SAPIENZA
UNIVERSITÀ DI ROMA

Agent Behavior Synthesis from Components

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
ANTONIO RUBERTI

Dottorato di Ricerca in Ingegneria Informatica – XXV Ciclo

Candidate

Paolo Felli

ID number 798378

Thesis Advisor

Prof. Giuseppe De Giacomo

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Ingegneria Informatica

1 October 2013

Thesis defended on 7 October 2013

Agent Behavior Synthesis from Components
Ph.D. thesis. Sapienza – University of Rome

Version: 1 October 2013

Website: http://www.dis.uniroma1.it/ felli

Author's email: felli@dis.uniroma1.it

Hemingway never did this.

Acknowledgments

I would like to acknowledge my advisor and all the people that helped me, in various ways, to complete my studies and to develop the ideas presented in this dissertation. They know who they are. Their continuous help, advise and encouragement have been fundamental for successfully accomplishing my work. I also thank the innumerable bartenders around the world who made this possible.

Contents

Chapter 1

Introduction

In this thesis we address the problem of agent behavior synthesis from logical specifications. The goal is to transform a specification into a program that is guaranteed to satisfy it. First of all, when talking about "*agents*", we do not wish to restrict ourselves to software, computer embedded or specific hardware-based systems. The terminology we shall be using is thus general enough to account for these and others; as so, we won't be specific about the final implementations such systems will have. From an AI perspective, an agent can be a physical or virtual entity that can reason and perform actions to achieve its goals, and its *behavior* refers to its abstract operational model, generally represented as a nondeterministic finite-state machine. Hence, we will describe agents through such models, and we will refer to them as (agent) behaviors.

There exist, in literature, various characterizations trying to identify those classes of non-trivial systems that require special methods and approaches. Roughly speaking, systems (seen as a whole) can be deterministic or non-deterministic (depending wether their evolution is uniquely defined by the actions they perform), terminating or non-terminating (whose execution is unbounded), synchronous/asynchronous, realt-time, and so on. In system design, a more fundamental distinction is considered, one that allows to characterize systems for which specific methods are needed from those that are relatively easy to deal with (Harel and Pnueli, 1985a;b). Systems are thus divided into *closed* and *open*. On one hand, closed systems are those whose computations are completely determined by their state; they may accept inputs from the external word, perform internal transformations and produce outputs. On the other hand, open systems (or *reactive systems*) are embedded into a certain *environment* $\mathcal{E}$ they continuously need to interact with via input and output signals. Their computations crucially depends on this interaction and therefore they can be hardly though of as input/output functions, as they depend on their *context*, i.e., the state of $\mathcal{E}$, which is not controllable. This is the kind of system we will be considering in this dissertation, as they are better suited to model an agent setting. To this end, we will use "*(agent) system*" to refer to a set of agents $\mathcal{S}$, opposed to the environment $\mathcal{E}$. Notice, however, that an environment can embody itself adversarial agents, thus this distinction is more conceptual than practical, and it will differ in practice, depending on the specific setting.

We can further characterize the synthesis task by adjusting the information avail-

able, namely distinguishing between settings with complete/incomplete *information* and full/partial *observability* of the environment. Moreover, we are interested in system that are also *dynamic*, in the sense that they respond to events depending not only on the current context, but also on the history of prior events or actions.
We began defining the synthesis problem as the problem of transforming a logical specification into a program that is guaranteed to satisfy it. Being now more specific, considering an agent system $\mathcal{S}$, an environment $\mathcal{E}$ and a logical specification φ, we can define the main reasoning tasks of interest as follows:

- Verification. It is the task of checking, for a given system $\mathcal{S}$ and specification φ, whether $\mathcal{S}$ satisfies φ in $\mathcal{E}$, denoted $(\mathcal{S} \parallel \mathcal{E}) \vDash \varphi$.
- Realizability. The realizability of φ is the problem of determining whether there exists a *controller* $\mathcal{C}$ able to *refine* $\mathcal{S}$ such that all its computations on $\mathcal{E}$ satisfy φ, i.e., $(\mathcal{C} \parallel \mathcal{S} \parallel \mathcal{E}) \vDash \varphi$ whenever φ is satisfiable.
- Synthesis Problem. The correct synthesis of φ amounts to construct such controller $\mathcal{C}$.

In general, satisfiability of φ does not imply that a required $\mathcal{C}$ does exists. In addition, an evidence of the satisfiability of φ is not of much help for synthesis.

In this thesis we deal with various problems of verification and synthesis, of increasing sophistication, that are concerned with aspects such as observability, controllability and realizability. The kinds of systems we will consider here are characterized by an ongoing, typically non-terminating and highly non-deterministic evolution, and they often model parallel or distributed programs. Finally, our analysis is regarded as offline. How $\mathcal{S}$, $\mathcal{E}$ and a controller $\mathcal{C}$ are defined, how the synthesis is performed, or how is possible to model the interaction between these fundamental components may vary considerably depending on the setting. Therefore, we will make use of several ingredients.

LTL synthesis for reactive systems. We will generally assume non-terminating systems, and a specification usually given as a linear temporal formula (LTL) (Pnueli, 1977).
More formally, we can reformulate the synthesis problem for reactive systems as follows (Kupferman and Vardi, 2000). Given two sets I and O of input and output signals, respectively, we can regard a controller $\mathcal{C}$ as derived from a strategy function $f_{\mathcal{C}} : (2^I)^* \to 2^O$ that maps a finite sequence of sets of input signals, i.e. the *history* of input signals received so far, into a set of output signals. When $\mathcal{C}$ interacts with $\mathcal{E}$ –which generates (infinite) input sequences–, it associates with each input sequence (infinite) computations over $2^{\{I \cup O\}}$. At each step $\ell = 0, 1, \ldots$, the strategy $f_{\mathcal{C}}$ outputs (assigns to O) the value $f_{\mathcal{C}}(i_0, \ldots, i_\ell)$, where $i_0, \ldots, i_\ell$ is the sequence of input values assumed by the variables of I over the ℓ steps. Since $\mathcal{E}$ is not controllable, all possible inputs have to be taken into account. As a consequence, even though the program $f_{\mathcal{C}}$ is deterministic, it induces a computation-tree whose branching corresponds to external non-determinism. The branching of such tree is indeed fixed, and each branch (path) corresponds to a different computation. A correct strategy for this linear paradigm is thus required to be such that the property

expressed by requirement φ holds in all these paths (Lamport, 1980; Emerson and Halpern, 1986).

The realizability and synthesis problems can be traced back to the Church's solvability problem (Church, 1963), which considered the synthesis problem for specifications written in monadic second order theory of one successor (S1S). In that setting, the specification is given by means of a regular relation $R \subseteq (2^I)^\omega \times (2^O)^\omega$ relating sequences of input and output signals. In other words, $\langle x, y \rangle \in R$ means that y is a permitted response to x. Church's problem was solved independently by Rabin (Rabin, 1969) using tree automata and from Büchi and Landweber (Büchi and Landweber, 1969) using infinite games, and indeed there is a close relation between finite automata over infinite objects and finite games of infinite length –or infinite games.

When dealing with closed systems, a program meeting the specification can be extracted from a constructive proof that the formula is satisfiable (Manna and Waldinger, 1980; Clarke and Emerson, 1982). Essentially, the automata-based algorithm for LTL realizability comprises the following steps: first we convert the negated specification $\neg\varphi$ into a non-deterministic Büchi automaton $A_{\neg\varphi}$ (Grädel et al., 2002) and check for non-emptiness the product $A_{\neg\varphi} \times \mathcal{S}$, where $\mathcal{S}$ denotes here the behavior of the system. However, such synthesis paradigm is not of much interest when applied to open systems, because we have to cope with the uncontrollable behavior of their environment as well (Pnueli and Rosner, 1989b). For these reasons, there is a growing need for reliable methods of designing correct reactive systems, and two main approaches to LTL synthesis exist in this setting. The first approach is based on a reduction to the emptiness problem of ω-tree automata (Pnueli and Rosner, 1989b). The second is to view it as the solution of a two-player ω-regular game (Grädel et al., 2002), i.e., a game played by two players, the controller $\mathcal{C}$ and its environment $\mathcal{E}$. In this game, player $\mathcal{C}$ tries to satisfy φ, whereas player $\mathcal{E}$ tries to violate it. Player $\mathcal{C}$ wins if it manages to keep the game in the winning region, i.e., those states from which, irrespectively of the actions chosen by $\mathcal{E}$, the combined behavior is such that $(\mathcal{C} \parallel \mathcal{S} \parallel \mathcal{E}) \vDash \varphi$. The first step consists in translating the LTL formula φ to an equivalent non-deterministic ω-automaton A_φ (which is exponential in φ), then we convert such automaton into a deterministic one (exponential blow-up) and we build a corresponding ω-regular game. The main complexity source of these procedure is hence the required determinization of a non-deterministic ω-automaton, which is also technically difficult when it comes to devise symbolic algorithms. Indeed, Safra's construction (Safra, 1988) generates (in the worst case) an equivalent deterministic Rabin automaton with an highly exponential number of states which can be hardly managed without sophisticated minimization techniques (Klein and Baier, 2006).

As a consequence, LTL synthesis is a well-known decidable setting but one that is 2EXPTIME-complete (Pnueli and Rosner, 1989b; Kupferman and Vardi, 2000). This is one of the reasons why, in spite of the rich theory developed for system synthesis, little of this theory has been reduced to practice in Computer Science. This is in contrast with *Model Checking* (Emerson, 1996; Baier and Katoen, 2008) techniques, i.e., the systematic, automated check of a formal, logical specification φ against a complete model, which has led to industrial development and verification tools.

As expected, the full synthesis process is considered hopelessly intractable, although there exist attempts to avoid determinization ("Safraless" approach (Kupferman and Vardi, 2005)). Solutions that are not complete but computationally attractive have also been put forward (e.g. (Harding et al., 2005)). Alternative approaches focus on restrictions imposed on the LTL specification. Request-response specifications of the form $\square(p_i \rightarrow \diamond q_i)$ are considered in (Wallmeier et al., 2003), and in (Piterman et al., 2006a; Bloem et al., 2012), authors proposed an approach to synthesis for linear-time specifications based on a wider class of formulas called General Reactivity of rank 1 (GR(1)), where the winning condition is of the form $(\square\diamond p_1 \wedge \cdots \wedge \square\diamond p_m) \rightarrow (\square\diamond q_1 \wedge \cdots \wedge \square\diamond q_n)$ -where each p_i and q_i is a boolean combination of atomic propositions. Such restriction leads to efficient symbolic algorithms for synthesis (Bloem et al., 2012).

Apart from computational and algorithmic concerns, this synthesis approach have been often questioned as it assumes that a complete, exhaustive specification is provided at design time. However this assumption is not completely realistic: it is often the case that such specification is not available from the beginning, but it needs to be computed or refined incrementally, or it may even evolve. In (Kupferman et al., 2006) authors address the problem of compositional synthesis. Supposing to have synthesized already different programs for two specifications φ_1 and φ_2, the idea is to use realizability proofs as a starting point for realizability testing and synthesis for $\varphi_2 \wedge \varphi_2$.

Incomplete information. So far, we considered the case where the specifications (either linear or branching) refer solely to signals in I and O, which are both known to $\mathcal{C}$. This is called synthesis with *complete information*. However, it is often the case that the program does not have complete information about its environment. Hence assume that $\mathcal{C}$ can not read some signals E, with $E \cap I = \varnothing$, even tough the specification is allowed to mention them. It can still be viewed as a strategy $f_{\mathcal{C}} : (2^I)^* \rightarrow 2^O$. Nevertheless, the computations of $\mathcal{C}$ are now infinite words over $2^{I \cup E \cup O}$, and the computation tree induced by $f_{\mathcal{C}}$ now has a fixed branching degree $|2^{I \cup E}|$. Also note that different nodes in this tree may have, according $f_{\mathcal{C}}$'s incomplete information, the same history of inputs, and can not be discriminated. Hence, $f_{\mathcal{C}}$ must output the same set of signals or, in other words, it must be *executable*.

It is known how to cope with incomplete information in the linear paradigm. In particular, the approach used in (Pnueli and Rosner, 1989b) can be extended to handle LTL synthesis with incomplete information (the complexity is 2EXPTIME-complete in both settings (Ronald Fagin and Vardi, 1995)). Essentially, the non-determinism of the automata can be used to guess the missing information, making sure that no guess violates the specification. On the other hand, coping with incomplete information is more difficult in the branching paradigm. The methods used in the linear paradigm are not applicable here, as we have to account for executability of the strategy. Therefore the synthesis problems for CTL and CTL* are complete for EXPTIME and 2-EXPTIME, respectively. Keeping in mind that the satisfiability problems for LTL, CTL, and CTL* are complete for PSPACE, EXPTIME, and 2-EXPTIME (Emerson, 1995), it follows that while the transition from closed to open systems dramatically increases the complexity of synthesis in the linear paradigm,

it does not influence the complexity in the branching paradigm.

Synthesis from shared components: agent behavior composition. An interesting observation about the general approach to automated synthesis is that it is traditionally assumed that the system is built from scratch. In the real world, this assumption is not always realistic, and it is often desirable to exploit reusable sub-systems. Indeed, many non-trivial every-day life applications are constructed composing libraries of reusable components that are not themselves part of the system, but they are rather accessed by it. As an example, this is the case of web-based service orchestration (Alonso et al., 2004; Berardi et al., 2003; Sardiña et al., 2007; Stroeder and Pagnucco, 2009) advocated for SOA (Su, 2008b), in which a web service is realized by orchestrating pre-existing web services to which one has access, but not control of. Apart from an obvious practical advantage when it comes to implementation, one fundamental aspect is that we can abstract from the details of each component. The specification thus only mentions the aspects of the components that are relevant for the synthesis of the system at large.
In particular, we will often deal, throughout this thesis, with the agent behavior composition problem, i.e., the problem of realizing a "virtual" behavior by suitably directing a set of available "concrete", i.e., already implemented, agent behaviors. It is a synthesis problem, whose solution amounts to synthesizing a controller that suitably directs the available behaviors. This problem has been studied in various areas of Computer Science, including (web) services (Balbiani et al., 2008), AI reasoning about action (Sardina et al., 2008; Stroeder and Pagnucco, 2009; De Giacomo et al., 2013), verification (Lustig and Vardi, 2009), and robotics (Bordignon et al., 2007). Many approaches (surveyed, e.g., in (ter Beek et al., 2007)) have been proposed in the last years in order to address this problem from different viewpoints. We follow here the approach proposed in (Stroeder and Pagnucco, 2009; Sardiña et al., 2007; Sardiña et al., 2008; De Giacomo et al., 2013).

Supervisory Control. At the same time in which these synthesis problems were being studied in AI, Ramadge and Wonham introduced the problem of Supervisory Control for discrete-event systems. Supervisory Control (Wonham and Ramadge, 1987; Ramadge and Wonham, 1987; 1989b; Cassandras and Lafortune, 2006) is the task of automatically synthesizing "supervisors" that restrict the behavior of a system (i.e., a discrete-event system, or DES, often referred to in the literature as the *plant*), which is assumed to spontaneously generate events, such that as much as possible of a given specifications is fulfilled. DES model a wide spectrum of physical systems, including manufacturing, traffic, logistics, and database systems. The assumption is that the overall behavior of the plant is *not* satisfactory and must be controlled. However, the events that the plant can generate are partitioned into *controllable* and *uncontrollable* ones. To that end, a supervisory action is to be imposed so as to meet a given specification on event orderings and legality of states. Supervisors observe (some of —in case of partial observability) the events executed by the plant and may choose to disable a subset of the controllable ones. Therefore, a specification is here a language over the alphabet of events, and a controllable specification is one for which there exists a supervisor that can guarantee it, i.e.,

such that the resulting *closed-loop* behavior of the supervised plant is equal to the specification language. When the given specification language is not realizable, it is possible to compute its maximal (largest) controllable sublanguage.
Hence, at least in the basic framework, the major difference wrt synthesis of reactive systems is that the systems considered in this theory are terminating, therefore the problem do not share the intrinsic complexity of LTL synthesis. Indeed, it has been shown that it is solvable in linear time in the size of the plant. Moreover, whereas a controller is able to instruct the system as to as meet the specification, a supervisor restricts instead the set of system choices –i.e., possible evolutions of the plant– but has no control over them.

Dealing with non-realizable targets in agent behavior composition. The classical behavior composition setting has been extensively investigated in the recent literature. However, one open issue has resisted principled solutions for a long time: *if the target specification is not fully realizable, is there a way to realize it "at best"?* The need for "approximations" in problem instances not admitting (easy) exact solutions was first highlighted in (Stroeder and Pagnucco, 2009) and the first attempt to define and study properties of such approximations was done in (Yadav and Sardina, 2012). Indeed, (Stroeder and Pagnucco, 2009) was the first to highlight this issue and proposed a search-based method that could eventually be adapted to compute approximate solutions "close" to the perfect one. In (Yadav and Sardina, 2012), a decision-theoretic version of the problem was proposed when solutions may not fully realize the target module, and the task is to return an alternative target closest to the original one, but fully solvable. Nonetheless, such framework deviates from the classical one, as it requires quantification of various aspects of the problem, and the solutions may bring the system to dead-end "blocking" situations. While their proposal, based on the formal notion of simulation, comes as a principled generalization of the classical framework, it did not provide ways to actually compute such solutions for the general case, but only for the special case of deterministic behaviors.

Synthesis and AI Planning. AI Planning is the field of Artificial Intelligence that is concerned with the generation of strategy, typically to be executed by an agent, given a declarative specification of the environment together with the possible actions and goals, the latter specified in terms of winning condition on the state space (Ghallab et al., 2004). There are several form of planning, and the field can be further divided by the kind of problems considered. While Classical Planning considers a single-agent deterministic setting and so it corresponds to reachability analysis in large state spaces, forms of Universal Planning involve synthesizing a reactive control program that can direct an agent toward its goal states.
It is easy to see that there is a strong connection between the areas of planning in AI and verification and synthesis of reactive programs. Actually, it has been argued that the two fields have the same subject matter, and are distinct only because of historical conditions (Kautz et al., 2006). Indeed, we can formulate the classical AI planning problem for reachability goals as to find, given a finite-state automaton A with accepting states, a word leading from the initial state to this

set. The realizability check thus correspond to check whether the language of A is non-empty, i.e. $L(A) \neq \varnothing$, whereas the synthesis task is to compute a word $L(A)$. When dealing instead with an LTL specification φ for non-terminating systems, we can define the planning domain as a Büchi automaton A_φ, and the planning task as to find an infinite word $w \in L(A_\varphi)$, i.e., such that $w \vDash \varphi$. In particular, such word w does exist iff there exist two finite words u, v such that $w = u \cdot v^\omega$ and hence one can build a finite-state program $\mathcal{C}$, i.e. a controller, that first performs u then repeats v (De Giacomo and Vardi, 1999) ("lasso" shape).

It is however natural to relax this assumption, and study the case where an agent has the capability of interacting with multiple, partially-observable, environments sharing a common interface. This is the setting of Generalized Planning for long-running goals (see, e.g., (Levesque, 2005; Srivastava et al., 2008; Bonet et al., 2009; Hu and De Giacomo, 2011)).

Multi-agent systems. The study of Multi-Agent Systems (MAS) (Wooldridge, 2009c) is concerned with the study of open, distributed systems, where the process entities (or *agents*) possess highly flexible and autonomous behaviour.

Differently from disciplines such as distributed systems and software engineering, the emphasis here is on the prominence given to concepts such as knowledge, beliefs, obligations, etc., that are used to model the agents in the system.

Since information technology is facing the task of delivering ever more complex distributed applications, MAS researchers argue that much is to be gained from an approach that focuses on high-level macroscopic characteristics of the entities, at least in the modeling stage. MAS theories involve the formal representation of agents' behaviour and attitudes. To this end, various modal logics have been studied and developed, including logics for knowledge, beliefs, actions, obligations, intentions, as well as combinations of these with temporal operators.

Data-aware processes. We focus here on processes operating on data in the context of *business processes*, i.e., those set of activities that are performed in coordination in an organizational and technical environment, which jointly realize a business goal. Business process management (BPM) (Weske, 2007) includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes.

Recent work in business processes, services and databases is bringing forward the need of considering both data and processes as first-class citizens in process and service design (Nigam and Caswell, 2003; Bhattacharya et al., 2007; Deutsch et al., 2009; Vianu, 2009). Explicitly representing data, data types, and data dependencies between activities of a business process puts a business process management system in a position to control relevant data as generated and processed during the processes execution. However, the verification of temporal properties in the presence of data represents a significant research challenge, since data makes the system infinite-state, and neither finite-state model checking (Clarke et al., 1999b) nor most of the current techniques for infinite-state model checking, which mostly tackle recursion (Burkart et al., 2001), apply to this case.

1.1 Thesis Structure and Contribution

In Chapter 3 we consider a range of agent behavior synthesis problems making use of shared components, all characterized by full observability and nondeterminism (i.e., partial controllability). In Section 3.1 we introduce one of such problems, the agent *behavior composition problem* as studied in the AI community and in particular in (Sardiña et al., 2007; Stroeder and Pagnucco, 2009; De Giacomo et al., 2013). It is the problem of checking whether a set of available, though partially controllable, behavior modules can be suitably coordinated (i.e., composed) in a way that it appears as if a desired but non-existent target behavior is being executed. Hence, its solution amounts to synthesizing a controller that suitably orchestrates all these available modules. The problem is appealing because, with computers now present in everyday devices, the trend is to build embedded complex systems from a collection of simple components. The composition problem has been indeed studied in various areas of Computer Science, including (web) services (Balbiani et al., 2009), AI reasoning about action (Sardiña et al., 2007; Stroeder and Pagnucco, 2009; De Giacomo et al., 2013), verification (Lustig and Vardi, 2009), and robotics (Bordignon et al., 2007) (see (De Giacomo et al., 2013) for an extensive review). In particular, we address this problem referring to the multi-agent setting, i.e., in which each module is represented by an agent behavior.

We show how the solution to this problem can be expressed as checking a certain safety ATL formula over a specific ATL game structure. In such game, the players represent the virtual target agent behavior, the concrete available agent behavior, and a controller, whose actual controlling strategy has yet to be defined. The players corresponding to the target and to the available agents team up togheter against the controller. The controller tries to realize the target by delegating, at each step, the action chosen by the target agent to one of the available behaviors. In doing this, the controller has to cope with the choice of the action to perform by the target agent, and the non-deterministic evolution of the available agent that has been selected to perform the action. Therefore, not any delegation is correct: even though the agent behavior is currently able to perform the action, the computation may still get "stuck" in the future, because the controller can not foresee the future target requests (the actual sequence of target actions). Solving the problem hence amounts to synthesize a strategy for one player in a multi-player game, from which we are able to extract a "composition generator". Observe that this is not different from solving a two-player ω-regular game as in the general solution schema for synthesis of linear temporal specifications, as the controller plays against an environment coalition formed by all other players.

In Section 3.2 we generalize this intuition, and show that many agent behavior synthesis problems (e.g., multi-target behavior composition (Sardina and De Giacomo, 2008), conditional planning (Rintanen, 2004a; Ghallab et al., 2004) and agent planning programs (De Giacomo et al., 2010b)) can be solved *uniformly* by model checking two-player game structures (that we call 2GS), and we introduce a variant of (modal) μ-calculus (Emerson, 1996) that allows for separately quantifying over both environment's and controller's moves. Finally, in Section 3.3, we report on a case study.

Part of this work has been published in (De Giacomo and Felli, 2010; De Giacomo et al., 2010a; 2012).

In Chapter 4 we formally relate the standard behavior composition problem to supervisory control theory in discrete-event systems, showing how composing a target module amounts to imposing a supervisor for a special discrete event system, thus establishing the formal relation between the two different synthesis tasks. Moreover, inspired by the notion of supremal controllable sublanguage, we address the case in which we are given a target behavior and a set of deterministic available behaviors such that, as often happens, there is no "exact" composition, i.e., the target cannot be completely realized in the system. There has been a recent interest in the literature to look beyond exact compositions. As it has been said, the need for "approximations" in problem instances not admitting (easy) exact solutions was first highlighted in (Stroeder and Pagnucco, 2009) and the first attempt to define and study properties of such approximations was done in (Yadav and Sardina, 2012). Adopting the notion of target approximations from (Yadav and Sardina, 2012), we show how to extract *supremal realizable target fragments* –closest to the original target module– out of adequate supervisors (rather than exact composition) for the special case of deterministic systems.

Part of this work has been submitted for publication (Felli et al., 2013b).

In Chapter 5 we present a novel technique to *effectively build* the largest realizable fragment—the "*supremal*"—of a given target specification for the general composition case in which available behaviors may be non-deterministic. The technique relies on two simple and well-known operations over transition systems (or state models), namely, cross product and belief-level state construction. In doing so, we provide an elegant result on the uniqueness of such fragments.
Then, we investigate—inspired by work on AI reasoning about action (Reiter, 2001b) and on Supervisory Control—the composition task in the presence of *exogenous events*.

Part of this work has been published in (Yadav et al., 2013).

In Chapter 6 we first resume a theoretical approach to planning for long-running (LTL) goals (De Giacomo and Vardi, 1999) in various settings, then adapt the approach also to Generalized Planning (Levesque, 2005; Bonet and Geffner, 2009) for LTL goals, showing that the solution of the generalized case is within the same complexity bound (Section 6.1). The main technical results is thus to give a sound and complete procedure for solving the generalized planning problem for LTL goals, within the same complexity bound. In particular, we show how solving the (generalized) planning problem for LTL goals amounts to synthesize a plan that can be represented finitely. However, this result does not provide any mechanism to transform such plans into finite-state controllers that can be "read" at runtime in order to retrieve the action to execute.

Therefore, in Section 6.3 we make use of the theoretical solution devised in Section 6.1 and we thus address this problem in the context of *Multi-Agent Systems* (MAS), linking the planning framework to the notion of agent protocol. Moreover, we will ground the work on Interpreted Systems (Ronald Fagin and Vardi, 1995), a popular agent-based semantics. In particular, we prove the complexity of optimal strategies based on perfect recall and, importantly, we show their reduction to finite state controllers, which can be embedded in the agent protocol, i.e., the set of rules that constrain the actions executable by the agent at each point in time.

Part of this work has been published in (Felli et al., 2012).

In Chapter 7 we address a further kind of synthesis from shared components in an multi-agent setting. Specifically, we consider programs written in the the family of Belief-Desire-Intention (BDI) agent programming systems (Bratman et al., 1988; Rao and Georgeff, 1992; Bordini et al., 2006). General BDI is a conceptual framework, not a specific implementation; however, it is considered a popular and successful approach for building agent systems (Menzies et al., 2003). An agent in a BDI system continually tries to achieve its goals/desires by selecting an adequate plan from the *plan library* given its current beliefs, and placing it into the *intention base* for execution. The agent's plan library Π encodes the standard operational knowledge of the domain by means of a set of *plan-rules* (or "recipes") of the form $\phi[\vec{\beta}]\psi$: *plan* $\vec{\beta}$ *is a reasonable plan for achieving* ψ *when (context) condition* ϕ *is believed true*. Most BDI-style programming languages come with a clear single-step semantics basically realizing (Rao and Georgeff, 1992)'s execution model in which (rational) behavior arises due to the execution of plans from the agent's plan library so as to achieve certain goals relative to the agent's beliefs. Hence we address here the problem of synthesizing rational strategies for BDI agents, in which know-how and motivations can be assigned to each agent by means of plan libraries and goals. Other approaches –e.g. (Alechina et al., 2007; 2008; Dastani and Jamroga, 2010)– restrict logic models to those conforming to agents' capabilities, therefore it is not possible to reason about the agent's know-how or what the agent could achieve *if it had* specific capabilities, i.e., access to a different set of plan libraries. Instead, we make use of the "ATL-like" BDI-ATELS logic introduced in (Yadav and Sardiña, 2012), that makes explicit such assignments in the language. The key construct $\langle\langle A\rangle\rangle_{\omega,\varrho}\varphi$ in the new framework states that coalition A has a joint strategy for ensuring φ, under the assumptions that some agents in the system are BDI-style agents with capabilities and (initial) goals as specified by assignments ω and ϱ, respectively. We extend here the work in (Yadav and Sardiña, 2012) to consider plan libraries with action sequences (not only atomic plans) and we present a reduction to standard ATL model checking.

Part of this work has been submitted for publication (Felli et al., 2013a).

In Chapter 8 we briefly comment on the work done in the context of verification for data-aware processes, what does this mean and which are the major difficulties involved when we turn to the synthesis task. Since this work is (at this stage) focused

on decidability issues of verification, and since a complete, thoughtful exposition would require an extensive tractation, details are omitted.

The reader is referred to the list of related publications, i.e., (Bagheri Hariri et al., 2011; Hariri et al., 2012; Bagheri Hariri et al., 2013).

Conclusions, discussions and related work are present in each chapter.

Chapter 2

Preliminaries

In this chapter we give the main definitions and notations that will be used throughout the entire thesis. First, we introduce transition systems, which are used to model agent behaviors, then we briefly overview the temporal logics that are needed for the technical development of the following chapters.

2.1 Transition systems

Kripke structures are finite state machines often used in computer science as models to describe the behavior of systems. They are basically directed graphs where nodes represent states, and edges model transitions, i.e., state changes. A state describes some information about a system at a certain moment of its evolution.

Definition 2.1. A *Kripke structure* is a tuple $M = \langle S, S_0, \delta, AP, \mathcal{V} \rangle$ where:

- S is a set of states;
- $S_0 \subseteq S$ is the set of initial states;
- $\delta : S \times S$ is a (left-total) transition relation;
- AP is a set of atomic propositions,;
- $\mathcal{V} : S \to 2^{AP}$ is a labeling function.

△

M is finite if S and AP are finite. Initially, M starts in some initial state $s_0 \in S_0$ and evolves according to the transition relation. If $\langle s, s' \rangle \in \delta$ then s' is a successor of s. The labeling function $\mathcal{V}$ is an interpretation function, i.e., it relates a set $\mathcal{V}(s) \in 2^{AP}$ of atomic propositions to any state s. Therefore, given a propositional formula φ over AP, we say that a state s satisfies φ iff the evaluation induced by $\mathcal{V}(s)$ makes φ true, namely $s \vDash \varphi$ iff $\mathcal{V}(s) \vDash \varphi$.
A *path* or *computation* of a Kripke structure M is a infinite sequence of states $\lambda = s_0, s_1, \ldots$ starting at some $s_0 \in S_0$ such that s_{i+1} is a successor of s_i for any $i \geq 0$. A *word* corresponding to a path λ is instead the sequence of state labels $word(\lambda) = \mathcal{V}(s_0), \mathcal{V}(s_1), \ldots$. Hence, we will write $Paths(s)$ to denote the set of

paths in M starting from s; $Words(s) = \{word(\lambda) \mid \lambda \in Paths(s)\}$ and $Words(M) = \bigcup_{s \in S} Words(s)$.

Finally, a Transition System is essentially a Kripke structure extended with actions.

Definition 2.2. A *transition system* is tuple $T = \langle S, Act, S_0, \delta, AP, \mathcal{V} \rangle$ where:

- S is a set of states;
- Act is a finite set of actions;
- $S_0 \subseteq S$ is the set of initial states;
- $\delta : S \times Act \times S$ is a transition relation;
- AP is a set of atomic propositions;
- $\mathcal{V} : S \to 2^{AP}$ is a labeling function.

△

Hence, if s is the current state, upon performing an action α the transition system evolves by *non-deterministically* selecting a transition $\langle s, \alpha, s' \rangle \in \delta$ (also denoted $s \xrightarrow{\alpha} s'$), and s' is said to be a successor (or α-successor) of s. If δ is a function, the T is said to be *deterministic*.

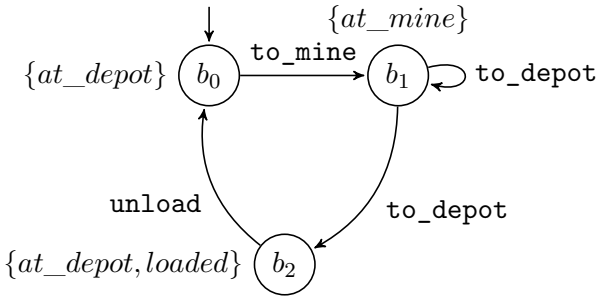


Figure 2.1. A TS modeling the behavior of a mining truck.

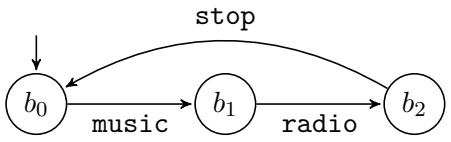


Figure 2.2. A TS modeling an audio device.

Example 1. *Consider Figure 2.1. It depicts a transition system modeling a truck in a mining scenario.* $AP = \{at\_depo, at\_mine, loaded\}$. *The initial state is* b_0 *(marked with an incoming arrow) and the proposition* $\mathcal{V}(b_0) = \{at\_depo\}$ *is used to keep track of the physical position of the truck. By performing action* `to_mine`*, the truck moves from state* b_0 *to state* b_1*, i.e.,* $\langle b_0, \texttt{to\_mine}, b_1 \rangle \in \delta$. *Similarly,*

by executing action `to_depot`*, the truck moves either to* b_2 *or stays in* b_1 *(i.e.,* $\{\langle b_1, \texttt{to\_depot}, b_1\rangle, \langle b_1, \texttt{to\_depot}, b_2\rangle\} \in \delta$*), and so on.*
Figure 2.2 shows instead the behavior of an audio device. It allows to play music or to listen to radio, but the execution of these actions is constrained by the structure of the transition system, so that, e.g., only `music` *is available from* b_0*. Note that, as often happens, we do not need any state proposition for modeling our behavior:* $AP = \varnothing$ *and thus* $\mathcal{V}(s) = \varnothing$ *for any* $s \in S$*. When this is the case, we will omit for brevity* AP *and* $\mathcal{V}$*. Finally, note that the truck behavior is non-deterministic, whereas the audio device is deterministic.* □

An *trace* (or *execution fragment*) τ of length n of T is an alternating sequence of states and actions ending with a state $\tau = s_0\alpha_1 s_1 \alpha_2 \cdots \alpha_n s_n$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i < n$. It can be either finite or infinite. A (maximal) *execution* of T is therefore either a finite execution fragment that ends in a state with no successors, or an infinite execution fragment.
The notion of path and words are analogous to the ones for Kripke structures: in this thesis we will call *path* or *computation* of a transition system T is a infinite sequence of states $\lambda = s_0, s_1, \ldots$ starting at some $s_0 \in S_0$ such that s_{i+1} is a successor of s_i for any $i \geq 0$. A *word* corresponding to a path λ is instead the sequence of state labels $word(\lambda) = \mathcal{V}(s_0), \mathcal{V}(s_1), \ldots$.

Example 2. *Referring to the transition system depicted in Figure 2.1, a possible trace is the sequence* $\tau = b_0\texttt{to\_mine}b_1\texttt{to\_depot}b_1\texttt{to\_depot}b_2 \ldots$*, with path* $\lambda = b_0, b_1, b_1, b_2, \ldots$ *. The corresponding word over the proposition alphabet would be* $word(\lambda) = \{at\_depo\}, \{at\_mine\}, \{at\_mine\}, \{at\_depo, loaded\}, \ldots$. □

2.2 Linear Temporal Logic

Linear temporal logic (LTL) was introduced in (Pnueli, 1977) for the specification and verification of reactive systems. LTL is called linear, because the qualitative notion of time is path-based and viewed to be linear: at each moment in time there is only one possible successor state and thus each time moment has a unique possible future. Technically speaking, this follows from the fact that the interpretation of LTL formulae is defined in terms of paths, i.e., sequences of states.
The basic ingredients of LTL-formulae are atomic propositions, the usual operators $\wedge, \neg$ and two basic temporal modalities $\bigcirc$ (next) and $\mathcal{U}$ (until).
LTL formulae over the set AP of atomic proposition are formed according to the following grammar:

$$\varphi := \texttt{true} \mid p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

where $p \in AP$ are atomic proposition $p \in AP$ standing for the state label a in a transition system. Typically, the atoms are assertions about the values of either control or program variables. We also use the usual boolean abbreviations.
Informally, $\bigcirc\varphi$ holds at the current moment iff φ holds in the next "step" (successor state), whereas $\varphi_1 \mathcal{U} \varphi_2$ holds at the current moment if there is some future moment

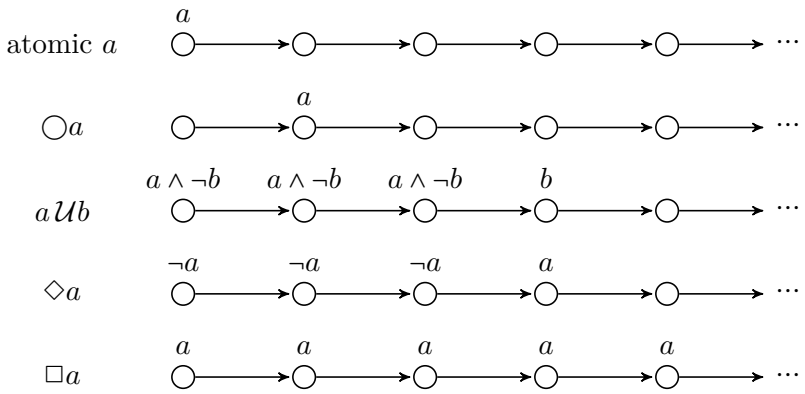


Figure 2.3. Intuitive semantics of temporal modalities on words.

for which φ_2 holds and φ_1 holds at all moments until then. Moreover, we define the derived temporal modalities $\Diamond$ (eventually) and $\Box$ (always) as follows:

$$\Diamond\varphi := \texttt{true}\,\mathcal{U}\varphi \qquad \Box\varphi := \neg\Diamond\neg\varphi$$

Hence, the intuitive meaning of $\Diamond\varphi$ is to ensures that φ will be true eventually in the future whereas $\Box\varphi$ is satisfied if and only if φ holds forever.
LTL formulae stand for properties of words over AP. This means that a path can either fulfill an LTL-formula or not. To precisely formulate when a path satisfies an LTL formula, we proceed as follows. First, we regard an LTL formula φ as a language $Words(\varphi)$ that contains all infinite words over the alphabet 2^{AP} that satisfy φ. Then, the semantics is extended to an interpretation over paths and states of a transition system.

Semantics over words in $(2^{AP})^\omega$. The satisfaction relation for infinite words is defined as follows. For $w = A_0, A_1, \cdots \in (2^{AP})^\omega$, $w[j, \infty] = A_j, A_{j+1}, \cdots$ denotes the suffix of w starting at the j-th element. The intuitive semantics is represented in Figure 2.3. Formally:

- $w \vDash \texttt{true}$
- $w \vDash p$ iff $p \in A_0$ (i.e. $A_0 \vDash p$)
- $w \vDash \varphi_1 \wedge \varphi_2$ iff $w \vDash \varphi_1$ and $w \vDash \varphi_2$
- $w \vDash \neg\varphi$ iff $w \nvDash \varphi$
- $w \vDash \bigcirc\varphi$ iff $w[1, \infty] \vDash \varphi$
- $w \vDash \varphi_1 \mathcal{U} \varphi_2$ iff $\exists j \geq 0.\ w[j, \infty] \vDash \varphi$ and $w[i] \vDash \varphi$ for all $0 \leq i < j$
- $w \vDash \Diamond\varphi$ iff $\exists j \geq 0.\ w[j, \infty] \vDash \varphi$ (derived)
- $w \vDash \Box\varphi$ iff $\forall j \geq 0.\ w[j, \infty] \vDash \varphi$ (derived)

Hence, consider an LTL formula φ over 2^{AP}. The set of words (the *LT property*) induced by φ is the set $Words(\varphi) = \{w \in (2^{AP})^\omega \mid w \vDash \varphi\}$.

Semantics over paths and states. As a subsequent step, we determine the semantics of LTL-formulae with respect to paths and states of a Kripke structure $M = \langle S, S_0, \delta, AP, \mathcal{V} \rangle$. Hence given an LTL formula φ, a path λ of M and a state $s \in S$:

$$\lambda \vDash \varphi \quad \text{iff} \quad word(\lambda) \vDash \varphi$$

$$s \vDash \varphi \quad \text{iff} \quad \forall \lambda \in Paths(s).\, \lambda \vDash \varphi$$

Hence the formula φ is true in M, i.e., $M \vDash \varphi$, iff $Words(M) \subseteq Words(\varphi)$ iff $\lambda \vDash \varphi$ for any $\lambda \in Paths(M)$. Notably, this implies that $M \vDash \varphi$ iff $s_0 \vDash \varphi$ for any initial state $s_0 \in S_0$ of M.
The problem of model-checking for LTL is PSPACE-complete.

Example 3. *Consider again Figure 2.1. The LTL formula* $\varphi_1 = \Box\Diamond loaded$ *(requiring that the truck visits infinitely often the depot) is false in* b_0*, as the transition system can remain forever in state* b_1 *due to the nondeterminism of action* `to_depot`*. Hence, there exists an infinite number of paths* $\lambda \in Paths(b_0)$ *such that* $\lambda \nvDash \varphi_1$. □

2.3 Branching Temporal Logic

We have seen how the interpretation of LTL formulae is defined in terms of paths, i.e., sequences of states. In turn, paths are obtained from a Kripke structure that might be branching, i.e., a state can have multiple successor for the same action. Consequently, several computations may start in a state. However, the interpretation of LTL-formulae in a state requires that a formula φ holds in state s if *all* possible computations that start in s satisfy it, but we can not quantify existentially over them. In other words, LTL assumes an implicit *universal* quantification over paths.
The temporal operators in branching temporal logic (CTL) (Emerson and Halpern, 1986; Lamport, 1980) allow instead the expression of properties of some or all computations starting from a state. To that end, it supports an *existential* path quantifier (denoted $\exists$) and an explicit universal path quantifier (denoted $\forall$) such that the formula $\exists\varphi$ denotes that there exists a computation along which φ holds, whereas $\forall\varphi$ denotes that φ holds in all possible paths.
The most interesting point is that we can "nest" quantifiers as to as express complex requirements such as $\forall\Box\exists\Diamond\Phi$: in any state ($\Box$) of any possible computation ($\forall$), there exists a possibility ($\exists$) to eventually ($\Diamond$) meet Φ.
While LTL formulas are *path* formulas, in the sense that their semantics is defined over paths, CTL formulas are divided into path formulas (here denoted by Φ) and *state* formulas (here denoted by φ).
CTL state formulas over the set AP of atomic proposition are formed according to the following grammar:

$$\Phi := \texttt{true} \mid p \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

whereas path formulas are formed according to the grammar:

$$\varphi := \bigcirc\Phi \mid \Phi_1 \,\mathcal{U}\, \Phi_2$$

where Φ, Φ_1, Φ_2 are state formulas. Intuitively, a state formula captures a property of a state, while path formulas capture a property of a (infinite) path. Similarly to LTL, $\exists\Diamond\Phi = \exists(\texttt{true}\mathcal{U}\Phi)$, $\forall\Diamond\Phi = \forall(\texttt{true}\mathcal{U}\Phi)$, $\exists\Box\Phi = \neg\forall\Diamond\neg\Phi$ and $\forall\Box\Phi = \neg\exists\Diamond\neg\Phi$.

Semantics (over paths). As for LTL, also CTL formulas are interpreted over the states and paths of a Kripke structure $M = \langle S, S_0, \delta, AP, \mathcal{V}\rangle$. Formally the semantics of CTL formulas is defined by two satisfaction relations (both denoted by $\models$): one for the state formulas and one for the path formulas. Hence, a state $s \in S$ satisfies a state formula Φ according to:

- $s \models p$ iff $p \in L(s)$
- $s \models \neg\Phi$ iff $s \not\models \Phi$
- $s \models \Phi_1 \wedge \Phi_2$ iff $s \models \Phi_1$ and $s \models \Phi_2$
- $s \models \exists\varphi$ iff $\exists\lambda \in Paths(s).\ \lambda \models \phi$
- $s \models \forall\varphi$ iff $\forall\lambda \in Paths(s).\ \lambda \models \phi$

For the path formulas, $\models$ is a relation between (infinite) paths in M and path formulas. Given a path λ, we denote with $\lambda[j]$ the j-th state of λ.

