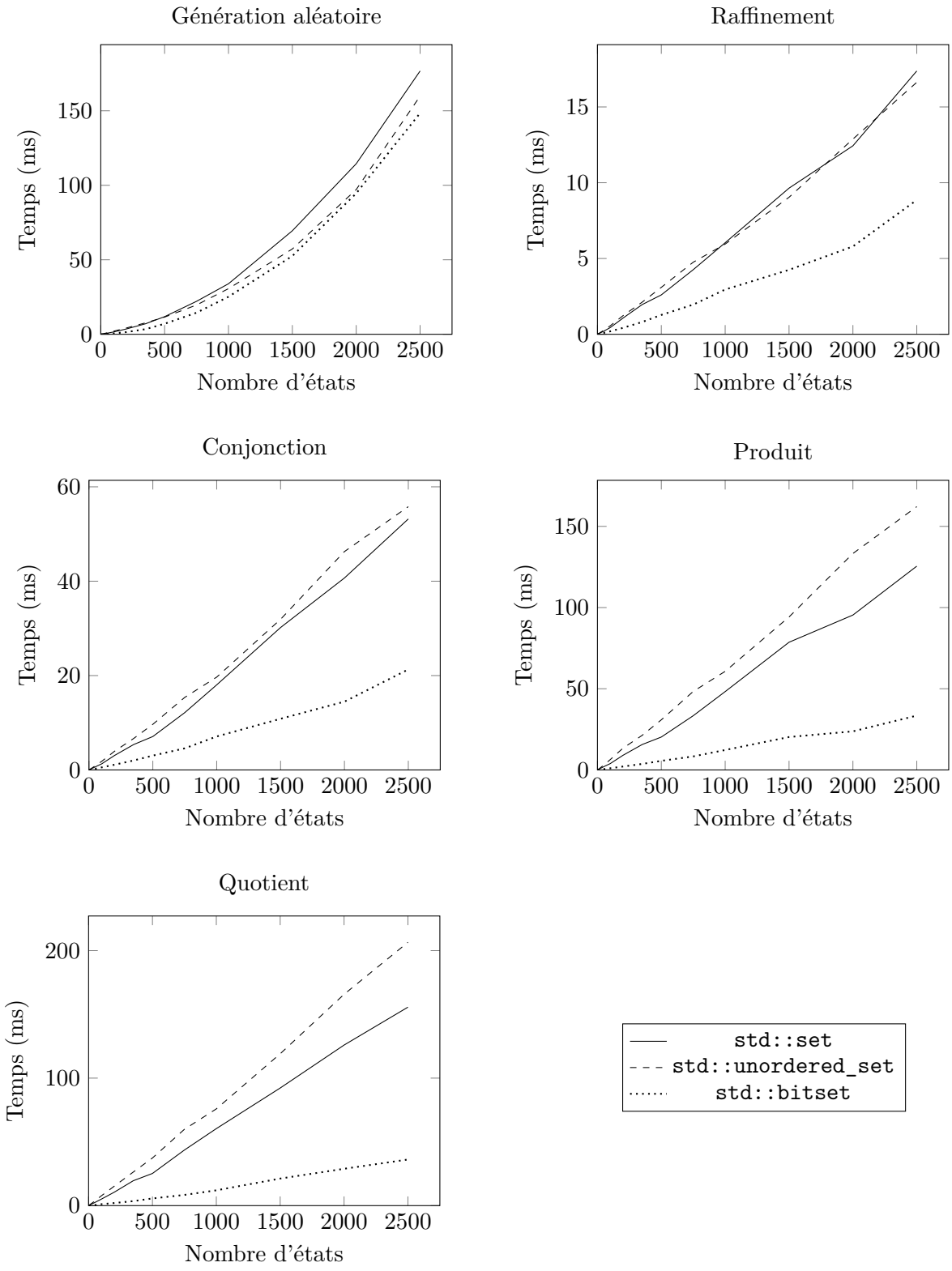


Figure 5.3 : Comparaison entre structures de données pour les ensembles

Chapitre 6

Conclusion

6.1 Contributions

Dans cette thèse, nous avons présenté deux principaux résultats théoriques sous la forme de deux nouveaux formalismes de spécification ; l'un offrant de meilleures performances en échange d'une expressivité un peu réduite et l'autre permettant d'exprimer un nouveau type de contraintes sur les chemins en utilisant des états marqués. Nous avons aussi implémenté ces formalismes de spécification dans un outil et présenté des résultats expérimentaux.