- $\lambda \models \bigcirc\Phi$ iff $\lambda[1] \models \Phi$
- $\lambda \models \Phi_1 \mathcal{U} \Phi_2$ iff $\exists j \geq 0.\ (\lambda[j] \models \Phi_2 \wedge (\forall 0 \leq i < j.\lambda[i] \models \Phi_1))$
- $\lambda \models \Diamond\Phi$ iff $\exists j \geq 0.\ \pi[j] \models \Phi$ (derived)
- $\lambda \models \Box\Phi$ iff $\forall j \geq 0.\ \pi[j] \models \Phi$ (derived)

The interpretations for atomic propositions, negation, and conjunction are as usual. Hence we say that M satisfies a CTL formula Φ iff Φ holds in all its initial states, i.e.,

$$M \models \Phi \quad \text{iff} \quad \forall s_0 \in S_0.\ s_0 \models \Phi$$

Example 4. *Consider once more Figure 2.1. We already noted that the LTL formula $\varphi_1 = \Box\Diamond at\_depot$ is false for the transition system (see Example 3). Now consider the CTL formula $\varphi_2 = \forall\Box\exists\Diamond loaded$. It is easy to check that it is true in the model, as it only requires the possibility of achieving $\Diamond loaded$ for any path.* □

2.4 Alternating Temporal Logic

ATL (Alternating-time Temporal Logic) (Alur et al., 2002) is a logic whose interpretation structures are multi-player game structures where players can cooperate or confront each other so as to satisfy certain formulae. Technically, ATL is quite close to CTL, with which it shares excellent model checking techniques (Clarke et al., 1999d; Baier and Katoen, 2008). Differently from CTL, when an ATL formula is satisfied then it means that there exists a strategy, for the players specified in the formula, that fullils the temporal/dynamic requirements in the formula.

Therefore, Alternating-time Temporal Logic (Alur et al., 2002) is a logic that can predicate on moves of a game played by a set of players. For example, let $\mathcal{A}$ be the set of players and $A \subseteq \mathcal{A}$ a subset of them, then the ATL formula $\langle\langle A\rangle\rangle\varphi$ asserts (intuitively) that there exists a *strategy* for players in A to satisfy the state predicate φ irrespective of how players in $\mathcal{A}\backslash A$ evolve. The temporal operators are "$\Diamond$" (eventually), "$\Box$" (always), "$\bigcirc$" (next) and "$\mathcal{U}$" (until). The ATL formula $\langle\langle p1, p2\rangle\rangle\Diamond\varphi$ captures the requirement "players $p1$ and $p2$ can cooperate to eventually make φ true". This means that there exists at a winning strategy that $p1$ and $p2$ can follow to force the game to reach a state where φ is true.
ATL formulae are constructed inductively as follows:

- p, for propositions $p \in AP$ are ATL formulae;
- $\neg\varphi$ and $\varphi_1 \vee \varphi_2$ where φ, φ_1 and φ_2 are ATL formulae, are ATL formulae;
- $\langle\langle A\rangle\rangle\bigcirc\varphi$ and $\langle\langle A\rangle\rangle\Box\varphi$ and $\langle\langle A\rangle\rangle\varphi_1\,\mathcal{U}\varphi_2$, where $A \subseteq \mathcal{A}$ is a set of players and φ, φ_1 and φ_2 are ATL formulae, are ATL formulae.

We also use the usual boolean abbreviations.
Differently from LTL and CTL, ATL formulae are interpreted over *concurrent game structures*: every state transition of a concurrent game structure results from a set of moves, one for each player.

Definition 2.3. A *Concurrent Game Structure* is a tuple $\mathcal{M} = \langle \mathcal{A}, Q, AP, \mathcal{V}, d, \delta\rangle$ where:

- $\mathcal{A}$ is a non-empty, finite set of players. $|\mathcal{A}| = k$, and players are identified with numbers in $\{1, \ldots, k\}$.
- Q is a finite set of (global) states of the system.
- AP is a finite set of atomic propositions.
- $\mathcal{V} : Q \to Pwr(AP)$ is a mapping that specifies which propositions are true in which states.
- $d : Ag \times Q \to \mathbb{N}$ is a function specifying how many move options each player has at a particular state. Namely, for each player $a \in Ag$ and each state $q \in Q$, $d_a(q) \geq 1$ is the number of moves available to a in q. Each of these moves is identified with a natural number, thus agent a can choose his decision from the set $\{1, \ldots, d_a(q)\}$. For each state $q \in Q$, we write $D(q)$ for the set $\{1, \ldots, d_1(q)\} \times \cdots \times \{1, \ldots, d_k(q)\}$ of *move vectors* (tuples of decisions, one for each player). The function D is called *move function*.
- δ is the transition function. For each state $q \in Q$ and each move vector $\langle j_1, \ldots, j_k\rangle \in D(q)$, the successor state $q' = \delta(q, j_1, \ldots, j_k) \in Q$ results from state q if every player $i \in Ag$ chooses move j_i.

$\triangle$

If $n = |Q|$ is the number of states and $m = \sum_{q \in Q} d_1(q) \times \cdots \times d_k(q)$ is the number of transitions in $\mathcal{M}$, then m is not bounded by n^2 as in Kripke structures, and the size of $\mathcal{M}$ is $O(m)$.

It is important to distinguish between the computational structure, defined explicitly in the model, and the behavioral structure, i.e. the model of how the system is supposed to behave in time. Indeed, the computational structure is finite, whereas the implied behavioral structure is infinite. In ATL, the finite state automaton lying at the core of every concurrent game structure can be seen as a way of imposing the tree of possible (infinite) computations that may occur in the system.

Definition 2.4. A *computation* of $\mathcal{M}$ is an infinite sequence $\lambda = q_0, q_1, q_2 \ldots$ of states such that for each $i \geq 0$, the state q_{i+1} is a successor of q_i, i.e., there exists a move vector $\langle j_1, \ldots, j_k \rangle \in D(q_i)$ such that $q_{i+1} = \delta(q_i, j_1, \ldots, j_k)$. △

It should be noted that the earliest the version only includes definitions for a *synchronous* turn-based structure and an *asynchronous* structure in which every transition is owned by a single agent. The concurrent game structure above is then a successiv version as it appears in (Alur et al., 2002). Indeed, different versions of ATL have been successively proposed, each with a slightly different definition of the semantic structure. Notably, a precedent work offers more general structures (called *alternating transitions systems*) with no action labels and a more sophisticated transition function. While in ordinary transition systems, each transition corresponds to a possible step of the system, in alternating transition systems each transition corresponds to a possible move in the game between the system and the environment.

Definition 2.5. An *Alternating Transition System* is a tuple $S = \langle \mathcal{A}, Q, AP, \mathcal{V}, \delta \rangle$ where:

- $\mathcal{A}, Q, AP, \mathcal{V}$ are as above;
- $\delta : Q \times Ag \to 2^{2^Q}$ is a transition function that maps a state and an agent to a nonempty set of choices, where each choice is a set of possible next states. Whenever the system is in state q, each agent a chooses a set $Q_a \in \delta(q, a)$. In this way, an agent a ensures that the next state of the system will be among its choice Q_a. However, which state in Q_a will be next depends on the choices made by the other agents, because the successor of q must lie in the intersection $\bigcap_{a \in Ag} Q_a$ of the choices made by all agents. The transition function is non-blocking and the agents together choose a unique next state state. Thus, it is required that this intersection always contains a unique state: for every state $q \in Q$ and every set $Q_1, \ldots, Q_k$ of choices $Q_a \in \delta(q, a)$, the intersection $Q_1 \cap \ldots \cap Q_k$ is a singleton.

△

In general, both kinds of semantics are equivalent (Goranko and Jamroga, 2004), but the concurrent game structures are smaller and easier to read in most cases (Jamroga, 2004).

Once the notion of successor is given, we can provide a formal definition of winning strategy.

Definition 2.6. Given a concurrent game structure $\mathcal{M}$ as above, a *strategy* for player $a \in \mathcal{A}$ is a function f_a that maps every non-empty finite state sequence $\lambda \in Q^+$ to one of its moves, i.e., a natural number such that if the last state of λ is q then $f_a(\lambda) \leq d_a(q)$. △

The strategy f_a determines, for every finite prefix λ of a computation, a move $f_a(\lambda)$ for player a. Hence, a strategy f_a induces a set of computations that player a can enforce. Given a state $q \in Q$, a set $A \subseteq \{1, \ldots, k\}$ of players, and a set $F_a = \{f_a \mid a \in A\}$ of strategies, one for each player in A, we define the *outcomes* of F_a from q to be the set $\mathsf{out}(q, F_A)$ of q-*computations* that the players in A collectively can enforce when they follow the strategies in F_A. A computation $\lambda = q0, q1, q2, \ldots$ is then in $\mathsf{out}(q, F_A)$ if $q_0 = q$ and for all positions $i > 0$ every player a follows the strategy f_a to reach the state q_{i+1}, that is, there is a move vector $\langle j_1, \ldots, j_k \rangle \in D(q_i)$ such that $j_a = f_a(\lambda[0, i])$ for all players $a \in A$, and $\delta(q_i, j_1, \ldots, j_k) = q_{i+1}$.
Now we can provide a formal definition of the satisfaction relation: we write $\mathcal{M}, q \models \varphi$ to indicate that the state q satisfies formula φ with respect to game structure $\mathcal{M}$. $\models$ is defined inductively as follows:

- $q \models p$, for propositions $p \in P$, iff $p \in \mathcal{V}(q)$.
- $q \models \neg\varphi$ iff $q \not\models \varphi$.
- $q \models \varphi_1 \vee \varphi_2$ iff $q \models \varphi_1$ or $q \models \varphi_2$.
- $q \models \langle\langle A \rangle\rangle \bigcirc \varphi$ iff there exists a set F_A of strategies, one for each player in A, such that for all computations $\lambda \in \mathsf{out}(q, F_A)$, we have $\lambda[1] \models \varphi$.
- $q \models \langle\langle A \rangle\rangle \, \Box \, \varphi$ iff there exists a set F_A of strategies, one for each player in A, such that for all computations $\lambda \in \mathsf{out}(q, F_A)$ and all positions $i \geq 0$, we have $\lambda[i] \models \varphi$.
- $q \models \langle\langle A \rangle\rangle (\varphi_1 \mathcal{U} \varphi_2)$ iff there exists a set F_A of strategies, one for each player in A, such that for all computations $\lambda \in \mathsf{out}(q, F_A)$, there exists a position $i \geq 0$ such that $\lambda[i] \models \varphi_2$ and for all positions $0 \leq j < i$, we have $\lambda[j] \models \varphi_1$.

As for operator $\Diamond$ (eventually), we observe that $\langle\langle A \rangle\rangle \Diamond \varphi$ is equivalent to $\langle\langle A \rangle\rangle (true \mathcal{U} \varphi)$.
Concerning computational complexity, the cost of ATL model-checking is linear in the size of the game structure, as for CTL, a very well-known temporal logic used in model checking(Clarke et al., 1999c), of which ATL is an extension.

Example 5. *Imagine a mining scenario in which there are two agents: a truck Ag_r and a crane Ag_c. The truck (like the one depicted in Figure 2.1) can move between a depot and a mine, where a crane is used to load it with materials. When the truck is at the mine, the crane can either load or wait; when the truck has been loaded and at the depot, it can unload its load. We can model this simple scenario with a Concurrent Game Structure $\mathcal{M}ine = \langle \mathcal{A}, Q, AP, \mathcal{V}, d, \delta \rangle$ (see Figure 2.4, where understandable action labels are used instead of integers) where:*

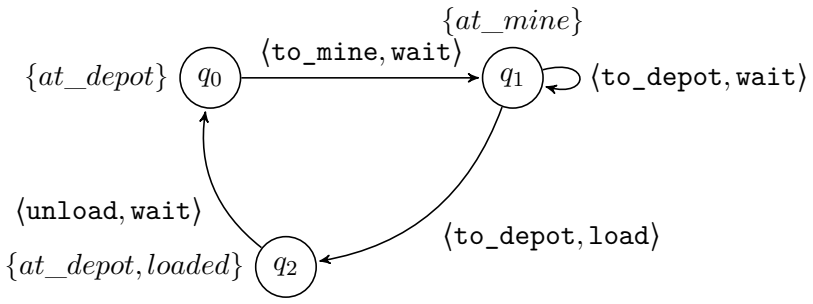


Figure 2.4. Game structure $\mathcal{M}ine$.

- $\mathcal{A} = \{Ag_r, Ag_c\}$, $k = 2$;
- $Q = \{q_0, q_1, q_2\}$;
- $AP = \{at\_mine, at\_depot, loaded\}$;
- $\mathcal{V}$ *is such that* $\mathcal{V}(q_0) = \{at\_depot\}$, $\mathcal{V}(q_1) = \{at\_mine\}$ *and* $\mathcal{V}(q_2) = \{at\_depot, loaded\}$;
- $d(1, q_0) = d(2, q_0) = d(1, q_1) = d(2, q_2) = d(1, q_1) = 1$ *and* $d(2, q_1) = 2$.
- $\delta(q_0, 1, 1) = \delta(q_1, 1, 1) = q1$; $\delta(q_1, 1, 2) = q2$; $\delta(q_2, 1, 1) = q0$.

Similarly to previous examples, the ATL formula $\varphi_1 = \langle\langle Ag_r \rangle\rangle \Diamond loaded$ *is false for* $q_0 \in Q$*: the truck can not force the game to reach state* q_2 *unless the crane performs action* 2 (`load`) *from* q_1*. On the other hand, both* $\varphi_2 = \langle\langle Ag_r, Ag_c \rangle\rangle \Diamond loaded$ *and* $\varphi_3 = \langle\langle Ag_c \rangle\rangle \Box \neg loaded$ *are true in* q_0*, i.e.,* $\mathcal{M}, q_0 \vDash \varphi_2$ *and* $\mathcal{M}, q_0 \vDash \varphi_3$. □

2.5 Modal μ-calculus

Formulas of the μ-calculus (Emerson, 1996) are basically constituted by three kinds of components: (*i*) propositions to denote properties of the global store in a given configuration (*ii*) modalities to denote the capability of performing certain actions in a given configuration (*iii*) least and greatest *fixpoint constructs* to denote "temporal" properties of the system, typically defined by induction and coinduction.
Formulas of μ-calculus are formed inductively from action in some fixed set Act, primitive (or atomic) proposition set AP, and *variable symbols* in some fixed set Var, according to the following abstract syntax:

$$\varphi := p \mid \texttt{true} \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle a \rangle \varphi \mid [a]\varphi \mid \mu X.\varphi \mid \nu X.\varphi \mid X$$

where p is a primitive proposition in AP, X is a variable symbol in Var, and a is an action in Act. The symbols μ and ν can be considered as quantifiers, and we make use of notions of scope, bound and free occurrences of variables, closed formulas, etc. The definitions of these notions are the same as in first-order logic, treating μ and ν as quantifiers.
Hence, for formulas of the form $\mu X.\varphi$ and $\nu X.\varphi$ we require the *syntactic* monotonicity of φ wrt X, i.e., every occurrence of X in φ must be within the scope af

an even number of negations. This is essential to guaranteed, by Tarski-Knaster theorem, the existence of least and greatest fixpoints.

Semantics. The semantics of μ-calculus is based on the notions of transition system and valuation of variables.
Given a transition system $T = \langle S, Act, S_0, \delta, AP, \mathcal{V}\rangle$, a valuation Π is a mapping from variables in Var to subsets of the states in T.
Given a valuation Π, we denote by $\Pi[X \leftarrow \mathcal{E}]$, the valuation identical to Π except for $\Pi[X \leftarrow \mathcal{E}](X) = E$, i.e. for every variable Y we have

$$\Pi[X \leftarrow \mathcal{E}](Y) = \begin{cases} \mathcal{E} & \text{if } Y = X \\ \Pi(Y) & \text{if } Y \neq X \end{cases}$$

We can now define the *extension function* $[\![\cdot]\!]^T_\Pi$ mapping formulas to states of S as follows (to ease the notation, T and Π are skipped where understood):

$$\begin{array}{ll}
[\![p]\!] & = \{s \in S \mid p \in \mathcal{V}(s)\} \\
[\![X]\!] & = \Pi(X) \subseteq S \\
[\![\texttt{true}]\!] & = S \\
[\![\texttt{false}]\!] & = \varnothing \\
[\![\neg\varphi]\!] & = S \setminus [\![\varphi]\!] \\
[\![\varphi_1 \wedge \varphi_2]\!] & = [\![\varphi_1]\!] \cap [\![\varphi_2]\!] \\
[\![\varphi_1 \vee \varphi_2]\!] & = [\![\varphi_1]\!] \cup [\![\varphi_2]\!] \\
[\![\langle a\rangle\varphi]\!] & = \{s \in S \mid \exists s'.\langle s, a, s'\rangle \in \delta \text{ and } s' \in [\![\varphi]\!]\} \\
[\![[a]\varphi]\!] & = \{s \in S \mid \forall s'.\langle s, a, s'\rangle \in \delta \text{ implies } s' \in [\![\varphi]\!]\} \\
[\![\mu X.\varphi]\!] & = \bigcap\{\mathcal{E} \subseteq S \mid [\![\varphi]\!]_{\Pi[X\leftarrow\mathcal{E}]} \subseteq \mathcal{E}\} \\
[\![\mu X.\varphi]\!] & = \bigcup\{\mathcal{E} \subseteq S \mid \mathcal{E} \subseteq [\![\varphi]\!]_{\Pi[X\leftarrow\mathcal{E}]}\}
\end{array}$$

Intuitively, the extension function $[\![\cdot]\!]^T_\Pi$ assigns to the constructs of μ-calculus the following meanings:

- the boolean connectives have the expected meaning;
- the extension of $\langle a\rangle\varphi$ includes the states $s \in S$ such that starting from s, there is an a-successor $s' \in [\![\varphi]\!]$;
- the extension of $[a]\varphi$ includes the states s such that any a-successor $s' \in [\![\varphi]\!]$;
- the extension of $\mu X.\varphi$ is the smallest subset $\mathcal{E} \subseteq S$ such that, assigning X to the extension $\mathcal{E}$, the resulting extension of φ is in $\mathcal{E}$. Namely, the extension of $\mu X.\varphi$ is the *least fixpoint* of the operator $\Omega\mathcal{E}.[\![\varphi]\!]^T_{\Pi[X\leftarrow\mathcal{E}]}$. Similarly $\nu X.\varphi$ captures the *greatest fixpoint*.

Note also that if φ is closed (no free variables are present) then the extension of $[\![\varphi]\!]^T_\Pi$ is in fact independent of the valuation Π, hence we can also write $[\![\varphi]\!]^T$. Finally, we say that a closed formula φ is true in $s \in S$ iff $s \in [\![\varphi]\!]^T$.

Example 6. *The formula*

$$\varphi_\Diamond = \mu X.\mathit{loaded} \vee (\langle -\rangle\texttt{true} \wedge [-]X)$$

expresses that for all evolutions of the system, loaded eventually holds. Indeed, its extension $\mathcal{E}$ is the smallest set that includes (1) the states in the extension of loaded and (2) the states that can actually make a transition (we use "$-$" to denote any action in Act) such that every transition leads to a state in $\mathcal{E}$ ($\langle - \rangle \mathtt{true}$ is used here to check that a successor state actually exists). In other words, the extension $\mathcal{E}$ includes each state s such that every run from s leads eventually (i.e. in a finite number of steps) to a state in the extension of loaded. Therefore, checking this formula on the transition system of Figure 2.1 will give us $[\![\varphi_\Diamond]\!] = \{b_2\}$ □

Chapter 3

Synthesis via Game Structures

AI has been long concerned with agent behavior synthesis problems. In this chapter we consider a variety of agent behavior synthesis problems characterized by *full observability* and *non-determinism* (i.e., partial controllability). In Section 3.1 we introduce one of such problems, the *agent behavior composition*, and show how it can be expressed as checking a certain ATL formula over a specific ATL game structure. In particular, we show how the problem can be reduced to checking the existence of a strategy for a player (hence synthesizing it) in a multi-agent setting. In Section 3.2 we then generalize this intuition, and show that many agent behavior synthesis problems can be solved by model checking *two-player* game structures, hence distinguishing between the actions/moves of two antagonistic players: the system and the environment.
Specifically, we focus on three problems of increasing sophistication. The simplest problem we consider is the standard *conditional planning* in non-deterministic fully observable domains (Rintanen, 2004a), which is well understood by the AI community. Then, we move to a sophisticated form of planning recently introduced in (De Giacomo and Felli, 2010), in which so-called *agent planning programs*—programs built only from achievement and maintenance goals—are meant to merge two traditions in AI research, namely, Automated Planning and Agent-Oriented Programming (Wooldridge, 2009b). Solving, that is, realizing, such planning programs requires temporally extended plans that loop and possibly do not even terminate, analogously to (Kerjean et al., 2006). Finally, we turn to and advanced form of agent behavior composition. Moreover, by exploiting the connections between such game structures and the interpretation structures for ATL, we also provide implementations of the solution techniques for the above problems by relying on a suitable use of an ATL model checker.

3.1 Agent Behavior Composition via ATL

Agent behavior composition is the problem of realizing a "virtual" agent by suitably directing a set of available "concrete", i.e., already implemented, agents. It is a synthesis problem, whose solution amounts to synthesizing a controller that suitably directs the available agents.

In this section, we show that agent composition can be solved by ATL model checking. ATL (see Section 2.4) has been widely adopted by the Agents community since it allows for naturally specifying properties of societies of agents (Wooldridge, 2009a; Lomuscio and Raimondi, 2006). The interest of the Agents community has led to active research on specific model checking tools for ATL, which by now are among the best model checkers for verification of temporal properties (Lomuscio et al., 2009).
We show that indeed agent composition can be naturally expressed as checking a certain ATL formula over a specific game structure where the players are the virtual target (agent) behavior, the concrete available agent behaviors, and a controller, whose actual controlling strategy has yet to be defined. The players corresponding to the target and to the available agent behaviors team up togheter against the controller. The controller tries to realize the target by looking, at each point in time, at the action chosen by the target agent behavior, and by selecting accordingly who, among the available agents, has to perform the action. In doing this the controller has to cope with the choice of the action to perform by the target agent and the non-deterministic choice of the next state of the available agent that has been selected to perform the action. The ATL formula essentially requires that the controller avoids errors, where an error is produced whenever no available agents are able to actually perform the target agent's action currently requested. If the controller has a strategy to satisfy the ATL formula, then, from such strategy, a refined controller realizing the composition can be synthesized. In fact, we show that by ATL model checking we get much more than a single controller realizing a composition: we get a "controller generator" (Sardiña et al., 2008) i.e., an implicit representation of all possible controllers realizing a composition.
This result is of interest for at least two contrasting reasons. First, from the point of view of agent composition, it gives access to some of the most modern model checking techniques and tools, such as MCMAS, that have been recently developed by the Agent community. Second, from the point of view of ATL verification tools, it gives a novel concrete problem to look at, which puts emphasis on actually synthesize winning policies (the refined controller) instead of just checking that they exist, as usual in many contexts where ATL is used for agent verification.

3.1.1 Agent Behavior Composition Problem

Composition of non-deterministic, fully observable available behaviors[1] for realizing a target behavior has its roots in certain forms of service composition advocated for SOA (Su, 2008a), and it is strictly related to composition of stateful, or "conversational", web services (Su, 2008c). Roughly speaking, it can be states as follows:

> Consider a set of available agent behaviors and an additional desired target behavior, all exporting a description of their operational model. Is it possible to coordinate (i.e., compose) the available agent behaviors

[1] This section will formally introduce the agent behavior composition problem. As said in the introduction, from an AI perspective, a behavior refers to an agent's abstract operational model, generally represented as a non-deterministic finite-state machine. Hence, in the technical development that follows, we will model agents through their behaviors.

> as to as realize (or "mimic"), at execution time, the behavior of the target (but non-existent) one, as if it is being executed?

Such composition problem has been studied in various areas of Computer Science, including (web) *services* (Balbiani et al., 2008), AI reasoning about action (Sardina et al., 2008; Stroeder and Pagnucco, 2009; De Giacomo et al., 2013), verification (Lustig and Vardi, 2009), and robotics (Bordignon et al., 2007), and it is often referred as "Service Composition Problem". Many approaches (surveyed, e.g., in (ter Beek et al., 2007)) have been proposed in the last years in order to address the service composition problem from different viewpoints. Works based on Planning in AI, such as (McIlraith and Son, 2002; Wu et al., 2003; Blythe and Ambite, 2004; Zhao and Doshi, 2006) consider only the input/output specification of available services, which is captured by atomic actions together with their pre- and post-conditions (a notable extension is (Beauche and Poizat, 2008)), and specify the overall semantics in terms of propositions/formulas (facts known to be true) and actions, affecting the proposition values. All these approaches consider *stateless* services, as the operations offered to clients do not depend on the past history, as services do not retain any information about past interactions. Also other works (e.g., (Yang and Papazoglou, 2004; Medjahed et al., 2003; Curbera et al., 2004; Cardose and Sheth, 2004)) consider available services as atomic actions, but, rather than on (planning-based) composition, they focus on modeling issues and automatic service discovery, by resorting to rich ontologies as a basic description mean. Many works (e.g., (Klein et al., 2010; Schuller et al., 2010; Baligand et al., 2007; Paoli et al., 2006) consider how to perform composition by taking into account Quality-of-Service (QoS) of the composite and component services. Some works consider non classical techniques (e.g., (Wang et al., 2010) adopts learning approaches) for solving the composition problem. There are also approaches (e.g., (Hassen et al., 2008)) that consider *stateful* services, which impose constraints on the possible sequences of interactions (a.k.a., conversations) that a client can engage with the service. Stateful services raise additional challenges, as the process coordinating such services should be correct w.r.t. the possible conversations allowed by the services themselves. An interesting approach of this type is the one of (Pistore et al., 2005), in which the specification is a set of atomic actions and propositions, like in planning, services are (finite-state) transition systems whose transitions correspond to action executions, which, in general, affect the truth values of propositions, and the client requests a (main) goal (i.e., a formula built from the above propositions) to be achieved, while requiring runtime failures to be properly handled by achieving a special exception handling goal.
The problem is appealing in that with computers now present in everyday devices like phones, cars and planes or places like homes, offices and factories, the trend is to build embedded complex systems from a collection of simple components. For instance, one may be interested in implementing a non-existent house entertainment systems by making use of various devices installed in the house, such as game consoles, TVs, music players, automatic lights, etc.

We address the problem following the approach proposed in (Stroeder and Pagnucco, 2009; Sardiña et al., 2007; Sardiña et al., 2008; De Giacomo et al., 2013) and, as we

do not limit ourselves to the service setting, we will adopt the general terminology of agent behaviors.
In this approach, behaviors are modeled by a transition system, which captures their operational model, as well as the available choices that, at each point, the behavior has available for continuing its execution. Given a *virtual target behavior*, i.e., a behavior of which we have the desired behavior but not its actual implementation, and a set of available concrete behaviors, i.e., a set of behaviors, each with its own behavior, that are indeed implemented, the composition's goal is to synthesize a *controller*, i.e., a suitable software module, capable of implementing the target behavior by suitably controlling the available behaviors. Such a module realizes a target behavior if and only if it's able, at every step, to delegate every action executable by the target to one of the available behavior. Notice that, in doing this, the controller has to take into account not only local states of both the target and the available behaviors, but also their future evolutions, delegating actions to available behaviors so that all possible future target behavior's actions can continue to be delegated. We call such a controller a *composition* of the available behavior that realizes the target behavior.

Definition 3.1. Formally, an *agent behavior* is a tuple $\mathcal{B} = \langle B, Act, b_0, \varrho, F \rangle$, where:

- B is the set of finite states;
- Act is the set of actions;
- $b_0 \in B$ is the initial state;
- $\varrho \subseteq B \times Act \times B$ is the behavior's (non-deterministic) transition relation: $\langle b, a, b' \rangle \in \varrho$, or $b \xrightarrow{\sigma} b'$ in $\mathcal{B}$, denotes that action a executed in behavior state b may lead the behavior to successor state b';
- $F \subseteq S$ is a subset of final states.

△

Final states are meant to capture those states in which the behavior can be safely "left", i.e., the computation can stop. A solution that manages to realize a target behavior without guaranteeing that, whenever the target is in a final states, all available behaviors are left in final states as well, is not a good solution. However, this requirement on final states will be sometimes omitted in some solution approaches presented in this dissertation, as it can be captured by introducing special "final" actions in the alphabet Act and therefore is not fundamental. Removing final states also corresponds to consider all states as final.
Observe that since agent behaviors may be non-deterministic, one cannot know beforehand what actions will be available to execute after an action is performed in a state, as the next set of applicable actions would depend on the successor state in which the behavior happens to be in. Hence, we say that non-deterministic behaviors are only partially controllable. A deterministic behavior is one where the successor state is always uniquely determined –a fully controllable behavior.

Example 7. *Consider the truck behavior* $\mathcal{B}_r = \langle B_r, Act, b_{r0}, \varrho_r, F_r \rangle$ *of Figure 3.1, which is the same transition system of Figure 2.1 without state propositions and*

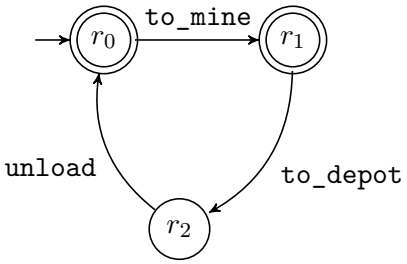


Figure 3.1. A behavior modeling a mining truck.

with the addition of final states. It is also deterministic. $B_r = \{r_0, r_1, r_2\}$, $Act = \{\texttt{to\_mine}, \texttt{to\_depot}, \texttt{unload}\}$, $b_{r0} = r0$, $\varrho_r(r_0, \texttt{to\_mine}, r_1)$, $\varrho_r(r_1, \texttt{to\_depot}, r_2)$, $\varrho_r(r_2, \texttt{unload}, r_0)$, *and* $F_r = \{r_0, r_1\}$.
Consider now a new agent behavior $\mathcal{B}_c = \langle B_c, Act, b_{c0}, \varrho_c, F_c \rangle$ *as in Figure 3.3 modeling an old crane: it is used to perform a* `load` *action and then it needs to be repaired in order to be able to load again. However, this repairing action is non-deterministic, and it could be the case that it has to be performed twice to get the crane operational again.* $B_c = \{c_0, c_1, c_2\}$, $Act = \{\texttt{load}, \texttt{maintenance}\}$, $b_{c0} = c0$, $\varrho_c(c_0, \texttt{load}, c_1)$, $\varrho_c(c_1, \texttt{maintenance}, c_0)$, $\varrho_c(c_1, \texttt{maintenance}, c_2)$, $\varrho_c(c_2, \texttt{maintenance}, c_0)$, *and* $F_c = \{c_0\}$.

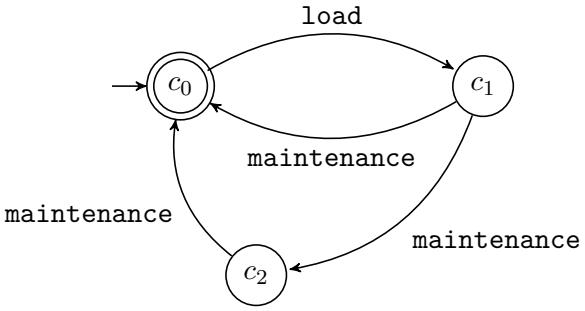


Figure 3.2. A behavior modeling an old crane.

□

Example 8. *Consider now the behaviors* $\mathcal{B}_a = \langle B_a, Act, b_{0a}, \varrho_a, F_a \rangle$ *and* $\mathcal{B}_g = \{B_g, Act, b_{0g}, \varrho_g, F_g\}$ *of Figure 3.2.* $\mathcal{B}_a$ *is the same transition system of Figure 2.2 with the addition of final states, whereas* $\mathcal{B}_g$ *is a behavior modeling a game device, which allows to play games, watch movies or access the internet.*

□

In this Chapter (and in the whole dissertation as well) we will make use of the terminology introduced in Chapter 2. Hence, a *trace* of $\mathcal{B}$ is the possibly infinite sequence $\tau = b_0 \xrightarrow{a_1} b_1 \xrightarrow{a_2} \cdots$ such that $b_{i+1} \in \varrho(b_i, a_{i+1})$, for $i \geq 0$ (for more details, refer to the preliminaries in Chapter 2). A *history* is a finite trace.

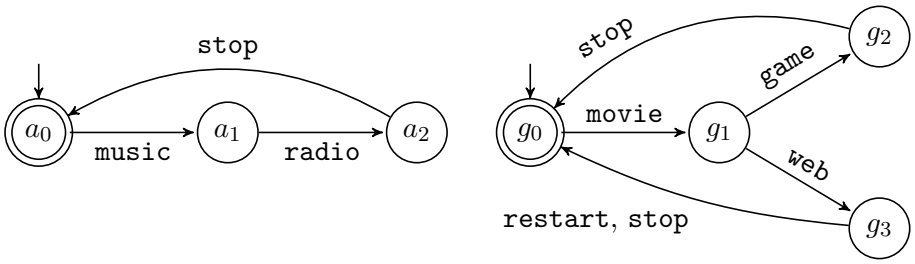


Figure 3.3. Two behaviors modeling an audio device and a game device.

Available System. The system stands for a collection of agent behaviors that are at disposal. Technically, an *available system* is a tuple $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n \rangle$, where $\mathcal{B}_i = \langle B_i, Act_i, b_{i0}, \varrho_i, F_i \rangle$, for $i \in \{1, \ldots, n\}$, is a, possibly non-deterministic, *available* agent behavior in the system. To refer to the behavior that emerges from the joint execution of available agent behaviors, the notion *enacted system behavior* is used in the literature (De Giacomo et al., 2013). In a nutshell, the *enacted system behavior* $\mathcal{E}_\mathcal{S}$ of an available system $\mathcal{S}$ (as above) is a transition system obtained as the asynchronous product of the available behaviors in $\mathcal{S}$. A transition $s \xrightarrow{a,k} s'$ in $\mathcal{E}_\mathcal{S}$ states that action a has been performed by behavior $\mathcal{B}_k$ (and all other behaviors remain still).

Definition 3.2. The *enacted system behavior* of $\mathcal{S}$ is a tuple $\mathcal{E}_\mathcal{S} = \{S, Act, s_0, \varrho, S_F\}$, where:

- $S = B_1 \times \cdots \times B_n$ is the finite set of $\mathcal{E}_\mathcal{S}$'s states;
- $Act = \bigcup_{i=1}^{n} Act_i$ is the set of actions of $\mathcal{E}_\mathcal{S}$;
- $s_0 = \langle b_{10}, \ldots, b_{n0} \rangle \in S$ is $\mathcal{E}_\mathcal{S}$'s initial state;
- $\varrho \subseteq S \times Act \times \{1, \ldots, n\} \times S$ is $\mathcal{E}_\mathcal{S}$'s transition relation, where $\langle s, a, k, s' \rangle \in \varrho$, or $s \xrightarrow{a,k} s'$ in $\mathcal{E}_\mathcal{S}$, with $s = \langle b_1, \ldots, b_n \rangle$ and $s' = \langle b'_1, \ldots, b'_n \rangle$, iff:
  - $b_k \xrightarrow{a} b'_k$ in $\mathcal{B}_k$; and
  - $b'_i = b_i$, for $i \in \{1, \ldots, n\} \setminus \{k\}$.
- $S_F = F_1 \times \cdots \times F_n$.

△

Example 9. *The available system for the two behaviors $\mathcal{B}_r$ and $\mathcal{B}_c$ as before (the truck and the crane) is the tuple $\mathcal{S} = \langle \mathcal{B}_r, \mathcal{B}_c \rangle$. The enacted system $\mathcal{E}_\mathcal{S}$ of $\mathcal{S}$ is depicted in Figure 3.4.* □

A *system history* is a straightforward generalization of behavior histories to an available system $\mathcal{E}_\mathcal{S}$, that is, a sequence of the form $h = \vec{b}_0 \xrightarrow{a_1,k_1} \vec{b}_1 \xrightarrow{a_2,k_2} \cdots \xrightarrow{a_\ell,k_\ell} \vec{b}_\ell$. We denote with $last(h)$ the last state $\vec{b}_\ell$ of h, with h^ℓ the prefix of h of length ℓ and with $\mathcal{H}$ the set of all system histories.

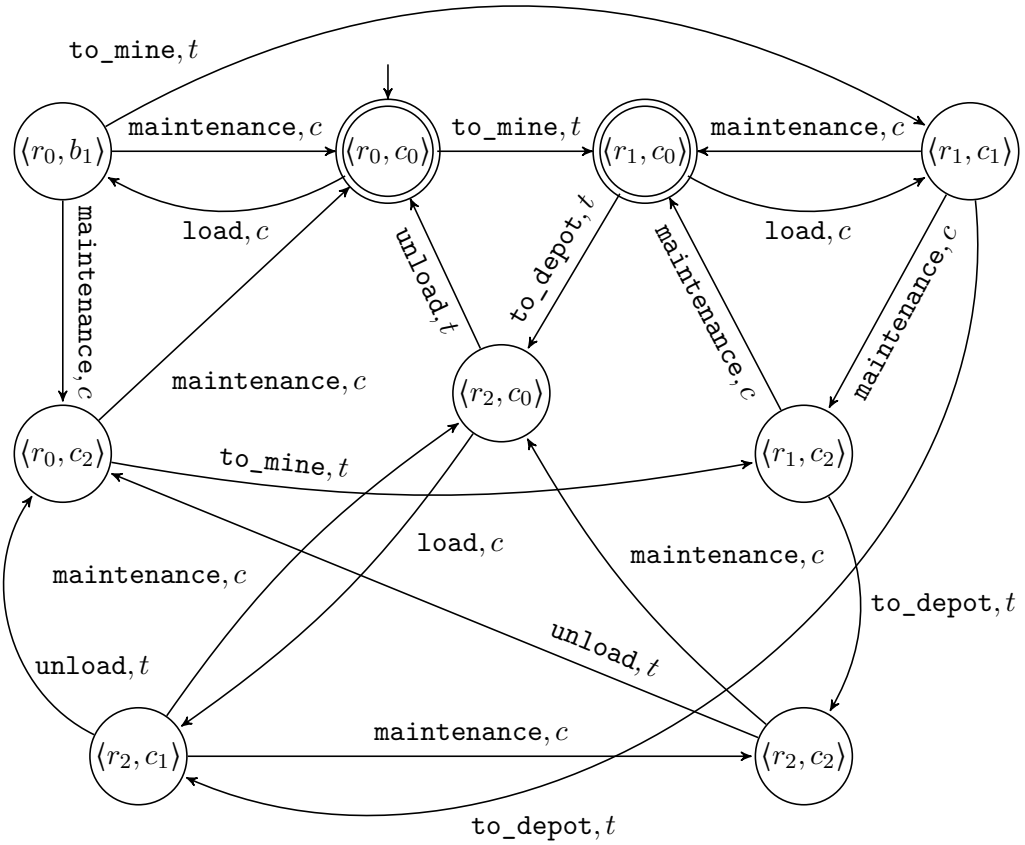


Figure 3.4. The enacted system $\mathcal{E}_{\mathcal{S}}$ for $\mathcal{S} = \{\mathcal{B}_r, \mathcal{B}_c\}$.

Target Specification. Finally, the *target agent behavior* specification is a deterministic agent behavior $\mathcal{T} = \langle T, Act_T, t_0, \varrho_T, F_T \rangle$.

Therefore, informally, the behavior composition task can be stated as follows: Given a system $\mathcal{S}$ and a target agent behavior $\mathcal{T}$, is it possible to (partially) control the available agent behaviors in $\mathcal{S}$ in a step-by-step manner—by instructing them on which action to execute next and observing, afterwards, the outcome in the behavior used—so as to "realize" the desired target behavior. In other words, by adequately controlling the system, it appears as if one was actually executing the target behavior.

Remark. Note that, even though we delegate a target action to an agent behavior which is currently able to perform it, such delegation may be incorrect, and we may still get "stuck" in the future. As a matter of fact, we can not foresee the future target requests (the actual sequence of target actions).

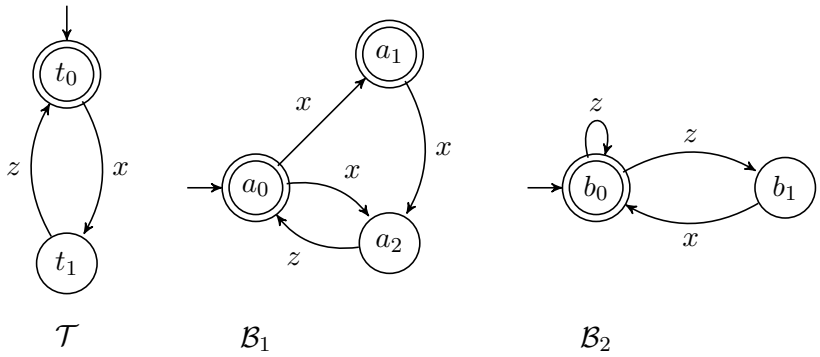


Figure 3.5. Target $\mathcal{T}$ and available agent behaviors $\mathcal{B}_1, \mathcal{B}_2$.

Example 10. *Figure 3.5 shows two available agent behaviors $\mathcal{B}_1$ and $\mathcal{B}_2$ and a simple target behavior $\mathcal{T}$. At the beginning, $\mathcal{T}$ is in state t_0, $\mathcal{B}_1$ in a_0 and $\mathcal{B}_2$ in b_0. As $\mathcal{T}$ performs action x, we need to delegate such an action to an available agent behavior. It is trivial to notice that the only available choice is to choose $\mathcal{B}_1$. However, as $\mathcal{B}_1$ performs x, it could non-deterministically evolve to state a_1, from which action z is not available. Therefore, as $\mathcal{T}$ performs z, this action can be delegated to $\mathcal{B}_1$ only if it is currently in a_2, otherwise $\mathcal{B}_2$ is the only choice. However (as we will see later), when it happens that $\mathcal{B}_1$ reaches state a_2 upon performing action x, the only correct choice is indeed to delegate z to $\mathcal{B}_1$: if we choose $\mathcal{B}_2$ we won't be able to mimic possible future actions of the target (due to non-determinism).* □

Now we are going to define the solution of our problem.

Compositions. A *controller* for system $\mathcal{S}$ is a function

$$\mathcal{C} : \mathcal{H} \times Act \to \{1, \ldots, n\}$$

which, given a system history $h \in \mathcal{H}$ and an action $a \in Act$ to perform, selects a behavior (actually, returns its index) to delegate a to for execution.
One can formally define when a controller realizes the target behavior –a solution to the problem– as done in (Sardiña et al., 2007; Sardina et al., 2008). In particular, one first defines when a controller $\mathcal{C}$ *realizes* a trace of the target $\mathcal{T}$. Then, since the target behavior is a deterministic transition system, and thus its behavior is completely characterized by its set of traces, one defines that a controller $\mathcal{C}$ realizes the target behavior $\mathcal{T}$ iff it realizes all its traces (Sardina et al., 2008). Such a controller is called (exact) *composition*.
The set $\mathcal{H}_{\mathcal{C},\tau}$ is defined as the subset of system histories *induced* by $\mathcal{C}$ on a target trace τ.

Definition 3.3. (Sardina et al., 2008) Given a target trace τ in $\mathcal{T}$ and a controller $\mathcal{C}$, $\mathcal{H}_{\mathcal{C},\tau} \subseteq \mathcal{H}$ is inductively defined as follows: (i) $h^0 = \langle b_{10}, \ldots, b_{n0} \rangle \in \mathcal{H}_{\mathcal{C},\tau}$; (ii) $h^{j+1} \in \mathcal{H}_{\mathcal{C},\tau}$ iff $h^{j+1} = \langle b_1^{j+1}, \ldots, b_n^{j+1} \rangle = \delta(h^j, a_{j+1})$ and $h^j = \langle b_1^j, \ldots, b_n^j \rangle \in \mathcal{H}_{\mathcal{C},\tau}$; a_{j+1} is the same as τ; $\mathcal{C}(h^j, a_{j+1}) = k$; $b_k^j \xrightarrow{a_{j+1}} b_{k+1}^j$ is in $\mathcal{B}_k$ and $b_i^{j+1} = b_i^j$ for any $i \neq k$. △

A controller $\mathcal{C}$ realizes a trace τ of $\mathcal{T}$ iff it is always defined for every prefix of any induce history h and, also, such histories agree on final states: whenever the target behavior is in a final state then all the available behaviors are. Formally, iff for all $h \in \mathcal{H}_{\mathcal{C},\tau}$ the function $\mathcal{C}(h, a_{length(h)+1})$ is defined, and for all target states $t_j \in F_T$ of τ it is $last(h^j) \in S_F$.

Exact Compositions via Simulation. Though technically involved, one can formally define when a so-called *controller*, a function taking a run of the system and the next action request and outputting the index of the available behavior where the action is being delegated, realizes the target behavior; see (De Giacomo and Sardina, 2007; De Giacomo et al., 2013). An interesting and much used result links exact compositions to the formal notion of simulation (Milner, 1971b). A simulation relation captures the behavioral equivalence of two transition systems. Intuitively, a (transition) system S_1 "simulates" another system S_2, denoted $S_2 \preceq S_1$, if S_1 is able to *match* all of S_2's moves. Thus, (Sardina et al., 2008) defined a so-called *ND-simulation* (non-deterministic simulation) relation between (the states of) the target behavior $\mathcal{T}$ and (the states of) the enacted system $\mathcal{E}_{\mathcal{S}}$, denoted $\preceq_{\text{ND}}$.

Definition 3.4. Given a target behavior $\mathcal{T}$ and an enacted system $\mathcal{E}_{\mathcal{S}}$, a *ND-simulation relation* of $\mathcal{T}$ by $\mathcal{E}_{\mathcal{S}}$ is a relation $Sim \subseteq T \times B_1 \times \cdots \times B_n$ such that $\langle b_t, b_1, \ldots b_n \rangle \in Sim$ implies:

- if $t \in F_T$ then $b_i \in F_i$ for $i = 1, \ldots, n$: all behaviors are in a final state whenever the target is in a final state;
- for each transition $t \xrightarrow{a} t'$ in $\mathcal{T}$ there exists an index $j \in \{1, \ldots, n\}$ such that the following holds:
  - there exists at least one transition $b_j \xrightarrow{a} b'_j$ in $\mathcal{B}_j$;
  - for *all* transitions $b_j \xrightarrow{a} b'_j$ in $\mathcal{B}_j$ we have that $\langle t', b_1 \ldots, b'_j \ldots, b_n \rangle \in Sim$ (all behaviors but $\mathcal{B}_j$ remain still).

△

It was proven that there exists an exact composition for a target behavior $\mathcal{T}$ on an available system $\mathcal{S}$ *iff* $\mathcal{T} \preceq_{\text{ND}} \mathcal{E}_{\mathcal{S}}$, that is, the enacted system can ND-simulate the target behavior.

Theorem 1. (Sardina et al., 2008) A composition of the available behaviors $\mathcal{B}_1, \ldots, \mathcal{B}_n$ realizing the target behavior $\mathcal{T}$ exists if and only if $\mathcal{T}$ is simulated by $\mathcal{E}_{\mathcal{S}}$.

Example 11. *Figure 3.6 shows the graphical representation of the simulation relation between the agent behavior $\mathcal{T}$ and two available behaviors $\mathcal{B}_1$ and $\mathcal{B}_2$ of Figure 3.5 ($\mathcal{T} \preceq \mathcal{E}_{\mathcal{S}}$). As a consequence of Theorem 1, there exists a composition of $\mathcal{T}$ by $\mathcal{S} = \{\mathcal{B}_1, \mathcal{B}_2\}$. Conversely, notice that there exists no composition for the entertaining room example of Figure 3.7.* □

Example 12. *Figure 3.7 shows the entertaining room example[2] ($\mathcal{B}_g$ and $\mathcal{B}_a$ are the same of Figure 3.2). Figure 3.8 shows instead the complete mining example.* □

[2]This example is based on one by Nitin Yadav.

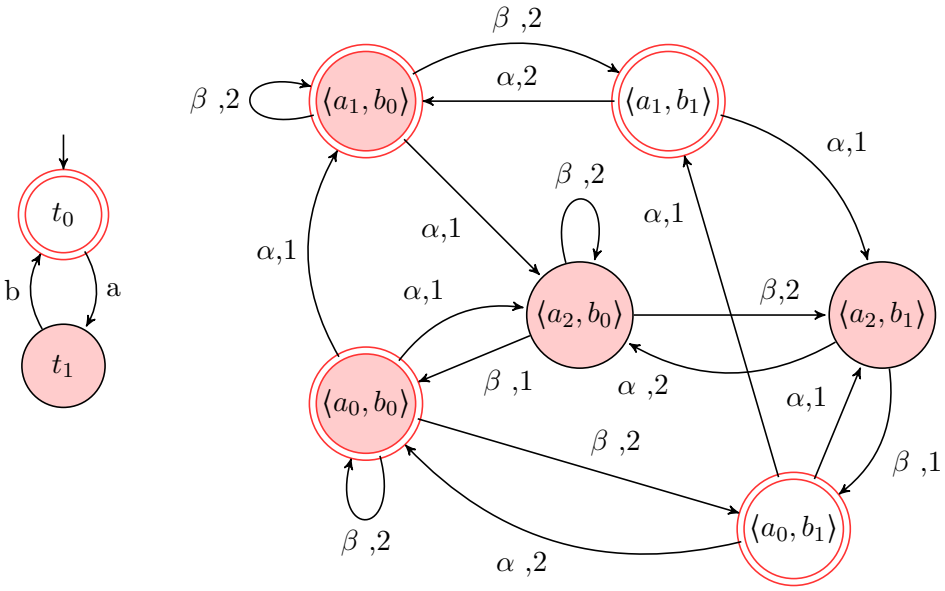


Figure 3.6. $\mathcal{T}$ simulated by the enacted system $\mathcal{E}_{\mathcal{S}}$ of $\mathcal{S} = \{\mathcal{B}_1, \mathcal{B}_2\}$.

Theorem 1 thus relates the notion of simulation relation to the one of behavior composition showing, basically, that checking for the *existence* of an behavior composition is equivalent to checking for the existence of a simulation relation between the target behavior and the available behaviors. To actually synthesize a controller from the simulation we compute the so called *composition generator*, or ω for short. Intuitively, the ω is a program that returns, for each state the available behaviors may potentially reach while realizing a target history, and for each action the target behavior may do in such a state, the set of all available behaviors able to perform the target behavior's action, while guaranteeing that every future target behavior's actions can still be fulfilled. The ω is directly obtained by the maximal simulation relation as follows:

Definition 3.5. (Sardiña et al., 2008) Let $\mathcal{T}$ be a target behavior and $\mathcal{S} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ be a system of n available behaviors such that $\mathcal{T}$ is simulated by $\mathcal{E}_S$. The *Composition Generator* (CG) for $\mathcal{T}$ by $\mathcal{S}$ is the function: $\omega : S \times Act \to 2^{\{1,\ldots,n\}}$ such that for $s = \langle t, b_1, \ldots, b_n\rangle \in S$ and $a \in Act$

$$\omega(s,a) = \{i \quad | \quad \begin{array}{l} t \xrightarrow{a} t' \text{ is in } \mathcal{T} \text{ and} \\ b_i \xrightarrow{a} b_i' \text{ is in } \mathcal{B}_i \text{ and} \\ t' \preceq_{\mathrm{ND}} \langle b_1, \ldots, b_i', \ldots, b_n\rangle\} \end{array}$$

△

ω is a function that, given the states of the target and available behaviors and an action, outputs the set of all available behaviors able to perform that action in their current state, while preserving the simulation. If there exists a composition of $\mathcal{T}$ by $\mathcal{S}$, then the composition generator *generates* compositions, called *generated compositions*, by picking up, at each step, one among the behaviors indices returned

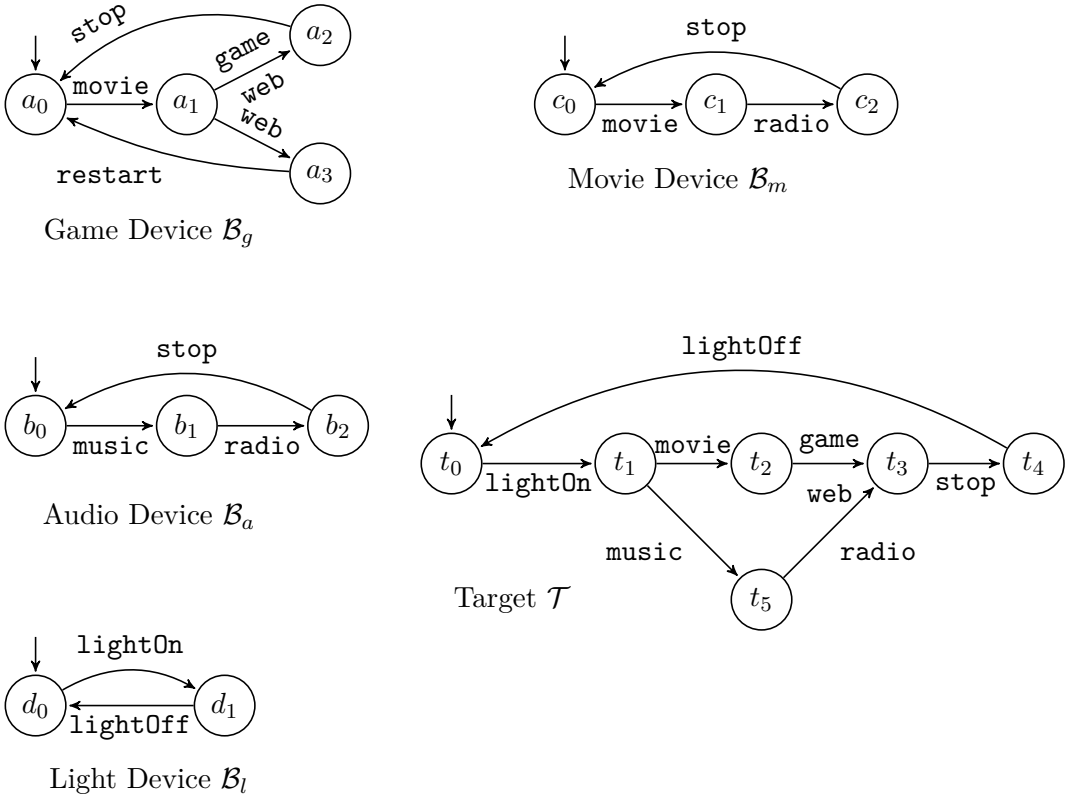


Figure 3.7. The entertaining room example.

by $\omega(last(h), a)$. A *generated compositions* of ω is thus a composition $\mathcal{C} : \mathcal{H} \times Act \to \{1, \ldots, n\}$ such that

$$\mathcal{C}(h, a) \subseteq \omega(last(h), a)$$

For further details please refer to (Sardiña et al., 2008). The next theorem guarantees that all compositions can be generated by the composition generator.

Theorem 2. (Sardiña et al., 2008) Let $\mathcal{T}$ and $\mathcal{S} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ be as above. A controller $\mathcal{C}$ is a composition of $\mathcal{T}$ by $\mathcal{B}_1, \ldots, \mathcal{B}_n$ if and only if it is a generated composition of ω.

As for complexity, it was proven that the problem is EXPTIME-complete (Sardiña et al., 2008).

3.1.2 Agent Behavior Synthesis via ATL

Now we look at how to use ATL for synthesizing compositions. To do so we introduce a concurrent game structure for the agent composition problem, reducing the search for possible compositions to the search for *winning strategies* in the multi-player game played over it. For simplicity, we will assume here, wlog, that all agents share the same action alphabet *Act*.

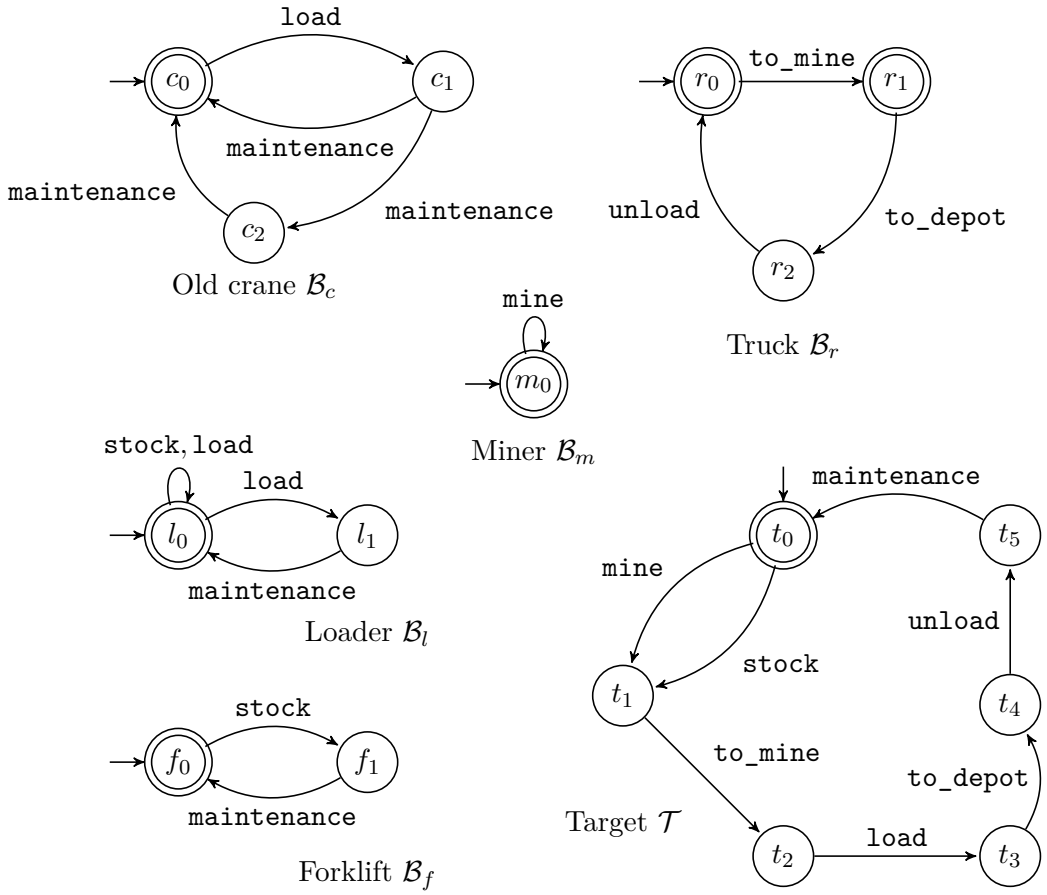


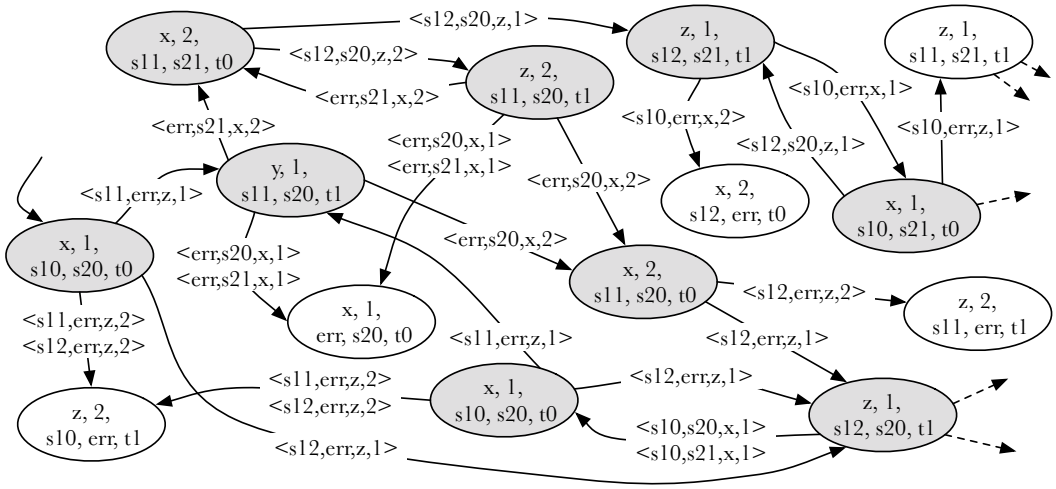
Figure 3.8. The mining example.

Given a target agent $\mathcal{T} = \langle T, Act, t_0, \varrho_T, F_T \rangle$ and a n available agents $\mathcal{B}_1, \ldots, \mathcal{B}_n$ with $\mathcal{B}_i = \langle B_i, Act, b_{0i}, \varrho_i, F_i \rangle$ with $i = 1, \ldots n$, we define a game structure $\mathcal{G}$ for our problem as follows.
We start by slightly modifying the available agents $\mathcal{B}_i$ $(i = 1, \ldots, n)$ by adding a new state err_i, disconnected, through ϱ_i, to the other states, and such that $err_i \notin F_i$.
We also define two convenient notations:

- $Act_i(b)$ that denotes the set of actions available to the agent i $(i = t, 1, \ldots, n)$ in its local state, i.e., for $i = 1, \ldots, n$ we have $Act_i(b) = \{\alpha \in Act \mid \langle b, \alpha, b' \rangle \in \varrho_i \text{ for some } b'\}$, whereas for $i = t$, it is $Act_t(b) = \{\alpha \in Act \mid \langle b, \alpha, b' \rangle \in \varrho_T \text{ for some } b'\}$.
- $Succ_i(b, \alpha)$ that denotes the set of possible successor states for player i $(i = t, 1, \ldots, n)$ when it performs action α from its local state b, i.e., $Succ_i(b, \alpha) = \{b' \in B_i \mid \langle b, a, b' \rangle \in \varrho_i\}$ for $i = 1, \ldots, n$ and $Succ_i(b, \alpha) = \{b' \in T \mid \langle b, \alpha, b' \rangle \in \varrho_T\}$ for $i = t$.

The game structure $\mathcal{G} = \langle k, Q, \Pi, \pi, d, \delta \rangle$ is defined as follows.

Players The set of players Σ is formed by one player for each available agent, one player for the target agent, and one player for the controller. Each player is



(a)

state_1	state_2	state_t	act	
a_0	b_0	t_0	x	$\{1\}$
a_0	b_0	t_1	z	$\{2\}$
a_0	b_1	t_0	x	$\{2\}$
a_1	b_0	t_0	x	$\{1\}$
a_1	b_0	t_1	z	$\{2\}$
a_1	b_1	t_0	x	$\{1,2\}$
a_2	b_0	t_1	z	$\{1\}$
a_2	b_1	t_1	z	$\{1\}$

(b)

Figure 3.9. A fragment of $\mathcal{G}$ and the corresponding $\omega_{\mathcal{G}}$ for the instance of Figure 3.5

identified by an integer $\Sigma = \{1, \ldots, k\}$.

- $i \in \{1 \ldots n\}$ for the available agents $(n = k - 2)$
- $t = k - 1$ is the target agent
- k is the controller

Game structure states The states of the game structure are characterized by the following finite range functions:

- $state_i$: returns the current state of the agent i $(i = t, 1, \ldots, n)$; it ranges over $b \in B_i$ for $i \neq t$ and $b \in T$ for $i = t$.
- sch : returns the scheduled available agent, i.e., the agent that performed the last action; it ranges over $i \in \{1, \ldots, n\}$.
- act_t : returns the action requested by the target, it ranges over $\alpha \in Act$.
- $final_i$: returns whether the current state $state_i$ of agent i is final or not $(i = t, 1, \ldots, n)$; it ranges over booleans.

Q is the set of states obtained by assigning a value to each of these functions, and Π is the set of propositions of the form $(f = v)$ corresponding to assert that function f has value v. Notice that we can use directly finite range functions, without fixing any specific encoding for the technical development that follows.
The function π, given a state q of the game structure returns the values for the various functions. For simplicity, we will use the notation $state_i(q) = b$ instead of $(state_i = b) \in \pi(q)$.

Initial states The initial states Q_0 of the game structure are those q_0 such that:

- every agent is in its local initial state, $state_i(q_0) = b_{0i}$ and $final_i(q_0) = true$ iff $b_{0i} \in F_i$ for any $i \in \{1, \ldots, n\}$, and $state_t(q_0) = t_0$, $final_t(q_0) = true$ iff $t_0 \in F_T$,
- $act_t(q_0) = \alpha$ for some action $\alpha \in Act(t_0)$, and
- $sch(q_0) = 1$ (this is a dummy value, which will be not used in any way during the game).

Note that regular concurrent game structures do not have initial states: model-checking an ATL formula over a game structure will return the subset of game states satisfying it. However, we are here interested in performing *local* model-checking, i.e., to check whether the formula is true in all states belonging to Q_0.

Players' moves The moves that the player i ($i = 1, \ldots, n$), representing the available agent $\mathcal{B}_i$, can perform in a state q are:

$$Moves_i(q) = \begin{cases} \{b' \mid b' \in Succ_i(state_i(q), act_t(q))\} \\ \qquad \text{if } Succ_i(state_i(q), act_t(q)) \neq \varnothing \\ \{err_i\} \qquad \text{otherwise.} \end{cases}$$

The moves that the player k, representing the controller, can do in a state q are:

$$Moves_k(q) = \{1, \ldots, n\}.$$

The moves that the player t, representing the target behavior $\mathcal{T}$, can perform in a state q are (with a little abuse of notation, and recalling that the target agent is deterministic):

$$Moves_t(q) = Act_t(Succ(state_t(q), act_t(q))).$$

Notice that the player t chooses in the *current* turn the action that will be executed *next*.
The number of moves is $d_i(q) = |Moves_i(q)|$ and, wlog, we can associate some enumeration of the elements in $Moves_i(q)$.

Game transitions The game transition function δ is defined as follows: $\delta(q, j_1, \ldots, j_k)$ is the game structure state q' such that:

- $sch(q') = j_k$
- $state_w(q') = j_w$ if $j_k = w$

- $state_i(q') = state_i(q)\ \forall i \neq w$
- $state_t(q') = b_t$, where $\{b_t\} = Succ(state_t(q), act_t(q))$
- $act_t(q') = j_t$
- $final_i(q') = true$ iff $state_i(q') \in F_i$.

Example 13. *Figure 3.9(a) shows a fragment of the game structure $\mathcal{G}$ for the example in Figure 3.5. Nodes represent states of the game and edges represent game transitions labelled with move vectors (for simplicity, states where one of the agents is in err are left as sink nodes).* □

ATL formula to check for composition Checking the existence of a composition is reduced to checking the ATL formula φ, over the game structure $\mathcal{G}$, defined as follows:

$$\begin{aligned}\varphi \ = \ \ll k \gg \Box\, (& \\ & \wedge_{i=1,\ldots,n}(state_i \neq err_i) \wedge \\ & (final_t \ \rightarrow \ (\wedge_{i=1,\ldots,n} final_i = true)) \\ )\end{aligned}$$

Given a target agent $\mathcal{T}$ and n available agents $\mathcal{B}_1, \ldots, \mathcal{B}_n$, let $\mathcal{G} = \langle k, Q, \Pi, \pi, d, \delta \rangle$ be the game structure and φ the ATL formula defined above. The set of winning states of the games is:

$$[\varphi]_{\mathcal{G}} = \{q \in Q \mid q \vDash \varphi\}$$

Referring to Figure 3.9(a) grey states are those in $[\varphi]_{\mathcal{G}}$.
From $[\varphi]_{\mathcal{G}}$ we can build an ATL Composition Generator $CG_{\mathcal{G}}$ for the composition of $\mathcal{S}$ for $\mathcal{T}$ exploiting the set $[\varphi]_{\mathcal{G}}$.

Definition 3.6. (ATL Composition Generator) Let $\mathcal{G}$ and φ be as above. We define the ATL Composition Generator $CG_{\mathcal{G}}$ as a tuple $CG_{\mathcal{G}} = \langle Act, \{1, \ldots, n\}, S_{\mathcal{G}}, S^0_{\mathcal{G}}, \omega_{\mathcal{G}}, \delta_{\mathcal{G}} \rangle$ where:

- Act is the set of actions, and $\{1, \ldots, n\}$ is the set of players representing the available agents, as in $\mathcal{G}$;
- $S_{\mathcal{G}} = \{\langle state_t(q), state_1(q), \ldots, state_n(q)\rangle \mid q \in [\varphi]_{\mathcal{G}}\}$;
- $S^0_{\mathcal{G}} = \{\langle state_t(q_0), state_1(q_0), \ldots, state_n(q_0)\rangle \mid q_0 \in Q_o \cap S_{\mathcal{G}}\}$;
- $\delta_{\mathcal{G}} : S_{\mathcal{G}} \times Act \times \{1, \ldots, n\} \rightarrow S_{\mathcal{G}}$ is the transition function, defined as follows: $\langle b'_t, b'_1, \ldots, b'_n\rangle \in \delta_{\mathcal{G}}(\langle b_t, b_1, \ldots, b_n\rangle, a, w)$ iff there exists $q \in [\varphi]_{\mathcal{G}}$ with $b_i = state_i(q)$ for $i = t, 1, \ldots, n$, $a = act_t(q)$, $b'_t \in Succ_t(b_t, a)$ such that *for each* $q' = \delta(q, b'_1, \cdots, b'_n, a', w)$, with $sch(q') = w$, $b'_w \in Succ_w(b_w, a)$, $b'_i = b_i$ for $i \neq w$, and $a' \in Act_t(q)$, we have $q' \in [\varphi]_{\mathcal{G}}$.
- $\omega_{\mathcal{G}} : S_{\mathcal{G}} \times Act \rightarrow 2^{\{1,\ldots,n\}}$ is the agent selection function: $\omega_{\mathcal{G}}(\langle b_t, b_1, \ldots, b_n\rangle, a) = \{i \mid \exists \langle b'_t, b'_1, \ldots, b'_n\rangle$ with $\langle b'_t, b'_1, \ldots, b'_n\rangle \in \delta_{\mathcal{G}}(\langle b_t, b_1, \ldots, b_n\rangle, a, i)\}$.

△

Figure 3.9(b) shows the agent selection function $\omega_{\mathcal{G}}$ of the ATL Composition Generator for the game structure of Figure 3.9(a). Next theorem states the soundness and completeness of the method based on the construction of $CG_{\mathcal{G}}$for computing agent compositions.

Theorem 3. Let $\mathcal{T}$ be a target agent and $\mathcal{S} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ be the set of available agents (the available system). Let $CG_{\mathcal{G}} = \langle Act, \{1, \ldots, n\}, S_{\mathcal{G}}, S^0_{\mathcal{G}}, \omega_{\mathcal{G}}, \delta_{\mathcal{G}} \rangle$ and ω be, respectively, the output function of the ATL Composition Generator and the Composition Generator for $\mathcal{T}$ by $\mathcal{B}_1, \ldots, \mathcal{B}_n$. Then

1. $\langle b_t, b_1, \ldots, b_n \rangle \in S_{\mathcal{G}}$ iff $b_t \preceq_{\mathrm{ND}} \langle b_1, \ldots, b_n \rangle$ and
2. for all $b_t, b_1, \ldots, b_n$ s.t. $b_t \preceq_{\mathrm{ND}} \langle b_1, \ldots, b_n \rangle$ and for all $a \in Act$, we have:

$$\omega_{\mathcal{G}}(\langle b_t, b_1, \ldots, b_n \rangle, a) = \omega(\langle b_t, b_1, \ldots, b_n \rangle, a)$$

Proof. We focus on (1) since (2) is a direct consequence of 1 and of the definition $\omega_{\mathcal{G}}$. $CG_{\mathcal{G}}$'s correctness is basically proven showing that the set S in $CG_{\mathcal{G}}$is a simulation relation (i.e., it satisfies the constraints (i) and (ii) in the definition of simulation relation), and it is hence contained in $\preceq_{\mathrm{ND}}$ which is the largest one.
As for completeness, we show that there exists no generated composition $\mathcal{C}$ for $\mathcal{T}$ and $\mathcal{B}_1, \ldots, \mathcal{B}_n$ which cannot be generated by $\omega_{\mathcal{G}}$. Toward contradiction let us assume that one such $\mathcal{C}$ exists. Then there exists a history of the system coherent with $\mathcal{C}$, such that, considering the definition of $CG_{\mathcal{G}}$ either (a) the requested action can't be performed in target's current state b_t, (b) target's current state b_t is final but at least one of the current states of the b_i $(i = 1, \ldots, n)$ available agents is not, or (c) no available agent is able to perform the requested action in its own current state b_i $(i = 1, \ldots, n)$, that is if all successor game states reached after performing it are error states. But (a) cannot happen by construction of $CG_{\mathcal{G}}$ being the history coherent with $\mathcal{C}$, and if either of (b) and (c) happens we get that $b_t \not\preceq_{ND} \langle b_1, \ldots, b_n \rangle$ contradicting the assumption that $\mathcal{C}$ is a generated composition. ■

Analogously of what done for the composition generator in Section 3.1.1, we can define the notion of $CG_{\mathcal{G}}$ *generated compositions*: i.e., the compositions obtained by picking up one among the available agents returned by function $\omega_{\mathcal{G}}$, at each step of the (virtual) target agent execution starting with all agents (target and available in their initial state). Then, as a direct consequence of Theorem 3 and the results of (Sardiña et al., 2008), we have that:

Theorem 4. Let $\mathcal{T}$ be a target agent and $\mathcal{S} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ be the set of available agents (the available system). Then (i) if $[\varphi]_{\mathcal{G}} \neq \varnothing$ then every controller generated by $CG_{\mathcal{G}}$ is a composition of $\mathcal{T}$ by $\mathcal{S}$ and (ii) if such composition does exist, then $[\varphi]_{\mathcal{G}} \neq \varnothing$ and every controller that is a composition of the target agent $\mathcal{T}$ by $\mathcal{S}$ can be generated by the ATL Composition Generator $CG_{\mathcal{G}}$.

By recalling that model checking ATL formulas is linear in the size of the game structure, analyzing the construction above we have:

Theorem 5. Computing ATL composition generator ($CG_{\mathcal{G}}$) is polynomial in the number of states of the target and available agents and exponential in the number of available agents.

Proof. The results follows by the construction of the game structure $\mathcal{G}$ above and from the fact that model checking ATL formula over game structure can be done in polynomial time. ■

From Theorem 4 and the EXPTIME-hardness of result in (Muscholl and Walukiewicz, 2008), we get a new proof of the complexity characterization of the agent composition problem (Sardiña et al., 2008).

Theorem 6. (Sardiña et al., 2008) Computing agent behavior composition is EXPTIME-complete.

3.1.3 Implementation

In this section we show how to use the ATL model checker MCMAS (Lomuscio et al., 2009) to solve agent composition via ATL model checking. In particular, following the definition of game structure $\mathcal{G}$, we show how to encode instances of the agent composition problem in ISPL (Interpreted Systems Programming Language) which is the input formalism for MCMAS. For readability, we show here a basic encoding, according to the definition of $\mathcal{G}$; some refinement will be discussed at the end of the section.

ISPL distinguishes between two kinds of agents: *ISPL standard agents* and one *ISPL Environment*. In brief, both ISPL standard agents and the ISPL Environment are characterized by (1) a set of local states, which are private with the exception of Environment's variables declared as `Obsvars`; (2) a set of actions, one of which is choosen by the ISPL agent in every state; (3) a rule describing which action can be performed by the ISPL agent in each local state (`Protocol`); and (4) a function describing how the local state evolve (`Evolution`).

We encode both the available agents and the target agent of our problem as ISPL standard agents, while we encode the controller in the Environment. Each ISPL standard agent features a variable `state`, holding the current state of the corresponding agent, while the ISPL Environment has two variables: `sch` and `act`, which correspond to propositions *sch* and act_t in Π, i.e., respectively, the available agent chosen by the controller to perform the requested target agent's action, and the target agent's action itself. The special value `start` is introduced for technical convenience: we need to "generate" a state for each possible action the target agent may request at the beginning of the game. All variables have enumeration type, ranging over the set of values they can assume according to the definition of $\mathcal{G}$.

We illustrate the ISPL encoding of our running example. Consider the same available and target agents as in Figure 3.5. The code for the ISPL Environment `Environment`:

```
Semantics = SA;

Agent Environment
   Obsvars:
      sch : {B1,B2,start};
      act : {a,b,start};
   end Obsvars
   Actions = {B1,B2,start};
   Protocol:
      act=start : {start};
      Other : {B1,B2};
   end Protocol
   Evolution:
      sch=B1 if Action=B1;
      sch=B2 if Action=B2;
      act=a if T.Action=a;
      act=b if T.Action=b;
   end Evolution
end Agent
```

Notice that the values of `sch` are unconstrained; they depend on the action chosen by the environment, which chooses them so as to satisfy the ATL formula of interest. Instead, `act` stores the action that the target agent has chosen to do next. The statement `Semantics = SA` specifies that only one assignment is allowed in each evolution line. This implies that evolution items are partitioned into groups such that two items belong to the same group if and only if they update the same variable and that they are not mutually excluded as long as they belong to different groups. Next we show the encoding as ISPL standard agents of the available agents `B1` and `B2`.

```
Agent B1
 Vars:
  state : {a0,a1,a2,err};
 end Vars
 Actions = {a0,a1,a2,err};
 Protocol:
  state=a0 and Environment.act=a : {a1,a2};
  state=a1 and Environment.act=a : {a2};
  state=a2 and Environment.act=b : {a0};
  Other : {err};
 end Protocol
 Evolution:
  state=err if Action=err and Environment.Action=B1;
  state=a0 if Action=a0 and Environment.Action=B1;
```

```
  state=a1 if Action=a1 and Environment.Action=B1;
  state=a2 if Action=a2 and Environment.Action=B1;
 end Evolution
end Agent

Agent B2
 Vars:
  state : {b0,b1,err};
 end Vars
 Actions = {b0,b1,err};
 Protocol:
  state=b0 and Environment.act=b : {b0,b1};
  state=b1 and Environment.act=a : {b0};
  Other : {err};
 end Protocol
 Evolution:
  state=err if Action=err and Environment.Action=B2;
  state=b0 if Action=b0 and Environment.Action=B2;
  state=b1 if Action=b1 and Environment.Action=B2;
 end Evolution
end Agent
```

Each ISPL standard agent for the available agents *reads* variable `Environment.act` which has been chosen in the previous game round and chooses a next state to go to among those reachable through that action. If such an action is not available to the agent in its current state, then `err` is chosen. If the ISPL standard agent is the one chosen by the controller, then by reaching such an error state, it falsifies the ATL formula.
Finally, we show the encoding as a ISPL standard agent `T` of the target agent `T` .

```
Agent T
   Vars:
      state : {t0,t1};
   end Vars
   Actions = {a,b};
   Protocol:
      Environment.act=start: {a};
      state=t0 and Environment.act=a : {b};
      state=t1 and Environment.act=b : {a};
   end Protocol
   Evolution:
      state=t1 if state=t0 and Environment.act=a;
      state=t0 if state=t1 and Environment.act=b;
   end Evolution
end Agent
```

The ISPL standard agent `T` reads the current action `Environment.act` in the ISPL Environment `Environment`, which stores its own previous choice, and virtually makes *the* corresponding transition (remember that the target agent is deterministic) getting to the new state. Then, it selects its next action among those available in its next state. Consider for example the first `Evolution` statement: `state=t1 if state=t0 and Environment.act=a`. Such a can be read as follows: *"if current state is `t0` and the (last) action requested is `a`, then request an action chosen among those available at state `t1`, namely the set `{b}` in this case"*. Note that, considering the definition of `Environment`, the ISPL standard agent `T` chooses the action to be stored in `Environment.act` at the *next* turn of the game.
The ISPL code is completed as follows.

```
Evaluation
  Error if B1.state=err or B2.state=err;
  B1Final if B1.state=a0 or B1.state=a1;
  B2Final if B2.state=b0 or B2.state=b1;
  TFinal if T.state=t0;
end Evaluation

InitStates
  B1.state=a0 and B2.state=b0 and T.state=t0 and
  Environment.act=start and Environment.sch=start;
end InitStates

Groups
  Controller = {Environment};
end Groups

Formulae
  <Controller> G (
    !Error and (TFinal -> (B1Final and B2Final))
  );
end Formulae
```

where we define some computed propositions for convenience (`Evaluation`), the initial state of the game (`InitState`), the group of agents appearing in the ATL formula (`Groups`), and the ATL formula itself (`Formulae`). All of these part directly correspond to what described in Section 3.1.2: in particular the ATL formula requires that all ISPL standard agents for available agents have to be in a final state if the one for the target does, and none of the ISPL standard agent for available agents can be in error state (since this can only be reached whenever the scheduled available agent cannot actually replicate requested action).
Standard MCMAS checks if the ATL formula φ is satisfied in the specified game structure $\mathcal{G}$. However, we used an available prototype of MCMAS that can actually return a convenient data structure with the set of all states of the game structure that satisfy the ATL formula, namely the set $[\varphi]_{\mathcal{G}}$, and the transitions among them.

Using such a data structure we wrote a simple Java program that actually computes $\omega_{\mathcal{G}}$ of Definition 3.6, thus obtaining a practical way to generate all compositions.
The whole approach is quite effective in practice. In particular, we have run some experimental comparisons with direct implementations of the simulation approach proposed in (Sardiña et al., 2008), and our MCMAS based system is generally two orders of magnitude faster. We also compare it with implementations based on TLV (Piterman et al., 2006a) that are tailored for the agent composition problem (Patrizi, 2009), and the results of the two systems are similar, even if we used a completely standard MCMAS implementation and prototypical additional components.
We close the section by observing that the ISPL encoding shown here, which directly reflects the theoretical construction done above, could be easily refined (though possibly at expenses of clarity) with at least two major improvements. First, we can massively reduce the number of error states from the resulting structure, giving ISPL Environment both a copy of available agents' states and a protocol function which only schedules ISPL standard agents that are actually able to perform the current target action, according to their own protocols. If none of the ISPL standard agents is able to fullfill this condition, then an error *action* is selected, and the successor state is flagged with a boolean variable set to true in the `Environment`. Local error states are no more needed and the `Error` definition in the `Evaluation` section needs to be changed accordingly. Second, the system can be forced to "loop" after such an error condition is reached, e.g. forcing all ISPL agents to select an error action, thus avoiding to generate further (error) states.

3.1.4 Discussion

We have shown that indeed agent behavior composition can be naturally expressed as checking a certain ATL formula over a specific game structure. This gives effective techniques for agent composition based on current ATL model checkers. The connection between agent composition and ATL and more generally the work on ATL-based agent verification can be quite fruitful in the future. Several advancements in the recent work on ATL within the Agent community can be of great interest for agent composition. For example, the recent work on using ATL together with forms of epistemic logics to capture the different knowledge of the various agents could give effective techniques to deal with partial observability of the behaviors of the available agents in the agent composition. Currently known techniques are mostly based on a belief-state construction (Rintanen, 2004b; Pistore et al., 2005; De Giacomo et al., 2009) and have mostly resisted effective implementations.
In the last years, the research on behavior composition within the AI community has been quite fruitful and several composition techniques have been devised, based on reduction to PDL satisfiability (Harel et al., 2000; De Giacomo and Sardina, 2007; Sardiña et al., 2007), on forms of simulation or bisimulation (Milner, 1971c; Sardiña et al., 2008; De Giacomo et al., 2009; Balbiani et al., 2009), on LTL (Linear time logic) synthesis (Pnueli and Rosner, 1989b; Piterman et al., 2006a; Lustig and Vardi, 2009; Patrizi, 2009) and on direct techniques (Stroeder and Pagnucco, 2009). In particular, we followed the approach proposed in (Stroeder and Pagnucco, 2009; Sardiña et al., 2007; Sardiña et al., 2008; De Giacomo et al., 2013). Observe how, in this framework, every interaction (of a controller) with a behavior is abstracted

away to a single action and no distinction is made between the inputs/output signals. Other works, such as (Lustig and Vardi, 2009) make this distinction more explicit along the line of (Pnueli and Rosner, 1989b). However, in (Lustig and Vardi, 2009) a component receives control when entering an initial state and releases control when entering a final state, and composing components amounts to deciding which of them will resume control when the control is relinquished. Therefore, given control to a component is not modeled as an atomic operation, hence the quantification structure on input and output signals is different: in our approach, a composition generator assigns action execution to components in a step-wise fashion.

3.2 Generalized 2GS

In this section we review a series of agent behavior synthesis problems under full observability and nondeterminism (partial controllability), ranging from conditional planning, to recently introduced agent planning programs, and to sophisticated forms of agent behavior compositions, and show that all of them can be solved by model checking two-player game structures. These structures are akin to transition systems/Kripke structures (see Chapter 2), usually adopted in model checking, except that they distinguish (and hence allow to separately quantify) between the actions/moves of two antagonistic players. We show that using them we can implement solvers for several agent behavior synthesis problems. Moreover, by exploiting the connections between such game structures and the interpretation structures for ATL, we also provide implementations of the solution techniques for the above problems by relying on a suitable use of an ATL model checker.
The simplest problem we consider is standard *conditional planning* in nondeterministic fully observable domains (Rintanen, 2004a), which is well understood by the AI community. Then, we move to a sophisticated form of planning recently introduced in (De Giacomo et al., 2010b), in which so-called *agent planning programs*—programs built only from achievement and maintenance goals—are meant to merge two traditions in AI research, namely, Automated Planning and Agent-Oriented Programming (Wooldridge, 2009b). Solving, that is, realizing, such planning programs requires temporally extended plans that loop and possibly do not even terminate, analogously to (Kerjean et al., 2006). Finally, we turn to an advanced form of *agent behavior composition* (see Section 3.1.1) in which we shall consider an advanced form of behavior composition, in which several devices are composed in order to realize *multiple virtual agents* simultaneously (Sardina and De Giacomo, 2008), which is relevant, e.g., for robot ecology, ubiquitous robots, or intelligent spaces (Lundh et al., 2008).
The techniques originally proposed for the above synthesis problems are quite diverse, ranging from specific forms of planning (for conditional planning), to simulation (for composition), to LTL-based synthesis (for agent planning programs and advanced forms of composition).
The main contribution of this work is to show that diverse agent behavior synthesis problems, including all the ones mentioned above, can be treated *uniformly* by relying on two foundational ingredients:

- making explicit the assumption of two different roles in reasoning/synthesis:

a role that works *against* the solution, the so-called "environment;" and a role that works *towards* the solution, the so-called "controller;"

- exploiting full observability even in the presence of nondeterminism (i.e., partial controllability) to do reasoning and synthesis based on model checking (Clarke et al., 1999c).

On the basis of these two points, we introduce *two-player game structures*, which are akin to the widely adopted transition systems/Kripke structures in model checking, except that they distinguish between the actions/moves of two antagonistic players: the *environment* and the *controller*. Such a distinction has its roots in discrete control theory (Ramadge and Wonham, 1989a), and has lately been adopted in Verification for dealing with synthesis from temporal specifications (Piterman et al., 2006b). Also, this distinction has been explicitly made in some AI work on reasoning about actions and on agents, e.g., (Lespérance et al., 2008; Genesereth and Nilsson, 1987; Wooldridge, 2009b). Formally, such a distinction allows for separately quantifying over both environment's and controller's moves. To fully exploit this possibility, we introduce a variant of (modal) μ-calculus (Emerson, 1996) (see Section 2.5) —possibly the most powerful formalism for temporal specification (Clarke et al., 1999c)—that takes into account such a distinction.
We demonstrate then that the resulting framework is indeed a very powerful one. To that end, we show that one can reformulate each of the above synthesis problems, as well as many others, as the task of model checking a (typically simple) μ-calculus formula over suitable two-player game structures. By exploiting the result on μ-calculus model checking, we are able to solve, optimally wrt computational complexity and effectively in practice, through model checking tools, several forms of agent behavior synthesis.

3.2.1 Two-player Game Structures

We start by introducing the notion of *two-player game structure* (2GS, for short), largely inspired by those game structures used in synthesis by model checking in Verification (Piterman et al., 2006b; de Alfaro et al., 2001), which in turn are at the base of ATL interpretation structures (Alur et al., 2002), often used in modeling multi-agent systems (Wooldridge, 2009b). 2GS's are akin to transition systems used to describe the systems to be checked in Verification (Clarke et al., 1999c), with a substantial difference, though: while a transition system describes the evolution of a system, a 2GS describes the *joint evolution* of two autonomous systems—the *environment* and the *controller*—running together and interacting at each step, as if engaged in a sort of game.
Formally, a *two-player game structure* (2GS) is a tuple $G = \langle \mathcal{X}, \mathcal{Y}, start, \rho_e, \rho_c \rangle$, where:

- $\mathcal{X} = \{x_1, \ldots, x_m\}$ and $\mathcal{Y} = \{y_1, \ldots, y_n\}$ are two disjoint finite sets representing the environment and controller variables, respectively. Each variable x_i (resp. y_i) ranges over finite domain X_i (resp. Y_i). Set $\mathcal{X} \cup \mathcal{Y}$ is the set of *game state variables*. A *valuation* of variables in $\mathcal{X} \cup \mathcal{Y}$ is a total function val assigning to each $x_i \in \mathcal{X}$ (resp. $y_i \in \mathcal{Y}$) a value $val(x_i) \in X_i$ (resp. $val(y_i) \in Y_i$). For convenience, we represent valuations as vectors $\langle \vec{x}, \vec{y} \rangle \in \vec{X} \times \vec{Y}$, where

$\vec{X} = X_1 \times \cdots \times X_m$ and $\vec{Y} = Y_1 \times \cdots \times Y_n$. Notice that $\vec{X}$ (resp. $\vec{Y}$) corresponds to the subvaluation of variables in $\mathcal{X}$ (resp. $\mathcal{Y}$). A valuation $\langle \vec{x}, \vec{y} \rangle$ is a *game state*, where $\vec{x}$ and $\vec{y}$ are the corresponding environment and controller states, respectively.