Spécifications à ensembles d'acceptation convexes. Elles permettent de définir des opérations plus efficaces que les spécifications à ensembles d'acceptation. Ces dernières offrent un formalisme de spécification très expressif, mais certaines opérations, en particulier le quotient, ont une complexité très élevée. Utiliser des ensembles d'acceptation convexes-clos nous permet de définir des opérations avec une complexité plus faible et, en particulier, d'éviter une explosion exponentielle par rapport à la taille de l'alphabet dans l'opération de quotient. De plus, les spécifications à ensembles d'acceptation convexes, bien que moins expressives que les spécifications à ensembles d'acceptation, restent plus expressives que beaucoup d'autre formalismes de spécification, comme les spécifications modales, les spécifications modales disjonctives et les spécifications modales avec obligations. Nous avons utilisé l'assistant de preuves Coq pour s'assurer de la validité de nos résultats.

Spécifications à ensembles d'acceptation marquées. Elles permettent d'exprimer des propriétés d'atteignabilité et disposent de toutes les opérations usuelles pour former une théorie de spécification complète, c'est-à-dire le raffinement, la conjonction, le produit et le quotient. Alors que de nombreux formalismes de spécification se concentrent seulement sur l'expression de propriétés locales avec, par exemple, des modalités, des ensembles d'acceptation ou des formules booléennes, les états marqués peuvent être utilisés pour exprimer des contraintes sur les chemins en garantissant que certains états seront toujours atteignables. Ceci peut être utilisé pour s'assurer de l'absence d'interblocages et donc, par exemple, qu'un système termine ou qu'un point de passage sera atteignable infiniment souvent.

MAccS. C'est un outil offrant une implémentation de ces théories. Son interface graphique peut être utilisée pour concevoir des spécifications et leur appliquer diverses opérations. Il peut aussi sauvegarder et charger des spécifications à partir d'une représentation textuelle et être intégré dans d'autres logiciels. Nous l'avons également utilisé pour faire des tests de performance : nous avons montré qu'utiliser des ensembles d'acceptation convexes-clos était

en effet bien plus efficace en pratique que des ensembles d'acceptation quelconques et nous avons montré que notre outil pouvait manipuler des spécifications avec des milliers d'états en une fraction de seconde.

6.2 Perspectives

Nous proposons maintenant des orientations possibles pour de futurs travaux. Nous commençons par décrire de possibles extensions des contributions de cette thèse, puis suggérons différentes directions pour l'étude de nouvelles théories de spécification.

6.2.1 Court terme

D'un point de vue pratique, nous pourrions continuer à travailler sur MAccS. En particulier, le code est pour l'instant exécuté de manière séquentielle. Certaines parties des algorithmes (comme le calcul des ensembles d'acceptation des états de la conjonction, du produit et du quotient) pourraient être exécutées en parallèle afin de mieux exploiter les possibilités des machines multicœurs.

Nous avons utilisé l'assistant de preuve Coq pour vérifier la validité des preuves sur les ensembles d'acceptation convexes-clos. Il serait intéressant de continuer ce travail afin de vérifier la théorie de spécification complète. Cependant, les graphes et automates, comme les ensembles, ne sont pas des structures de données inductives et il est donc difficile de les manipuler dans Coq ou des langages similaires comme Isabelle/HOL ou Agda. Alors que la bibliothèque standard de Coq offre de bonnes fondations pour manipuler les ensembles, il n'y a pas d'équivalent pour les automates. Des thèses entières ont été consacrées à ce sujet, par exemple [Pic12] pour la représentation de graphes en utilisant la coinduction en Coq et [Gio13] pour la représentation de structures de pointeurs, incluant les graphes, en Isabelle/HOL.

D'un point de vue plus théorique, plusieurs améliorations pourraient être apportées aux théories présentées dans cette thèse. D'abord, nous avons introduit des opérations d'extension d'alphabet sur les spécifications à ensembles d'acceptation afin de gérer les alphabets dissemblables. Ces opérations devraient être étendues aux spécifications à ensembles d'acceptation marquées. C'est une étape importante pour concevoir et construire des systèmes complexes avec des propriétés d'atteignabilité dans lesquels les différents composants peuvent avoir des alphabets différents.

D'autre part, nous pourrions étendre nos formalismes de spécification avec des entrées/sorties : les entrées représenteraient les actions émises par l'environnement du système en cours de conception tandis que les sorties correspondraient aux actions issues du système. La question du produit dans un cadre ouvert a été étudiée pour les automates d'interface dans [dAH01]. Les états d'erreur sont identifiés comme étant les états dans lesquels un automate d'interface peut émettre une action sans que l'autre automate d'interface ait une transition étiquetée par l'entrée correspondante. La présence d'états d'erreur ne pousse pas à interdire la composition ; au lieu de cela, une approche optimiste est proposée : la composition est autorisée s'il existe un troisième automate, appelé *environnement*, qui ferme le système et permet de ne pas atteindre les états d'erreur. Adapter la composition optimiste aux spécifications à ensembles d'acceptation marquées conduirait à considérer un environnement plus *coopératif* qui pourrait aider à atteindre les états marqués.