- $start = \langle \vec{x}_o, \vec{y}_o \rangle$ is the *initial state* of the game.
- $\rho_e \subseteq \vec{X} \times \vec{Y} \times \vec{X}$ is the *environment transition relation*, which relates each game state to its possible successor environment states (or *moves*).
- $\rho_c \subseteq \vec{X} \times \vec{Y} \times \vec{X} \times \vec{Y}$ is the *controller transition relation*, which relates each game state and environment move to the possible controller replies. Notice that formally the projection of ρ_c on $\vec{X} \times \vec{Y} \times \vec{X}$, which does not include the controller response, is trivially ρ_e.

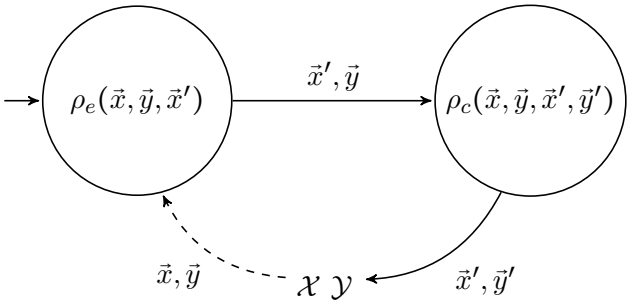


Figure 3.10. A turn of a 2GS.

The idea of 2GS is that from a current game state $\langle \vec{x}, \vec{y} \rangle$, the environment *moves* by choosing an $\vec{x}'$ such that $\rho_e(\vec{x}, \vec{y}, \vec{x}')$ holds, and, after this, the controller *replies back* by choosing a $\vec{y}'$ such that $\rho_c(\vec{x}, \vec{y}, \vec{x}', \vec{y}')$ holds. Intuitively, a game structure represents the rules of a game played by these two adversaries, the *environment* and the *controller*. More precisely, the game structure defines the *constraints* each player is subject to when moving (but not the goal of the game).
Given a 2GS G as above, one can express the *winning conditions for the controller* (i.e., its goal) through a *goal formula* over G. To express such goal formulas, we use a variant of the μ-calculus (Emerson, 1996) interpreted over game structures. The key building block is the operator $\odot\Psi$ interpreted as follows

$$\begin{aligned} &\langle \vec{x}, \vec{y} \rangle \vDash \odot\Psi \text{ iff} \\ &\quad \exists \vec{x}'.\rho_e(\vec{x}, \vec{y}, \vec{x}') \wedge \\ &\quad \forall \vec{x}'.\rho_e(\vec{x}, \vec{y}, \vec{x}') \rightarrow \exists \vec{y}'.\rho_c(\vec{x}, \vec{y}, \vec{x}', \vec{y}') \text{ s.t. } \langle \vec{x}', \vec{y}' \rangle \vDash \Psi. \end{aligned}$$

In English, this operator expresses the following: *for every move $\vec{x}$ of the environment from the game state $\langle \vec{x}, \vec{y} \rangle$, there is a move $\vec{y}'$ of controller such that in the resulting state of the game $\langle \vec{x}', \vec{y}' \rangle$ the property Ψ holds.* With this operator at hand, we develop the whole μ-calculus as follows:

$$\Psi \leftarrow \varphi \mid Z \mid \Psi_1 \wedge \Psi_2 \mid \Psi_1 \vee \Psi_2 \mid \odot\Psi \mid \mu Z.\Psi \mid \nu Z.\Psi,$$

where φ is an arbitrary boolean expression built from propositions of the form $(x_i = \bar{x}_i)$ and $(y_i = \bar{y}_i)$; Z is a predicate variable; $\odot\Psi$ is as defined above; and μ (resp. ν) is the least (resp. greatest) fixpoint operator from the μ-calculus. We say that a 2GS G satisfies goal formula Ψ, written $G \vDash \Psi$, if and only if $start \vDash \Psi$.
We recall that one can express arbitrary temporal/dynamic properties using least and greatest fixpoints constructions (Emerson, 1996). For instance, to express that the controller wins the game if a state satisfying a formula φ is reached from the initial state, one can write $G \vDash \diamondsuit\varphi$, where:

$$\diamondsuit\varphi \doteq \mu Z.\ \varphi \vee \odot Z.$$

Similarly, a greatest fixpoint construction can be used to express the ability of the controller to maintain a property φ, namely, we write $G \vDash \boxdot\varphi$, where:

$$\boxdot\varphi \doteq \nu Z.\varphi \wedge \odot Z.$$

Fixpoints can be also nested into each other, for example:

$$\boxdot\diamondsuit\varphi \doteq \nu Z_1.(\mu Z_2.((\varphi \wedge \odot Z_1) \vee \odot Z_2))$$

expresses that the controller has a strategy to force the game so that it is *always* the case that *eventually* a state where φ holds is reached.
In general, we shall be interested in checking whether the goal formula is satisfied in a game structure, which amounts to *model checking* the game structure. In fact, such a form of model checking is essentially identical to the standard model checking of transition systems (Clarke et al., 1999c), except for the computation of the pre-images, which, in the case of game structures are based on the operator $\odot\Psi$. Hence, one can apply classical results in model checking for μ-calculus (Emerson, 1996), thus obtaining a computational characterization of the complexity of checking goal formulas in 2GSs.

Theorem 7. Checking a goal formula Ψ over a game structure $G = \langle \mathcal{X}, \mathcal{Y}, start, \rho_e, \rho_c \rangle$ can be done in time

$$O((|G| \cdot |\Psi|)^k),$$

where $|G|$ denotes the number of game states of G plus $|\rho_e| + |\rho_c|$, $|\Psi|$ is the size of formula Ψ (considering propositional formulas as atomic), and k is the number of nested fixpoints sharing the same free variables in Ψ.

Proof. The thesis follows from the results in (Emerson, 1996) on fixpoints computations and from the definition of $\odot\Psi$, which, though more sophisticated than in standard μ-calculus, only involves local checks, that is, checks on transitions and states directly connected to the current state. ■

We are not merely interested in verifying goal formulas, but, also and more importantly, in *synthesizing* strategies to actually fulfill them. A (controller) *strategy* is a partial function $f : (\vec{X} \times \vec{Y})^+ \times \vec{X} \mapsto \vec{Y}$ such that for every sequence $\lambda = \langle \vec{x}_0, \vec{y}_0 \rangle \cdots \langle \vec{x}_n, \vec{y}_n \rangle$ and every $\vec{x}' \in \vec{X}$ such that $\rho_e(\vec{x}_n, \vec{y}_n, \vec{x}')$ holds, it is the case that $\rho_c(\vec{x}_n, \vec{y}_n, \vec{x}', f(\lambda, \vec{x}'))$ applies. We say that a strategy f is *winning* if by resolving

the controller existential choice in evaluating the formulas of the form $\odot\Psi$ according to f, the goal formula is satisfied. Notably, model checking algorithms provide a *witness* of the checked property (Clarke et al., 1999c; Piterman et al., 2006b), which, in our case, consists of a *labeling* of the game structure produced during the model checking process. From *labelled* game states, one can read how the controller is meant to react to the environment in order to fulfill the formulas that *label* the state itself, and from this, define a strategy to fulfill the goal formula.

3.2.2 Conditional Planning

To better understand how game structures work, we first use them to capture conditional planning with full observability (Rintanen, 2004a; Ghallab et al., 2004).
Let $\mathcal{D} = \langle P, A, S_0, \rho\rangle$ be a (nondeterministic) dynamic domain, where: *(i)* $P = \{p_1, \ldots, p_n\}$ is a finite set of *domain propositions*, and a *state* is a subset of 2^P; *(ii)* $A = \{a_1, \ldots, a_r\}$ is the finite set of *domain actions*; *(iii)* $S_0 \in 2^P$ is the *initial state*; *(iv)* $\rho \subseteq 2^P \times A \times 2^P$ is the *domain transition relation*. We freely interchange notations $\langle S, a, S'\rangle \in \rho$ and $S \xrightarrow{a} S'$.
Suppose next that φ is the propositional formula over P expressing the (reachability) goal for $\mathcal{D}$. We then define the game structure $G_{\mathcal{D}} = \langle \mathcal{X}, \mathcal{Y}, \mathit{start}, \rho_e, \rho_c\rangle$ as follows:

- $\mathcal{X} = P$ and $\mathcal{Y} = \{act\}$, with act ranging over $A \cup \{a_{init}\}$;
- $\mathit{start} = \langle S_0, a_{init}\rangle$;
- $\rho_e = \rho \cup \{\langle S_0, a_{init}, S_0\rangle\}$;
- $\rho_c(S, a, S', a')$ iff for some $\mathcal{S}'' \in 2^P$, $\rho(S', a', S'')$ holds (i.e., the action a' is executable next).

In words, the environment plays the role of the the nondeterministic domain $\mathcal{D}$, while the controller is meant to capture a plan. At each step, the controller chooses an action act, which must be executable in the current state of the domain (fourth point above). Once the controller has selected a certain action, the environment plays its turn by choosing the next state to evolve to (third point above). This move basically involves resolving the nondeterminism of the action act selected by the controller. A special action a_{init} is used as the initial (dummy) controller move, which keeps the environment in $\mathcal{D}$'s initial state S_0.
Finally, we represent the planning goal φ as the goal formula $\diamondsuit\varphi$ over $G_{\mathcal{D}}$. Such a formula requires that, no matter how the environment moves (i.e., how domain $\mathcal{D}$ happens to evolve), the controller guarantees reachability of a game state where φ holds (i.e., a domain state satisfying φ).

Theorem 8. There exists a conditional plan for reaching goal φ in the dynamic domain $\mathcal{D}$ iff $G_{\mathcal{D}} \models \diamondsuit\varphi$.

As discussed above, we can check such a property by standard model checking. What is more, from a witness, we can directly compute a winning strategy f, which corresponds to a conditional plan for goal φ in domain $\mathcal{D}$.
As for computational complexity, by applying Theorem 7 and considering that the goal formula $\diamondsuit\varphi \doteq \mu Z.\ \varphi \vee \odot Z$ has no nested fixpoints, we get that such a technique

computes conditional plans in $O(|G|) = O(|2^P| \cdot |A| + |\rho|)$. That is, its complexity is polynomial in the size of the domain and exponential in its representation, matching the problem complexity, which is EXPTIME-complete (Rintanen, 2004a).
It is worth noting that, although we have focused on standard conditional planning, similar reductions can be done for other forms of planning with full observability. In particular, all planning accounts tackled via model checking of CTL, including strong cyclic planning (Cimatti et al., 2003), can be directly recast as finding a winning strategy for a goal formula without nesting of fixpoints in a 2GS as above. Also the propositional variant of frameworks where both the agent and the domain behaviors are modeled as a Golog-like program (Lespérance et al., 2008) can be easily captured, see (Fritz et al., 2008).

3.2.3 Agent Planning Programs

Next, we consider an advanced form of planning that requires loops and possibly non-terminating plans (De Giacomo et al., 2010b). Given a dynamic domain $\mathcal{D} = \langle P, A, S_0, \rho \rangle$ as above, an *agent planning program* for $\mathcal{D}$ is a tuple $\mathcal{T} = \langle T, \mathcal{G}, t_0, \delta \rangle$, where:

- $T = \{t_0, \ldots, t_q\}$ is the finite set of *program states*;
- $\mathcal{G}$ is a finite set of goals of the form *"achieve ϕ while maintaining ψ,"* denoted by pairs $g = \langle \psi, \phi \rangle$, where ψ and ϕ are propositional formulae over P;
- $t_0 \in T$ is the *program initial state*;
- $\delta \subseteq T \times \mathcal{G} \times T$ is the *program transition relation*. We freely interchange notations $\langle t, g, t' \rangle \in \delta$ and $t \xrightarrow{g} t'$ in $\mathcal{T}$.

Intuitively, an agent planning program provides a structured representation of the different goals that an agent may need to satisfy—it encodes the agent's space of deliberation.
Agent planning programs are *realized* as follows (for the formal definition see (De Giacomo et al., 2010b)): at any point in time, the planning program is in a state t and the dynamic domain in a state S; the agent requests a transition $t \xrightarrow{\langle \psi, \phi \rangle} t'$ in $\mathcal{T}$ (e.g., $t \xrightarrow{\langle \neg Driving, At(pub) \rangle} t'$, i.e., be at the pub and never be driving); then, a plan π from S that leads the dynamic domain to a state satisfying ϕ, while only traversing states where ψ holds, is synthesized—notice that such a plan *must also guarantee the continuation of the program*; upon plan completion, the agent planning program moves to t' and requests a new transition, and so on. Notice also that, at any point in time, all possible choices available in the agent planning program must be guaranteed by the system, since the actual request that will be made is not known in advance—the whole agent's space of deliberation ought to be accounted for.
For example, imagine a planning program for specifying the routine habits of a young researcher. Initially, the researcher is at home, from where she may choose to go to work or to a friend's house. After work, she may want to go back home or to a pub, and so on. So, in order to fulfill the initial possible request to go to work, plans involving driving or taking a bus to the lab are calculated and one of them is chosen. Further plans are then calculated to fulfill the request of going to

the pub, and so on. Now suppose that the researcher would like to always leave the car home when going to the pub. Then, plans involving driving to work are not appropriate, not because they fail to satisfy the goal of being at work, but because they would prevent the fulfillment of further goals (namely, going to the pub with the car left at home). Thus, plans must not only fulfill their goals, but must also make fulfilling later requests encoded in the structure of the program, possibly within loops, possible.

Let us now show how the problem of finding a realization of a planning program $\mathcal{T}$ can be reduced to building a winning strategy for a goal over a 2GS. Precisely, from $\mathcal{D}$ and $\mathcal{T}$, we shall build a 2GS G and a goal formula φ, such that $G \vDash \varphi$ iff $\mathcal{T}$ is realizable in $\mathcal{D}$. Also, from a witness of the check $G \vDash \varphi$, we shall extract a winning strategy f that corresponds to a realization of $\mathcal{T}$.

The construction of $G = \langle \mathcal{X}, \mathcal{Y}, \mathit{start}, \rho_e, \rho_c \rangle$ is as follows. The set of *environment variables* is $\mathcal{X} = \{P, tr_{\mathcal{T}}\}$, where $tr_{\mathcal{T}}$ ranges over $\delta \cup \{tr_{init}\}$, that is, the set of $\mathcal{T}$'s transitions (plus tr_{init}, for technical convenience only). The set of *controller variables* is $\mathcal{Y} = \{act, last\}$, where variable act ranges over $A \cup \{a_{init}\}$ (again, a_{init} is introduced for convenience), and $last$ is a propositional variable. Variable act stands for the action to be executed next, while $last$ marks the end of current plan's execution.

As for the transitions of the game structure, we have:

- $\mathit{start} = \langle S_0, tr_{init}, a_{init}, \bot \rangle$;
- the *environment transition relation* ρ_e is such that $\rho_e(\langle S, tr \rangle, \langle a, l \rangle, \langle S', tr' \rangle)$ iff:
  - tr is a transition $t \xrightarrow{\langle \psi, \phi \rangle} t' \in \delta$;
  - $S \vDash \psi$ and $S \xrightarrow{a} S' \in \rho$, i.e., S both fulfills the maintenance goal required by tr and enables a's execution;
  - if $l = \bot$, then $tr'_{\mathcal{T}} = tr_{\mathcal{T}}$, i.e., if a is not the last action and thus the transition realization is not completed, then the planning program keeps requesting the same (current) transition $tr_{\mathcal{T}}$;
  - if $l = \top$, then $S' \vDash \phi$ and $tr'_{\mathcal{T}}$ is any $t' \xrightarrow{\langle \psi', \phi' \rangle} t'' \in \delta$, i.e., if a is indeed the last action for tr realization, then goal ϕ is indeed achieved and a new $\mathcal{T}$ transition is chosen according to δ.

  Furthermore, for each initial transition $t_0 \xrightarrow{\psi, \phi} t' \in \delta$ we have that $\rho_e(\langle S_0, tr_{init} \rangle, \langle a_{init}, \bot \rangle, \langle S_0, t_0 \rightarrow t' \rangle)$, capturing all possible initial moves for the environment;
- the *controller transition relation* ρ_s is such that $\rho_s(\langle S, tr \rangle, \langle a, l \rangle, \langle S', tr' \rangle, \langle a', l' \rangle)$ iff there exists a transition $S' \xrightarrow{a'} S''$ in $\mathcal{D}$ for some state S'' (i.e., action a' is executable in current domain state S'). Observe no constraint is required on controller variable l'.

Intuitively, G represents the synchronous evolution of domain $\mathcal{D}$ and program $\mathcal{T}$, which together form the environment, operated by the controller. The environment includes all $\mathcal{D}$'s and all $\mathcal{T}$'s transitions, both chosen nondeterministically from the

controller's viewpoint. The controller, on the other hand, represents the possible decisions made at each step, namely, the action to be performed next and the notification for plan completion, in order to fulfill the requests issued by the environment. Observe that $\mathcal{T}$ can issue a new transition request *only after* its current request is deemed fulfilled by the controller (i.e., *last* holds).
As for the goal formula, we have $\varphi = \boxdot\diamondsuit last$, which requires that it is *always* the case that *eventually* the controller does reach the end of the (current) plan, thus fulfilling the (current) request issued by program $\mathcal{T}$.

Theorem 9. Let $\mathcal{T}$ be a planning program over a dynamic domain $\mathcal{D}$, and G be the corresponding 2GS built as above. Then, there exists a realization of $\mathcal{T}$ in $\mathcal{D}$ iff $G \vDash \boxdot\diamondsuit last$.

Since $\varphi = \boxdot\diamondsuit last$ has two nested fixpoints, we get that such a technique computes a realization of $\mathcal{T}$ in $\mathcal{D}$ in $O(|G|) = O((|2^P|\cdot(|\delta|+|\rho|))^2)$, i.e., in polynomial time in the size of $\mathcal{D}$ and $\mathcal{T}$ and in exponential time in their representations. This upperbound is indeed *tight*, as conditional planning, which is a special case of agent planning program realization, is already EXPTIME-complete.

3.2.4 Multitarget Agent Composition

We now turn to agent composition in the style of (Sardina et al., 2008; Su, 2008c) (and introduced in Section 3.1.1). In particular, we focus on a sophisticated form of composition originally proposed in (Sardina and De Giacomo, 2008), extending the formalization of Section 3.1.1, which is instead limited to classical case of a single target behavior. Namely, we want to realize a *collection* of independent (target) *virtual agent behaviors* that are meant to act autonomously and asynchronously on a shared environment. For example, a surveillance agent and a cleaning agent behaviors, among others, may all operate in the same smart house environment. Such agents have no fixed embodiment, but must be concretely realized by a set of available *devices* (e.g., a vacuum cleaner, microwave, or video camera) that are allowed to "join" and "leave" the various agents, dynamically depending on agent's requests. Each agent's embodiment is dynamically transformed while in execution. Both agent behaviors and devices (i.e., their logics) are described by *transition systems* of the form $TS = \langle A, S, s_0, \delta\rangle$, where: *(i)* A is a finite set of actions; *(ii)* S is the finite set of possible states; *(iii)* $s_0 \in S$ is the initial state of TS; and *(iv)* $\delta \subseteq S \times A \times S$ is the transition relation, with $\langle s, a, s'\rangle \in \delta$ or $s \xrightarrow{a} s'$ denoting that TS may evolve to state s' when action a is executed in state s. We assume, wlog, that each state may evolve to at least one next state. Intuitively, the transition systems for the agents encode the space of deliberation of these agents, whereas the transition systems for the devices encode the *capabilities* of such artifacts.
So, we consider a tuple of *available devices* $\langle \mathcal{B}_1, \ldots, \mathcal{B}_n\rangle$, where $\mathcal{B}_i = \langle A^{\mathcal{B}}, B_i, b_{0i}, \delta_i\rangle$, and a tuple of *virtual agents* $\langle \mathcal{T}_1, \ldots, \mathcal{T}_m\rangle$, where $\mathcal{T}_i = \langle A_{\mathcal{T}}, T_i, t_{0i}, \varrho_i\rangle$. Observe that, differently from the setting of Section 3.1.1, we disregard here the final states of devices.
All $\mathcal{T}_i$'s are *deterministic*, i.e., fully controllable—in the sense that there is no uncertainty on the resulting state obtained by executing an action—whereas each $\mathcal{B}_i$ may

be *nondeterministic*, i.e., only partially controllable (though their current state is fully observable). The composition problem we are concerned with involves guaranteeing the concurrent execution of the virtual agents *as if each of them were acting in isolation*, though, in reality, they are all collectively realized by the same set of (actual) available devices. A solution to this problem amounts to synthesizing a *controller* that intelligently delegates the actions requested by virtual agents to concrete devices. The original problem, which was shown to be EXPTIME-complete, allowed the controller to assign agents' requested actions to devices *without* simultaneously progressing those agents whose actions have been done. In other words, the controller may instruct the execution of an action in a certain device without stating to which particular agent it corresponds. Obviously, after m steps (i.e., the number of agents), no more requests are pending so some agent is allowed to issue a new request (Sardina and De Giacomo, 2008). 2GSs can be used to encode and solve this problem within the same complexity bound as the original solution.
Here we detail an interesting variant of this problem, in which the served agent is progressed, simultaneously, when an action is executed by a device. Differently from the original problem, such a variant requires to actually identify which devices are "embodying" each agent *at every point in time*. Also, it shows how easy it is to tackle composition variants using 2GSs.
Specifically, from tuples $\langle \mathcal{B}_1, \ldots, \mathcal{B}_n \rangle$ and $\langle \mathcal{T}_1, \ldots, \mathcal{T}_m \rangle$, we build a 2GS $G = \langle \mathcal{X}, \mathcal{Y}, \mathit{start}, \rho_e, \rho_c \rangle$ as follows. The *environment variables* are $\mathcal{X} = \{s_1^{\mathcal{B}}, \ldots, s_n^{\mathcal{B}}, s_1^{\mathcal{T}}, \ldots, s_m^{\mathcal{T}}, r_1^{\mathcal{T}}, \ldots, r_m^{\mathcal{T}}\}$, where variables $s_i^{\mathcal{B}}$, $s_i^{\mathcal{T}}$ and $r_i^{\mathcal{T}}$ range over B_i, T_i, and $A_{\mathcal{T}}$, respectively. Each $s_i^{\mathcal{B}}$ corresponds to the current state of $\mathcal{B}_i$, $s_i^{\mathcal{T}}$ to the current state of $\mathcal{T}_i$, and $r_i^{\mathcal{T}}$ to the last request issued by $\mathcal{T}_i$. The *controller variables*, on the other hand, are $\mathcal{Y} = \{\mathit{dev}, \mathit{full}\}$, where *dev* ranges over $\{0, \ldots, n\}$ and *full* over $\{1, \ldots, m\}$. Variable *dev* stores the index of the available device selected to execute the action (0 being a dummy value denoting the game's initial state). Variable *full* stores the index of the virtual agent whose request is to be fulfilled. As before, we denote assignments to $\mathcal{X}$-variables as $\vec{x} \in \vec{X} = B_1 \times \cdots \times B_n \times T_1 \times \cdots \times T_m \times (A_{\mathcal{T}})^m$, and assignments to $\mathcal{Y}$-variables as $\vec{y} \in \vec{Y} = \{0, \ldots, n\} \times \{1, \ldots, m\}$.
The *initial state* is $\mathit{start} = \langle \vec{x}_0, \vec{y}_0 \rangle$, with $\vec{x}_0 = \langle b_{01}, \ldots, b_{0n}, t_{01}, \ldots, t_{0m}, a, \ldots, a \rangle$, for a arbitrarily chosen in $A_{\mathcal{T}}$, and $\vec{y}_0 = \langle 0, 1 \rangle$, i.e., the only state where $\mathit{dev} = 0$.
The *environment transition relation* $\rho_e \subseteq \vec{X} \times \vec{Y} \times \vec{X}$ is such that, for $\langle \vec{x}, \vec{y} \rangle \neq \mathit{start}$, $\langle \vec{x}, \vec{y}, \vec{x}' \rangle \in \rho_e$ iff for $\vec{x} = \langle b_1, \ldots, b_n, t_1, \ldots, t_m, r_1, \ldots, r_m \rangle$, $\vec{y} = \langle k, f \rangle$, and $\vec{x}' = \langle b'_1, \ldots, b'_n, t'_1, \ldots, t'_m, r'_1, \ldots, r'_m \rangle$ we have that:

- there exists a transition $\langle b_k, r_f, b'_k \rangle$ in δ_k, i.e., the selected device actually executes the action requested by the agent to be fulfilled (i.e., agent $\mathcal{T}_f$); while for each $i \neq k$, $b'_i = b_i$ applies, that is, non-selected devices remain still;
- the f-th agent moves (deterministically) according to transition $\langle t_f, r_f, t'_f \rangle \in \tau_f$, and from t'_f there exists a further transition $\langle t'_f, r'_f, t''_f \rangle \in \tau_f$, for some t''_f, i.e., the virtual agent whose request is fulfilled moves according to its transition function and issues a new (legal) request; while for each $i \neq f$, $t'_i = t_i$ and $r'_i = r_i$ apply, i.e., all virtual agents whose request are not yet fulfilled remain still.

In addition, from $\mathit{start} = \langle \vec{x}_0, \vec{y}_0 \rangle$, we have that $\langle \vec{x}_0, \vec{y}_0, \vec{x} \rangle \in \rho_e$ with $\vec{x} = \langle b_{01}, \ldots, b_{0n}, t_{01}, \ldots, t_{0m}, r_1, \ldots$ for r_i such that $\langle t_{0i}, r_i, t \rangle \in \varrho_i$, for some $t \in T_i$, i.e., from the initial state each virtual

agent issues an initial legal request.
Finally, the *controller transition relation* $\rho_c \subseteq \vec{X} \times \vec{Y} \times \vec{X} \times \vec{Y}$ is such that $\langle \vec{x}, \vec{y}, \vec{x}', \vec{y}' \rangle \in \rho_c$ iff $\langle \vec{x}, \vec{y}, \vec{x}' \rangle \in \rho_e$ and for $\vec{x}' = \langle b'_1, \ldots, b'_n, t'_1, \ldots, t'_m, r'_1, \ldots, r'_m \rangle$ and $\vec{y}' = \langle k', f' \rangle$, there exists a transition $\langle b'_{k'}, r_{f'}, b''_{k'} \rangle \in \delta_{k'}$, for some $b''_{k'}$, i.e., the action requested by agent $\mathcal{T}_{f'}$ is actually executable by device $\mathcal{B}_{k'}$. That completes the definition of 2GS G.
Now, on such 2GS, we define the *goal formula* φ simply as $\varphi = \bigwedge_{f=1}^{m} \boxdot\diamondsuit(full = f)$, thus requiring that each time a virtual agent request is issued, it is eventually fulfilled.

Theorem 10. There exists a composition for the virtual agent behaviors $\langle \mathcal{T}_1, \ldots, \mathcal{T}_m \rangle$ by the available devices $\langle \mathcal{B}_1, \ldots, \mathcal{B}_n \rangle$ (according to the assumptions considered here) iff $G \models \varphi$, for G and φ constructed as above.

From a witness of $G \models \varphi$, one can extract an actual controller able to do the composition. As for complexity, since the goal formula contains two nested fixpoints, the time needed to check the existence of a composition (and to actually compute one) is $O(|G|^2) = O((|\vec{X}| \cdot |\vec{Y}| + |\rho_e| + |\rho_c|)^2)$, where $|\vec{X}| = O(|B_{max}|^n \cdot |T_{max}|^m \cdot |A_{\mathcal{T}}|^m)$; $|\vec{Y}| = n \cdot m + 1$; $|\rho_e| = O(|\vec{X}| \cdot |\vec{Y}| \cdot |B_{max}| \cdot |A_{\mathcal{T}}|)$; and $|\rho_c| = O(|\rho_e| \cdot m \cdot n)$, with B_{max} and T_{max} being the maximum number of states among devices and virtual agents, respectively. Such a bound, under the natural assumption that the number of actions is at most polynomial in the number of states of virtual agents, reduces to $O(u^{m+n})$, where $u = \max\{B_{max}, T_{max}\}$, which is essentially the complexity of the problem (Sardina and De Giacomo, 2008).

3.2.5 Implementation

The approach presented here is ready implementable with model checking tools. However, the need of quantifying separately on environment and controller moves, requires the use of μ-calculus, and not simply CTL or LTL (Clarke et al., 1999c) for which model checking tools are much more mature. As an alternative, if the goal formula does not require nested fixpoints, we can adopt ATL model checkers (Lomuscio et al., 2009). ATL interpretation structures are indeed quite related to 2GSs, and we can split the set of ATL agents into two coalitions representing (possibly in a modular way) the environment and the controller—e.g., (De Giacomo and Felli, 2010) encodes single target composition in ATL.
Another alternative is to use (procedure *synth* in) the TLV system (Piterman et al., 2006b), which is a framework for the development of BDD-based procedures for formal synthesis/verification. TLV takes a generic 2GS G expressed in the concrete specification language SMV (Clarke et al., 1999c), and uses special SMV *modules* `Env` and `Sys` to distinguish the environment and the controller. Also, it takes an LTL formula φ of "GR1" form: $\varphi = \bigwedge_{i=1}^{n} \Box\diamondsuit p_i \longrightarrow \bigwedge_{j=1}^{m} \Box\diamondsuit q_j$, with p_i and q_j propositions over G. Then, it synthesizes, if any, a controller strategy to satisfy φ, by transforming φ into a μ-calculus formula φ_μ over G, which quantifies separately on the environment and the controller. Because of the fixed structure of φ, the resulting φ_μ has 3 nested fixpoints.
It turns out that TLV can be used to solve both agent planning programs and multi target composition problems; indeed, the goal formulas φ_g of such problems, are

special cases of the φ_μ that TLV uses. Specifically, for the multi target problem, we can proceed as follows. To encode a 2GS in SMV, one essentially needs to describe both controller's and environment's behaviors. They are actually encoded as SMV modules—each representing a transition system, with its initial state, state variables and transition rules, specified in SMV sections `VAR`, `INIT` and `TRANS`, respectively—possibly communicating via input/output parameters. We do so by defining: *(i)* one SMV module for each available device $\mathcal{B}_i$ and one for each virtual agent $\mathcal{T}_i$, wrapping them in an `Env` module to form the 2GS environment; and *(ii)* a further `Sys` module, representing the controller, which communicates with all available devices, to delegate them action executions, and with all virtual agents, to inform them of request fulfillment. The controller is modeled so as to guarantee that action delegations are always *safe*, i.e, the device instructed to execute the action can actually do it in its current state. As for goal formula, this corresponds to the μ-calculus formula obtained in TLV by processing the GR1 LTL formula $\Box\Diamond\texttt{true} \longrightarrow \bigwedge_{i=1}^{n} \Box\Diamond\texttt{fulfilled}_i$ (n number of virtual agents), which technically is encoded as a list $\texttt{fulfilled}_1 \ldots \texttt{fulfilled}_n$ in the `JUSTICE` section of controller's module. We used TLV to actually solve several variants of the example in (Sardina and De Giacomo, 2008), using the version of the multitarget composition introduced above. The tool was able to find the solution for those simple problems in around 10 seconds (on a Core2Duo 2.4Ghz laptop with 2GB of RAM).

3.2.6 Discussion

One of the main contributions of this work is to show that various agent behavior synthesis problems can be treated uniformly by relying on the explicit distinction between the roles of the system and the environment. Formally, such a distinction allows for separately quantifying over both environment's and controller's moves. As seen in the introduction to this thesis, this concept is not new in synthesis of reactive systems, and was largely inspired by those game structures used in synthesis by model checking in Verification (see, e.g., (Piterman et al., 2006a)). Indeed, when considering a linear temporal specifications $\varphi(x, y)$ over input x and output y, the synthesis task amounts to build a program satisfying the branching-time formula $(\forall x)(\exists y)A\varphi(x, y)$ that explicitly captures the alternation of environment's and controller's moves (Pnueli and Rosner, 1989b).

We have shown that model checking 2GS's is a very powerful, yet fully automatically manageable, technique for synthesis in a variety of AI contexts under full observability. By observing that most current model checking algorithms are based on the so-called "global" model checking, we conclude that these can be seen as a generalization of planning by backward reasoning when applied to 2GSs. Model checking algorithms based on forward reasoning are also available, the so-called "local" model checking techniques (Stirling and Walker, 1991), though they are currently considered less effective in Verification (Clarke et al., 1999c). On the contrary, within AI, planning by forward reasoning is currently deemed the most effective, mainly due to use of heuristics that seriously reduce the search space. An interesting direction for future research, thus, would be to apply local model checking to 2GSs, while taking advantage of such heuristics developed by the automated planning community.

3.3 A case study: Smart Homes

The emerging trend in process management and in service-oriented applications is to enable the composition of new distributed processes on the basis of user requests, through (parts of) available (and often embedded in the environment) behaviors to be composed and orchestrated in order to satisfy such requests. Here, we consider a user process as specified in terms of repeated goals that the user may choose to get fulfilled, namely a Planning Program. Observe that this is the same setting as in Section 3.2.3. As customary in behavior composition, available agent behaviors are suitably composed and orchestrated in order to realize such a process. In particular we focus on smart homes, in which available devices (corresponding to a materialization of agent behaviors) are those ones offered by sensor and actuator devices deployed in the home, and the target user process is directly and continuously controlled by the inhabitants, through actual goal choices.
As done previously in the previous sections, we adopt here the approach as in (Berardi et al., 2005b), often referred to as *Roman Model*, in which again agent behaviors are abstracted as transition systems and the objective is to obtain a composite behavior that preserves a desired interaction, expressed as a (virtual) target behavior.
However, this is a notable extension of the Roman Model, where *goal-based processes* -inspired by the agent planning programs (De Giacomo and Felli, 2010) first introduced in Section 3.2.3- are used, instead of target behaviors, to specify what the user desires to achieve. Such processes can be thought of as *routines* built from virtual tasks expressed declaratively simply as *goals*, which allow users to specify the desired state of affair to bring about. Such goals are organized in a control flow structure, possibly involving loops that regulates their sequencing, as well as the decision points where the user can choose the next goal to request. Hence, clients are able to request new goals, once the current one is achieved (by a plan). These routines typically involve loops, thus ruling out naïve approaches based on (classical, conditional or conformant) planning. Indeed, not all plans that achieve a requested goal are successful: some might lead the system to states preventing future client requests fulfillment. Such *bad* plans could be recognized by taking into account *all* goals the client can request in the future, which, in the presence of loops, span over an infinite horizon (though finite-state).
The approach proposed here is strongly motivated by challenging applications in the domain of smart houses[3] , i.e., buildings pervasively equipped with sensors and actuators making their functionalities available according to the service-oriented paradigm. In order to be dynamically configurable and composable, embedded services need to expose semantically rich service descriptions, comprising *(i)* interface specifications and *(ii)* specifications of the externally visible behaviors. Moreover, human actors in the environment can be abstracted as behaviors, and actually "wrapped" by a semantic description (e.g., a nurse offering medical services). This allows them to be involved in orchestrations and to collaborate with devices, to

[3]The European-funded project SM4All (Smart hoMes for All – `http://www.sm4all-project.eu`), from which the model proposed here has been inspired, aims at developing an innovative platform for embedded services in ubiquitous and person-centric environments.

reach certain goals. See, e.g., (Kaldeli et al., 2010).
We envision a user that can express processes she would like to have realized in the house, in the form of routines consisting of goals (e.g., states of the house she would like to have realized); an engine automatically synthesizes the right orchestration of behaviors able to satisfy the goals. Users can interact with the house through different kinds of interfaces, either centralized (e.g., in a home control station) or distributed, and embedded in specific interface devices. Brain Computer Interfaces (BCIs) allow also people with disabilities to interact with the system. Using such interfaces, users issue specific goals to the system, which is, in turn, expected to react and satisfy the request.
In this chapter, we detail this approach. We first provide a framework for composition of goal-oriented processes from available (non-atomic) behaviors. A said before, this framework is inspired to the agent planning programs (De Giacomo and Felli, 2010) first introduced in Section 3.2.3. We then present a case study where the framework is applied in a real smart home setting and provide a effective solver, which synthesize an orchestrator that realizes the target goal-oriented processes, by detailing subprocesses that fulfil the various goals at the various point in time. Our solver is sound and complete, and far more practical than other solutions proposed in literature, also because it easily allows for exploiting heuristic in the search for the solution. Finally, some practical experiments are illustrated.

3.3.1 Framework

We assume that the user acts on an environment that is formalized as a possibly nondeterministic *dynamic domain* $\mathcal{D}$, which provides a symbolic abstraction of the world that the user acts in. Formally, a *dynamic domain* is a tuple $\mathcal{D} = \langle P, A, D_0, \rho \rangle$, where:

- $P = \{p_1, \ldots, p_n\}$ is a finite set of *domain propositions*. $D \in 2^P$ is a *state*;
- $A = \{a_1, \ldots, a_r\}$ is the finite set of *domain actions*;
- $D_0 \in 2^P$ is the *initial state*;
- $\rho \subseteq 2^P \times A \times 2^P$ is the *transition relation*. We freely interchange notations $\langle D, a, D' \rangle \in \rho$ and $D \xrightarrow{a} D'$ in $\mathcal{D}$.

Intuitively, a dynamic domain models an environment whose states are described by the set P of boolean propositions, holding all relevant information about the current situation. For instance, the state of a room can be defined by the light being on or off and the door being open or closed, using two propositions *light_on* and *door_open*. By convention, we say that if one of such propositions is in the current state of $\mathcal{D}$, then it evaluates to $\top$ (true), otherwise it is $\bot$ (false). Hence, a propositional formula φ over P *holds* in a domain state $D \in 2^P$ ($D \vDash \varphi$) if φ evaluates to $\top$ when all of its propositions occurring in D are replaced by $\top$. However, such domain can not be manipulated directly, i.e., domain actions can not be accessed directly by the user: they are provided through *available behaviors*. The idea is that, at each moment, a behavior offers a set of possible actions, and the user can interact with the domain $\mathcal{D}$ *only* by means of available behaviors.

Given a dynamic domain $\mathcal{D}$, a *behavior* over $\mathcal{D}$ is a tuple $\mathcal{B} = \langle B, O, b_0, \varrho \rangle$, where: *(i)* B is the finite set of behavior states; *(ii)* O is the finite set of behavior actions over the domain, i.e., $O \cap A \neq \varnothing$; *(iii)* $b_0 \in B$ is the behavior initial state; *(iv)* $\varrho \subseteq B \times O \times B$ is the behavior transition relation. We will interchange notations $\langle b, a, b' \rangle \in \varrho$ and $b \xrightarrow{a} b'$ in $\mathcal{B}$.

As a behavior is instructed to perform an action over $\mathcal{D}$, both the behavior and the domain evolve *synchronously* (and possibly *nondeterministically*) according to their respective transition relations. So, for a domain action to be carried out, it needs to be both compatible with the domain and (currently) available in some behavior. However, behaviors can also feature *local* actions, i.e., actions whose execution does not affect the domain evolution. For instance, a behavior representing a physical device might require to be switched on to use all its functionalities, a fact that is not captured by $\mathcal{D}$ alone. To define formally this idea, we introduce the notiton of executability for actions of a behavior $\mathcal{B} = \langle B, O, b_0, \varrho \rangle$: given $\mathcal{B}$ in its own behavior state b and a domain $\mathcal{D}$ in domain state D, action $a \in O$ is said to be *executable* by $\mathcal{B}$ in b iff *(i)* it is available in b, i.e. $b \xrightarrow{a} b'$ in $\mathcal{B}$ for some state $b' \in B$ and *(ii)* it is either a local action ($a \notin O \cap A$) or it is allowed in D, i.e., there exists a domain state D' such that $D \xrightarrow{a} D'$. Notice that behaviors are loosely-coupled with the domain they are interacting with: new behaviors can be easily added to the systems and modifications to the description of the underlying domain don't affect them.

Example 14. *Consider a dynamic domain $\mathcal{D} = \langle P, A, D_0, \rho \rangle$ describing (among othet components) a simple door as in Figure 3.12a. A domain proposition* `door_open` $\in P$ *is used to keep its state, and the door can be either closed or opened executing domain actions $\{doClose, doOpen\} \subseteq A$. However, the door can only be managed through a behavior $doorSrv = \langle \{open, closed\}, \{doOpen, doClose\}, open, \varrho \rangle$ where ϱ is such that $open \xrightarrow{doClose} closed$ and $closed \xrightarrow{doOpen} open$. Assume that $doorSrv$ is in its state open, and the current domain state D to be such that* `door_open` $\in D$ *(i.e., it evaluates to true in D). As soon as action $doClose$ is executed, both the $doorSrv$ behavior evolves changing its state to closed and,* synchronously*, the domain evolves to a state D' such that* `door_open` $\notin D'$ *(i.e., it evaluates to false in D').*

□

Given a dynamic domain $\mathcal{D}$ and a fixed set of available behaviors over it, we define a *dynamic system* to be the resulting global system, seen as a whole: it is an abstract structure used to capture the interaction of available behaviors with the environment. Formally, given a dynamic domain $\mathcal{D}$ and a set of available behaviors $\mathcal{B}_1, \ldots, \mathcal{B}_n$, with $\mathcal{B}_i = \langle B_i, O_i, b_{i0}, \varrho_i \rangle$, the corresponding *dynamic system* is the tuple $\mathcal{S} = \langle S, \Gamma, s_0, \vartheta \rangle$, where:

- $S = (B_1 \times \cdots \times B_n) \times 2^P$ is the set of system states;
- $\Gamma = A \cup \bigcup_{i=1}^{n} O_i$ is the set of system actions;
- $s_0 = \langle \langle b_{10}, \ldots, b_{n0} \rangle, D_0 \rangle \in S$ is the system initial state;
- $\vartheta \subseteq S \times (\Gamma \times \{1, \ldots, n\}) \times S$ is the system transition relation such that $\langle \langle b_1, \ldots b_n \rangle, D \rangle \xrightarrow{a,i} \langle \langle b'_1, \ldots b'_n \rangle, D' \rangle$ is in ϑ iff:

(i) $\langle b_i, a, b_i' \rangle \in \varrho_i$;

(ii) for each $j \in \{1, \ldots, n\}$, if $j \neq i$ then $b_j' = b_j$.

(iii) if $a \in A$ then $\langle D, a, D' \rangle \in \rho$, otherwise $D' = D$;

(i)-(ii) require that only one behavior $\mathcal{B}_i$ moves from its own state b_i to b_i' performing action a, and *(iii)* requires that, if the action performed is not a local action, the domain evolves accordingly. Indeed, the set of system operations Γ includes operations local to behaviors, i.e. whose execution, according to ϑ, does not affect the domain evolution.
We stress the fact that a dynamic system does not correspond to any actual structure: it is a convenient representation of the interaction between the available behaviors and the domain. Indeed, a dynamic system captures the joint execution of a dynamic domain and a set of behaviors where, at each step, only one behavior moves, and possibly affects, through operation execution, the state of the underlying domain. The evolutions of a system $\mathcal{S}$ are captured by its *histories*, herehence $\mathcal{S}$-*histories*. One such history is a finite sequence of the form $\tau = s^0 \xrightarrow{a^1, j^1} s^1 \cdots s^{\ell-1} \xrightarrow{a^\ell, j^\ell} s^\ell$ of length $|\tau| \doteq \ell{+}1$ such that *(i)* $s^i \in S$ for $i \in \{0, \ldots, \ell\}$; *(ii)* $s^0 = s_0$; *(iii)* $s^i \xrightarrow{a^{i+1}, j^{i+1}} s^{i+1}$ in $\mathcal{S}$, for each $i \in \{0, \ldots, \ell - 1\}$. We denote with $\tau|_k$ its k-length (finite) prefix, and with $\mathcal{H}$ the set of all possible $\mathcal{S}$-histories. Given a dynamic system $\mathcal{S}$, a *general plan* is a (possibly partial) function $\pi : \mathcal{H} \longrightarrow \Gamma \times \{1, \ldots, n\}$ that outputs, given an $\mathcal{S}$-history, a pair representing the action to be executed and the index of the behavior which has to execute it. An *execution* of a general plan π from a state $s \in S$ is a possibly infinite sequence $\tau = s^0 \xrightarrow{a^1, j^1} s^1 \xrightarrow{a^2, j^2} \cdots$ such that *(i)* $s^0 = s$; *(ii)* $\tau|_k$ is an $\mathcal{S}$-history, for all $0 < k \leq |\tau|$; and *(iii)* $\langle a^k, j^k \rangle = \pi(\tau|_k)$, for all $0 < k < |\tau|$. When all possible executions of a general plan are finite, the plan is a *conditional plan*. The set of all conditional plans over $\mathcal{S}$ is referred to as Π. Note that, being finite, executions of conditional plans are $\mathcal{S}$-histories. A finite execution τ such that $\pi(\tau)$ is undefined is a *complete execution*, which means, informally, that the execution cannot be extended further. In the following, we shall consider only conditional plans.

We say that an execution $\tau = s^0 \xrightarrow{a^1, j^1} s^1 \cdots s^{\ell-1} \xrightarrow{a^\ell, j^\ell} s^\ell$ of a conditional plan π, with $s^i = \langle \langle b_1^i, \ldots b_n^i \rangle, D^i \rangle$:

- *achieves* a goal ϕ iff $D^\ell \vDash \phi$
- *maintains* a goal ψ iff $D^i \vDash \psi$ for every $i \in \{0, \ldots, \ell - 1\}$

Such notions can be extended to conditional plans: a conditional plan π *achieves* ϕ from state s if all of its complete executions from s do so; and π *maintains* ψ from s if all of its (complete or not) executions from s do.
Finally, we can formally define the notion of *(goal-based) target process* for a dynamic domain $\mathcal{D}$ as a tuple $\mathcal{T} = \langle T, \mathcal{G}, t_0, \delta \rangle$, where:

- $T = \{t_0, \ldots, t_q\}$ is the finite set of *process states*;
- $\mathcal{G}$ is a finite set of goals of the form *achieve* ϕ *while maintaining* ψ, denoted by pairs $g = \langle \psi, \phi \rangle$, where ψ and ϕ are propositional formulae over P;

- $t_0 \in T$ is the *process initial state*;
- $\delta \subseteq T \times \mathcal{G} \times T$ is the *transition relation*. We will also write $t \xrightarrow{g} t'$ in $\mathcal{P}$.

A target process $\mathcal{T}$ is a transition system whose states represent *choice points*, and whose transitions specify pairs of *maintenance* and *achievement* goals that the user can request at each step. Hence, $\mathcal{T}$ allows to combine achievement and maintenance goals so that they can be requested (and hence fulfilled) according to a specific temporal arrangement, which is specified by the relation δ of $\mathcal{T}$. Intuitively, a target process $\mathcal{T}$ is *realized* when a conditional plan π is available for the goal couple $g = \langle \phi, \psi \rangle$ chosen from initial state t_0 and, upon plan's completion, a new conditional plan π' is available for the new selected goal, and so on. In other words, all potential target requests respecting $\mathcal{T}$'s structure (possibly infinite) have to be fulfilled by a conditional plan, which is meant to be executed starting from the state that previous plan execution left the dynamic system $\mathcal{S}$ in (initially from s_0). Since the sequences of goals actually chosen by the user can not be forseen, a realization has to take into account all possible ones: at any point in time, all possible choices available in the target process must be guaranteed by the system, i.e., every legal request needs to be satisfied. We are going to give a formal definition of this intuition (De Giacomo et al., 2010b) in the remainder of this section.

Let $\mathcal{S}$ be a dynamic system and $\mathcal{T}$ a target process. A *PLAN-simulation relation*, is a relation $R \subseteq T \times S$ such that $\langle t, s \rangle \in R$ implies that for each transition $t \xrightarrow{\langle \psi, \phi \rangle} t'$ in $\mathcal{T}$, there exists a conditional plan π such that: *(i)* π *achieves* ϕ and *maintains* ψ from state s and *(ii)* for all π's possible complete executions from s of the form $s \xrightarrow{\pi(\tau|_1)} \cdots \xrightarrow{\pi(\tau|_\ell)} s^\ell$, it is the case that $\langle t', s^\ell \rangle \in R$. A plan π *preserves* R from $\langle t, s \rangle$ for a given transition $t \xrightarrow{\langle \psi, \phi \rangle} t'$ in $\mathcal{T}$ if requirement *(ii)* above holds. Also, we say that a target process state $t \in T$ is *PLAN-simulated* by a system state $s \in S$, denoted $t \preceq_{\mathsf{PLAN}} s$, if there exists a PLAN-simulation relation R such that $\langle t, s \rangle \in R$. Moreover, we say that a target process $\mathcal{T}$ is *realizable* in a dynamic system $\mathcal{S}$ if $t_0 \preceq_{\mathsf{PLAN}} s_0$. When the target process is realizable, one can compute *once for all* (*offline*) a function $\Omega : S \times \delta \longrightarrow \Pi$ that, if at any point in time the dynamic system reaches state s and the process requests transition $t \xrightarrow{\langle \psi, \phi \rangle} t'$ of $\mathcal{T}$, outputs a conditional plan π that its execution starting from s *(i)* achieves ϕ while maintaining ψ and *(ii)* preserves $\preceq_{\mathsf{PLAN}}$, i.e., it guarantees that, for all possible states the system can reach upon π's execution, all target transitions outgoing from t' (according to δ) can still be realized by a conditional plan (possibly returned by the function itself). Such function Ω is referred to as *process realization*.

We can now formally state the problem of concern: *Given a dynamic domain $\mathcal{D}$, available behaviors $\mathcal{B}_1, \ldots, \mathcal{B}_n$, and a target process $\mathcal{T}$, build, if it exists, a realization of $\mathcal{T}$ in the dynamic system $\mathcal{S}$ corresponding to $\mathcal{D}$ and $\mathcal{B}_1, \ldots, \mathcal{B}_n$.* In previous work (De Giacomo et al., 2010b;a), a solution to a simplified variant of our problem has been proposed[4]. Here, as discussed above, we explicitly distinguish between dynamic domain and available behaviors, thus obtaining a different, more sophisti-

[4]In particular, in (De Giacomo et al., 2010b;a) behaviors are modelled directly in terms of restrictions on the domain.

cated problem. Nonetheless, the techniques presented there still apply, as we can reduce our problem to that case. This allows us to claim this result:

Theorem 11. (De Giacomo et al., 2010b) Building a realization of a target process $\mathcal{T}$ in a dynamic system $\mathcal{S}$ is an EXPTIME-complete problem.

3.3.2 Case Study

Here we present a case study, freely inspired by a real storyboard of a live demo held in a smart home located in Rome, whose houseplant is depicted in Figure 3.11. The home is equipped with many devices and a central reasoning system, whose domotic core is based on the framework here described, in order to orchestrate all the offered services. Imagine here lives Niels, a man affected by Amyotrophic Lateral Sclerosis (ALS). He is unable to walk, thus he needs a wheelchair to move around the house. The other human actors are Dan, a guest sleeping in the living room, and Wilma, the nurse. At the beginning of the story, Niels is sleeping in his automated bed. The behaviors the system can manage are the `bedService`, i.e., an automated bed, which can be either *down* or *up*; the `door`Num`Service`, i.e., the doors, for $Num \in \{1, 7\}$ (Figure 3.12a); the `alarmService`, i.e., an alarm, that can be either *set* or not; the `light`$Room$`Service`, i.e., light bulbs and lamps, for each *Room* in Figure 3.11; the `kitchenService`, i.e., a cooking service with preset dishes (Figure 3.12c); the `pantryService`, i.e., an automated pantry, able to check whether ingredients are in or not and buy them, if missing (see Figure 3.12b); the `bathroomService`, i.e., a bathroom management system, able to warm the temperature inside and fill or empty the tub (Figure 3.12e); and finally the `tvService`, i.e., a TV, either *on* or *off*.

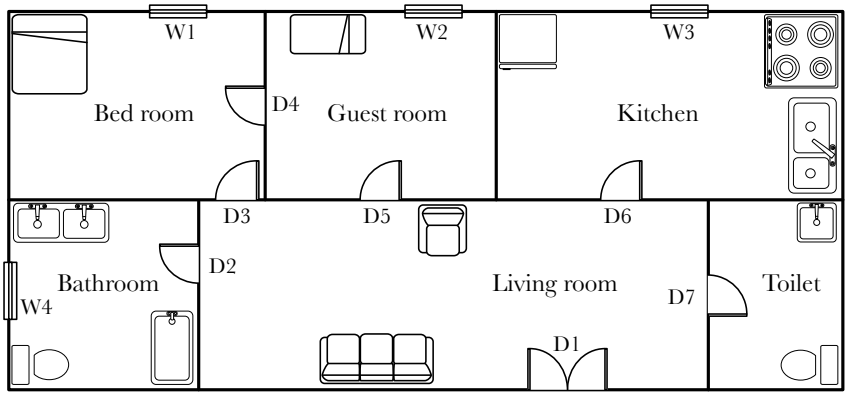
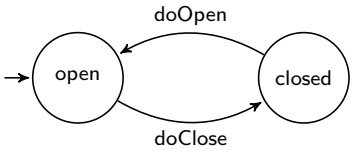
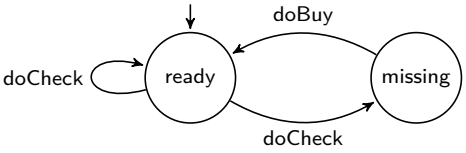


Figure 3.11. The smart home plant

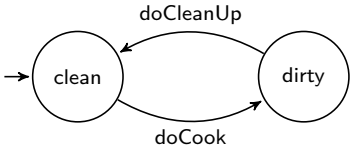
Finally, we consider a very particular behavior, that we call `nurseService`: it is Wilma, the nurse, who is in charge of moving Niels around the house. Despite the fact that an analogous service could be provided by some mechanical device, we refer to an human to illustrate how actors can be abstracted as behaviors as well, wrapped by a semantic description. All behaviors are depicted in Figure 3.12, except for `light`$Room$`Service`, `bedService`, `alarmService` and `tvService` that have very simple on/off behaviors. As described in Section 3.3.1, a dynamic domain state is a



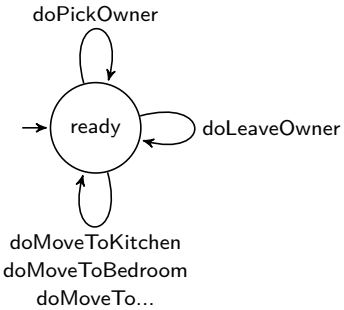
(a) doorNumService



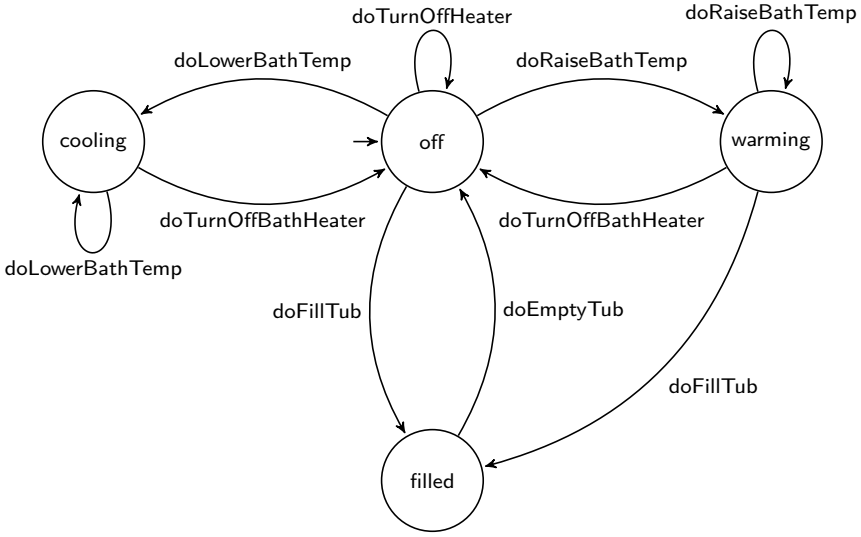
(b) pantryService



(c) kitchenService



(d) nurseService



(e) bathroomService

Figure 3.12. Case study services

subset of 2^P, where $P = \{p_1, \ldots, p_n\}$ is a finite set of boolean domain propositions. In order to express that, e.g., the bathroom temperature is mild, we could make use of a grounded propositional letter such as $bathroomTemperatureIsMild$. Nevertheless, we would have a grounded proposition for each value that the sensed temperature may assume ($bathroomTemperatureIsHot$, etc.) with the implicit constraint that only one of them can be evaluated to $\top$ at a time (and all the others to $\bot$). Thus, for sake of simplicity, here we make use of statements of the form "$var = val$" (e.g., "$varBathroomTemperature = warm$"). We call var the domain variable; val can be equal to any expected value which var can assume. Using such abbreviations we can phrase concepts like "a domain variable var is *set to* the val value" to easily refer to a transition in the dynamic domain moving from the current state to a following one where the proposition $var = val$ holds. For the sake of readability, actions are identified by the do- prefix (e.g., doRing).

Now we comment the case study. All services affect, through their actions, the related domain variables representing the state of the context. As an example, consider Figure 3.12e. Action doRaiseBathTemp causes the `bathroomService` to reach warming state, and affects the domain setting $varBathroomTemperature$ either to *(i)* $mild$ if it was equal to $cold$, or *(ii)* $warm$, if previously $mild$. However, we can imagine also *indirect* effects: e.g., the `door4Service` and `door5Service`'s doOpen actions trivially turn the $varDoor4$ and $varDoor5$ domain variables from *closed* to *open* and, at the same time, change the $varGuestDisturbed$ domain variable from $false$ to $true$, since, as depicted in Figure 3.11, they leed to the guest room, which we supposed Dan, the guest, to sleep in. The dynamic domain constrains the execution of service actions, allowing *executable* transitions only to take place (as explained in Section 3.3.1). For instance, consider the doPick and doLeave actions in `nurseService`: they represent Wilma taking and releasing Niels' wheelchair. Even if they are always available according to the service's description (Figure 3.12d), they are allowed by the domain iff $varPositionOwner$ and $varPositionNurse$ are equal (i.e., iff $\bigvee_{r \in Rooms}(varPositionOwner = r \wedge varPositionNurse = r)$ for $Rooms = \{livingRoom, bedRoom, bathRoom, guestRoom, toilet, kitchen\}$). Further on, it is stated that you can activate the doPick transition only if $varOwnerPicked = false$ (conversely, activate the doPick only if $varOwnerPicked = true$) and, when $varOwnerPicked = true$, doMoveTo$Room$ causes both $varPositionOwner$ and $varPositionNurse$ to be set to the same $Room$. As an example of interaction between behaviors, consider `kitchenService` and `pantryService`. As depicted in Figure 3.12, they do not have any action in common. Though, cooking any dish (namely, invoking doCook action on the `kitchenService` service) is *not* possible if some ingredients are missing (i.e., if $varIngredients = false$). The `pantryService` can buy them (indeed, doBuyIngredients sets $varIngredients = true$), but only after the execution of a check (doCheckIngredients). The evolution of doCheckIngredients is constrained by the $varIngredients$ domain variable: if $varIngredients = false$, then the next state of `pantryService` is missing (and the doBuyIngredients action *executable*), otherwise it remains in the ready state.

Such comments motivate the advantages of decoupling behaviors and dynamic domain as in the framework: the evolution of the system is not straightforward from the inspection of services or dynamic domains alone. Indeed, a service represents the behavior of a real device or application plugged in the environment, and it is

distributed by vendors who do not know the actual context in which it will be used. The same service could affect (or be affected by) the world in different ways, according to the environment with which it is interacting.

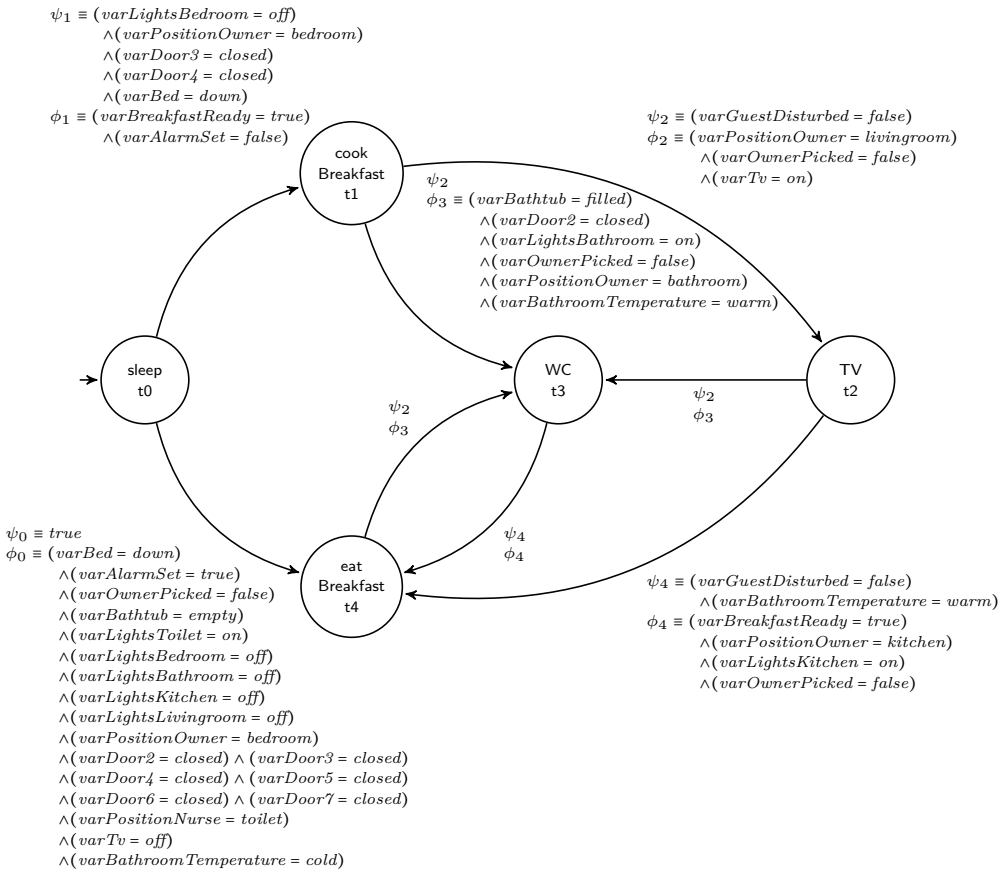


Figure 3.13. The sample target process

Next we turn to the target process itself, shown in Figure 3.13, representing what Niels wants to happen, when waking up in the morning. First, the home system must let Niels get awaken only after the breakfast is ready: this is the aim of the first transition, where the reachability goal is to have $(varBreakfastReady = true) \wedge (varAlarmSet = false)$, while all conditions that make Niels sleep comfortable must be kept: $(varBed = down) \wedge (varLightsBedroom = off) \wedge \ldots$ Then, once the alarm rang out, we let Niels decide whether he prefers to have a bath or to watch TV (and optionally have a bath afterwards). In both cases, we do not want to wake up Dan ($\psi_2 \equiv (varGuestDisturbed = false)$). Niels can successively have breakfast, but we suppose that further he can go back to the bathroom and eat a little more again how many times he wishes: this is the rationale beneath the formulation of: $\psi_4 \equiv (varGuestDisturbed = false) \wedge (varBathroomTemperature = warm)$. Finally, Niels can get back to the bed room. The transition from the eatBreakfast (t4) state to the sleep (t0) one has no maintenance goal (i.e., $\phi_0 = true$), whereas the reachability goal is just to reset the domain variables to their initial setup.

3.3.3 Solver

As we can see from the case study above, goal-based processes can be used to naturally specify the behavior of complex long-running intelligent systems. In order to apply this framework to real applications, however, we need a practical and efficient solver for such composition tasks. The solution in (De Giacomo et al., 2010b;a) reduces the composition problem to LTL synthesis by model checking. As a result, an efficient model checker and the approach is viable in practice only if large computational resources are available; on typical hardware for the smart home applications, only simple examples can be solved with that approach.

In light of the success of heuristic search in classical planning, it is interesting to ask whether this problem can be more efficiently solved by a direct search method. We pursue here this idea, and propose a novel solution to the composition problem based on an AND-OR search in the space of execution traces of incremental partial policies. Intuitively, the search keeps a partial policy at each step, and simulates its execution, taking into account all possibilities of the goal requests and the nondeterministic effects of the actions. If no action is specified for some situation yet, the policy is augmented by trying all possible actions for it. Starting from an empty policy, this process is repeated until either a valid policy is found that works in all contingencies, or all policy extensions are tried yet no solution is found. In this process, the nondeterministic goal requests and action effects are handled as "AND steps," whereas the free choice of actions during expansion represents an "OR step." Figure 3.14 shows the Prolog code of the body of our solver.

The composition starts from an empty policy [ ] with the initial goal state and initial world state S_0, and simulates (while incrementally building) its execution, until all possible goal requests in the target process can always be achieved by some policy C (line 2). In our implementation, we always assume that the intial goal state is 0. To handle all the possible evolutions of the system from goal state T and world state S, the `compose/4` predicate first finds all goal requests GL that originate from T, and augments the current policy C_0 to obtain C_1 that handles these requests (lines 4–6). This is done by `compGoals/5`, which represents the first AND step in the search cycle. It recursively processes each goal request in the list GL by using `planForGoal/8` (lines 8–11). Notice that the policy is updated in each recursive step with the intermediate variable C in line 11. The predicate `planForGoal/8` essentially performs conditional planning with full observability and nondeterministic effects (lines 13–20). Lines 14 and 15 prevent the found partial policy from containing deadloops or violating the maintenance goal. Line 16 detects visited states in achieved goals so that they can be realized in the same way, and thus no further search is needed. Line 17 checks whether the current achievement goal has been realized, and if so, it goes on to recursively compose for the next goal state. Finally, Lines 18–20 capture the last case where no action is associated to the current situation, in which case the current policy needs expansion to handle it. This is done by the OR step of the search cycle, which proposes a best candidate action with the predicate `bestAct/3`, and planning goes on for the resulting world states.

Since the actions are nondeterministic, it means that executing an action may lead to multiple possible states, and the policy we find must work for them all. In our

algorithm, `tryStates/8` handles all these states by recursing into `planForGoal/8` with updated policy for each state, which represents the second AND step in the search cycle (lines 22–25).
Recall that the exploration of a search branch may fail in Lines 14 and 15, due to a deadloop and violation of a maintenance goal, respectively. When either case occurs, the program backtracks to the most recent predicate with a different succeeding assignment, which is always `bestAct/3`. From there, the next best action is proposed and tried, and so on. If all the possible actions have been tried, yet none leads to a valid policy, the program backtracks to the next most recent `bestAct/3` instance, and the same process is performed similarly.
Notice that our algorithm is applicable to any goal-based process composition task, as the predicates `initial_state/1`, `holds/2`, `bestAct/3`, `next_states/3` and `goal/4` behave according to the actual target goal-based process and its underlying dynamic environment which are specified using a problem definition language detailed below. It is not hard to see that the our algorithm strategically enumerates all valid policies, generating *on-the-fly* action mappings for *reachable* situations only. The algorithm can be shown to be sound and complete.

Theorem 12 (Soundness and completeness)**.** Let $\mathcal{T}$ be a target goal-based process and $\mathcal{S}$ its underlying dynamic system. If `compose(C)` succeeds, then `C` is a realization of $\mathcal{T}$ in $\mathcal{S}$. Moreover, if $\mathcal{T}$ is realizable in $\mathcal{S}$, then `compose(C)` succeeds.

This algorithm can be used for solving small composition problems even with a simple enumeration-based implementation of `bestAct/3`. However, as the problem size grows, this naïve implementation quickly becomes intractable, due to the large branching factor and deep search tree. Therefore, some intelligent ordering is needed for the succeeding bindings of `bestAct/3`, in order to make our solver efficient for large composition tasks. In our implementation of the solver[5], we make use of the well-known *delete-relaxation heuristics* (Hoffmann and Nebel, 2001), although other heuristics in classical planning could be adapted as well.
The delete-relaxation heuristics for a state is computed by solving a relaxed goal-reachability problem where all negative conditions and delete effects are eliminated from the original planning problem. It can be shown that the relaxed problem can always be solved (or proven unsolvable) in polynomial time. If a relaxed plan is found, then the number of actions in the plan is used as a heuristic estimation for the cost of achieving the goal from the current state; otherwise it is guaranteed that no plan exists to achieve the goal from the current state, so it is safe to prune this search branch and backtrack to other alternatives. In our implementation, when choosing the best action, `bestAct/3` first sorts all legal actions according to the optimistic goal distance of their successor states using the delete-relaxation heuristics,[6] and unifies with each of the actions in ascending order when the predicate

[5]The code of both solver and case study, as well as the experimental results, are available at the URL: `http://dl.dropbox.com/u/8845400/Code.zip` ("reviewer" is the password to open the password-protected zip file).

[6]In our experiments, we also take into account the conjunction of the maintenance goals of the next goal state, so that states violating future maintenance goals are pruned earlier, leading to further gain in efficiency.

is (re-)evaluated. Notice that the heuristics only changes the ordering of branch exploration in the search tree, with possible sound pruning for deadends, and does not affect the correctness guarantee of our algorithm.
For our solver, a problem specification is a regular Prolog source file which contains the following components:

- the instruction to load the solver `:- include(planner).`
- a list of primitive fluents, each fluent `F` specified by `prim_fluent(F).`
- a list of primitive actions, each action `A` specified by `prim_action(A).`
- action preconditions, one for each action `A`, by `poss(A,P).` where `A` is the action, and `P` is its precondition formula.
- conditional effects of actions of the form `causes(A,F,V,C).` meaning that fluent `F` will take value `V` if action `A` is executed in a state where condition `C` holds.
- initial assignment of fluents of the form `init(F,V).` where `F` is the fluent and `V` is its initial value.
- the process by a list of goal transitions of the form `goal(T,M,G,T').` where `T` and `T'` are the source and target goal states of the transition, `M` is the maintanence goal, and `G` the achievement goal. By default, the initial goal state is always 0.

3.3.4 Experiments on the case study

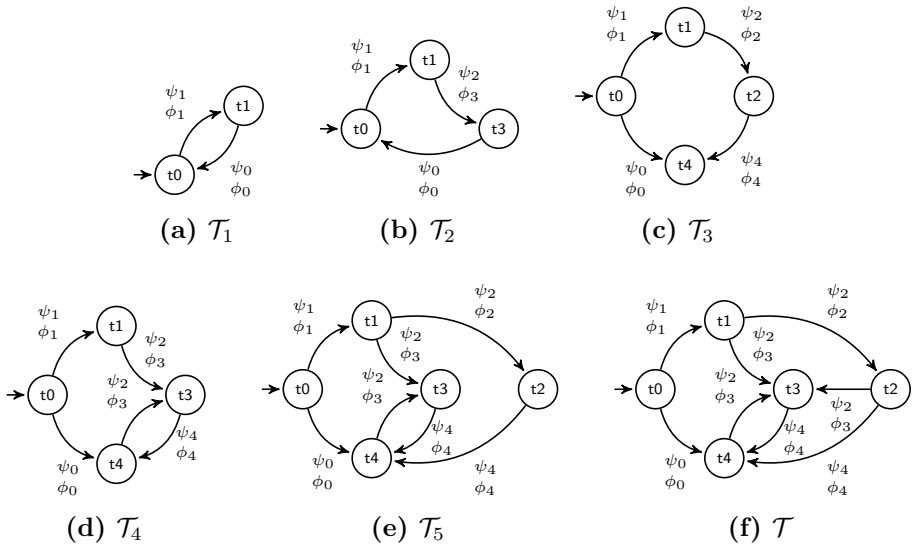
In order to test the efficiency of the solution presented in the previous section, we conducted some experiments based on the case study.
Given the dynamic system $\mathcal{S}$ and the target process $\mathcal{T}$ illustrated before, we considered both $\mathcal{T}$ and its restrictions $\mathcal{T}_{i\in\{1,\ldots,5\}}$, shown in Figure 3.15, where states and goals refer to the ones depicted in Figure 3.13.
Hence, for each $\mathcal{T}_i$, we run the solver 10 times in a SWI-Prolog 5.10.2 environment, on top of an Intel Core Duo 1.66 GHz (2 GB DDR2 RAM, Ubuntu 10.04) laptop. We gathered the results listed in Figure 3.16. The solution for the complete problem was found in about 239 seconds. Performances for simpler formulations followed an almost linear trend with respect to the input dimension, measured in terms of number of transitions in the target process (see Figure 3.17a). Figure 3.17b [7] shows that such results are quite reliable, since the Coefficient of Variation (i.e., the ratio between the Standard Deviation σ and the Mean Value $\mathcal{M}$) is fair little (ca. 2‰, excluding the first value, which is not significant, being the solution for that instance computed in too few milliseconds) and keeps constant as the $\mathcal{M}$ value grows. The performances are notable, especially if compared to the previous tests. Indeed, we ran a solver based on model checking techniques, built on top of TLV (version 4.18.4, see (Pnueli and Shahar, 1996)), on the laptop mentioned above. Notice that such a solver requires the usage of high computational resources, which are not affordable

[7] There, the base for the Logarithm of the Mean Time $\mathcal{M}$ is the least $\mathcal{M}$ value

in a smart home scenario. Indeed, on our laptop it took more than 24 hours to terminate, whereas our solver returned a solution within less than 4 minutes.

```
 0: % planner.pl - a generic solver for goal-based process composition.
 1: % Usage: call compose(C) to find a realization C.
 2: compose(C) :- initial_state(S0), compose(0, S0, [ ], C).
 3:
 4: % compose(T, S, C0, C1) compose for goal state T.
 5: compose(T, S, C0, C1) :-
 6:     findall(⟨M, G, T'⟩, goal(T, M, G, T'), GL), !,
        compGoals(T, GL, S, C0, C1).
 7:
 8: % compGoals(T, GL, S, C0, C1) compose for all goal requests in GL.
 9: compGoals(_, [ ], _, C, C).
10: compGoals(T, [⟨M, G, T'⟩|GL], S, C0, C1) :-
11:     planForGoal(T, M, G, T', S, [ ], C0, C),
        compGoals(T, GL, S, C, C1).
12:
13: % planForGoal(T, M, G, T', S, H, C0, C1)
    % update policy for a specific goal.
14: planForGoal(_, _, _, _, S, H, _, _) :- member(S, H), !, fail.
15: planForGoal(_, M, _, _, S, _, _, _) :- \+ holds(M, S), !, fail.
16: planForGoal(T, _, _, T', S, _, C, C) :- member(⟨T, T', S, _⟩, C), !.
17: planForGoal(_, _, G, T, S, _, C0, C1) :- holds(G, S), !,
    compose(T, S, C0, C1).
18: planForGoal(T, M, G, T', S, H, C0, C1) :-
19:     bestAct(G, A, S), next_states(S, A, SL),
20:     tryStates(T, M, G, T', SL, [S|H], [⟨T, T', S, A⟩|C0], C1).
21:
22: % tryStates(T, M, G, T', SL, H, C0, C1)
    % compose for all progressed world states.
23: tryStates(_, _, _, _, [ ], _, C, C).
24: tryStates(T, M, G, T', [S|SL], H, C0, C1) :-
25:     planForGoal(T, M, G, T', S, H, C0, C),
        tryStates(T, M, G, T', SL, H, C, C1).
```

Figure 3.14. Prolog implementation of our search-based solver.



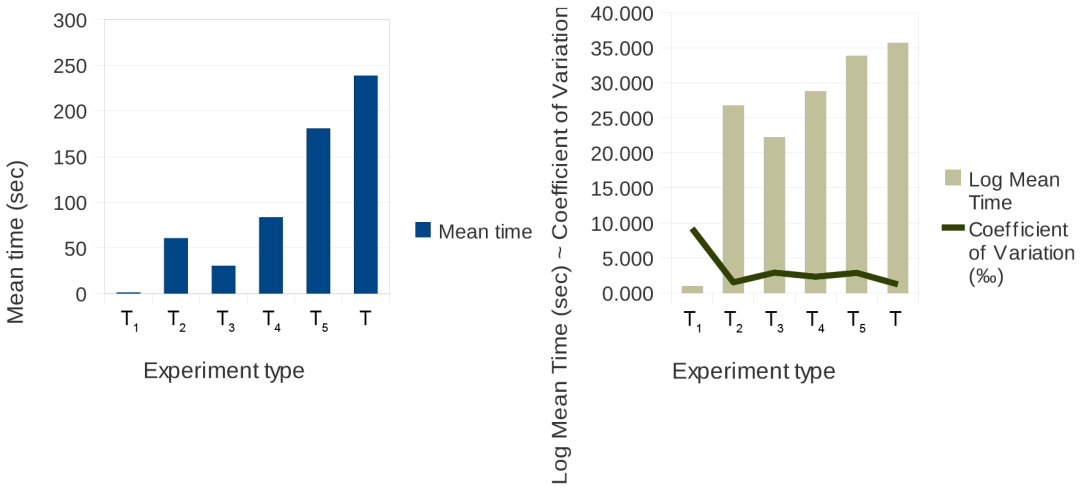
(a) $\mathcal{T}_1$ (b) $\mathcal{T}_2$ (c) $\mathcal{T}_3$

(d) $\mathcal{T}_4$ (e) $\mathcal{T}_5$ (f) $\mathcal{T}$

Figure 3.15. Test target processes

	Trans.'s	Time $[sec]$			Std. Dev.	Coeff. of Var. [‰]
		Mean ($\mathcal{M}$)	min (m)	Max (M)	(σ)	($\sigma/\mathcal{M}$)
$\mathcal{T}_1$	2	1.166	1.15	1.19	0.011	9.219
$\mathcal{T}_2$	3	60.688	60.54	60.82	0.094	1.545
$\mathcal{T}_3$	4	30.448	30.39	30.61	0.088	2.896
$\mathcal{T}_4$	5	83.497	83.26	83.88	0.195	2.331
$\mathcal{T}_5$	7	180.894	180.29	182.05	0.523	2.889
$\mathcal{T}$	8	238.841	238.38	239.19	0.291	1.219

Figure 3.16. Test results



(a) Tests results (b) Coefficient of variation

Chapter 4

Supervisory Control for Behavior Composition

In this chapter we formally relate the agent behavior composition problem, as studied within the AI community (e.g., (Berardi et al., 2005a; De Giacomo and Sardina, 2007; Stroeder and Pagnucco, 2009)) to *Supervisory Control Theory* (STC) in discrete-event systems (DES) (Wonham and Ramadge, 1987; Ramadge and Wonham, 1987; 1989b; Cassandras and Lafortune, 2006).
As formally introduced in Section 3.1.1, by *behavior composition problem* we refer to the task of automatically "realizing" (i.e., implementing) a desired though virtual *target* behavior module by suitably coordinating the execution of a set of concrete *available* behavior modules. Refer to Section 3.1.1 for details.
Supervisory control, on the other hand, is the task of automatically synthesizing "supervisors" that restrict the behavior of a discrete-event system (DES), which is assumed to *spontaneously* generate events, such that as much as possible of a given specifications is fulfilled. DES model a wide spectrum of physical systems, including manufacturing, traffic, logistics, and database systems. In supervisory control, an automaton $\mathcal{G}$—known as *"the plant"*—is used to model both controllable and uncontrollable behaviors of a given DES. The assumption is that the overall behavior of $\mathcal{G}$ is *not* satisfactory and must be controlled. To that end, a so-called *supervisor* sup is imposed on $\mathcal{G}$ so as to meet a given specification on event orderings and legality of states. Supervisors observe (some of) the events executed by $\mathcal{G}$ and can disable those that are "controllable." A controllable specification is one for which there exists a supervisor that can guarantee it.
The motivations behind linking behavior composition to supervisory control theory are threefold. Supervisory control theory have a rigorous *foundations rooted in formal languages*. It was first developed by (Ramadge and Wonham, 1989b) et alii in the 80's and then further studied by the control community w.r.t. both theory and applications (Cassandras and Lafortune, 2006). Hence, recasting the composition task as the supervision of DESs provides us with a solid foundation for studying composition. In addition, computational properties for supervisor synthesis have been substantially advanced, and some tools for supervisor synthesis are now available (e.g. TCT/STCT (Zhang and Wonham, 2001), GRAIL (Reiser et al., 2006), DESUMA (Ricker et al., 2006), and SUPREMICA (Åkesson et al., 2003)). As a

consequence, we can apply very different techniques for solving the composition problem than those already available within the AI literature (e.g., PDL satisfiability (De Giacomo and Sardina, 2007), direct search (Stroeder and Pagnucco, 2009), LTL/ATL synthesis (Lustig and Vardi, 2009; De Giacomo and Felli, 2010), and computation of special kind of simulation relations (Sardina et al., 2008; Berardi et al., 2008)). Finally, once linked, we expected cross-fertilization between AI-style composition and DES supervisory theory. In fact, we would like to import notions and techniques common in DES into the composition setting, such as decentralized supervision, hierarchical and tolerance supervision/composition (Cassandras and Lafortune, 2006).

4.1 Supervisory Control Theory

Discrete event systems range across a wide variety of physical systems that arise in technology (e.g., manufacturing and logistic systems, DBMSs, communication protocols and networks, etc.), whose processes are discrete (in time and state space), event-driven, and nondeterministic (Cassandras and Lafortune, 2006). Roughly speaking, Supervisory Control Theory (SCT) is concerned with the *controllability* of the sequences or strings of events that such process/systems (*plant*), can generate (Ramadge and Wonham, 1989b).

Discrete-event systems encompass a wide variety of physical systems that arise in technology. These include manufacturing systems, traffic and logistic systems, DBMSs, communication protocols and networks. Typically, the processes associated with these systems may be thought of as discrete (in time and state space), event-driven and nondeterministic. The supervisory control framework, also known as the Ramadge-Wonham framework (Ramadge and Wonham, 1987; Wonham and Ramadge, 1987; Wonham, 2012; Cassandras and Lafortune, 2006), comprises a set of methodologies for automatically synthesizing supervisors for restricting the behavior of a plant such that as much as possible of the given specifications are fulfilled. To this end, the set of events is partitioned into *controllable* and *uncontrollable*, and the plant is assumed to spontaneously generate them. While the supervisor can observe the string of all events generated by the plant, its control action can only prevent those that are controllable. Hence, the supervisor observes the events generated by the plant and the control problem is to suitably interact with the process, by disabling (part of the) controllable events, so as to confine its behavior to within specified legal bounds. In other words, a supervisor is conceptually slightly different from the usual notion of controller, in the sense that plant's events are both spontaneous and process-generated.

In its original formulation, the SCT regards the plant as language generator and, accordingly, the specification is to be modeled by formal languages of finite strings. Notably, these languages are not necessarily regular languages generated by finite automata as was done in most subsequent work. In this chapter, we will deal with unbounded, yet terminating processes: the specifications ares thus different from the usual "long-running" (LTL) specifications considered up to now. However, there exist in literature some extensions to infinite languages: more recent extensions of the framework (see, e.g., (Thistle and Wonham, 1994)) also deal with non-terminating

executions of the plant, allowing the investigation of both liveness and safety issues in the control of discrete-event systems.
Specifications are interpreted as follows. The language of the plant $\mathcal{G}$ contains strings that are not acceptable because they violate some safety or nonblocking condition that we wish to impose on the system. It could be that certain states of $\mathcal{G}$ are undesirable and should be avoided. These could be states where $\mathcal{G}$ blocks, via deadlock or livelock; or they could be states that are physically inadmissible, for example, a collision of a robot with an automated guided vehicle or an attempt to place a part in a full buffer in an automated manufacturing system. Moreover, it could be that some strings in the language $L(\mathcal{G})$ of the plant contain substrings that are not allowed ("bad prefixes"). These substrings may violate a desired ordering of certain events; for example, requests for the use of a common resource should be granted in a "first-come first-served" manner. Therefore, we will be considering as specification a sublanguage of $L(\mathcal{G})$ representing the "legal" or "admissible" behaviors. Various control-theoretic questions such as controllability (Ramadge and Wonham, 1987; Wonham and Ramadge, 1987), observability (Lin and Wonham, 1988b; Ramadge, 1986), decentralized and hierarchical control (Wong and Wonham, 1996; Lin and Wonham, 1988a), as well as such questions as computational complexity (Gohari and Wonham, 2000) and others were studied in this framework. For what concerns this thesis, we will focus on controllability concerns.

4.1.1 Generators and languages

We regard the discrete-event system to be controlled, i.e. the *plant* in the traditional control terminology, as the generator of a formal language. A *language* L over an alphabet of events Σ is any subset of Σ^*, i.e., $L \subseteq \Sigma^*$. A *word* of L is then an element $w \in \Sigma^*$, and when $w_1, w_2 \in \Sigma^*$, then $w_1 \cdot w_2$ denotes their concatenation. The *prefix-closure* of a language L, denoted by $\overline{L}$, is the language of all prefixes of words in L, i.e., $w \in \overline{L}$ iff $w \cdot w' \in L$, for some $w' \in \Sigma^*$. A language L is said to be *closed* if $L = \overline{L}$. In SCT, the plant is viewed as a so-called *generator* (of the language of string of events characterizing its processes).

Definition 4.1. A *generator* is a *deterministic* FSM $\mathcal{G} = \langle \Sigma, G, g_0, \gamma, G_m \rangle$, where:

- Σ is the finite alphabet of events ($\epsilon \in \Sigma$ being the empty symbol);
- G is a finite set of states;
- $g_0 \in G$ is the initial state;
- $\gamma : G \times \Sigma \times G$ is the transition relation; and
- $G_m \subseteq G$ is the set of "marked" states. These states are used to distinguish words that, somehow, represent correct or complete computations of the plant.

△

We generalize γ to words as follows: $\gamma(g, \epsilon) = g$ and $\gamma(g, w \cdot \sigma) = \gamma(\gamma(g, w), \sigma)$, with $w \in \Sigma^*$, $\sigma \in \Sigma$ and ϵ the empty string.

Definition 4.2. We define the two languages of a generator $\mathcal{G}$:

- the language *generated* by $\mathcal{G}$ is $L(\mathcal{G}) = \{w \in \Sigma^* \mid \gamma(g_0, w) \text{ is defined}\}$, whereas
- the *marked* language of $\mathcal{G}$ is $L_m(\mathcal{G}) = \{w \in L(\mathcal{G}) \mid \gamma(g_0, w) \in G_m\}$.

△

Words in the former language stand for, possibly partial, operations or tasks, while words in the marked language usually represent the completion of some operations or tasks. Note that $L(\mathcal{G})$ is always closed, but $L_m(\mathcal{G})$ may not be so.
A state g is *reachable* if $g = \gamma(g_0, w)$ for some word $w \in \Sigma^*$ and *coreachable* if there exists a word $w \in \Sigma^*$ such that $\gamma(g', w) \in G_m$. A generator $\mathcal{G}$ is *nonblocking* if every reachable state is also coreachable, i.e., $L(\mathcal{G}) = \overline{L_m(\mathcal{G})}$. It is instead said to be *trim* if it is both reachable and coreachable. Moreover, $\mathcal{G}$ *represents* a language L if $\mathcal{G}$ is nonblocking and $L_m(\mathcal{G}) = L$. Given two languages $K \subseteq L$, K is said to be *L-marked* if $\overline{K} \cap L \subseteq K$, and *$L$-closed* if $K = \overline{K} \cap L$, i.e., if K contains all its prefixes that belong to L.
We partition the alphabet $\Sigma = \Sigma_c \cup \Sigma_u$ into *controllable* (Σ_c) and *uncontrollable* (Σ_u) events, with $\Sigma_c \cap \Sigma_u = \varnothing$. Events may occur only when *enabled* by a supervisor: whereas controllable events may be enabled or disabled, uncontrollable events are always enabled.