6.2.2 Moyen terme

Nous avons étudié deux problèmes différents dans cette thèse : d'un côté nous avons optimisé la représentation des ensembles d'acceptation convexes-clos et d'un autre côté nous avons introduit

un formalisme plus expressif avec des états marqués. Il serait intéressant de combiner ces deux résultats pour faire des « spécifications à ensembles d'acceptation convexes marquées ». Cependant, s'assurer que la convexité est préservée — en particulier par l'opération de quotient et la partie assez complexe supprimant les *livelocks* — risque d'être difficile.

Dans cette thèse, nous n'avons étudié que des spécifications déterministes, comme expliqué dans la section 2.3. Il y a de nombreux travaux sur les spécifications non déterministes et il pourrait donc être intéressant de voir si nos résultats sont préservés lorsque l'on considère des spécifications non déterministes. Ceci pourrait être assez difficile dans la mesure où les opérations non déterministes sont typiquement bien plus difficiles à définir que leurs variantes déterministes. De plus, le quotient d'automates à ensembles d'acceptation non déterministes défini dans [BDF$^+$13] a une explosion exponentielle par rapport au nombre d'états ; comme expliqué dans la section 3.3.7, la convexité pourrait aider à améliorer partiellement cette situation. Une autre possibilité serait d'étudier une autre sémantique pour les spécifications non déterministes basée sur des *failure traces* [BHR84] au lieu d'une simple sémantique de simulation, comme défendu dans [BV15].

6.2.3 Long terme

D'autre types de propriétés de compatibilité pourraient être étudiés dans le contexte d'une théorie de spécification comme, par exemple, l'opacité [Maz04], définie initialement dans la communauté sécurité. Par définition, un système est dit opaque si un ensemble donné de traces, appelé le secret, ne peut pas être inféré à partir d'une observation partielle. À notre connaissance, il n'existe pas de résultat de compositionalité pour cette propriété. Les points de départ pour une théorie de spécification offrant des systèmes opaques corrects par construction serait [AČZ06] pour le raffinement et [DDM10b] pour le quotient.

Une motivation pour l'introduction des spécifications à ensembles d'acceptation marquées était le besoin d'un formalisme de spécification pour modéliser des services sous-spécifiés accompagnés de leur possible terminaison de session grâce aux états marqués. Maintenant, une nouvelle étape pourrait être l'étude de l'orchestration de services qui pourrait être représentée par une spécification modale dont les transitions seraient étiquetées par l'identifiant d'un service modélisé par une spécification à ensembles d'acceptation marquée. Chaque transition serait alors interprétée comme un appel au service correspondant qui terminerait en atteignant un état marqué. Une autre possibilité serait d'étudier des *modal visible pushdown automata*.

Nous avons mentionné la possibilité de continuer la mécanisation de nos résultats dans l'assistant de preuve Coq pour s'assurer de leur validité. Lorsque l'on considère des extensions de théories de spécification avec des paramètres ou des données, en particulier sur des domaines infinis, les procédures de décision typiques deviennent souvent très inefficaces et certains problèmes sont même indécidables. Une manière de résoudre ce type de problèmes est de générer des obligations de preuve qui peuvent alors être prouvées avec des prouveurs automatiques ou des assistants de preuve. Par exemple, c'est l'approche utilisée par l'Atelier B et des projets de recherche comme BWare [DDMM14]. Alors, une mécanisation Coq pourrait être utile non seulement pour augmenter le degré de confiance dans les résultats, mais aussi pour offrir aux utilisateurs un moyen de prouver des propriétés sur leurs spécifications.

Bibliography

[AČZ06] Rajeev Alur, Pavol Černý, and Steve Zdancewic. Preserving secrecy under refinement. In *ICALP*, pages 107–118, 2006.

[AH99] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[AtBFG10] Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi. A logical framework to deal with variability. In *IFM*, volume 6396 of *LNCS*, pages 43–58. Springer, 2010.

[AVW03] André Arnold, Aymeric Vincent, and Igor Walukiewicz. Games for synthesis of controllers with partial observation. *Theor. Comput. Sci.*, 1(303):7–34, 2003.