4.1.2 Specifications and supervisors

A *specification* for a generator plant $\mathcal{G}$ is a language $K \subseteq L(\mathcal{G})$. We say that K is *controllable* in plant $\mathcal{G}$ iff

$$\overline{K}\Sigma_u \cap L(\mathcal{G}) \subseteq \overline{K}$$

that is, every prefix of K followed by an uncontrollable event and compatible with the plant can be extended to a word in the specification itself. Informally, a specification is controllable if it is not possible to be "pushed" outside of it.

Definition 4.3. A *supervisor* for specification K and plant $\mathcal{G}$ is a function

$$V : L(\mathcal{G}) \to \{\Sigma_a \in Pwr(\Sigma) \mid \Sigma_a \supseteq \Sigma_u\}$$

that returns, for each word in $L(\mathcal{G})$, the set of events Σ_a that are enabled (i.e., allowed) next — again, uncontrollable events are always enabled. △

A plant $\mathcal{G}$ under supervisor V yields the *controlled system* $V/\mathcal{G}$ (sometimes referred to as "closed-loop" behavior) whose generated and marked languages are defined as:

$$L(V/\mathcal{G}) = \{w \cdot \sigma \in L(\mathcal{G}) \mid \ w \in L(V/\mathcal{G}),\ \sigma \in V(w)\} \cup \{\epsilon\};$$
$$L_m(V/\mathcal{G}) = L(V/\mathcal{G}) \cap L_m(\mathcal{G}).$$

Informally, $L(V/\mathcal{G})$ represents all operations, processes, or tasks, that plant $\mathcal{G}$ may yield while supervised by V, whereas $L_m(V/\mathcal{G})$ stand for the subset of those operations or tasks that are, in some manner, *complete*.
A key result in SCT states that being able to control a specification in a plant amounts to finding a supervisor for such specification (Wonham, 2012).

Theorem 13. Let $\mathcal{G}$ be a generator and $K \subseteq L(\mathcal{G})$ be a closed and non-empty specification. Then, there exists a supervisor V such that $L(V/\mathcal{G}) = K$ *iff* K is controllable.

The reason for not allowing $K = \varnothing$ is because we always have $\epsilon \in L(V/\mathcal{G})$, by definition. Indeed, there is a difference between $L(V/\mathcal{G}) = \{\epsilon\}$ and $L(V/\mathcal{G}) = \varnothing$: while in the first case there is no possible controlled behavior, in the latter the controlled system stays in its initial state and all events are disabled by the supervisor.

4.1.3 Nonblocking Supervisors

In addition, in many settings, one would like to control the language representing *complete* operations or tasks, that is, the marked fragment of the plant. Thus, we shall focus on *nonblocking* supervisors V that can always force the plant $\mathcal{G}$ to generate words that can eventually be extended into the marked (supervised) language. Technically, supervisor V is *nonblocking* in plant $\mathcal{G}$ iff

$$L(V/\mathcal{G}) = \overline{L_m(V/\mathcal{G})}$$

that is, the set of words generated by the supervised plant are exactly all the prefixes of marked words, i.e., they can be extended to a marked word.
However, observe that the previous result of Theorem 13 does not guarantee that the supervisor V is nonblocking. To ensure this property, a further requirement is needed.

Theorem 14. Given a generator $\mathcal{G}$ and a specification $K \subseteq L_m(\mathcal{G})$, $K \neq \varnothing$, there exists a nonblocking supervisor V for $\mathcal{G}$ such that $L(V/\mathcal{G}) = \overline{K}$ and $L_m(V/\mathcal{G}) = K$ iff *(i)* K is controllable wrt $\mathcal{G}$ and *(ii)* it is $L_m(\mathcal{G})$-closed.

If K is closed then condition (ii) is not needed.

4.1.4 On the supremal controllable sublanguage

When the specification K is not (guaranteed to be) controllable, one then looks for controlling the "largest" (in terms of set inclusion) sublanguage possible.
Given a generator $\mathcal{G}$ and a specification K, the set of all sublanguages of K that are controllable wrt $\mathcal{G}$ is defined as

$$\mathcal{C}(K) := \{L \subseteq K \mid L \text{ is controllable wrt } \mathcal{G}\}$$

Interestingly, this set is closed under arbitrary union and contains a (unique) supremal element (Wonham and Ramadge, 1987): the *supremal controllable sublanguage* of K, denoted $sup\mathcal{C}(K)$.
Putting both parts together, one is usually interested in (controlling) the K's sublanguage

$$K^{\uparrow} = sup\mathcal{C}(K \cap L_m(\mathcal{G}))$$

i.e. the supremal marked specification. It turns out that, under a plausible assumption, a supervisor does exist for non-empty $K^{\uparrow}$.

Theorem 15. If $\overline{K} \cap L_m(\mathcal{G}) \subseteq K$ (i.e., K is $L_m(\mathcal{G})$-marked) and $K^\uparrow \neq \varnothing$, then there exists a nonblocking supervisor V for $\mathcal{G}$ such that $L_m(V/\mathcal{G}) = K^\uparrow$.

The assumption states that K is closed under marked-prefixes: every prefix from K representing a complete task is part of K. Theorem 15 will play a key role in our results.
If we disregard the plant's marked states, i.e., the marked language of the plant $\mathcal{G}$ under the supervision of a supervisor V is defined as

$$L_m(V/\mathcal{G}) = L(V/\mathcal{G}) \cap K \tag{4.1}$$

then K is not required to be marked. This assumption matches the *Marking* Nonblocking Supervisory Control problem (MNSC) for the pair $(K, \mathcal{G})$ in the Wonham/Ramadge framework (Wonham, 2012) – see Section 4.1.4.

4.2 A fixpoint computation of $sup\mathcal{C}(K)$

We can characterize the supremal controllable sublanguage $K^\uparrow$ of a given language K as the largest fixpoint of a certain operator Ω (Wonham and Ramadge, 1987). Let $Pwr(\Sigma^*)$ be the set of all languages over Σ. Define the operator

$$\Omega : Pwr(\Sigma^*) \to Pwr(\Sigma^*)$$

according to

$$\Omega(L) = K \cap L_m(\mathcal{G}) \cap sup\{E \subseteq \Sigma^* \mid E = \overline{E} \text{ and } E\Sigma_u \cap L(\mathcal{G}) \subseteq \overline{L}\}$$

Observe how this definition is dependent on the specific plant $\mathcal{G}$ and specification K. Given a language L, $\Omega(L)$ computes its largest controllable sublanguage compatible with both $\mathcal{G}$ and K. A language L such that $L = \Omega(L)$ is a *fixpoint* of Ω. The following proposition describes $sup\mathcal{C}(K)$ as the largest fixpoint of Ω, i.e., the largest language L such that $\Omega(L) = L$.

Proposition 1. Let $K^\uparrow := sup\mathcal{C}(K)$. Then $K^\uparrow = \Omega(K^\uparrow)$ and $L \subseteq K^\uparrow$ for every language L such that $\overline{L} = \Omega(L)$. △

Furthermore, if we set

$$K_0 = K \quad \text{and} \quad K_{i+1} = \Omega(K_i) \tag{4.2}$$

then

$$\lim_{i \to \infty} K_i = K^\uparrow$$

If we restrict to the *regular* case, i.e., when $\mathcal{G}$ and K are regular and represented with generators, then the following fundamental result holds:

Theorem 16. (Wonham, 2012) The sequence of languages K_i as in (4.2) converges after a finite number of terms to $K^\uparrow = sup\mathcal{C}(K)$.

In other words, $K^\uparrow = \bigcap_{i=0}^{\infty} K_i$ is a fixpoint of the operator Ω. In the next section we will show how this iteration scheme yields an effective procedure for the computation of $K^\uparrow$.

4.2.1 On the complexity of the regular case

Assuming that the generator for K and the plant $\mathcal{G}$ have, respectively, m and n states, then the scheme illustrated in the previous section converges after at most mn iterations. Since the computation of Ω is itself bounded by a polynomial in m and n (Wonham and Ramadge, 1987), this implies that the computation of $K^{\uparrow}$ is of polynomial complexity in m and n, in contrast to the prefix-closed special case where it is linear in nm (Ramadge and Wonham, 1987). Moreover, if K and $L(\mathcal{G})$ are regular, then $K^{\uparrow}$ is regular as well and it can be represented (i.e., marked) by a trim automaton with at most nm states (Cassandras and Lafortune, 2006).

In (Wonham and Ramadge, 1987) it is shown how the operator Ω can be effectively computed. In Section 4.2.2 we present an intuitive algorithm as in (Cassandras and Lafortune, 2006), of which the worst-case complexity is $O(n^2m^2 \times |\Sigma|)$, i.e., it is quadratic in the product nm. In Section 4.2.3 we introduce instead a novel approach, based on the construction of a game arena as customary in Verification and Synthesis, which is polynomial as well.

Notably, authors of (Gohari and Wonham, 2000) point out that these complexity results are limited to the case in which we have a single plant component and a single specification generator. In the "generalized" case, namely in which there are N plant modules running concurrently and the specification is composed by M modules each rejecting certain event sequences as illegal, there is no way to avoid constructing a state space exponential in both N and M, and the problem is proved to be NP-complete in the size of the input. Although this setting does not provide more expressive power, since a problem with arbitrary N, M can be converted into the standard supervisory control problem (single plant, single specification), it shows that it is possible to obtain a more concise representation.

4.2.2 Computation of $\sup\mathcal{C}(K)$ by iterative refinement

Instead of computing the operator Ω, this approach (Cassandras and Lafortune, 2006) employs the technique of "refinement by product". This technique consists in building a product automaton $\mathcal{G} \times K$ which is then iteratively refined (ergo the quadratic cost in nm).

In order to check which strings in K, if any, violate the controllability condition, we need to check whether uncontrollable events are always feasible in this product automaton. This is due to the fact that uncontrollable events that are not allowed by the specification may still happen inside the plant. A correct supervisor must prevent this situation from happening. To this end, given a generic state g of a generic generator $\mathcal{G} = \langle \Sigma, G, g_0, \gamma, G_m \rangle$, we will denote with $\mathsf{Act}_{\mathcal{G}}(g)$ the *active* event set of a state g, i.e., $\mathsf{Act}_{\mathcal{G}}(g) = \{\sigma \in \Sigma \mid \gamma(g, \sigma) \text{ is defined } \}$. Iterating this reasoning, we will be able to compute the maximal controllable sublanguage though the procedure formally defined below.

Here, as in the following section, plant's marked states will be disregarded. As a consequence, the resulting marked language will be only determined by the specification (marked) language. Observe how this matches the *Marking* Nonblocking Supervisory Control problem (MNSC) as expressed in 4.1.

step 0 Let $\mathcal{G} = \langle \Sigma, G, g_0, \gamma \rangle$ be the plant (marked states are disregarded here) and let $\mathcal{K} = \langle \Sigma, X, x_0, \gamma_x, X_m \rangle$ be the automaton representing the specification language, i.e., such that $L_m(\mathcal{K}) = K$ and $L(\mathcal{K}) = \overline{K}$. Wlog, we assume $K \subseteq L(\mathcal{G})$.

step1: build the product. Let

$$\mathcal{G}_0 = \langle \Sigma, G_0, \langle g_0, x_0 \rangle, \gamma_0, G_{m,0} \rangle = \mathcal{G} \times \mathcal{K}$$

where $G_0 \subseteq G \times X$. Moreover $G_{m,0} \subseteq G \times X_m$, i.e., the marking is solely determined by the states of $\mathcal{K}$. Notice that, since $K \subseteq L(\mathcal{G})$, we have $L_m(\mathcal{G}_0) = K$ and $L(\mathcal{G}_0) = \overline{K}$. Set $i = 0$.

step2: iterative refinement of $\mathcal{G}_i$. The refinement works as follows: first we compute the set of states of $\mathcal{G}_i$ that guarantee that no uncontrollable event will violate the specification, then we build $\mathcal{G}_{i+1}$ by projecting $\mathcal{G}_i$ on this subset.

$$G_{i+1} = \{ \langle g, x \rangle \in G_i \mid \mathsf{Act}_{\mathcal{G}}(g) \cap \Sigma_u \subseteq \mathsf{Act}_{\mathcal{G}_i}(\langle g, x \rangle) \} \tag{4.3}$$

$$\gamma_{i+1} = \gamma_i \mid_{G_{i+1}} \tag{4.4}$$

$$G_{m,i+1} = G_{m,i} \cap G_{i+1} \tag{4.5}$$

(4.3) computes the set of states for which all uncontrollable events (that are feasible in the plant $\mathcal{G}$) are also feasible in the product automaton $\mathcal{G}_i$ computed in the previous iteration; (4.4) restricts the transition function to the new state space; finally, (4.5) shrinks the set of marked states accordingly.
Hence, consider the trim automaton

$$\mathcal{G}_{i+1} := \mathsf{Trim}(\langle \Sigma, G_{i+1}, \gamma_{i+1}, \langle g_0, x_0 \rangle, G_{m,i+1} \rangle) \tag{4.6}$$

where Trim is a procedure removing states that are either not reachable or not coreachable.
If G_{i+1} is empty (thus also the initial state $\langle g_0, x_0 \rangle$ was deleted by (4.3)), then $K^{\uparrow} = \varnothing$. Otherwise we repeat step2 for $i + 1$.

fixpoint condition (termination). If $\mathcal{G}_{i+1} = \mathcal{G}_i$ then we reached the refinement fixpoint and thus

$$L_m(\mathcal{G}_{i+1}) = K^{\uparrow} \quad \text{and} \quad L(\mathcal{G}_{i+1}) = \overline{K^{\uparrow}}$$

Since $\mathcal{G}_i$ is trim, it is trivial to notice that $\mathcal{G}_i$ *represents* $K^{\uparrow}$.

4.2.3 SCT as DFA game

Section 4.2.2 presented a *procedural* approach to SCT, i.e., for computing the supremal controllable sublanguage $sup\mathcal{C}(K)$ of a given specification K wrt a given plant $\mathcal{G}$. However, although it captures the notion of $sup\mathcal{C}(K)$ as the fixpoint of the operator Ω –as defined in Section 4.2– it does not provide any insight or link between Discrete Control Theory and the work on Verification and Synthesis in Computer Science.

DFA games

In Chapter 1 we defined an open system as a system that interacts with its environment via input and output signals and whose behavior depends on this interaction. In particular, when dealing with open systems, we distinguish between output signals O (generated by the system) over which we have control, and input signals I (generated by the environment) over which we have no control. We thus can see an open system as playing a game with an adversarial environment, with the specifications being the game winning condition.
Formally, we can formalize this intuition as follows. Consider a game Game in which:

- the environment chooses input values in I;
- the system chooses output values in O;
- in each round both players set their values;
- a *play* is a sequence of such couples, namely, a word in $(I \times O)^*$;
- the specification is a language recognizer $\mathcal{S}$ (a deterministic finite automaton on words, or DFA);
- the system wins iff the play is accepted by $\mathcal{S}$.

A strategy for the system is thus a function $f : I^* \to O$ mapping finite sequences of input signals into output signals. As the system interacts with an environment generating infinite input sequences, even though f is deterministic, it induces a computation tree with fixed branching degree $|I|$, i.e. it associates a play to each input sequence. The branches of such computation tree correspond to external nondeterminism, caused by different possible inputs.
The game can be solved computing the subset of winning states, i.e., those states from which there exists a strategy that can always induce a play accepted by $\mathcal{S}$.

SCT as DFA game

Consider again the basic problem of SCT. Given a plant $\mathcal{G}$ and a specification K, compute a nonblocking supervisor V for $\mathcal{G}$ such that "as much as possible" of K is fulfilled, i.e., the closed loop behavior of $\mathcal{G}$ under V is such that $L_m(V/\mathcal{G}) = K^{\uparrow}$, with $K^{\uparrow} = sup\mathcal{C}(K)$. In particular, we focus once more on the special case of nonblocking supervisors, namely, requiring that $L(V/\mathcal{G}) = \overline{K^{\uparrow}}$. As in Section 4.2.2 , we disregard the plant's marking states, hence focusing on the MNSC problem for $(K, \mathcal{G})$ – see Section 4.1.4.
For notational convenience, given a transition function γ of a generator $\mathcal{G}$, in this section we will denote by $\gamma(g, \alpha)!$ the fact that $\exists g' \in G$ such that $g' = \delta(g, \alpha)$, i.e., $\gamma(g, \alpha)$ is defined. Otherwise we write $\neg\delta(g, \alpha)$.

As before, a plant is a trim –i.e. reachable and coreachable– generator $\mathcal{G} = \langle \Sigma, G, g_0, \gamma \rangle$ (marked states are disregarded) where:

- $\Sigma = \Sigma_c \cup \Sigma_u$, $\Sigma_c \cap \Sigma_u = \varnothing$ is the set of events;
- $\gamma : S \times \Sigma \to S$ is the deterministic transition function.

Now let K be a specification language provided by means of a trim recognizer $\mathcal{S} = \langle \Sigma, S, s_0, \varsigma, S_m \rangle$, i.e., such that $L_m(\mathcal{S}) = K$ and $L(\mathcal{S}) = \overline{K}$.
First, we build the product automaton $\mathcal{K} = \mathcal{S} \times \mathcal{G}$, which is meant to compile away from $\mathcal{S}$ everything that is not compatible with $\mathcal{G}$.
$\mathcal{K} = \langle \Sigma, Q, q_0, \rho, Q_m \rangle$ is such that:

- $Q = S \times G$
- $Q_m = S_m \times G$

Finally, we build the controller $\mathsf{Game} = \langle Pwr(\Sigma_c) \times \Sigma, Q \cup \{\top, \bot\}, q_0, \delta, Q_m \rangle$ where:

- $Pwr(\Sigma_c) \times \Sigma$ is the new set of events. An element $\langle P, \sigma \rangle$ is meant to capture the execution of event σ in $\mathcal{G}$ whereas P is the set of events allowed by a supervisor.
- The state space Q is the same of $\mathcal{K}$, with the addition of two special states $\bot$ and $\top$. The former is intended to capture system runs whose last event is controllable and not allowed by the supervisor – see $(\delta, 4)$. The latter instead capture a failure condition – see $(\delta, 1)$.
- Depending whether $\sigma \in P$ or not, the event can be carried out or prevented from being executed. Remember that uncontrollable events are always allowed by definition. Hence the partial transition function $\delta : (Pwr(\Sigma_c) \times \Sigma) \times Q \to Q$ is defined as follows:

  1. $\delta((P, \sigma), \langle s, g \rangle) = \bot$ if $\sigma \notin \Sigma_c$ and $\gamma(g, \sigma)!$ but $\neg\rho(\langle s, g \rangle, \sigma)$
  2. $\delta((P, \sigma), q) = \rho(q, \sigma)$ if $q = \langle s, g \rangle$, $\sigma \notin \Sigma_c$ and $\rho(g, \sigma)!$
  3. $\delta((P, \sigma), q) = \rho(q, \sigma)$ if $\sigma \in \Sigma_c$ and $\sigma \in P$
  4. $\delta((P, \sigma), q) = \top$ if $\rho(q, \sigma)!$ and $\sigma \in \Sigma_c \wedge \sigma \notin P$

In words, case $(\delta, 1)$ copes with uncontrollable events that are defined in plant $\mathcal{G}$ but are not allowed by specification $\mathcal{K}$. Observe how this condition is strictly connected with the definition of controllable language in SCT: any of such events violates the specification K. Case $(\delta, 2)$ represents an uncontrollable event happening in plant $\mathcal{G}$, hence set P is ignored. Finally, the last two conditions represent a controllable events being prevented from happening $(\delta, 3)$ or being allowed $(\delta, 4)$. Observe that by $(\delta, 4)$ the sink state $\top$ is reached: forbidding a controllable action is sufficient to ensure the controllability of plant $\mathcal{G}$ wrt the current run.

Solving the game

We compute the set of winning states as a least fixpoint:

$$\mathsf{Win}_0(Q) = q \in Q_m \cup \{\top\} \text{ s.t. } \delta((P, \sigma), q)! \to \sigma \in \Sigma_c \tag{4.7}$$

$$\begin{aligned} \mathsf{Win}_{i+1}(Q) = \mathsf{Win}_i(Q) \cup \\ \{q \mid \exists P \; \forall \sigma \in (P \cup \Sigma_u) \; . \; \delta((P, \sigma), q)! \in \mathsf{Win}_i(Q)\} \end{aligned} \tag{4.8}$$

Therefore:

$$\mathsf{Win}(Q) = \mathsf{Win}_j(Q) \text{ s.t. } \mathsf{Win}_j(Q) = \mathsf{Win}_{j+1}(Q)$$

is the set of Game's states that are either marked in $\mathcal{K}$ or such that for every (uncontrollable as well as controllable) event, there exists a control action (i.e. a set $P \in Pwr(\Sigma_c)$) able to ensure that any successor is in $\mathsf{Win}(Q)$. Note that (4.7) also deals with the condition captured by $(\delta, 1)$, as discussed before.

Theorem 17. A winning strategy $f : \Sigma^* \to Pwr(\Sigma)$ exists iff $q_0 \in \mathsf{Win}(Q)$.

This known realizability result follows from the fact that every regular game is (finite memory-)*determined*, i.e., the winning regions for the two players partition the set of vertices of the game (Grädel et al., 2002).
What about Synthesis? We compute the *maximal controller* $\mathcal{C}^\uparrow$ from the set $\mathsf{Win}(Q)$ as the finite state *transducer* (DFA with output) computed from Game as:

$$\mathcal{C}^\uparrow = \langle Pwr(\Sigma_c) \times \Sigma, \mathsf{Win}(Q), q_0, \delta' \rangle = \mathsf{Trim}(\mathsf{Game}\,|_{\mathsf{Win}(Q)})$$

The *output function* $\omega : Q \times \Sigma \to 2^{Pwr(\Sigma_c)}$ is thus obtained by simply reading the transition function, namely $P \in \omega(q, \sigma)$ iff $\delta'((P, \sigma), q)!$. We can extract the set F of winning strategies $f : \Sigma^* \to Pwr(\Sigma)$. First we define the outcome $out_f(\sigma_0 \cdots \sigma_n)$ of given strategy f from state q_0 as the (unique – since $\mathcal{C}^\uparrow$ is deterministic) computation $q_0, q_1, \ldots, q_m$ such that for all positions $i \geq 0$ it is $q_{i+1} = \delta'((P, \sigma_i), q_i)$ with $P = f(\sigma_0 \cdots \sigma_i)$. If instead $\delta'((P, \sigma_i), q_i)$ is not defined for any P, then $m = i$ and $f(\sigma_0 \cdots \sigma_i)$ is undefined. This happens whenever a controllable event is forbidden by the strategy, hence the computation stops. Let $out_f(w)_i$ denote its i-th position. We extract a winning strategy $f \in F$ as:

$$f(\sigma) \in \omega(q_0, \sigma)$$

$$f(\sigma_0 \cdots \sigma_n) \in \omega(q, \sigma_n) \text{ where } q = out_f(\sigma_0 \cdots \sigma_{n-1})_n$$

Adopting the SCT terminology, and recalling that controllable languages are closed under union, we derive the corresponding supervisor $V : L(\mathcal{G}) \to Pwr(\Sigma)$ as follows (below, $w \in \Sigma^*$):

$$V(w) = \bigcup \{ f(w \cdot \sigma) \mid \sigma \in \Sigma \wedge f \in F \} \cup \Sigma_u$$

Theorem 18. V is a nonblocking supervisor such that $L_m(V/\mathcal{G}) = K^\uparrow$.

Proof. Assume $L_m(V/\mathcal{G}) = L(V/\mathcal{G}) \cap K \subset K^\uparrow$. It implies that there exists a word $w \in L(\mathcal{G}) \cap K$ which is controllable but is not in $L(V/\mathcal{G})$. Let w be $w = \sigma_0 \cdots \sigma_n$. Applying the definition of $L(V/\mathcal{G})$ we deduce that $\sigma_\ell \notin V(\sigma_0 \cdots \sigma_{\ell-1})$ for $\ell \leq n$, namely, $\sigma_\ell \notin f(\sigma_0 \cdots \sigma_\ell)$ for any strategy $f \in F$, and $\sigma_\ell \notin \Sigma_u$ either (because uncontrollable actions are always allowed by V). This implies that $\sigma_\ell \notin \omega(q, \sigma_\ell)$ with $q = out_f(\sigma_0 \cdots \sigma_{\ell-1})_\ell$ for any winning f. By construction of $\mathcal{C}^\uparrow$, this means that either $q \notin Q_m \cup \{\top\}$ (4.7) or not all (P, σ_ℓ)-successors of q with $\sigma_\ell \in (P \cup \Sigma_u)$ are in $\mathsf{Win}(Q)$ (4.8). As a consequence, w is not controllable.Consider now the case that $K^\uparrow \subset L_m(V/\mathcal{G}) = L(V/\mathcal{G}) \cap K$. Again, let $w = \sigma_0 \cdots \sigma_n$ a word such that

$w \in L(V/\mathcal{G}) \setminus K^{\uparrow}$. w is not controllable by definition, hence there exists at least one index $\ell < n$ such that $\sigma_0 \cdots \sigma_\ell \cdot \Sigma_u \cap L(\mathcal{G}) \notin \overline{K^{\uparrow}}$; namely, $\sigma_\ell \in \Sigma_c$ whereas $\sigma_{\ell+1} \in \Sigma_u$. Because $\sigma_\ell \in V(\sigma_0 \cdots \sigma_{\ell-1})$ is controllable, then there exists a strategy f such that $\sigma_\ell \in f(\sigma_0 \cdots \sigma_\ell)$. Assume $q = out_f(\sigma_0 \cdots \sigma_{\ell-1})_\ell$. Then there exists an integer i such that $q \in \mathsf{Win}_i(Q)$, otherwise such f can not exist by construction of $\mathcal{C}^{\uparrow}$. Hence either $q \in Q_m \cup \{\top\}$ (4.7) or $\exists P \forall \sigma \in P$ we have $\delta((P,\sigma),q) \in \mathsf{Win}_i(Q)$ (4.8). By substituting σ with σ_ℓ, we get that both $\sigma_0 \cdots \sigma_\ell$ and $\sigma_0 \cdots \sigma_{\ell+1}$ are in $\overline{K^{\uparrow}}$, and thus a contradiction. ■

As a consequence of Theorem 17, we get also the following result.

Theorem 19. *$K^{\uparrow} \neq \varnothing$ iff $q_0 \in \mathsf{Win}(Q)$.*

As for complexity, it is trivial to see that this approach is polynomial.

4.3 SCT for Agent Behavior Composition

As formalized in Section 3.1.1, the agent behavior composition problem amounts to synthesising a *controller* that is able to "realize" (i.e., implement) a desired, but nonexistent, *target agent behavior*, by suitably coordinating a set of *available agent behaviors*. As done in Chapter 3, we adopt the framework proposed in (Stroeder and Pagnucco, 2009; Sardiña et al., 2007; Sardiña et al., 2008; De Giacomo et al., 2013). In this approach, agent behaviors represent the operational logic of a device or a program and they are modeled using finite transition systems. The reader is referred to Section 3.1.1 for any detail[1].

Example 15. *Consider the example depicted in Figure 4.1. Target $\mathcal{T}$ encapsulates the desired functionality for a house cleaning system, which involves first turning on the lights to sense which area needs cleaning, and then either vacuuming or mopping the area. At every step, the user can request an action compatible with this specification, and a (good) controller should guarantee that it can fulfill such request by delegating the action to one of the three available devices installed in the house, namely, a vacuum device $\mathcal{B}_1$, a mopping device $\mathcal{B}_2$, a light cum sensor $\mathcal{B}_3$ and a lamp $\mathcal{B}_4$. Note that action* `clean` *in device $\mathcal{B}_1$ is non-deterministic: the controller may not know whether the device will stay in a_1 or evolve to a_2 (signaling that its internal dustbin becomes full and must be empty), though it can observe the evolution once it happens.* □

In Section 3.1.1 (please refer to it for definitions and details) we introduced the notion of *controller*, i.e., a function taking a history (run) of the system and the next action request and outputting the index of the available agent behavior to which the action is delegated. We then defined what does it mean when a controller *realizes the target agent behavior* – see also (De Giacomo and Sardina, 2007; De Giacomo et al., 2013). We called these controllers *exact compositions*, solutions to

[1] Observe, however, that we disregard here agent behaviors' final states, hence all states are considered as final — see discussion after Definition 3.1

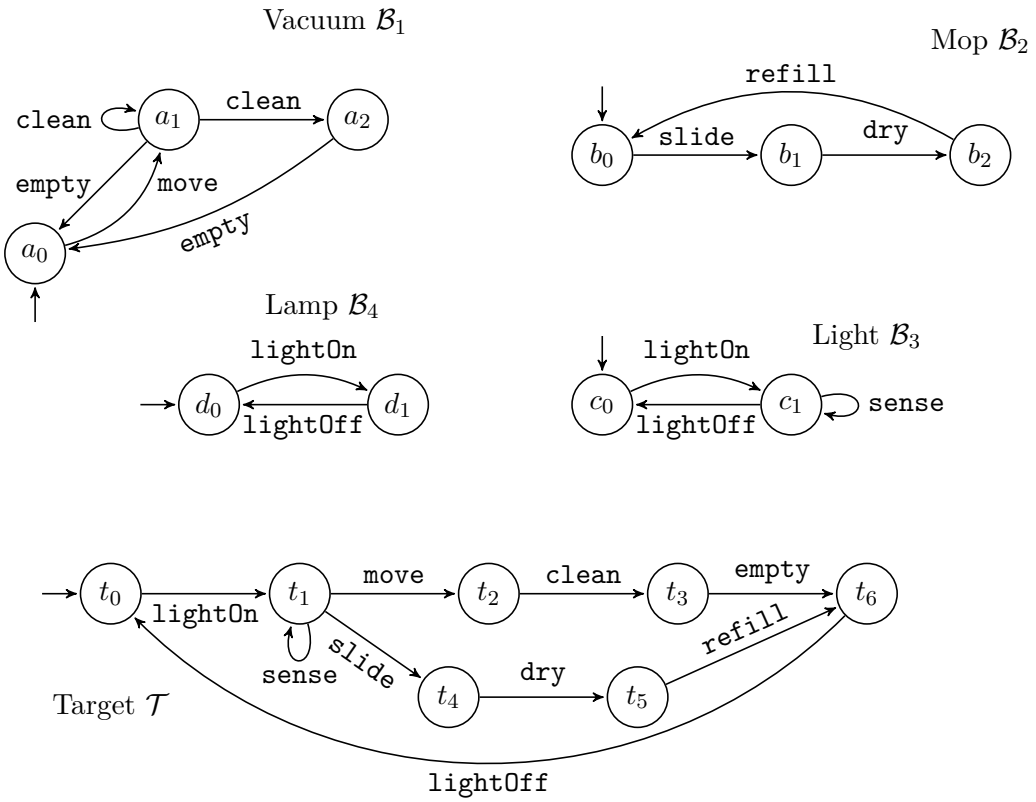


Figure 4.1. A smart house with four available behaviors.

the composition problem guaranteeing the complete realization of the target in the system. For example, it can be shown that there is indeed an exact composition for the example in Figure 4.1: all actions requested as per the target will *always* be fulfilled by the controller.

Since exact compositions are not very informative on unsolvable instances—a mere "no solution" answer is highly unsatisfactory— (Yadav and Sardina, 2012) proposed to look for the *optimal target approximations* of $\mathcal{T}$ in $\mathcal{S}$ instead: the *closest* (w.r.t target $\mathcal{T}$) *alternative* target $\tilde{\mathcal{T}}$ that has an exact composition. To capture the notion of "closeness," the authors relied on a special kind of simulation relation. Importantly, it turns out that there is an exact solution for the original target iff there exists an optimal approximation that is simulation equivalent to it (a property that can be checked in polynomial time). More surprising is the fact that optimal approximations are in fact unique (up to isomorphism). Though not necessary to develop our approach, we refer the reader to (Yadav and Sardina, 2012) for more details on optimal target approximations. However, the notion of target approximation as "*supremal realizable target fragments*" will be investigated in the next Chapter.

4.3.1 DES-based Agent Behavior Composition

Consider an available system (see Section 3.1.1) $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n \rangle$, where $\mathcal{B}_i = \langle B_i, \Sigma, b_{0i}, \varrho_i \rangle$ for $i \in \{1, \ldots, n\}$, and a (deterministic) target $\mathcal{T} = \langle T, \Sigma_T, t_0, \varrho_T \rangle$ (wlog we assume $\mathcal{T}$ to be connected – also, to be consistent with the SCT, we will regard here actions as alphabet elements, thus using symbol Σ).
So, let us next build a product generator $\mathcal{G}$—the *plant* to be controlled—capturing the composition of $\mathcal{T}$ on $\mathcal{S}$. Below, we use two auxiliarly sets $\mathsf{Indx} = \{1, \ldots, n\}$ and $\mathsf{Succ} = \bigcup_{i \in \{1,\ldots,n\}} B_i$.

Definition 4.4. Let the *composition plant* $\mathcal{G} = \langle \Sigma, G, g_0, \gamma, G_m \rangle$ be defined as follows:

- $\Sigma = \Sigma_c \cup \Sigma_u$, where $\Sigma_c = \mathsf{Indx}$ and $\Sigma_u = \Sigma_T \cup \mathsf{Succ}$, is the finite set of controllable and uncontrollable events.
- $G \subseteq T \times B_1 \times \ldots \times B_n \times \Sigma \times (\mathsf{Indx} \cup \{0\})$ is the finite set of states of the plant, encoding the state of all behaviors together with the last event and behavior delegation (0 stands for no delegation).
- $g_0 = \langle t_0, b_{01}, \ldots, b_{0n}, e, 0 \rangle$ is the initial state, encoding the initial configuration of the system and target, and the fact that there has been no event or delegation.
- Transition function $\gamma : G \times \Sigma \to G$ of $\mathcal{G}$ is such that:
  1. $\gamma(\langle t, b_1, \ldots, b_n, e, 0 \rangle, \sigma) = \langle t', b_1, \ldots, b_n, \sigma, 0 \rangle$ *iff* $\sigma \in \Sigma_T$ and $t' = \varrho_T(t, \sigma)$;
  2. $\gamma(\langle t, b_1, \ldots, b_n, \sigma, 0 \rangle, j) = \langle t, b_1, \ldots, b_n, \sigma, j \rangle$ *iff* $\sigma \in \Sigma_T$, $j \in \mathsf{Indx}$;
  3. $\gamma(\langle t, b_1, \ldots, b_n, \sigma, j \rangle, b'_j) = \langle t, b_1, \ldots, b'_j, \ldots, b_n, e, 0 \rangle$ *iff* $b'_j \in \varrho_j(b_j, \sigma)$.
- $G_m = T \times B_1 \times \ldots \times B_n \times \{e\} \times \{0\}$ are the marked states.

△

Let us explain the main ingredients of the composition plant. First, the controllable aspect of the plant amounts to behavior delegations: we can control where each target request is delegated to. On the other hand, the actual target request and the evolution of the selected behavior are uncontrollable events.
A state in the plant stands for a snapshot of the whole composition instance.
In the composition plant, the whole process for one target request involves three γ-transitions in the plant, namely, target action request, behavior delegation, and lastly behavior evolution. Initially and after each target request has been fulfilled, the plant is in a state with no active request (e) and no behavior delegation (0), ready to accept and process a new target request—a *marked* state. So, given a legal target request (uncontrollable event) $\sigma \in \Sigma_T$, the plant evolves to a state recording the request and the corresponding target evolution (case 1 of γ). After that, the composition plant may evolve relative to events representing behavior delegations, one per available behavior, of the pending action to states recording

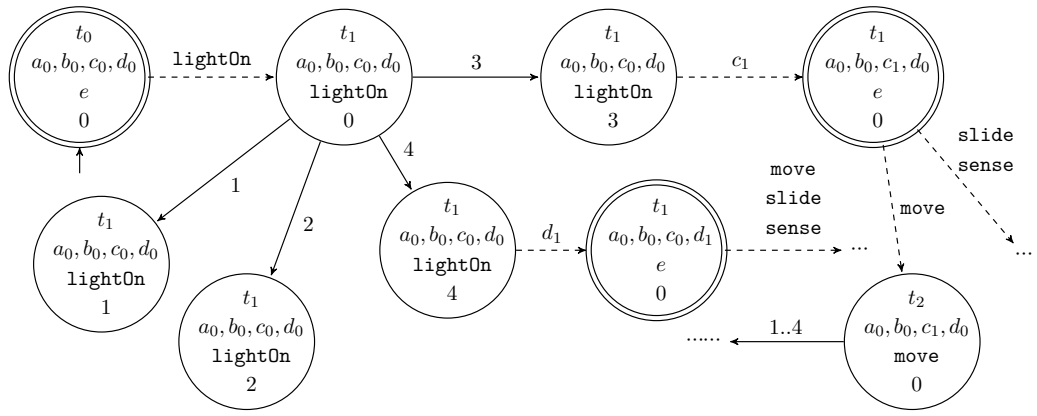


Figure 4.2. Plant $\mathcal{G}$ for the example in Figure 4.1

such delegations (case 2 of γ). At this point, the plant state includes the current state of every behavior, together with the current pending action and its delegation index. Two cases are then possible depending on whether the selected behavior $\mathcal{B}_j$ is capable of performing such action. If it is capable (case 3 of γ), an uncontrollable event representing each potential evolution of the delegated behavior may occur, evolving the state of such behavior and resetting the plant to a state where a new action request can be processed, that is, a marked state. If instead the action can not be performed in the delegated behavior $\mathcal{B}_j$, then the current (non-marked) state is a dead-end. As one can see, marked states represent those states where the whole action request and behavior delegation process has been fully completed.

Example 16. *Figure 7.1 depicts the (partial) plant for the composition problem of Figure 4.1. Transitions with uncontrollable events are depicted as dashed. Note how each complete delegation of actions to behaviors corresponds, in the plant, to three consecutive events in* $(\Sigma_T \cdot \mathit{Indx} \cdot \bigcup_i B_i)$*. Observe also that in the state resulting from a* `lightOn` *uncontrollable event, there are 4 possible ways of delegating such request. However, only two of them (*$\mathcal{B}_3$ *and* $\mathcal{B}_4$*) will not result in immediate dead-end states, and only one of them—namely delegation to* $\mathcal{B}_3$*—will be part of the solution. (By selecting* $\mathcal{B}_4$ *the plant will reach dead-end states when* `sense` *actions is requested.)* □

With the plant built, we next consider the specification to be controlled to be exactly the language $K = L_m(\mathcal{G})$. In other words, we aim at controlling the marked language of the plant, that is, those runs that represent *complete* request-delegation processes. It is important to note that the target behavior $\mathcal{T}$ is *not* used to derive the language specification, but it is embedded instead into the plant itself. The fact is that the target is one of the component generating uncontrollable events (the other being the evolution of available behaviors).

An important technical result is that given any target trace τ of $\mathcal{T}$, the set of system histories that a composition $\mathcal{C}$—a controller solving the problem— yields when τ is executed (denoted $\mathcal{H}_{\mathcal{C},\tau}$ — see Definition 3.3), is in bijection with the set of traces

of plant $\mathcal{G}$. This appears evident by inspecting Figure 7.1, and is formalized in the following lemma. The result uses mapping $\texttt{word}(h) \in (\Sigma_t \cdot \mathsf{Indx} \cdot \bigcup_i B_i)^{|h|}$ to translate a behavior composition system history (i.e., finite traces of the enacted system $\mathcal{E}_\mathcal{S}$) into words generated by composition plant $\mathcal{G}$.

Lemma 1. A controller $\mathcal{C}$ is a composition for target $\mathcal{T}$ on system $\mathcal{S}$ *iff* for each target trace τ and system history $h \in \mathcal{H}_{\mathcal{C},\tau}$ we have that $\texttt{word}(h) \in K^\uparrow$, where:

$$\texttt{word}(\vec{b}_0 \xrightarrow{\sigma_1,j_1} \vec{b}_1 \xrightarrow{\sigma_2,j_2} \dots \xrightarrow{\sigma_k,j_k} \vec{b}_k) = (\sigma_1 \cdot j_1 \cdot \mathsf{st}_{j_1}(\vec{b}_1)) \cdot \dots \cdot (\sigma_k \cdot j_k \cdot \mathsf{st}_{j_k}(\vec{b}_k)).$$

Proof. For convenience, we define a set of ranged functions $\mathsf{st}_i : G \to B_i$, with $i \in \mathsf{Indx}$, such that, given a plant state $g = \langle t, \vec{b}, \sigma, j \rangle$, $\mathsf{st}_i(g)$ is the state component of behaviour $\mathcal{B}_i$ or target $\mathcal{T}$.

($\Rightarrow$) Assume by contradiction that there exists a composition P such that for some target trace τ and induced history $h \xrightarrow{\sigma,j} \vec{b}$, we have $P(h,\sigma) = j$ but $\texttt{word}(h) \cdot \sigma \cdot j \cdot b' \notin K^\uparrow$, with $b' = \mathsf{st}_j(\vec{b})$. This implies that $\texttt{word}(h) \cdot \sigma \cdot j$ is not allowed from the initial state g_0 of the plant, according to supervisor V, i.e., either

(a) $\texttt{word}(h) \cdot \sigma \notin L(\mathcal{G})$ or

(b) $\texttt{word}(h) \cdot \sigma \cdot j \cdot b' \notin L(\mathcal{G})$ or

(c) for all words $w \in L_m(\mathcal{G})$ with $w > \texttt{word}(h) \cdot \sigma \cdot j \cdot b'$ we have $w \notin \overline{K^\uparrow}$.

Case (a) is not possible by construction of $\mathcal{G}$. Indeed, according to γ, it is $w \cdot \sigma \in L(\mathcal{G})$ for every $w \in L_m(\mathcal{G})$ such that $\varrho_T(\mathsf{st}_t(\gamma(g_0, w)), \sigma)$ is defined in $\mathcal{T}$. Case (b) implies, by definition of γ, that $b'_j \notin \varrho_j(b_j, \sigma)$, with $b_j = \mathsf{st}_j(last(h))$. Hence, the action σ can not be replicated by behavior $\mathcal{B}_j$ and, as a consequence, the plant's state reached with σ is a dead-end. This contradicts the fact that P is a composition for $\mathcal{T}$ by $\mathcal{S}$. Finally, case (c) implies that for any such word w we have $w \cdot \Sigma_u \cap L(\mathcal{G}) \notin \overline{K^\uparrow}$, i.e., there exists a sequence of (uncontrollable) events leading to a state which is not coreachable, i.e., from where a marked state is not reachable. Indeed, remember that $K = L_m(\mathcal{G})$. Hence, since every action $\sigma \in \Sigma_t \subset \Sigma_u$ is always allowed by any supervisor and, by construction of $\mathcal{G}$, $w \cdot \sigma \in L(\mathcal{G})$ for every $w \in L_m(\mathcal{G})$ such that $\varrho_T(\mathsf{st}_t(w), \sigma)$ is defined in $\mathcal{T}$, we can apply the same reasoning of (b) and deduce that $\mathcal{C}$ is not a composition. More precisely, there exists a target trace $\tau' = \tau \xrightarrow{\sigma} t_k$, with $h \in \mathcal{H}_{\mathcal{C},\tau}$, which is not realized by $\mathcal{C}$.

($\Leftarrow$) First of all, since $\overline{K} \cap L(\mathcal{G}) \subseteq K$ and by the previous assumption $K^\uparrow \neq \varnothing$, then by Theorem 15 a supervisor V does exist. Hence, $\texttt{word}(h) \cdot \sigma \cdot j \cdot b' \in K^\uparrow$ iff there exists a supervisor V such that $L_m(V/\mathcal{G}) = K^\uparrow$, $\sigma \in V(\texttt{word}(h))$, $j \in V(\texttt{word}(h) \cdot \sigma)$ and $b' \in V(\texttt{word}(h) \cdot \sigma \cdot j)$. Then, remember that $\Sigma_t \subset \Sigma_u$ and hence all target action are always allowed by V. Similarly, the event set Succ is uncontrollable as well. Assume by contradiction that $\texttt{word}(h) \cdot \sigma \cdot j \cdot b' \in K^\uparrow$ but it does not exist any composition $\mathcal{C}$ such that $\mathcal{C}(h, \sigma) = j$. By definition of composition, this

implies that there exists a target trace $\tau = t_0 \xrightarrow{\sigma_1} \cdots t_k$ with $\sigma = \sigma_{|\tau|}$ such that for some history $h \in \mathcal{H}_{\mathcal{C},\tau}$ we have that $\varrho(last(h), \sigma, j)$ is not defined in the enacted system behavior $\mathcal{E}_{\mathcal{S}} = \langle S, \Sigma, s_0, \varrho, S_F \rangle$ built out of $\mathcal{B}_1, \ldots, \mathcal{B}_n$. This means that either $\varrho_T(\mathtt{st}_t(last(h)), \sigma)$ is not defined or behavior $\mathcal{B}_j$ can not perform this action from its current state $\mathtt{st}_j(last(h))$, i.e., $b' \neq \varrho_j(\mathtt{st}_j(last(h)), \sigma)$. In other words, according to γ, $\mathtt{word}(h) \cdot \sigma \cdot j \cdot b' \notin L(\mathcal{G})$. If this is the case, then either $\sigma \notin V(\mathtt{word}(h))$ or $j \notin V(\mathtt{word}(h) \cdot \sigma)$ or $b' \notin V(\mathtt{word}(h) \cdot \sigma \cdot j)$ and we get a contradiction. ■

This result is key to prove our main results of this section, namely, that supervisors able to control the specification K in plant $\mathcal{G}$ correspond one-to-one with composition solution controllers for $\mathcal{T}$ on $\mathcal{S}$. To express such results, we need to relate supervisors and controllers. So, when V is a supervisor V for plant $\mathcal{G}$, we say that a controller $\mathcal{C}_V : \mathcal{H} \times A \to \{1, \ldots, n\}$ is *induced by* V *iff* $\mathcal{C}_V(h, \sigma) \in V(\mathtt{word}(h) \cdot \sigma)$, for every $h \in \mathcal{H}$ and $\sigma \in A$.