[BBG+04] Steffen Becker, Antonio Brogi, Ian Gorton, Sven Overhage, Alexander Romanovsky, and Massimo Tivoli. Towards an engineering approach to component adaptation. In *Architecting Systems with Trustworthy Components. Revised Selected Papers*, volume 3938 of *LNCS*, pages 193–215. Springer, 2004.

[BČK11] Nikola Beneš, Ivana Černá, and Jan Křetínský. Modal transition systems: Composition and LTL model checking. In *ATVA*, volume 6996 of *LNCS*, pages 228–242. Springer, 2011.

[BCN+12] Albert Benveniste, Benoît Caillaud, Dejan Ničković, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, and Kim G. Larsen. Contracts for System Design. Research Report RR-8147, INRIA, November 2012.

[BDD+92] Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, and Rainer Weber. The Design of Distributed Systems — An Introduction to FOCUS. Technical Report TUM-I9202, Technisch Universität München, January 1992.

[BDF+13] Nikola Beneš, Benoît Delahaye, Uli Fahrenberg, Jan Křetínský, and Axel Legay. Hennessy-Milner logic with greatest fixed points as a complete behavioural specification theory. In *CONCUR*, volume 8052 of *LNCS*, pages 76–90. Springer, 2013.

[BDH+12] Sebastian S. Bauer, Alexandre David, Rolf Hennicker, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Moving from specifications to contracts in component-based design. In *Proc. of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE'12)*, pages 43–58, 2012.

[BDH+15] Nikola Beneš, Przemysław Daca, Thomas A. Henzinger, Jan Křetínský, and Dejan Ničković. Complete composition operators for ioco-testing theory. In *Proc. of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE '15)*, pages 101–110. ACM, 2015.

[BFJ+13] Sebastian S. Bauer, Uli Fahrenberg, Line Juhl, Kim G. Larsen, Axel Legay, and Claus Thrane. Weighted modal transition systems. *Formal Methods in System Design*, 42(2):193–220, 2013.

[BFK+14] Nikola Beneš, Uli Fahrenberg, Jan Křetínský, Axel Legay, and Louis-Marie Traonouez. Logical vs. Behavioural Specifications. Research report, Inria Rennes, 2014.

[BFLV15] Ferenc Bujtor, Sascha Fendrich, Gerald Lüttgen, and Walter Vogler. Nondeterministic modal interfaces. In *SOFSEM*, volume 8939 of *LNCS*, pages 152–163. Springer, 2015.

[BG00] Glenn Bruns and Patrice Godefroid. Generalized model checking: Reasoning about partial state spaces. In *CONCUR*, volume 1877 of *LNCS*, pages 168–182. Springer, 2000.

[BHB10] Sebastian S. Bauer, Rolf Hennicker, and Michel Bidoit. A modal interface theory with data constraints. In *BMF, Revised Selected Papers*, volume 6527 of *LNCS*, pages 80–95. Springer, 2010.

[BHL14] Sebastian S. Bauer, Rolf Hennicker, and Axel Legay. A meta-theory for component interfaces with contracts on ports. *Sci. Comput. Program.*, 91:70–89, 2014.

[BHR84] Stephen D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[BHW11] Sebastian S. Bauer, Rolf Hennicker, and Martin Wirsing. Interface theories for concurrency and data. *Theor. Comput. Sci.*, 412(28):3101–3121, 2011.

[BJL+12] Sebastian S. Bauer, Line Juhl, Kim G. Larsen, Axel Legay, and Jiří Srba. Extending modal transition systems with structured labels. *Mathematical Structures in Computer Science*, 22(4):581–617, 2012.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BK10] Nikola Beneš and Jan Křetínský. Process algebra for modal transition systemses. In *MEMICS*, pages 9–18, 2010.

[BKL+11] Nikola Beneš, Jan Křetínský, Kim G. Larsen, Mikael H. Møller, and Jiří Srba. Parametric modal transition systems. In *ATVA*, volume 6996 of *LNCS*, pages 275–289. Springer, 2011.

[BL92] Gérard Boudol and Kim G. Larsen. Graphical versus logical specifications. *Theor. Comput. Sci.*, 106(1):3–20, 1992.

[BLL+14] Sebastian S. Bauer, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wąsowski. A modal specification theory for components with data. *Sci. Comput. Program.*, 83:106–128, 2014.

[BLPR09] Nathalie Bertrand, Axel Legay, Sophie Pinchinat, and Jean-Baptiste Raclet. A compositional approach on modal specifications for timed systems. In *ICFEM*, volume 5885 of *LNCS*, pages 679–697. Springer, 2009.

[BLPR12] Nathalie Bertrand, Axel Legay, Sophie Pinchinat, and Jean-Baptiste Raclet. Modal event-clock specifications for timed component-based design. *Sci. Comput. Program.*, 77(12):1212–1234, 2012.

[BLS95] Anders Børjesson, Kim G. Larsen, and Arne Skou. Generality in design and compositional verification using TAV. *Formal Methods in System Design*, 6(3):239–258, 1995.

[BML11] Sebastian S. Bauer, Philip Mayer, and Axel Legay. MIO Workbench: A tool for compositional design with modal input/output interfaces. In *ATVA*, volume 6996 of *LNCS*, pages 418–421. Springer, 2011.

[BMSH10] Sebastian S. Bauer, Philip Mayer, Andreas Schroeder, and Rolf Hennicker. On weak modal compatibility, refinement, and the MIO workbench. In *TACAS*, volume 6015 of *LNCS*, pages 175–189. Springer, 2010.

[BSBM04] Lucas Bordeaux, Gwen Salaün, Daniela Berardi, and Massimo Mecella. When are two web services compatible? In *TES, Revised Selected Papers*, volume 3324 of *LNCS*, pages 15–28. Springer, 2004.

[BV15] Ferenc Bujtor and Walter Vogler. Failure semantics for modal transition systems. *ACM Trans. Embed. Comput. Syst.*, 14(4):67:1–67:30, 2015.

[Cai11] Benoît Caillaud. Mica: A modal interface compositional analysis library, 2011. `http://www.irisa.fr/s4/tools/mica`.

[CCJK12] Taolue Chen, Chris Chilton, Bengt Jonsson, and Marta Kwiatkowska. A compositional specification theory for component behaviours. In *ESOP*, volume 7211 of *LNCS*, pages 148–168. Springer, 2012.

[CDEG03] Marsha Chechik, Benet Devereux, Steve Easterbrook, and Arie Gurfinkel. Multi-valued symbolic model-checking. *ACM Trans. Softw. Eng. Methodol.*, 12(4):371–408, 2003.

[CDL+11] Benoît Caillaud, Benoît Delahaye, Kim G. Larsen, Axel Legay, Mikkel L. Pedersen, and Andrzej Wąsowski. Constraint markov chains. *Theor. Comput. Sci.*, 412(34):4373–4404, 2011.

[ČGL93] Kārlis Čerāns, Jens Chr. Godskesen, and Kim G. Larsen. Timed modal specification — Theory and tools. In *CAV*, volume 697 of *LNCS*, pages 253–267. Springer, 1993.

[CL08] Christos G. Cassandras and Stéphane Lafortune. *Introduction to discrete event systems*. Springer Science & Business Media, 2008.

[CMP06] Carlos Canal, Juan Manuel Murillo, and Pascal Poizat. Software adaptation. *L'objet*, 12(1):9–31, 2006.

[CPM+12] Giovanni Castagnetti, Matteo Piccolo, Alan Mishchenko, Tiziano Villa, Robert K. Brayton, and Nina Yevtushenko. Solving parallel equations with BALM-II. Technical report, University of Verona, 2012.

[CPS08] Carlos Canal, Pascal Poizat, and Gwen Salaün. Model-based adaptation of behavioral mismatching components. *IEEE Trans. Software Eng.*, 34(4):546–563, 2008.

[CR12] Benoît Caillaud and Jean-Baptiste Raclet. Ensuring reachability by design. In *ICTAC*, volume 7521 of *LNCS*, pages 213–227. Springer, 2012.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ACM SIGSOFT Software Engineering Notes*, pages 109–120. ACM, 2001.

[DDM10a] Philippe Darondeau, Jérémy Dubreil, and Hervé Marchand. Supervisory control for modal specifications of services. In *WODES*, pages 428–435, 2010.

[DDM10b] Jérémy Dubreil, Philippe Darondeau, and Hervé Marchand. Supervisory control for opacity. *IEEE Trans. Automat. Contr.*, 55(5):1089–1100, 2010.

[DDMM14] David Delahaye, Catherine Dubois, Claude Marché, and David Mentré. The BWare Project: Building a Proof Platform for the Automated Verification of B Proof Obligations. In *Abstract State Machines, Alloy, B, VDM, and Z*, volume 8477 of *LNCS*, pages 290–293. Springer, 2014.

[DFCU08] Nicolás D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastian Uchitel. MTSA: The modal transition system analyser. In *ASE*, pages 475–476. IEEE, 2008.

[DGG97] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):253–291, 1997.

[DHJP08] L. Doyen, T. A. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *EMSOFT*, pages 79–88. ACM Press, 2008.

[DKL+13] Benoît Delahaye, Joost-Pieter Katoen, Kim G. Larsen, Axel Legay, Mikkel L. Pedersen, Falak Sher, and Andrzej Wąsowski. Abstract probabilistic automata. *Information and Computation*, 232(0):66 – 116, 2013.