Theorem 20 (Soundness)**.** There exists a nonblocking supervisor V such that $L_m(V/\mathcal{G}) = K^{\uparrow} \neq \varnothing$ *iff* there exists a composition $\mathcal{C}$ for $\mathcal{T}$ on $\mathcal{S}$. In particular, every controller P_V induced by V is a composition for $\mathcal{T}$ on $\mathcal{S}$.

Proof. ($\Rightarrow$) Assume by contradiction that for some controller $\mathcal{C}_V$ there exists a target trace $\tau = t_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_k} t_k$ and an induced system history $h \in \mathcal{H}_{\mathcal{C}_V,\tau}$ such that either (a) $\mathcal{C}_V(h, \sigma_k)$ is not defined or (b) $\mathcal{C}_V(h, \sigma_k) = j$ but $\varrho_j(\mathtt{st}_j(last(h)), \sigma_k)$ is not defined. By Lemma 1, it means that $\mathtt{word}(h) \cdot \sigma_k \cdot j \notin \overline{K^{\uparrow}}$. More precisely, (a) implies that $\mathtt{word}(h) \cdot \sigma_k \notin L(\mathcal{G})$ whereas (b) implies that $\sigma_k \notin V(\mathtt{word}(h))$ and $j \notin V(\mathtt{word}(h) \cdot \sigma_k)$. Also, by construction of $\mathcal{G}$, it is $K^{\uparrow} = \varnothing$. ($\Leftarrow$) By Lemma 1, if a composition exists then $K^{\uparrow} \neq \varnothing$. ■

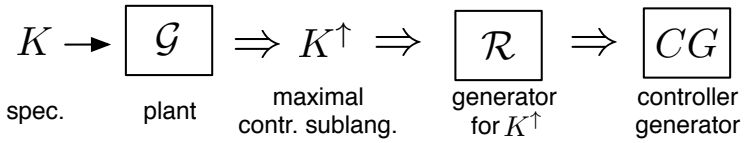
In words, the supremal of the specification is controllable if and only if a solution to the composition problem exists, and every controller extracted from a supervisor is in fact a composition solution. Furthermore the following results show that (nonblocking) supervisors are "complete", in that they embedd every possible composition controller possible.

Theorem 21 (Completeness)**.** Given a nonblocking supervisor V such that $L_m(V/\mathcal{G}) = K^{\uparrow}$, every composition $\mathcal{C}$ for $\mathcal{T}$ by $\mathcal{S}$ is obtained by V.

Proof. Assume by contradiction that there exists a composition $\mathcal{C}'$ which can not be obtained by V, i.e., it is such that $\mathcal{C}'(h, \sigma_{|\tau|}) \notin V(\mathtt{word}(h) \cdot \sigma_{|\tau|})$ for some target trace $\tau = t_0 \xrightarrow{\sigma_1} \cdots t_k$ and some induced system history $h \in \mathcal{H}_{\mathcal{C}',\tau}$. It is easy to see that this contradicts Lemma 1. ■

These two results show the formal link between the two synthesis tasks, namely, synthesis of a composition controller and supervisor synthesis. It remains to be seen how we can actually extract finite representations of composition controllers from supervisors.

From Supervisors to Controller Generators A *compositin controller generator* is a finite structure encoding all possible composition solutions. It is sort of a universal solution for the composition problem at hand. Once the controller generator has been computed offline, it can be stored and used at runtime the implement the target realization. It has been shown how such structure provides flexibility and robustness when it comes for the agent to run the target module. See (De Giacomo et al., 2013; Sardiña et al., 2008) for details on composition controller generators.
We start by noting that, since both languages $L(\mathcal{G})$ and K are regular, they can be implementable. In fact (Wonham and Ramadge, 1987) has shown that it is possible to compute a generator $\mathcal{R}$ that represents exactly the behavior of controllde system $V/\mathcal{G}$, for some supervisor V able to control $K^{\uparrow}$. As such, $\mathcal{R}$ captures not only the control actions of supervisor V, but also all internal events of the plant. Extracting the controller generator amounts to projecting the latter and transforming control actions into behavior delegations. The whole procedure can be depicted as follows: So, suppose the composition task is solvable (and hence $K^{\uparrow} \neq \varnothing$), and take



generator $\mathcal{R} = \langle \Sigma, Y, y_0, \rho, Y_m \rangle$ representing a "good" supervisor for plant $\mathcal{G}$. For any $y \in Y$ we denote with $[y]$ the tuple $\langle \mathsf{st}_t(y), \mathsf{st}_1(y), \ldots, \mathsf{st}_n(y) \rangle$, where function $\mathsf{st}_i(y)$ outputs the local state of target and available behaviors in $\mathcal{R}$'s state y. The *DES controller generator* is a tuple $CG = \langle A_t, \mathsf{Indx}, Q, [y_0], \vartheta, \omega \rangle$, where:

- $Q = \{[y] \mid y \in Y, p \in (\Sigma_t \cdot \mathsf{Indx} \cdot \mathsf{Succ})^*, y = \rho(y_0, p)\}$;
- $\vartheta : Q \times \Sigma_T \times \mathsf{Indx} \to Q$ is such that $\vartheta([y], \sigma, j) = [y']$ *iff* $y' = \rho(y, \sigma \cdot j \cdot b'_j)$ for some $b'_j \in \mathsf{Succ}$;
- $\omega : Q \times \Sigma_T \to Pwr(\mathsf{Indx})$ is the behavior selection function, such that $\omega(q, \sigma) = \{j \mid \exists q' = \vartheta(q, \sigma, j)\}$.

As explained in (De Giacomo et al., 2013), ω *generates* controllers P such that $\mathcal{C}(h, \sigma) \in \omega(last(h), \sigma)$, where h is a system history and σ is a domain action request. The following result demonstrates the correctness of our DES-based approach to compute controller generators.

Theorem 22. A controller $\mathcal{C}$ is a composition of $\mathcal{T}$ by $\mathcal{S}$ iff it is generated by CG.

We note also that the approach is optimal wrt worst-case complexity. Indeed, the plant $\mathcal{G}$ is exponential in the number of behaviors, and the procedure to synthesize the supervisor (that is, to extract $\mathcal{R}$) is polynomial in the size of the plant and the generator for the specification (Wonham and Ramadge, 1987; Gohari and Wonham, 2000). It follows then that computing the DES controller generator can be done in exponential time in the number of behaviors, which is the best we can hope for (De Giacomo and Sardina, 2007).

Q	
0	$\langle a_0, b_0, c_0, d_0 \rangle$
1	$\langle a_0, b_0, c_1, d_0 \rangle$
2	$\langle a_0, b_1, c_1, d_0 \rangle$
3	$\langle a_1, b_0, c_1, d_0 \rangle$
4	$\langle a_0, b_2, c_1, d_0 \rangle$
5	$\langle a_1, b_0, c_1, d_0 \rangle$
6	$\langle a_2, b_0, c_1, d_0 \rangle$
7	$\langle a_0, b_0, c_1, d_0 \rangle$

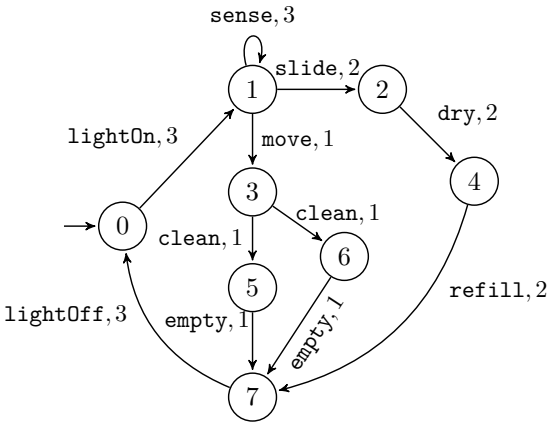


Figure 4.3. DES controller generator for the example in Figure 4.1

We close this section by noting that there are in fact several tools available for the automated synthesis of supervisors for a discrete event system. In particular, we have used TCT (Zhang and Wonham, 2001), in which both the plant to be controlled and the specification are formalized as generators, to extract the generator $\mathcal{R}$ encoding a supervisor for $K^{\uparrow}$.

4.3.2 Composition under Constraints

In practical applications, it may be useful to restrict to composition solutions conforming to some additional constraints. Such constraints can pertain to the evolution of one or more available behaviors or to the way behaviors are scheduled by the controller. For example, we may require that a given available behavior never reaches, under a given global context, an internal (dangerous) state. Similarly, we may impose fairness or load balancing restrictions on how tasks are delegated to behaviors. One may state then that, under certain conditions, an action is never delegated to the same behavior twice in a row, or that at at least half of the modules are to be used for the composition. We show how to do this next.
To represent such extension to the classical composition framework, we consider the problem of computing a composition $\mathcal{C}$ for a target $\mathcal{T}$ on a system $\mathcal{S}$ *under a constraint specification language* $H_c \subseteq \mathcal{H}$. That is, a constraint specification amounts to the set of system histories that are "legal."

Example 17. *As an example, consider again the setting in Figure 4.1 but with* $\mathcal{B}_4$ *modified so as to be identical to light device* $\mathcal{B}_3$*. It is not hard to see that now, differently from before, action* `sense` *can be correctly delegated to both such behaviors. Imagine now to construct a constraint set* H_c *so that fairness among* $\mathcal{B}_3$ *and* $\mathcal{B}_4$ *is achieved by requiring round-robin style selection. In other words, the set* H_c *contains all and only those system traces conforming to the fairness requirement.* □

We say that a controller $\mathcal{C}$ *conforms to* constraints H_c iff for any target trace τ in $\mathcal{T}$ it is the case that $\mathcal{H}_{\mathcal{C},\tau} \subseteq H_c$ (i.e., executions induced by $\mathcal{C}$ when running the

target yields constraint-compatible histories). It is not hard to see that the classical setting with no constraints is obtained by taking $H_c = \mathcal{H}$.

To solve a (constrained) composition problem using our DES-based approach, we first map the constraint specification H_c to its DES version by taking $K_{\mathcal{C}} = \{\mathtt{word}^{-1}(h) \mid h \in H_c\}$. It is not hard to see that $K_{\mathcal{C}} \subseteq \Sigma^*$ (where Σ is the set of events of the composition plant $\mathcal{G}$), even though, of course, at most the regular language $K_{\mathcal{C}} \cap L_m(\mathcal{G})$ can be controlled. We require the set H_c to be prefix-closed, that is, if a history h belongs to H_c so do all its prefixes.

Finally, we apply the approach from Section 4.3.1 but now using $K_{\mathcal{C}}$ directly as the specification of the language to be controlled. Note that if $L_m(\mathcal{G}) \subseteq K_{\mathcal{C}}$—everything that the plant may generate is constraint-compatible—then $K_{\mathcal{C}}$ plays no role and the problem reduces to the standard constraint free setting. On the other hand, when $L_m(\mathcal{G}) \setminus K_{\mathcal{C}} \neq \varnothing$, not any composition solution is adequate.

Since H_c is a prefix-closed set of histories, language $K_{\mathcal{C}}$ is closed as well, that is, $K_{\mathcal{C}} = \overline{K_{\mathcal{C}}}$. As a consequence, we have $\overline{K_{\mathcal{C}}} \cap L_m(\mathcal{G}) \subseteq K_{\mathcal{C}}$. Hence, from Theorem 15, we conclude that if ${K_{\mathcal{C}}}^{\uparrow} \neq \varnothing$, then there exists a nonblocking supervisor V for $\mathcal{G}$ with $L_m(V/\mathcal{G}) = {K_{\mathcal{C}}}^{\uparrow} = \overline{{K_{\mathcal{C}}}^{\uparrow}}$. This leads us to the main result of this section.

Theorem 23 (Correctness)**.** There exists a nonblocking supervisor V such that $L_m(V/\mathcal{G}) = {K_{\mathcal{C}}}^{\uparrow} \neq \varnothing$ iff there exists a composition $\mathcal{C}$ for $\mathcal{T}$ by $\mathcal{S}$ conforming to H_c.

As before, once we have a supervisor V as in the theorem, we can reconstruct all composition controllers $\mathcal{C}$ conforming with $\mathcal{C}$. In turn, assuming regularity of constraint specification H_c, we can finitely implement the supervisor with a generator $\mathcal{R}$ as explained above. However, the procedure for building the DES controller generator from $\mathcal{R}$ needs to be adapted, as we need more information than the current state of the behaviors and the target. More concretely, generator $\mathcal{R}$ may require additional bounded memory to witness the satisfaction of constraints. Therefore, given the automaton $\mathcal{R}$, we define the *memory-based* DES controller generator $CG = \langle \Sigma_t, \mathsf{Indx}, Q', q_0', \vartheta', \omega' \rangle$ as before, except that the state space Q' is just a subset of $\mathcal{R}$ state space Y, namely, $Q' = \{y \in Y \mid p \in (\Sigma_t \cdot \mathsf{Indx} \cdot \mathsf{Succ})^*, y = \rho(y_0, p)\}$. Summarizing, the fact that supervisory control theory is based entirely on languages allows us to integrate sophisticated constraints for the composition task in an almost straightforward manner, by just seeing constraints as sets of "allowed" words that the composition plant may generate.

4.3.3 Supremal Realizable Target Fragment

In Section 3.1.1 we have formalized the notion of (exact) target composition. However, suppose now we are given a target behavior $\mathcal{T}$ and a set of deterministic available behaviors $\mathcal{B}_1, \ldots, \mathcal{B}_n$ such that, as often happens, there is no exact composition for $\mathcal{T}$, i.e., the target cannot be completely realized in the system. There has been a recent interest in the literature to look beyond such exact solutions when such solutions can not be found. The need for "approximations" in problem instances not admitting (easy) exact solutions was first highlighted in (Stroeder and Pagnucco, 2009) and the first attempt to define and study properties of such approximations was done in (Yadav and Sardina, 2012). Roughly speaking, these approximations

are realizable target behaviors (of the original target behavior) that do have exact solutions, and the best approximation is that one closest to the original one. Here, we call such optimal approximations *supremal realizable target behaviors* (SRTB). In this section, we show how to adapt the composition plant $\mathcal{G}$ from Section 4.3.1 to extract SRTBs out of adequate supervisors (rather than exact composition) for the special case of deterministic systems. Indeed, due to intrinsic limitations of the present framework (which is based on languages) we restrict here to the case of deterministic available behaviors only.

A more complete and general approach to such problem will be presented in the next chapter. However, in the context of SCT, it is quite natural to investigate to which extent notions such as the maximality of realizable sublanguage of a given specification can be transferred in a AI synthesis setting.

Let us start then by quickly reviewing the notion of SRTBs by (Yadav and Sardina, 2012). We shall follow their account of *nondeterministic* target behaviors and requests on target transitions (instead of simply actions) to maintain the full controllability of the target module.

The definition of SRTBs relies on the formal (standard) notion of simulation (Milner, 1971a). Intuitively, a (transition) system S_1 "simulates" another system S_2, denoted $S_2 \preceq S_1$, if S_1 is able to always *match* all of S_2's moves.

The definition of simulation relation here is quite similar to Definition 3.4 of ND-simulation. The only difference is that final states are omitted. We report it for clarity.

Definition 4.5. Let $\mathcal{T}_i = \langle S_i, \mathcal{A}, s_{i0}, \varrho_i \rangle$, where $i \in \{1, 2\}$, be two transition systems. A *simulation relation* of $\mathcal{T}_2$ by $\mathcal{T}_1$ is a relation $Sim \subseteq S_2 \times S_1$ such that $\langle s_2, s_1 \rangle \in Sim$ iff: $\forall a, s_2'.\langle s_2, a, s_2' \rangle \in \varrho_2 \rightarrow \exists s_1' \langle s_1, a, s_1' \rangle \in \varrho_1 \wedge \langle s_2', s_1' \rangle \in Sim$. △

Informally, a target behavior $\tilde{\mathcal{T}} = \langle \tilde{T}, \tilde{A}_t, \tilde{t}_0, \tilde{\varrho_T} \rangle$ is a *realizable target behavior* (RTB) of original target specification $\mathcal{T}$ in available system $\mathcal{S}$ iff *(i)* $\tilde{\mathcal{T}}$ is simulated by $\mathcal{T}$ (i.e., $\tilde{\mathcal{T}} \preceq \mathcal{T}$); and *(ii)* $\tilde{\mathcal{T}}$ has an exact composition in $\mathcal{S}$. In addition, an RTB $\tilde{\mathcal{T}}$ is *supremal* (SRTB) iff there is no other RTB $\tilde{\mathcal{T}}'$ such that $\tilde{\mathcal{T}} \prec \tilde{\mathcal{T}}'$ (i.e., $\tilde{\mathcal{T}} \preceq \tilde{\mathcal{T}}'$ but $\tilde{\mathcal{T}} \not\preceq \tilde{\mathcal{T}}'$). Intuitively, a supremal RTB is the closest alternative to the original target that can indeed be completely realized.

The question then is: *can we adapt the DES framework of Section 4.3.1 to obtain SRTBs rather than exact compositions?* In what follows we answer positively to this question for the case when available behaviors in $\mathcal{S}$ are deterministic.

The key to synthesize supremal RTBs by controlling a new composition plant is the fact that since we are no longer committed to realize *all* target traces, the events corresponding to target's requests are now *controllable*. Hence, the supervisor can choose which target's requests to fulfill. In building the new composition plant, we observe that because we assume $\mathcal{S}$ to be deterministic, the whole process for one target request involves now only *two* γ-transitions, both with controllable events. Moreover, to accommodate fully controllable nondeterministic targets, we consider transitions (tuples $\theta \in \varrho_T$) instead of action requests. Thus, a (complete) target action delegation corresponds to two events $(\theta \cdot j) \in (\varrho_T \cdot \mathsf{Indx})$.

So, consider an available system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n \rangle$, where $\mathcal{B}_i = \langle B_i, \Sigma, b_{0i}, \varrho_i \rangle$ for $i \in \{1, \ldots, n\}$ are deterministic, and a target behavior $\mathcal{T} = \langle T, \Sigma_T, t_0, \varrho_T \rangle$. We define the *maximal composition plant* $\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle} = \langle \Sigma, G, g_0, \gamma, G_m \rangle$ as follows:

- $\Sigma = \Sigma_c \cup \Sigma_u$ is the set of events of the plant, where $\Sigma_u = \varnothing$ and $\Sigma_c = \mathsf{Indx} \cup \varrho_T$, i.e., both target transition requests and behavior delegations are controllable;
- $G \subseteq T \times B_1 \times \ldots \times B_n \times \Sigma$ is the finite set of states;
- $g_0 = \langle t_0, b_{01}, \ldots, b_{0n}, e \rangle$ is the initial state of $\hat{\mathcal{G}}_{\langle \mathcal{S},\mathcal{T} \rangle}$;
- $\gamma : G \times \Sigma \to G$ is the transition function, such that:
  1. $\gamma(\langle t, b_1, \ldots, b_n, e \rangle, \theta) = \langle t, b_1, \ldots, b_n, \theta \rangle$ *iff* $\theta = \langle t, \sigma, t' \rangle \in \varrho_T$;
  2. $\gamma(\langle t, b_1, \ldots, b_n, \theta \rangle, j) = \langle t', b_1, \ldots, b'_j, \ldots, b_n, e \rangle$ *iff* $\theta = \langle t, \sigma, t' \rangle \in \varrho_T$, $j \in \mathsf{Indx}$, and $b'_j = \varrho_j(b_j, \sigma)$;
- $G_m = T \times B_1 \times \ldots \times B_n \times \{e\}$.

As before, we take $K = L_m(\hat{\mathcal{G}}_{\langle \mathcal{S},\mathcal{T} \rangle})$ as the specification language to control (in the maximal composition plant). To compute a SRTB for target $\mathcal{T}$ in $\mathcal{S}$ we first compute the language $K^{\uparrow}$, and then build its corresponding generator $\mathcal{R} = \langle \Sigma, Y, y_0, \rho, Y_m \rangle$ as before. Finally, from generator $\mathcal{R}$, we extract the alternative, possibly nondeterministic, target behavior $\mathcal{T}^*_{\langle \mathcal{S},\mathcal{T} \rangle} = \langle T^*, \Sigma_T, y_0, \varrho^*_T \rangle$, where:

- $T^* = \{y \mid y \in Y, p \in (\varrho_T \cdot \mathsf{Indx})^*, y = \rho(y_0, p)\}$;
- $\varrho^*_T \subseteq T^* \times \Sigma_T \times T^*$ is such that $y' \in \varrho^*_T(y, \sigma)$ *iff* $y' = \rho(y, \theta \cdot j)$, where $j \in \mathsf{Indx}$, $\theta \in \varrho_T$, and $\theta = \langle t, \sigma, t' \rangle$.

Next, we present the key results for our technique.

Theorem 24 (Soundness)**.** Let $\mathcal{T}$ be a target and $\mathcal{S}$ an deterministic system. Then, $\mathcal{T}^*_{\langle \mathcal{S},\mathcal{T} \rangle}$ is a SRTB of $\mathcal{T}$ in $\mathcal{S}$.

We note that one can compute also the controller generator for $\mathcal{T}^*_{\langle \mathcal{S},\mathcal{T} \rangle}$ *while* building the SRTB itself, by just keeping track of delegations in $\mathcal{R}$ too, as done in Section 4.3.1.

Theorem 25 (Completeness)**.** Given a nonblocking supervisor V such that $L_m(V/\hat{\mathcal{G}}_{\langle \mathcal{S},\mathcal{T} \rangle}) = K^{\uparrow}$, every composition $\mathcal{C}$ for $\mathcal{T}^*_{\langle \mathcal{S},\mathcal{T} \rangle}$ in system $\mathcal{S}$ can be obtained by V.

In words, every supervisor that can control language $K^{\uparrow}$ in the maximal composition plant $\hat{\mathcal{G}}_{\langle \mathcal{S},\mathcal{T} \rangle}$ encodes all exact compositions of the SRTB $\mathcal{T}^*_{\langle \mathcal{S},\mathcal{T} \rangle}$ built above.

We conclude this section by providing the proofs of Theorem 24 and Theorem 25. In order to do this, we first prove the following two lemmas.

Lemma 2. If $\mathcal{C}$ is a composition for an RTB $\tilde{\mathcal{T}}$ of $\mathcal{T}$ in $\mathcal{S}$ then $W_{\mathcal{C},\tilde{\mathcal{T}},\mathcal{S}} \subseteq K^{\uparrow}$.

Proof. Assume by contradiction that there exists a composition P such that for some target trace $\tau = t_0 \xrightarrow{\sigma_1} t_1 \xrightarrow{\sigma_2} \cdots \to t_k$ of $\tilde{\mathcal{T}}$ and induced history $h \in \mathcal{H}_{\mathcal{C},\tau}$, we have $P(h, \theta) = j$ but $\mathtt{word}(\tau, h) \cdot \theta \cdot j \notin K^{\uparrow}$, where $\theta = \langle t_k, \sigma, t_{k+1} \rangle$ is the new target transition been requested. This implies that $\mathtt{word}(\tau, h) \cdot \theta \cdot j$ is not allowed from the initial state g_0 of the plant, according to supervisor V, i.e., either

(a) $\texttt{word}(\tau, h) \cdot \theta \notin L(\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle})$ or

(b) $\texttt{word}(\tau, h) \cdot \theta \cdot j \notin L_m(\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle})$ or

(c) for all words $w \in L_m(\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle})$ with $w > \texttt{word}(\tau, h) \cdot \theta \cdot j$ we have $w \notin \overline{K^{\uparrow}}$.

Case (a) is not possible by construction of $\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$. Indeed, according to γ, it is $w \cdot \theta \in L(\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle})$ for every $w \in L_m(\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle})$ such that $t = \texttt{st}_t(\gamma(g_0, w))$ and $\varrho_T(t, \sigma)$ is defined in $\mathcal{T}$. Case (b) implies, by definition of γ, that $b'_j \notin \varrho_j(b_j, \sigma)$, with $b_j = \texttt{st}_j(last(h))$. Hence, the action σ can not be replicated by behavior $\mathcal{B}_j$ and, as a consequence, a looping event $e \in \Sigma_u$ is the only one available in $\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$ after the event θ. This contradicts the fact that P is a composition for $\tilde{\mathcal{T}}$ by $\mathcal{S}$. Finally, case (c) implies that θ is not allowed in $\gamma(g_0, w)$ for $w = \texttt{word}(\tau, h)$. By construction of $\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$, this means that for any such a word w we have $w \cdot \Sigma_u \cap L(\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle}) \notin \overline{K^{\uparrow}}$, i.e., there exists an events (corresponding to a target action) leading to a sink state, i.e., a state which is not coreachable. Hence this leads to the same reasoning of case (b). Indeed, by construction of $\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$, $w \cdot \theta \in L(\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle})$ for every $w \in L_m(\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle})$ and $\theta = \langle t, \sigma, t' \rangle$ such that $\varrho_T(\texttt{st}_t(\gamma(g_0, w)), \sigma)$ is defined in $\mathcal{T}$. Hence, this implies there exists a target trace $\tau' = \tau \xrightarrow{\sigma} t_\ell$, with $h \in \mathcal{H}_{\mathcal{C}, \tau}$, which is not realized by $\mathcal{C}$, hence it is not a composition. ■

Lemma 3. $W_{\mathcal{T}^*_{\langle \mathcal{S}, \mathcal{T} \rangle}, \mathcal{S}} = K^{\uparrow}$.

Proof. Assume by contradiction that there exists a word $w \in K^{\uparrow}$ that can not be induced on $\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$ by $\mathcal{T}^*_{\langle \mathcal{S}, \mathcal{T} \rangle}$, namely, $w \notin W_{\mathcal{C}, \mathcal{T}^*_{\langle \mathcal{S}, \mathcal{T} \rangle}, \mathcal{S}}$ for any $\mathcal{C}$. This means that there is no target trace $\tau \in \mathcal{T}^*_{\langle \mathcal{S}, \mathcal{T} \rangle}$ and history $h \in \mathcal{H}_{\mathcal{C}, \tau}$ such that $\texttt{word}(\tau, h) = w$ for some $\mathcal{C}$. Hence, applying the definition of $\texttt{word}(h, \tau)$, consider any word

$$\texttt{word}(\tau, h) = (\langle t_0, \sigma_1, t_1 \rangle \cdot j_1) \cdot \ \ldots \ \cdot (\langle t_{k-1}, \sigma_k, t_k \rangle \cdot j_k)$$

corresponding to any trace $\tau = t_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_k} t_k$ of $\mathcal{T}^*_{\langle \mathcal{S}, \mathcal{T} \rangle}$ and history h induced by any $\mathcal{C}$. Moreover, let

$$w = (\langle t_0, \sigma'_1, t'_1 \rangle \cdot j'_1) \cdot \ \ldots \ \cdot (\langle t'_{k-1}, \sigma'_k, t'_k \rangle \cdot j'_k)$$

the word $w \in K^{\uparrow}$ as before. We want to show that such w can not exist. Hence, for every such trace τ in $\mathcal{T}^*_{\langle \mathcal{S}, \mathcal{T} \rangle}$ and induced history, either (a) $\sigma_i \neq \sigma'_i$ or (b) $t_i \neq t'_i$ or (c) $j_i \neq j'_i$ for some $i \in [1, \ldots, k]$. Case (a) implies that from a state g of $\hat{\mathcal{G}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$ such that $\texttt{st}_t(g) = t_{i-1}$, there is no event $\theta = \langle t_{i-1}, \sigma_i, \_ \rangle$ available. This is impossible, as every transition $\theta \in \varrho_T$ is mapped to an uncontrollable event θ. Similarly, case (b), because of (a) and the fact that t_0 is unique, implies that from a state g such that $\texttt{st}_t(g) = t_{i-1}$, there is no event $\theta = \langle t_{i-1}, \_, t_i \rangle$ available, i.e., to a nondeterministic outcome of the original $\mathcal{T}$ does not correspond any event in the plant. Again, by construction, this is not the case. Finally, if case (c) holds it means that a possible delegation (namely, inducing a controllable word) is not considered by composition $\mathcal{C}$, but this, by construction of $\mathcal{T}^*_{\langle \mathcal{S}, \mathcal{T} \rangle}$ and $\mathcal{R}$, implies

that there exists no plant state y in $\mathcal{R}$ with $y = \rho(y_0, \hat{w})$ where $\hat{w}$ is a prefix of w of length $i \leq k$, i.e., $\hat{w} = \langle t_0, \sigma'_1, t'_1\rangle \cdot j'_1 \cdot \ \ldots \ \cdot \langle t'_{i-1}, \sigma'_i, t'_i\rangle \cdot j'_i$. In other words, it means that $\hat{w} \notin K^{\uparrow}$ and thus it is not controllable. Observe that in this proof we ignored available behaviors' states as they are deterministic, thus univocally determined by the sequence of target actions. ■

With Lemma 3 at hand, we can easily prove the theorems.

Proof. (Proof of Theorem 24) First, we observe that $\mathcal{T}^*_{\langle \mathcal{S},\mathcal{T}\rangle}$ is an RTB of $\mathcal{T}$ in $\mathcal{S}$. It is trivial to prove this by inspection of $\hat{\mathcal{G}}_{\langle \mathcal{S},\mathcal{T}\rangle}$ and the definition of $\mathcal{R}$ (in particular, the fact that $L(\mathcal{R}) = \overline{K^{\uparrow}} \subseteq L(\hat{\mathcal{G}}_{\langle \mathcal{S},\mathcal{T}\rangle})$). Indeed, it is always the case that $\mathcal{T}^*_{\langle \mathcal{S},\mathcal{T}\rangle} \preceq \mathcal{T}$. In particular, transitions in $\hat{\mathcal{G}}_{\langle \mathcal{S},\mathcal{T}\rangle}$ are only defined wrt $\mathcal{T}$'s evolution only (function $\varrho_{\mathcal{T}}$). Then, we prove that it is the maximal one. Assume by contradiction that there exists a RTB $\tilde{\mathcal{T}}$ such that $\mathcal{T}^*_{\langle \mathcal{S},\mathcal{T}\rangle} \prec \tilde{\mathcal{T}}$. First, observe that this implies that there exists a target trace $\tilde{\tau} \in \tilde{\mathcal{T}}$ which is not in $\mathcal{T}^*_{\langle \mathcal{S},\mathcal{T}\rangle}$, and thus $W_{\mathcal{T}^*_{\langle \mathcal{S},\mathcal{T}\rangle},\mathcal{S}} \subset W_{\tilde{\mathcal{T}},\mathcal{S}}$. Indeed there exists a word $w \in W_{\tilde{\mathcal{T}},\mathcal{S}}$ of the form $w = (\theta_1 \cdot j_1) \cdot \ \ldots \ \cdot (\theta_k \cdot j_k)$ such that the transition θ_i is not in $\mathcal{T}^*_{\langle \mathcal{S},\mathcal{T}\rangle}$ for some $i \leq k$. Hence we get a contradiction: applying Lemma 2 for every composition P for $\tilde{\mathcal{T}}$ we deduce that $W_{\tilde{\mathcal{T}},\mathcal{S}} \subseteq K^{\uparrow}$, but for Lemma 3 it is also $W_{\mathcal{T}^*_{\langle \mathcal{S},\mathcal{T}\rangle},\mathcal{S}} = K^{\uparrow}$ ■

Proof. (Proof of Theorem 25) It follows from Lemma 3. Assume by contradiction that there exists a composition $\mathcal{C}'$ which can not be obtained by V, i.e., it is such that $\mathcal{C}'(h,\theta) \notin V(\texttt{word}(\tau,h)\cdot\theta)$ for some target transition $\theta = \langle t_{k-1}, \sigma_k, t_k\rangle$ and target trace $\tau = t_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_k} t_k$ in $\mathcal{T}^*_{\langle \mathcal{S},\mathcal{T}\rangle}$ and some induced system history $h \in \mathcal{H}_{\mathcal{C}',\tau}$. Assume $\mathcal{C}'(h,\theta) = j$. It follows that $\texttt{word}(\tau,h)\cdot\theta\cdot j \notin K^{\uparrow}$, wich contradicts the lemma. ■

4.4 Discussion

In this chapter we have formally shown how solving the agent behavior composition problem can be translated as finding a supervisor able to successfully control a specific discrete event systems (Section 4.3.1). One of the contributions of this work is thus to relate behavior composition and supervisory of discrete-event systems. In doing so, one can expect to leverage on the solid foundations and extensive work in supervisory control theory, as well as on the tools available in those communities.
To this end, we defined a specific encoding of the behavior composition problem as a DES plant. Observe how the target behavior is embedded inside the plant, so as to rule out those evolutions which not correspond to any target computation. A straight-forward approach would be, as customary in Supervisory Control Theory, to use the asynchronous crossproduct of all agent behaviors as the plant (thus capturing their joint asynchronous evolution –i.e. the enacted system $\mathcal{E}_{\mathcal{S}}$) and to use the target behavior as a generator for the specification.
However this does not work. It is well known that, when dealing with non-deterministic systems, language equivalence is weaker than simulation equivalence (and language

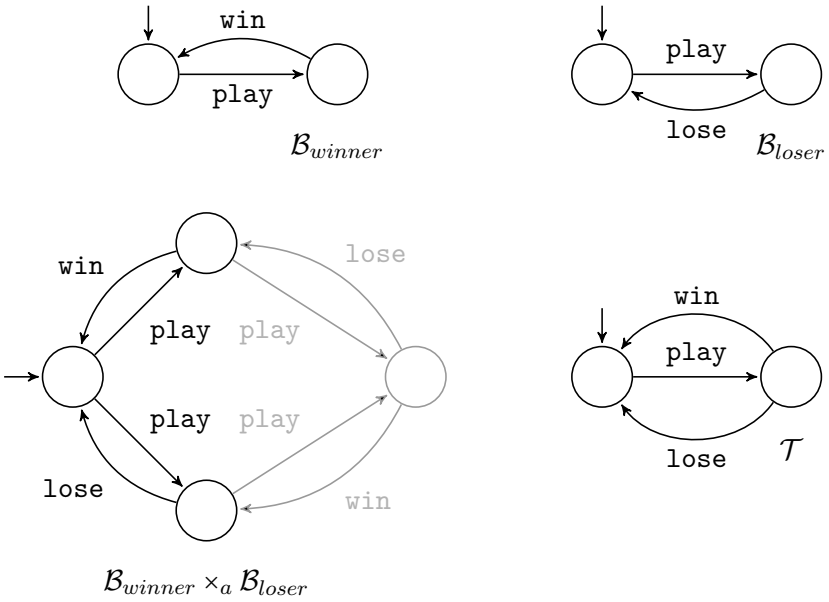


Figure 4.4. The (supervised) asynchronous crossproduct has the same language as the target, but they are not simulation equivalent.

containment weaker than simulation). As a result, although the controlled plant and the target behavior would result to have the same language, this would not be enough to solve the problem. Figure 4.4 illustrates this situation (for the sake of clarity, non-determinism is here allowed, instead of devising a suitable transformation of event labels in the plant, as well as in the specification, along the line of Definition 4.4).
In addition, we have shown how to slightly adapt the encoding to look for "the best possible" target realization when double a perfect one does not exist, though only for the special case of deterministic available systems. In the next chapter, we will investigate the issue of computing the "supremal realizable behavior" in the general case, borrowing notions from SCT.

Chapter 5

On the Supremal Realizable Target

As discussed in Section 3.1.1, the classical agent behavior composition problem has been extensively investigated in the recent literature (refer also to (De Giacomo et al., 2013) for an extensive review). However, one open issue has resisted principled solutions: *if the target specification is not fully realizable, is there a way to realize it "at best"?* (Stroeder and Pagnucco, 2009) were the first to highlight this issue and proposed a search-based method that could eventually be adapted to compute approximate solutions "close" to the perfect one. However, they did not detail what those "approximations" look like. Then, (Yadav and Sardina, 2012) developed a account of "approximate" composition where the task is to return an alternative target agent closest to the original one but fully solvable. While their proposal, based on the formal notion of simulation, comes as a principled generalization of the classical framework, it did not provide ways to actually compute such solutions for the general case, but only for the special case of deterministic behaviors.
In Section 4.3.3 we have investigated this idea in the context of supervisory control, showing how to adapt the composition plant $\mathcal{G}$ from Section 4.3.1 to extract *Supremal Realizable Target Behavior* (SRTBs) out of adequate supervisors for the special case of deterministic systems. In particular, due to intrinsic limitations of the framework (which is based on languages) we could not address the case of nondeterministic available behaviors. Moreover, that technique was aimed at showing a link between agent behavior composition and SCT rather than developing a principled approach to the general problem.
Instead, in this chapter we present a novel, general technique to *effectively build* the largest realizable fragment—the "*supremal*"—of a given target specification for the general composition case in which available behaviors may be nondeterministic. The technique relies on two simple and well-known operations over transition systems (or state models), namely, cross product and belief-level state construction. In doing so, we provide an elegant result on the uniqueness of such fragments.
Then, we investigate—inspired by work on AI reasoning about action (Reiter, 2001b) and on discrete event systems (Cassandras and Lafortune, 2006)—the agent composition task in the presence of *exogenous events*. These are special events that behaviors may *spontaneously generate*, such as the light bulb of a projector fusing

when turned on. Importantly, such events are *uncontrollable* and their occurrence cannot be blocked. As a result, we obtain a strictly more general composition framework. We demonstrate that the supremal realizable target can again be defined and computed. However, this time, solutions come into two variants, depending on the ability of the target's user to observe such events. If exogenous events can be observed by the user, then the supremal fragment may be *conditional* on such events (e.g., if the projector's light bulb fuses, the user may only request changing the bulb). Otherwise, the supremal ought to be *comformant* to all possible exogenous events that may ensue.

5.1 Preliminaries

The problem of concern is the one already illustrated in Section 3.1.1, i.e., the classical agent behavior composition framework as in (De Giacomo and Sardina, 2007; Sardina et al., 2008; Stroeder and Pagnucco, 2009; De Giacomo et al., 2013). As seen in that section, though technically involved, one can formally define when a so-called *controller*, a function taking a run of the system and the next action request and outputting the index of the available behavior to which the action is being delegated, realizes the target behavior. Such controllers are called *exact compositions*, solutions to the composition problem guaranteeing the complete realization of the target in the system.

Example 18. *Consider the presentation room scenario depicted in Figure 5.2, ignoring all dashed transitions. There are two available behaviors, a projector $\mathcal{B}_p$ and a speaker system $\mathcal{B}_a$. The projector allows setting of the* `source` *and* `warmup` *of the device in any order, followed by turning it* `off`*. The speaker on the other hand can simply be toggled on/off. The question then is whether these two devices are enough to be able to run the desired target behavior $\mathcal{T}$, which allows presentations with and without sound. The answer, in this case, is* yes. □

Exact Compositions via Simulation We have seen how it is possible to links exact compositions to the formal notion of simulation (Milner, 1971b). However, for the technical development that follows, we get rid of behaviors' final states that were defined in Section 3.1.1, thus referring to the standard notion of simulation.
For the sake of unambiguity, we repeat here some definition.
A simulation relation captures the behavioral equivalence of two transition systems. Intuitively, a (transition) system S_1 simulates another system S_2, denoted $S_2 \preceq S_1$, if S_1 is able to *match* all of S_2's moves. Thus, (Sardina et al., 2008) defined a so-called ND-simulation (nondeterministic simulation) relation between (the states of) the target behavior $\mathcal{T}$ and (the states of) the enacted system $\mathcal{E}_\mathcal{S}$, denoted $\preceq_{\text{ND}}$, and prove that there exists an exact composition for a target behavior $\mathcal{T}$ on an available system $\mathcal{S}$ *iff* $\mathcal{T} \preceq_{\text{ND}} \mathcal{E}_\mathcal{S}$, that is, the enacted system can ND-simulate the target behavior. While in this work we do not really need the details of ND-simulation, the plain notion of simulation plays a key role. For readability, we report here the definition of simulation already provided in Section 3.1.1.

Definition 5.1. Let $\mathcal{T}_i = \langle S_i, \mathcal{A}, s_{i0}, \varrho_i \rangle$, where $i \in \{1, 2\}$, be two transition systems. A *simulation relation* of $\mathcal{T}_2$ by $\mathcal{T}_1$ is a relation $Sim \subseteq S_2 \times S_1$ such that $\langle s_2, s_1 \rangle \in Sim$ iff: $\forall a, s_2'. \langle s_2, a, s_2' \rangle \in \varrho_2 \rightarrow \exists s_1' \langle s_1, a, s_1' \rangle \in \varrho_1 \wedge \langle s_2', s_1' \rangle \in Sim$. △

We say that a state $s_2 \in T_2$ is *simulated* by a state $s_1 \in T_1$ (or s_1 simulates s_2), denoted $s_2 \preceq s_1$, *iff* there exists a simulation relation Sim of T_2 by T_1 such that $\langle s_2, s_1 \rangle \in Sim$. Observe that relation $\preceq$ is itself a simulation relation (of $\mathcal{T}_2$ by $\mathcal{T}_1$), and in fact, it is the largest simulation relation, in that all simulation relations are contained in it. We say that a transition system $\mathcal{T}_1$ *simulates* another transition system $\mathcal{T}_2$, denoted $\mathcal{T}_2 \preceq \mathcal{T}_1$, if it is the case that $s_{20} \preceq s_{10}$. Two behaviors are said to be *simulation equivalent*, denoted $\mathcal{B}_1 \sim \mathcal{B}_2$, whenever they simulate each other.

Approximated Compositions The classical composition task described above has been extensively studied in the literature and various extensions have been developed (De Giacomo et al., 2013). However, such framework may prove insufficient for composition instances admitting no exact solutions (i.e., unsolvable instances)—a mere "no solution" answer may be highly unsatisfactory in many settings.
The first one to concretely deal with this issue account was (Stroeder and Pagnucco, 2009). In their work, they claimed that their search-based method "can easily be used to calculate approximations," that is, controllers that may not qualify as exact solutions but come "close" (enough) to them. They argue approximations are useful when no exact solution exists and when one is willing to trade faster solutions at the expense of incompleteness (of target realizability). Nonetheless, the authors did not provide a semantic of what these "approximations" are and what "closeness" means, both were left as important future work.
Later, (Yadav and Sardina, 2012) looked closer at a composition framework that can better accommodate unsolvable instances. In doing so, however, they proposed to focus on approximating the target, rather than the controller. To that end, they defined, based on the notion of simulation, what they called *target approximations*, namely, alternative target behaviors that are "contained" in the original target while enjoying exact composition solutions. In turn, they defined the *optimal* target approximation as that one which is "closest" possible to the original target (and that is fully realized by some controller). In fact, they showed that such an optimal target is unique. They also provided a technique to compute such a solution, but only for the special case of deterministic behaviors. Here, we will provide a general technique as well as the complexity characterization of the problem.