[DLL+10a] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wąsowski. ECDAR: An environment for compositional design and analysis of real time systems. In *ATVA*, volume 6252 of *LNCS*, pages 365–370. Springer, 2010.

[DLL+10b] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wąsowski. Timed I/O automata: A complete specification theory for real-time systems. In *HSCC*, pages 91–100. ACM, 2010.

[EHH12] Dorsaf Elhog-Benzina, Serge Haddad, and Rolf Hennicker. Refinement and asynchronous composition of modal Petri nets. *T. Petri Nets and Other Models of Concurrency*, 6900:96–120, 2012.

[FKLT15] Uli Fahrenberg, Jan Křetínský, Axel Legay, and Louis-Marie Traonouez. Compositionality for quantitative specifications. In *FACS*, volume 8997 of *LNCS*, pages 306–324. Springer, 2015.

[FS08] Harald Fecher and Heiko Schmidt. Comparing disjunctive modal transition systems with an one-selecting variant. *J. Log. Algebr. Program.*, 77(1-2):20–39, 2008.

[Gio13] Mathieu Giorgino. *Inductive representation, proofs and refinement of pointer structures.* PhD thesis, Université Toulouse III Paul Sabatier, 2013.

[GLZ89] Jens Chr. Godskesen, Kim G. Larsen, and Michael Zeeberg. TAV (Tools for Automatic Verification)—User's Manual. Technical report, Aalborg University, 1989.

[GMW12] Christian Gierds, Arjan J. Mooij, and Karsten Wolf. Reducing adapter synthesis to controller synthesis. *IEEE Transactions on Services Computing*, 5(1):72–85, 2012.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[GR09] Gregor Goessler and Jean-Baptiste Raclet. Modal contracts for component-based design. In *SEFM*, pages 295–303. IEEE, 2009.

[Hen85] Matthew Hennessy. Acceptance trees. *J. ACM*, 32(4):896–928, 1985.

[HHM13] Serge Haddad, Rolf Hennicker, and Mikael H. Møller. Specification of asynchronous component systems with modal I/O-Petri nets. In *TGC*, volume 8358 of *LNCS*, pages 219–234. Springer, 2013.

[HJS01] Michael Huth, Radha Jagadeesan, and David Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *ESOP*, volume 2028 of *LNCS*, pages 155–169. Springer, 2001.

[HM80] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *Automata, Languages and Programming*, LNCS, pages 299–309. Springer, 1980.

[HN12] Thomas A. Henzinger and Dejan Ničković. Independent implementability of viewpoints. In *Large-Scale Complex IT Systems. Development, Operation and Management — 17th Monterey Workshop*, pages 380–395, 2012.

[HS07] Thomas A. Henzinger and Joseph Sifakis. The discipline of embedded systems design. *IEEE Computer*, 40(10):32–40, 2007.

[JL91] Bengt Jonsson and Kim G. Larsen. Specification and refinement of probabilistic processes. In *LICS*, pages 266–277. IEEE, 1991.

[KS13] Jan Křetínský and Salomon Sickert. MoTraS: A tool for modal transition systems and their extensions. In *ATVA*, volume 8172 of *LNCS*, pages 487–491. Springer, 2013.

[KSL13] Andrew King, Oleg Sokolsky, and Insup Lee. A modal specification theory for timing variability. Technical report, Penn Engineering, 2013.

[Lar89] Kim G. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246. Springer, 1989.

[LNW07a] Kim G. Larsen, Ulrik Nyman, and Andrzej Wąsowski. Modal I/O automata for interface and product line theories. In *ESOP*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.

[LNW07b] Kim G. Larsen, Ulrik Nyman, and Andrzej Wąsowski. On modal refinement and consistency. In *CONCUR*, pages 105–119. Springer, 2007.

[LT88] Kim G. Larsen and Bent Thomsen. A modal process logic. In *LICS*, pages 203–210. IEEE, 1988.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[LT13] Axel Legay and Louis-Marie Traonouez. PyECDAR: Towards open source implementation for timed systems. In *ATVA*, volume 8172 of *LNCS*, pages 460–463. Springer, 2013.

[LV06] Gerald Lüttgen and Walter Vogler. Conjunction on processes: full-abstraction via ready-tree semantics. In *FOSSACS*, volume 3921 of *LNCS*, pages 261–276. Springer, 2006.

[LV12] Gerald Lüttgen and Walter Vogler. Modal interface automata. In *Theoretical Computer Science*, volume 7604 of *LNCS*, pages 265–279. Springer, 2012.

[LV13] G. Lüttgen and W. Vogler. Modal interface automata. *Logical Methods in Computer Science*, 9(3), 2013.

[LW11] N. Lohmann and K. Wolf. Compact representations and efficient algorithms for operating guidelines. *Fundam. Inform.*, 108(1-2):43–62, 2011.

[LX90] Kim G. Larsen and Liu Xinxin. Equation solving using modal transition systems. In *LICS*, pages 108–117. IEEE, 1990.

[Maz04] Laurent Mazaré. Decidability of opacity with non-atomic keys. In *FAST*, pages 71–84, 2004.

[MPS12] Radu Mateescu, Pascal Poizat, and Gwen Salaün. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. *IEEE Trans. Software Eng.*, 38(4):755–777, 2012.

[MS05] P. Massuthe and K. Schmidt. Operating guidelines - an automata-theoretic foundation for the service-oriented architecture. In *QSIC*, pages 452–457, 2005.

[NITS14] Pierluigi Nuzzo, Antonio Iannopollo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. Are interface theories equivalent to contract theories? In *Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE*, pages 104–113, 2014.

[Pic12] Célia Picard. *Coinductive graph representation.* PhD thesis, Université Toulouse III Paul Sabatier, 2012.

[Rac08] Jean-Baptiste Raclet. Residual for component specifications. In *FACS*, volume 215 of *Electr. Notes Theor. Comput. Sci.*, pages 93–110, 2008.

[RBB+09] Jean-Baptiste Raclet, Éric Badouel, Albert Benveniste, Benoît Caillaud, and Roberto Passerone. Why are modalities good for interface theories? In *Proc. of the 9th International Conference on Application of Concurrency to System Design (ACSD'09).* IEEE Computer Society Press, July 2009.

[RBB+11] Jean-Baptiste Raclet, Éric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. A modal interface theory for component-based design. *Fundamenta Informaticae*, 108(1-2):119–149, 2011.

[RT14] Jan Reineke and Stavros Tripakis. Basic problems in multi-view modeling. In *TACAS*, volume 8413 of *LNCS*, pages 217–232. Springer, 2014.

[RW89] Peter J. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[SLL02] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library*. Addison-Wesley, 2002.

[UC04] Sebastian Uchitel and Marsha Chechik. Merging partial behavioural models. In *SIGSOFT FSE*, pages 43–52. ACM, 2004.

[VPY+15] Tiziano Villa, Alexandre Petrenko, Nina Yevtushenko, Alan Mishchenko, and Robert K. Brayton. Component-based design by solving language equations. *Proceedings of the IEEE*, 103(11):2152–2167, 2015.

[VR15a] Guillaume Verdier and Jean-Baptiste Raclet. MAccS: a tool for reachability by design. In *FACS*, volume 8997 of *LNCS*, 2015.

[VR15b] Guillaume Verdier and Jean-Baptiste Raclet. Quotient of acceptance specifications under reachability constraints. In *LATA*, volume 8977 of *LNCS*, 2015.

[VYB+11] Tiziano Villa, Nina Yevtushenko, Robert K. Brayton, Alan Mishchenko, Alexandre Petrenko, and Alberto Sangiovanni-Vincentelli. *The unknown component problem: theory and applications*. Springer Science & Business Media, 2011.

[WMN05] A. Wombacher, B. Mahleko, and E. J. Neuhold. IPSI-PF - a business process matchmaking engine based on annotated finite state automata. *Inf. Syst. E-Business Management*, 3(2):127–150, 2005.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):292–333, 1997.

Variants of Acceptance Specifications for Modular System Design

Guillaume Verdier

Abstract

Software programs are taking a more and more important place in our lives. Some of these programs, like the control systems of power plants, aircraft, or medical devices for instance, are critical: a failure or malfunction could cause loss of human lives, damages to equipments, or environmental harm. Formal methods aim at offering means to design and verify such systems in order to guarantee that they will work as expected. As time passes, these systems grow in scope and size, yielding new challenges. It becomes necessary to develop these systems in a modular fashion to be able to distribute the implementation task to engineering teams. Moreover, being able to reuse some trustworthy parts of the systems and extend them to answer new needs in functionalities is increasingly required. As a consequence, formal methods also have to evolve in order to accommodate both the design and the verification of these larger, modular systems and thus address their scalability challenge.

We promote an algebraic approach for the design of correct-by-construction systems. It defines a formalism to express high-level specifications of systems and allows to incrementally refine these specifications into more concrete ones while preserving their properties, until an implementation is obtained. It also defines several operations allowing to assemble complex systems from simpler components, by merging several viewpoints of a specific system or composing several subsystems together, as well as decomposing a complex specification in order to reuse existing components and ease the implementation task. The specification formalism we use is based on modal specifications. In essence, a modal specification is an automaton with two kinds of transitions allowing to express mandatory and optional behaviors. Refining a modal specification amounts to deciding whether some optional parts should be removed or made mandatory.

This thesis contains two main theoretical contributions, based on an extension of modal specifications called acceptance specifications. The first contribution is the identification of a subclass of acceptance specifications, called convex acceptance specifications, which allows to define much more efficient operations while maintaining a high level of expressiveness. The second contribution is the definition of a new formalism, called marked acceptance specifications, that allows to express some reachability properties. This could be used for example to ensure that a system is terminating or to express a liveness property for a reactive system. Usual operations are defined on this new formalism and guarantee the preservation of the reachability properties as well as independent implementability. This thesis also describes some more practical results. All the theoretical results on convex acceptance specifications have been proved using the Coq proof assistant. The tool MAccS has been developed to implement the formalisms and operations presented in this thesis. It allows to test them easily on some examples, as well as run some experimentations and benchmarks.

Keywords: modular design, correctness-by-construction, acceptance specifications, specification theory, reachability, convexity

Variantes de spécifications à ensembles d'acceptation pour la conception modulaire de systèmes

Guillaume Verdier

Résumé

Les programmes informatiques prennent une place de plus en plus importante dans nos vies. Certains de ces programmes, comme par exemple les systèmes de contrôle de centrales électriques, d'avions ou de systèmes médicaux, sont critiques : une panne ou un dysfonctionnement pourraient causer la perte de vies humaines ou des dommages matériels ou environnementaux importants. Les méthodes formelles visent à offrir des moyens de concevoir et vérifier de tels systèmes afin de garantir qu'ils fonctionneront comme prévu. Au fil du temps, ces systèmes deviennent de plus en plus évolués et complexes, ce qui est source de nouveaux défis pour leur vérification. Il devient nécessaire de développer ces systèmes de manière modulaire afin de pouvoir distribuer la tâche d'implémentation à différentes équipes d'ingénieurs. De plus, il est important de pouvoir réutiliser des éléments certifiés et les adapter pour répondre à de nouveaux besoins. Aussi les méthodes formelles doivent évoluer afin de s'adapter à la conception et à la vérification de ces systèmes modulaires de taille toujours croissante.

Nous travaillons sur une approche algébrique pour la conception de systèmes corrects par construction. Elle définit un formalisme pour exprimer des spécifications de haut niveau et permet de les raffiner de manière incrémentale en des spécifications plus concrètes tout en préservant leurs propriétés, jusqu'à obtenir une implémentation. Elle définit également plusieurs opérations permettant de construire des systèmes complexes à partir de composants plus simples en fusionnant différents points de vue d'un même système ou en composant plusieurs sous-systèmes ensemble, ainsi que de décomposer une spécification complexe afin de réutiliser des composants existants et de simplifier la tâche d'implémentation. Le formalisme de spécification que nous utilisons est basé sur des spécifications modales. Intuitivement, une spécification modale est un automate doté de deux types de transitions permettant d'exprimer des comportements optionnels ou obligatoires. Raffiner une spécification modale revient à décider si les parties optionnelles devraient être supprimées ou rendues obligatoires.

Cette thèse contient deux principales contributions théoriques basées sur une extension des spécifications modales appelée « spécifications à ensembles d'acceptation ». La première contribution est l'identification d'une sous-classe des spécifications à ensembles d'acceptation, appelée « spécifications à ensembles d'acceptation convexes », qui permet de définir des opérations bien plus efficaces tout en gardant un haut niveau d'expressivité. La seconde contribution est la définition d'un nouveau formalisme, appelé « spécifications à ensembles d'acceptation marquées », qui permet d'exprimer des propriétés d'atteignabilité. Ceci peut, par exemple, être utilisé pour s'assurer qu'un système termine ou exprimer une propriété de vivacité dans un système réactif. Les opérations usuelles sont définies sur ce nouveau formalisme et elles garantissent la préservation des propriétés d'atteignabilité. Cette thèse présente également des résultats d'ordre plus pratique. Tous les résultats théoriques sur les spécifications à ensembles d'acceptation convexes ont été prouvés en utilisant l'assistant de preuves Coq. L'outil MAccS a été développé pour implémenter les formalismes et opérations présentés dans cette thèse. Il permet de les tester aisément sur des exemples, ainsi que d'étudier leur efficacité sur des cas concrets.

Mots-clés : conception modulaire, correction par construction, spécifications à ensembles d'acceptation, théorie de spécification, atteignabilité, convexité