5.2 Supremal Realizable Target Behavior

We adopt the approach of (Yadav and Sardina, 2012) to define the notion of realizable target from a target specification. We do not call it "approximation" in light of the extension with exogenous events that we study later. Indeed, once exogenous events are mentioned in the target specification, such a specification is not directly a target behavior anymore.
Formally, we say that behavior $\tilde{\mathcal{T}}$ is a *realizable target behavior* (RTB) of target specification $\mathcal{T}$ on system $\mathcal{S}$ *iff*

1. $\tilde{\mathcal{T}} \preceq \mathcal{T}$ (that is, $\tilde{\mathcal{T}}$ is "contained" in $\mathcal{T}$);
2. $\tilde{\mathcal{T}} \preceq_{\mathrm{ND}} \mathcal{E}_{\mathcal{S}}$, i.e, there is an exact composition for $\tilde{\mathcal{T}}$ on $\mathcal{S}$ (that is, it is fully realizable).

Notice that we elected "simulation" as the measure for comparing target behaviors. In particular if $\mathcal{T}_1 \preceq \mathcal{T}_2$ this means that an agent can mimic the behavior $\mathcal{T}_1$ by suitably choosing the transitions to traverse in $\mathcal{T}_2$. If $\mathcal{T}_1$ and $\mathcal{T}_2$ are simulation equivalent (i.e., $\mathcal{T}_1 \preceq \mathcal{T}_2$ and $\mathcal{T}_2 \preceq \mathcal{T}_1$) then the agent can mimic exactly one behavior using the other one, hence from the point of view of the agent the two behaviors are identical.

Definition 5.2. A behavior $\tilde{\mathcal{T}}$ is *"the"* *supremal realizable target behavior* (SRTB) of target $\mathcal{T}$ on system $\mathcal{S}$ *iff* $\tilde{\mathcal{T}}$ is a RTB of $\mathcal{T}$ in $\mathcal{S}$ and there is no RTB $\tilde{\mathcal{T}}'$ such that $\tilde{\mathcal{T}} \prec \tilde{\mathcal{T}}'$—$\tilde{\mathcal{T}}$ is the largest realizable fragment of $\mathcal{T}$. It can be shown that SRTB is unique up to simulation equivalence. △

We provide a simple and elegant characterization of SRTB as follows. Let $\mathcal{T}_1 \cup \mathcal{T}_2 = \langle T, Act, t_{10}, \varrho \rangle$, where $\mathcal{T}_i = \langle T_i, Act_i, t_{i0}, \varrho_i \rangle$ have disjoint states, be the resulting unified (target) behavior where T_2's initial state is merged with T_1's: *(i)* $T = T_1 \cup (T_2 \setminus \{t_{20}\})$; *(ii)* $Act = Act_1 \cup Act_2$; and $\varrho = \varrho_1 \cup \varrho_2|_{t_{20}}^{t_{10}}$ ($\varrho|_t^{t'}$ is relation ϱ with all states t replaced with t').

Theorem 26. Let $\tilde{\mathcal{T}}_1$ and $\tilde{\mathcal{T}}_2$ be two RTB for target specification $\mathcal{T}$ in system $\mathcal{S}$. Then $\mathcal{T}_1 \cup T_2$ is an RTB for $\mathcal{T}$ in $\mathcal{S}$ too.

In words, RTBs are closed under union. With this in mind, it is not hard to see that one can build the largest RTB—the supremal—by taking the union of all realizable targets.

Theorem 27. Let $\mathcal{S}$ be an system and $\mathcal{T}$ be a target. Then the SRTB $\mathcal{T}^*$ of $\mathcal{T}$ in $\mathcal{S}$ is:

$$\mathcal{T}^* = \bigcup_{\tilde{\mathcal{T}} \text{ is a RTB of } \mathcal{T} \text{ in } \mathcal{S}} \tilde{\mathcal{T}}$$

Notice that any $\tilde{\mathcal{T}}$ which is simulation equivalent to $\mathcal{T}^*$ is also "the" SRTB (we focus on semantics not syntax here).
Obviously, it remains to be seen if the SRTB can actually be computed and represented finitely. This is what we do next. Our technique to synthesize the SRTB relies on two simple operations on transition systems, namely, a specific synchronous product and a *conformance* enforcing procedure. Roughly speaking, the technique is as follows:

1. We take the *synchronous product* of the enacted system $\mathcal{E}_{\mathcal{S}}$ and the target spec. $\mathcal{T}$, yielding the structure $\mathcal{F}_{\langle \mathcal{S}, \mathcal{T} \rangle}$.
2. We modify $\mathcal{F}_{\langle \mathcal{S}, \mathcal{T} \rangle}$ to *enforce conformance* on its states which cannot be distinguished by the user of the target.

In fact the second step is needed only when the system includes *nondeterministic available behaviors*.

Full enacted system The full enacted system models the behavior that emerges from joint parallel execution of the enacted system and the target.

Definition 5.3. Given the enacted system $\mathcal{E}_{\mathcal{S}} = \langle S, \mathcal{A}_{\mathcal{S}}, s_0, \delta \rangle$ for a system $\mathcal{S} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ and a target specification $\mathcal{T} = \langle T, Act_T, t_0, \varrho_T \rangle$, the *full enacted system* of $\mathcal{T}$ and $\mathcal{S}$, denoted by $\mathcal{T} \times \mathcal{E}_{\mathcal{S}}$, is a tuple $\mathcal{F}_{\langle \mathcal{S},\mathcal{T} \rangle} = \langle F, \mathcal{A}_{\mathcal{F}}, f_0, \gamma \rangle$, where:

- $F = S \times T$ is the finite set of $\mathcal{F}_{\langle \mathcal{S},\mathcal{T} \rangle}$'s states; when $f = \langle s, t \rangle$, we denote s by $\mathsf{sys}(f)$ and t by $\mathsf{tgt}(f)$;
- $f_0 = \langle s_0, t_0 \rangle \in F$, is $\mathcal{F}_{\langle \mathcal{S},\mathcal{T} \rangle}$'s initial state;
- $Act_{\mathcal{F}} = Act_{\mathcal{S}} \cup Act_T$ (note that we allow for $Act_{\mathcal{S}} \neq Act_T$);
- $\gamma \subseteq F \times Act \times \{1, \ldots, n\} \times F$ is $\mathcal{F}_{\langle \mathcal{S},\mathcal{T} \rangle}$'s transition relation, where $\langle f, a, k, f' \rangle \in \gamma$, or $f \xrightarrow{a,k} f'$ in $\mathcal{F}_{\langle \mathcal{S},\mathcal{T} \rangle}$ *iff*
  - $\mathsf{tgt}(f) \xrightarrow{a} \mathsf{tgt}(f')$ in $\mathcal{T}$; and
  - $\mathsf{sys}(f) \xrightarrow{a,k} \mathsf{sys}(f')$ in $\mathcal{E}_{\mathcal{S}}$.

△

Observe that the transition relation of the full enacted system requires both the enacted system and the target to evolve *jointly*: the full enacted system is the *synchronous product* of the target specification and the enacted system.
As expected, the synchronous product (once we project out the indexes $\{1, \ldots, n\}$) is simulated by both the enacted system and the target (i.e., $\mathcal{T} \times \mathcal{E}_{\mathcal{S}} \preceq \mathcal{E}_{\mathcal{S}}$ and $\mathcal{T} \times \mathcal{E}_{\mathcal{S}} \preceq \mathcal{T}$). If the system includes only deterministic available behavior the regular simulation $\mathcal{T} \times \mathcal{E}_{\mathcal{S}} \preceq \mathcal{E}_{\mathcal{S}}$ suffices to conclude that the composition exists (ND-simulation is not needed in this case) (Sardina et al., 2008) . Hence, by Theorem 27, if available behaviours are deterministic $\mathcal{T} \times \mathcal{E}_{\mathcal{S}}$ is included in, and simulated by, $\mathcal{T}^*$. The converse can be shown along the line suggested in (Yadav and Sardina, 2012). Hence:

Theorem 28. Let $\mathcal{S} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ be a *deterministic* available system and $\mathcal{T} = \langle T, Act_T, t_0, \varrho_T \rangle$ a target specification behavior. Then, $\mathcal{F}_{\langle \mathcal{S},\mathcal{T} \rangle}$ is the SRTB of $\mathcal{T}$ in $\mathcal{S}$.

Also from the construction of $\mathcal{F}_{\langle \mathcal{S},\mathcal{T} \rangle}$ we can conclude that building the SRTB of $\mathcal{T}$ in $\mathcal{S}$ can be done in exponential time in the number of behaviors and polynomial in the number of states in each behavior.
When we consider nondeterministic available modules, and hence resort to ND-simulation, this is not true anymore. Indeed, there are examples where $\mathcal{T} \times \mathcal{E}_{\mathcal{S}} \not\preceq_{ND} \mathcal{E}_{\mathcal{S}}$ does *not* hold due to the nondeterminism present in $\mathcal{E}_{\mathcal{S}}$. In those cases, the full enacted system is a sort of target behavior in which agent transition requests are *conditional* on the nondeterministic execution of available behaviors. However, the agent using the target is not meant to have observability on such behaviors, and so it cannot decide its request upon such contingencies. Figure 5.1 shows one

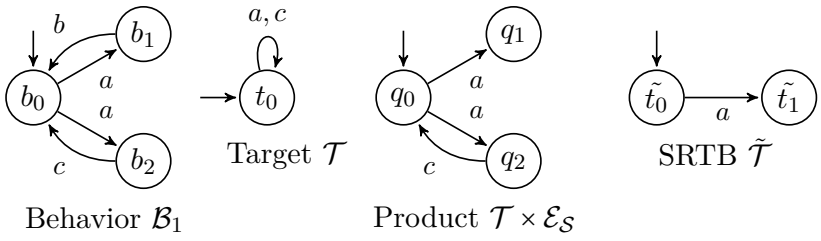


Figure 5.1. Instance where full enacted system is not a RTB

such case. Take product $\mathcal{T} \times \mathcal{E}_{\mathcal{S}}$ as a candidate for SRTB. After fulfilling transition request $q_0 \xrightarrow{a} q_2$ using module $\mathcal{B}_1$, the next request $q_2 \xrightarrow{c} q_0$ can only be honored *if* $\mathcal{B}_1$ happens to evolve to state b_2, but this is not guaranteed. Therefore, $\mathcal{T} \times \mathcal{E}_{\mathcal{S}}$ cannot be realized by $\mathcal{B}_1$ and hence it is not an RTB of $\mathcal{T}$ in $\mathcal{S}$.
What we need, is the target to be *conformant*, i.e., independent of conditions on the available behaviors states. Hence inspired by the literature on planning under uncertainty we construct a sort of belief states, and in turn, the *belief level full enacted system*. The idea behind generating the belief states is to track the states where the enacted system could evolve. Given a full enacted system $\mathcal{F}_{\langle \mathcal{S},\mathcal{T} \rangle} = \langle F, \mathcal{A}_{\mathcal{F}}, f_0, \gamma \rangle$ for a target $\mathcal{T} = \langle T, Act_T, t_0, \varrho_T \rangle$ and a system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n \rangle$ where $\mathcal{B}_i = \langle B_i, Act_i, b_{i0}, \varrho_i \rangle$ for $i \leq n$, the *belief-level full enacted system* is a tuple $\mathcal{K}_{\langle \mathcal{S},\mathcal{T} \rangle} = \langle Q, Act_{\mathcal{F}}, q_0, \delta_K \rangle$, where:

- $Q = 2^{(B_1 \times \cdots \times B_n)} \times T$ is $\mathcal{K}_{\langle \mathcal{S},\mathcal{T} \rangle}$'s set of states; when $q = \langle \{s_1, \ldots, s_\ell\}, t \rangle \in Q$ we denote $\{s_1, \ldots, s_\ell\}$ by $\mathsf{sys}(q)$ and t by $\mathsf{tgt}(q)$;
- $q_0 = \langle \{s_0\}, t_0 \rangle$ such that $f_0 = \langle s_0, t_0 \rangle$, is the initial state;
- $\delta_K \subseteq Q \times Act \times Q$ is $\mathcal{K}_{\langle \mathcal{S},\mathcal{T} \rangle}$'s transition relation, with $\langle \langle S, t \rangle, a, \langle S', t' \rangle \rangle \in \delta_K$ iff :
  - there exists a set $\mathsf{Indx} = \{\langle s_1 : k_1 \rangle, \ldots, \langle s_\ell : k_\ell \rangle\}$ such that $\{s_1, \ldots, s_\ell\} = S$; and $\langle s_i, t \rangle \xrightarrow{a,k_i} \langle s', t' \rangle$ in $\mathcal{F}_{\langle \mathcal{S},\mathcal{T} \rangle}$ for all $i \leq \ell$; that is, the action a must be executable from all enacted system states in S; and
  - $S' = \bigcup_{\langle s:i \rangle \in \mathsf{Indx}} \{s' \mid \langle \langle s, t \rangle, a, i, \langle s', t' \rangle \rangle \in \gamma\}$; that is, S' should contain all successors of enacted system states in S resulting from action a.

Observe, $\mathcal{K}_{\langle \mathcal{S},\mathcal{T} \rangle}$ is nondeterministic with respect to target evolutions and different behavior delegations. Note also that $\mathcal{K}_{\langle \mathcal{S},\mathcal{T} \rangle}$ can be built in time $2^{O(|\mathcal{B}|^n)}$ where $|\mathcal{B}|$ is the number of states of the largest behavior in $\mathcal{S}$, and n is the number of available behaviors in $\mathcal{S}$. Observe, however, that $\mathcal{K}_{\langle \mathcal{S},\mathcal{T} \rangle}$ can be computed *on-the-fly* in a step-wise fashion: given the current belief state q we can generate the next possible states without looking at any other state in Q.
Next, we show that $\mathcal{K}_{\langle \mathcal{S},\mathcal{T} \rangle}$ is the finite representation of SRTB $\mathcal{T}^*$ of target $\mathcal{T}$ in system $\mathcal{S}$ (see Theorem 27).

Theorem 29. Let $\mathcal{S}$ be an available system and $\mathcal{T}$ a target specification behavior. Then, $\mathcal{K}_{\langle \mathcal{S},\mathcal{T} \rangle}$ is the SRTB of $\mathcal{T}$ in $\mathcal{S}$.

Proof. First, we define few technical notions required for the proofs. Given a trace $\tau = s_0 \xrightarrow{a^1} \cdots \xrightarrow{a^n} s_n$, we denote the state s_i by $\tau[i]$, the label a^i by $\tau\langle i\rangle$, and prefix $s_0 \xrightarrow{a^1} \cdots \xrightarrow{a^i} s_i$ by $\tau[0,i]$, where $i \leq n$. Given a set of traces Γ, let $\mathsf{Pos}(\Gamma, i) = \{s \mid s = \tau[i], \tau \in \Gamma\}$ be the function that returns the set of i^{th} state from all traces in Γ.

Then, we prove that $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$ and the SRTB $\mathcal{T}^*$ of $\mathcal{T}$ in $\mathcal{S}$ are simulation equivalent. Proof for $\mathcal{T}^* \preceq \mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$: First, will show that all RTB's are simulated by $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$. Let $\mathcal{T}' = \langle T', Act', t'_0, \varrho'_T\rangle$ be a RTB of $\mathcal{T}$ in $\mathcal{S}$. Assume $\mathcal{T}' \not\preceq \mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$, that is, $\mathcal{T}'$ is not simulated by $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$. Let $\tau_{\mathcal{T}'} = t'_0 \xrightarrow{a^1} \cdots \xrightarrow{a^n} t'_n$ be a trace of $\mathcal{T}'$ such that $\tau_{\mathcal{T}'}$ cannot be simulated state-wise by any trace of $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$ and the simulation breaks at a state t'_{n-1}. We show that this is impossible since, we can build a legal trace of $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$ which can simulate the entire τ'.
As $\mathcal{T}'$ is a RTB of $\mathcal{T}$ in $\mathcal{S}$, it holds that $\mathcal{T}' \preceq \mathcal{T}$ ($\mathcal{T}'$ is simulated by $\mathcal{T}$) and $\mathcal{T}' \preceq_{\mathrm{ND}} \mathcal{E}_{\mathcal{S}}$ ($\mathcal{T}'$ has an exact solution in $\mathcal{S}$). Therefore, there exists a trace of $\mathcal{T}$ $\tau_{\mathcal{T}} = t_0 \xrightarrow{a^1} \cdots \xrightarrow{a^n} t_n$ such that $t'_i \preceq t_i$ for all $i \leq n$;
Let us define $\Gamma_{\mathcal{S}}$ as the maximal set of traces $s_0 \xrightarrow{a^1,k_1} \cdots \xrightarrow{a^n,k_n} s_n$ of enacted system $\mathcal{E}_{\mathcal{S}}$ of $\mathcal{S}$, such that:

1. $t'_i \preceq_{\mathrm{ND}} s_i, i \leq n$, i.e., which copy the target trace $\tau_{\mathcal{T}'} = t'_0 \xrightarrow{a^1} \cdots \xrightarrow{a^n} t'_n$;
2. they do so through transitions labelled by a_i, k_i for $i \leq n$ such that for any two traces $\tau_1, \tau_2 \in \Gamma_{\mathcal{S}}$ it is the case that if $\tau_1[i] = \tau_2[i]$, then $\tau_1\langle i\rangle = \tau_2\langle i\rangle$.

Since, $\mathcal{T}'$ is realizable in $\mathcal{S}$ we know that at least one composition exists. Therefore, $\Gamma_{\mathcal{S}}$ will not be empty. Notice that, because of condition 2 above, there may be several such maximal sets. We nondeterministically take one.
Now, consider a trace $\tau_{\mathcal{K}} = q_0 \xrightarrow{a^1} \cdots \xrightarrow{a^n} q_n$ such that $q_i = \langle \mathsf{Pos}(\Gamma_{\mathcal{S}}, i), \tau_{\mathcal{T}}[i]\rangle$ for all $i \leq n$. The idea behind Pos is to return all states where the enacted system could be in. We show $\tau_{\mathcal{K}}$ is a *legal trace of* $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$, that is, it consists of legal states and transitions. We start by observing that:

- $\tau_{\mathcal{K}}[i] = \langle \{s_1, \ldots, s_\ell\}, t\rangle$, where $\{s_1, \ldots, s_\ell\} = \mathsf{Pos}(\Gamma_{\mathcal{S}}, i)$ and $t = \tau_{\mathcal{T}}[i]$, is a legal state of $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$ for all $i \leq n$;
- $\tau_{\mathcal{K}}[0]$ is the initial state of $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$.

Then we proceed by induction on n.

- For $n = 0$, we have that the trace $\tau_{\mathcal{K}}[0]$ consisting only of the initial state is trivially legal.
- By inductive hypothesis let us assume that $q_0 \xrightarrow{a^1} \cdots \xrightarrow{a^i} q_i$ (for $i < n$) is a legal trace of $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$, and we show that also $q_0 \xrightarrow{a^1} \cdots \xrightarrow{a^{i+1}} q_{i+1}$ is a legal trace of $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$.

Consider the transition $q_i \xrightarrow{a^{i+1}} q_{i+1}$ of $\tau_{\mathcal{K}}$. Let $\mathsf{Pos}(\Gamma_{\mathcal{S}}, i) = \{s_1, \ldots, s_\ell\}$. Since τ' is realizable, there exists $s_j \xrightarrow{a^{i+1}, k_j^{i+1}} s'_j$ in $\mathcal{E}_{\mathcal{S}}$ for $j \leq \ell$ and $t_i \xrightarrow{a^{i+1}} t_{i+1}$ in $\mathcal{T}$. Hence, there exists exactly one set of indices (see definition of Γ_S, condition 2), $\mathsf{Indx} = \{\langle s_1 : k_1\rangle, \ldots, \langle s_\ell : k_\ell\rangle\}$, one per each element in $\mathsf{Pos}(\Gamma_{\mathcal{S}}, i)$, such that $\langle s, \tau_{\mathcal{T}}[i]\rangle \xrightarrow{a^{i+1}, k^{i+1}} \langle s', \tau_{\mathcal{T}}[i{+}1]\rangle$ in $\mathcal{F}_{\langle \mathcal{S},\mathcal{T}\rangle}$ where $s \in \mathsf{Pos}(\Gamma_{\mathcal{S}}, i)$, $s' \in \mathsf{Pos}(\Gamma_{\mathcal{S}}, i+1)$ and $\langle s : k^{i+1}\rangle \in \mathsf{Indx}$. That is, $q_i \xrightarrow{a^{i+1}} q_{i+1}$ in $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$.

So, RTB $\mathcal{T}'$ is simulated by $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$ (once we project out the indexes $\{1, \ldots, n\}$), that is, $\mathcal{T}' \preceq \mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$. From theorem 26 we know that union of two RTB's is an RTB, therefore $\mathcal{T}^*$ is also a RTB. Consequently, $\mathcal{T}^* \preceq \mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$.

To proof $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle} \preceq \mathcal{T}^*$, we simply observe that $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$ is an RTB, since by construction, we have $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle} \preceq \mathcal{T}$ and $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle} \preceq_{\mathrm{ND}} \mathcal{E}_{\mathcal{S}}$. Hence $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$ by theorem 26 $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$ is included in, and thus simulated by, $\mathcal{T}^*$.
To proove $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle} \preceq \mathcal{T}^*$, we simply observe that $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$ is an RTB, by construction, we have $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle} \preceq \mathcal{T}$ and $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle} \preceq_{\mathrm{ND}} \mathcal{E}_{\mathcal{S}}$. Hence $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$ by theorem 26 $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle}$ is included in, and thus simulated by, $\mathcal{T}^*$. ■

We note some similarities in the use of belief-level behaviors with the work in (De Giacomo et al., 2009) for composition under partial observability of the available behaviors. There the *controller* required to be conformant, here instead the *target behavior* must be so.

5.3 Composition with Exogenous Events

With an effective technique to synthesize the supremal realizable target at hand, we now turn to the second contribution. Inspired by discrete event systems (Cassandras and Lafortune, 2006) and reasoning about action work for dynamic systems (Reiter, 2001b), we show here how to accommodate *exogenous uncontrollable events* into the composition framework in a parsimonious manner. In doing so, it will come clear how robust and elaboration tolerant the definition of SRTBs and the technique to compute them are.

Example 19. *Let us return to our presentation room example in Figure 5.2. Suppose that when the projector's light bulb is on—after* `warmup` *has been executed—it may* `fuse` *anytime and requires the device to be repaired. Similarly, if a source is set before warming the projector up, an* `error` *may be thrown and the projector will need to be* `reset`*. The occurrence of both events—*`fuse` *and* `error`*—is outside the control of the client or the controller, they occur* spontaneously*. Hence, they are akin to* exogenous events *in reasoning about action literature (Reiter, 2001b) and* uncontrollable events *in discrete event systems (Cassandras and Lafortune, 2006).* □

Next, we extend the classical composition framework from Section 3.1.1 with *exogenous events*. To that end, we assume that the set of actions $\mathcal{A}$ in a behavior is

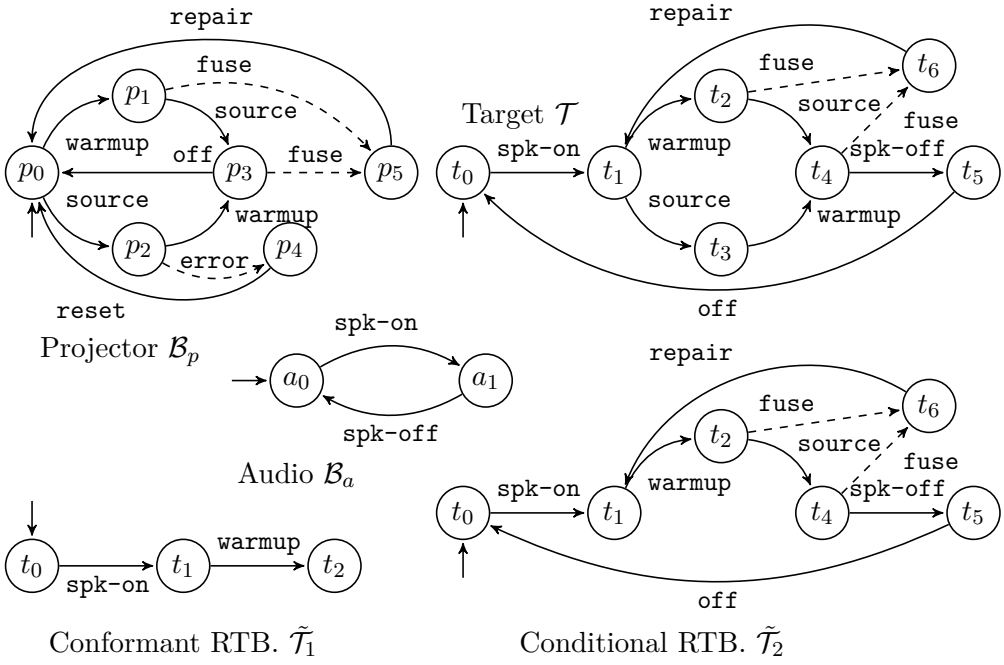


Figure 5.2. Media room scenario consisting of a projector, speaker and a target specification (see text for details). Dashed transitions denote uncontrollable exogenous events.

partitioned into *domain* ($\mathcal{A}^C$) and exogenous ($\mathcal{A}^U$) events, that is, $\mathcal{A} = \mathcal{A}^C \cup \mathcal{A}^U$ and $\mathcal{A}^C \cap \mathcal{A}^U = \varnothing$. Furthermore, as standard in discrete event systems, we assume exogenous events to be deterministic.[1]

We note that exogenous events play a inherently different role in available behaviors than nondeterminism. Exogenous (uncontrollable) events may happen *anytime* from a relevant state (e.g., p_1 in $\mathcal{B}_\mathcal{P}$), which allows modeling of concepts such as delayed uncertainty. Moreover, whereas nondeterminism is *not* observable to the target's user (in fact, the user agent is not even aware of the internal logic of available behaviors), exogenous events may be. Hence, the user of the projector room may be able to observe the light bulb fusing.

When it comes to the target specification, exogenous event transitions represent those transitions that are accounted—accepted—by the target but outside the controller of the user of the target. Thus, when the target is in state t_2, it only allows one exogenous event, namely, event `fuse`, whose occurrence will cause the target to evolve to state t_6 where its user is only allowed to request repairing the projector.

Since the user may be able to observe exogenous events, we can now consider—unlike standard composition—two types of composition solutions. Following planning terminology, a *conditional* SRTB is one that assumes the user is able to observe exogenous events, whereas a *conformant* SRTB is one where such events are non-observable to the user.

[1] Should this not be the case, we can model the various outcomes with different uncontrollable exogenous events.

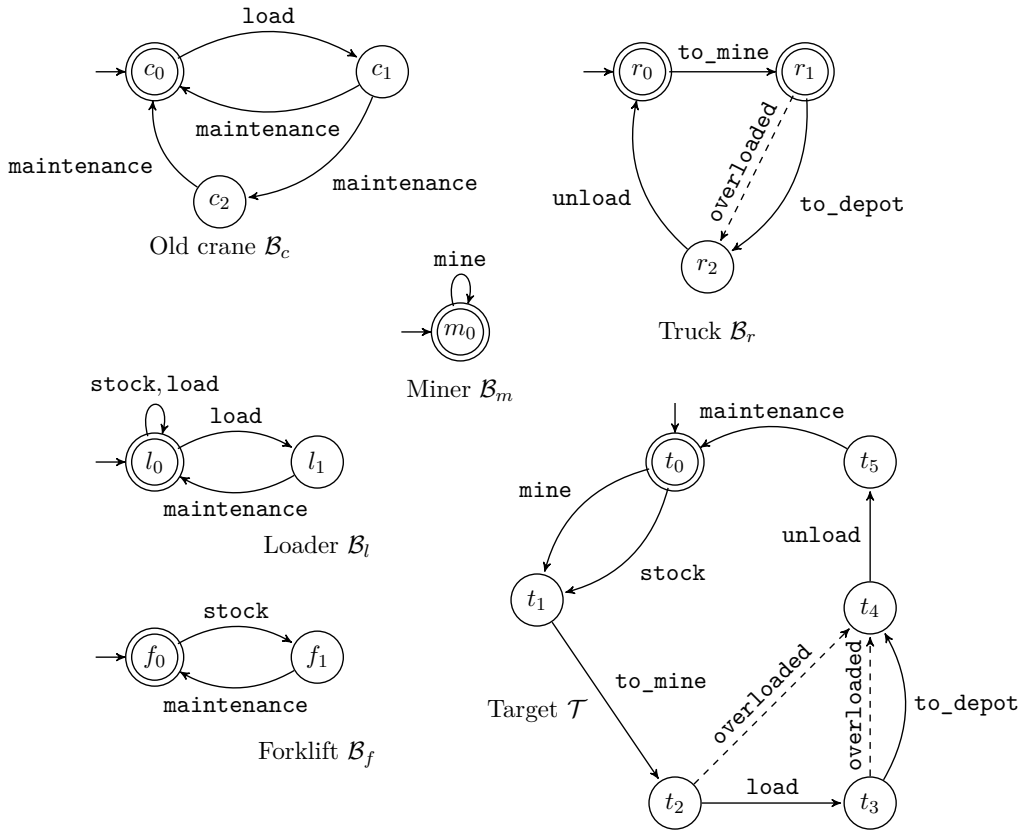


Figure 5.3. A modification of the mining example of previous Chapters.

In this section, we formally define conditional and conformant solution concepts and explain how to generalize the technique developed in Section 5.2 to compute such solutions.

Enacted and full enacted system The formal definition of the enacted system and the full enacted system remains same, except we assume the action set to be partitioned into controllable actions and uncontrollable exogenous events. Figure 5.4 depicts the reachable full enacted system for the media room example. See that states from where `error` may fire are excluded. In addition, given a full enacted system $\mathcal{F}_{\langle\mathcal{S},\mathcal{T}\rangle} = \langle F, \mathcal{A}^C_{\mathcal{F}} \cup \mathcal{A}^U_{\mathcal{F}}, f_0, \gamma\rangle$ for an enacted system $\mathcal{E}_{\mathcal{S}} = \langle S, \mathcal{A}^C_{\mathcal{S}} \cup \mathcal{A}^U_{\mathcal{S}}, s_0, \delta\rangle$ and a target specification $\mathcal{T} = \langle T, \mathcal{A}^C_T \cup \mathcal{A}^U_T, t_0, \varrho_T\rangle$, we define set $\Delta_{\langle\mathcal{S},\mathcal{T}\rangle}$ as those states in $\mathcal{F}_{\langle\mathcal{S},\mathcal{T}\rangle}$ from where prohibited exogenous events may fire. Formally,

$$\Delta_{\langle\mathcal{S},\mathcal{T}\rangle} = \{\langle s,t\rangle \,|\, \langle s,\alpha,k,s'\rangle \in \delta, \forall t' \langle t,\alpha,t'\rangle \notin \varrho_T : \alpha \in \mathcal{A}^U_{\mathcal{S}}\}.$$

5.3.1 Conditional SRTBs

When it comes to formally defining conditional SRTBs, interestingly, the definition of SRTBs from the classical framework (see Section 5.2) fits as is. However, we need to define exact solutions in the context of exogenous events. We do this by extending the ND-simulation relation in the light of exogenous events.

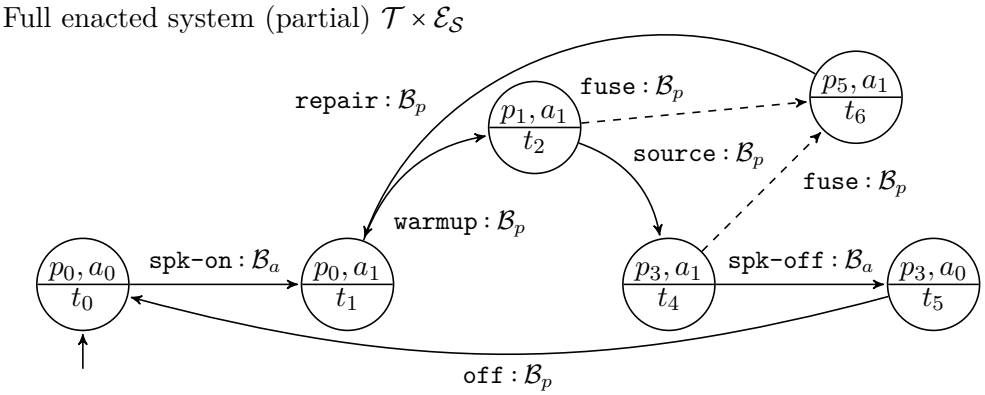


Figure 5.4. $\mathcal{F}_{\langle \mathcal{S},\mathcal{T} \rangle}$ with exogenous events for Example 5.2.

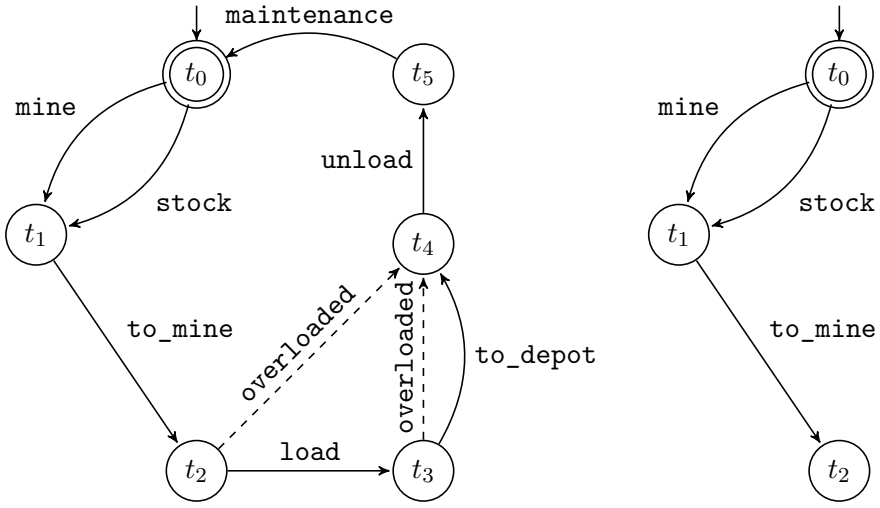
A transition system $\tilde{\mathcal{T}} = \langle \tilde{T}, \tilde{\mathcal{A}}_T^C \cup \tilde{\mathcal{A}}_T^U, \tilde{t_0}, \tilde{\varrho_T} \rangle$ is a *conditional-RTB* for a target $\mathcal{T} = \langle T, \mathcal{A}_T^C \cup \mathcal{A}_T^U, t_0, \varrho_T \rangle$ in system $\mathcal{S}$ with enacted system $\mathcal{E}_{\mathcal{S}} = \langle S, \mathcal{A}_{\mathcal{S}}^C \cup \mathcal{A}_{\mathcal{S}}^U, s_0, \delta \rangle$ iff $\tilde{\mathcal{T}} \preceq \mathcal{T}$ and $\langle \tilde{t_0}, s_0 \rangle \in \mathcal{C}$ where $\mathcal{C} \subseteq \tilde{T} \times S$ is the *conditional simulation relation* of $\tilde{\mathcal{T}}$ by $\mathcal{E}_{\mathcal{S}}$ such that $\langle \tilde{t}, s \rangle \in \mathcal{C}$ iff:

1. $\forall \tilde{t'}\ \forall a \in \tilde{\mathcal{A}_T^C}\ \exists k\ \forall s'(\langle \tilde{t}, a, \tilde{t'} \rangle \in \tilde{\varrho_T} \rightarrow \langle s, a, k, s' \rangle \in \delta)$ such that $\langle \tilde{t'}, s' \rangle \in \mathcal{C}$; and
2. $\forall \alpha \in \mathcal{A}_{\mathcal{S}}^U, \forall k(\langle s, \alpha, k, s' \rangle \in \delta \rightarrow \langle \tilde{t}, \alpha, \tilde{t'} \rangle \in \tilde{\varrho_T})$ such that $\langle \tilde{t'}, s' \rangle \in \mathcal{C}$.

The first condition (analogous to ND-simulation) requires all controllable actions of the RTB to be *feasible*. The second defines how uncontrollable exogenous events should be treated: since they are uncontrollable, their executions must be allowed in the target. If we want to prevent the occurrence of some exogenous event this can only be done by cutting some controllable action ahead of exogenous event's possible occurrence. This is related to the notion of *controllability* in discrete event systems (Wonham and Ramadge, 1987).
As usual, a conditional RTB is *supremal* iff it is not strictly simulated by any other conditional RTB. Consider our media room example (Figure 5.2), $\tilde{\mathcal{T}}_2$ is conditioned on `fuse` and prohibits `error`. Indeed, while realizing $\tilde{\mathcal{T}}_2$, it is guaranteed that `error` will never occur.

Example 20. *Figure 5.2 shows a conditional RTB for the media room example. Figure 5.3 shows instead a conditional and conformant RTBs for the mining example of Figure 5.3.* □

Computing conditional SRTBs When it comes to computing conditional-SRTBs, we modify the belief level construction to allow for exogenous events. Notice that exogenous events are considered to be observable in this case, so we can use their occurrence to refine the belief states in the belief-level full enacted system. This leads to the following definition.



Conditional RTB $\tilde{\mathcal{T}}_1$ Conformant RTB $\tilde{\mathcal{T}}_2$

Figure 5.5. A conditional RTB and a conformant RTB for the mining example of Figure 5.3

Definition 5.4. Given a belief level full enacted system $\mathcal{K}_{\langle \mathcal{S},\mathcal{T}\rangle} = \langle Q, \mathcal{A}^C_{\mathcal{F}} \cup \mathcal{A}^U_{\mathcal{F}}, q_0, \delta_K\rangle$ for full enacted system $\mathcal{F}_{\langle \mathcal{S},\mathcal{T}\rangle} = \langle F, \mathcal{A}^C_{\mathcal{F}} \cup \mathcal{A}^U_{\mathcal{F}}, f_0, \gamma\rangle$, the *conditional belief-level full enacted system* is a tuple $\mathcal{K}^{\mathcal{C}}_{\langle \mathcal{S},\mathcal{T}\rangle} = \langle Q^{\mathcal{C}}, \mathcal{A}^C_{\mathcal{F}} \cup \mathcal{A}^U_{\mathcal{F}}, q_0, \delta^{\mathcal{C}}_K\rangle$, where:

- $Q^{\mathcal{C}} = Q \setminus \{\langle S,t\rangle \mid \langle s,t\rangle \in \Delta_{\langle \mathcal{S},\mathcal{T}\rangle}, s \in S\}$; that is, prohibited exogenous events should never occur;
- $\delta^{\mathcal{C}}_K \subseteq Q \times \mathcal{A} \times Q$ is $\mathcal{K}^{\mathcal{C}}_{\langle \mathcal{S},\mathcal{T}\rangle}$'s transition relation where $\langle\langle S,t\rangle, a, \langle S',t'\rangle\rangle \in \delta^{\mathcal{C}}_K$ iff :
  - $a \in \mathcal{A}^C_{\mathcal{F}}$ and $\langle\langle S,t\rangle, a, \langle S',t'\rangle\rangle \in \delta_K$; that is, action a should be executable from all enacted states; and
  - $a \in \mathcal{A}^U_{\mathcal{F}}$ and $S' = \{s' \mid \langle\langle s,t\rangle, a, k, \langle s',t'\rangle\rangle \in \gamma, s \in S\}$; we revise belief state if an exogenous event occurs.

△

Next result shows that the conditional belief-level full enacted system is the SRTB in this context.

Theorem 30. Let $\mathcal{S}$ be an available system and $\mathcal{T}$ a target spec. Then, $\mathcal{K}^{\mathcal{C}}_{\langle \mathcal{S},\mathcal{T}\rangle}$ is the conditional-SRTB of $\mathcal{T}$ in $\mathcal{S}$.

The proof is similar to the one of Theorem 29, but now we exclude belief states that can do prohibited exogenous events (first item above), and we consider observation of exogenous events in refining belief states (second item).

Proof. We will prove that $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$ and the conditional SRTB $\mathcal{T}^*$ of $\mathcal{T}$ in $\mathcal{S}$ are simulation equivalent. The proof is similar to that of theorem 29 except here we consider exogenous events.

Proof for $\mathcal{T}^* \preceq \mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$: First, will show that all conditional RTB's are simulated by $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$. Let $\mathcal{T}' = \langle T', \mathcal{A}'^{C} \cup \mathcal{A}'^{U}, t'_0, \varrho'_T\rangle$ be a conditional RTB of $\mathcal{T}$ in $\mathcal{S}$. Assume $\mathcal{T}' \not\preceq \mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$, that is, $\mathcal{T}'$ is not simulated by $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$. Let $\tau_{\mathcal{T}'} = t'_0 \xrightarrow{a^1} \cdots \xrightarrow{a^n} t'_n$ be a trace of $\mathcal{T}'$ such that $\tau_{\mathcal{T}'}$ cannot be simulated state-wise by any trace of $\mathcal{K}_{\langle\mathcal{S},\mathcal{T}\rangle}$ and the simulation breaks at a state t'_{n-1}. We show that this is impossible since, we can build a legal trace of $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$ which can simulate the entire τ'. Note, now the traces can have both controllable actions and allowed exogenous events.

As $\mathcal{T}'$ is a RTB of $\mathcal{T}$ in $\mathcal{S}$, it holds that $\mathcal{T}' \preceq \mathcal{T}$ ($\mathcal{T}'$ is simulated by $\mathcal{T}$) and $\mathcal{T}' \preceq_{\mathcal{C}} \mathcal{E}_{\mathcal{S}}$ ($\mathcal{T}'$ has an exact solution in $\mathcal{S}$). Therefore, there exists a trace of $\mathcal{T}$ $\tau_{\mathcal{T}} = t_0 \xrightarrow{a^1} \cdots \xrightarrow{a^n} t_n$ such that $t'_i \preceq t_i$ for all $i \leq n$;

Let us define $\Gamma_{\mathcal{S}}$ as the maximal set of traces $s_0 \xrightarrow{a^1,k_1} \cdots \xrightarrow{a^\ell,k_\ell} s_\ell$, where $\ell \leq n$, of enacted system $\mathcal{E}_{\mathcal{S}}$ of $\mathcal{S}$, such that:

1. $t'_i \preceq_{\mathcal{C}} s_i, i \leq n$, i.e., which may be induced while realizing the RTB trace $\tau_{\mathcal{T}'} = t'_0 \xrightarrow{a^1} \cdots \xrightarrow{a^n} t'_n$;

2. for all traces $\tau_{\mathcal{S}} \in \Gamma_{\mathcal{S}}$ it is the case that $\mathsf{act\text{-}seq}(\tau_{\mathcal{S}}, \mathcal{A}^C) = \mathsf{act\text{-}seq}(\tau'[0,i], \mathcal{A}^C)$ for some $i \leq n$, where the function $\mathsf{act\text{-}seq}(\tau, Act)$ returns the action sequence of τ consisting only of actions included in Act. Formally,

$$\mathsf{act\text{-}seq}(s \xrightarrow{a} s', Act) = a \text{ if } a \in Act, \epsilon \text{ otherwise}$$

$$\mathsf{act\text{-}seq}(s_0 \xrightarrow{a^1} \cdots \xrightarrow{a^n} s_n, Act) = \mathsf{act\text{-}seq}(s_0 \xrightarrow{a^1} s_1, Act) \cdots \mathsf{act\text{-}seq}(s_{n-1} \xrightarrow{a^n} s_n, Act)$$

3. they do so through transitions labelled by a_i, k_i for $i \leq n$ such that for any two traces $\tau_1, \tau_2 \in \Gamma_{\mathcal{S}}$ it is the case that if $\tau_1[i] = \tau_2[i]$, then $\tau_1\langle i\rangle = \tau_2\langle i\rangle$ for controllable actions in τ_1 and τ_2. Since exogenous events are uncontrollable, we cannot put any restrictions on them.

Note, since exogenous events are uncontrollable $\Gamma_{\mathcal{S}}$ may include system traces where the exogenous event may not fire as per τ'. That is, for every exogenous event at location i of τ', there will be a system trace exactly of length i. Since, $\mathcal{T}'$ is realizable in $\mathcal{S}$ we know that at least one composition exists. Therefore, $\Gamma_{\mathcal{S}}$ will not be empty. Notice that, because of condition 2 above, there may be several such maximal sets. We nondeterministically take one.

Now, consider a trace $\tau_{\mathcal{K}} = q_0 \xrightarrow{a^1} \cdots \xrightarrow{a^n} q_n$ such that $q_i = \langle \mathsf{Pos}(\Gamma_{\mathcal{S}}, i), \tau_{\mathcal{T}}[i]\rangle$ for all $i \leq n$. The idea behind Pos is to return all states where the enacted system could be in. We show $\tau_{\mathcal{K}}$ is a *legal trace of* $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$, that is, it consists of legal states and transitions. We start by observing that:

- $\tau_{\mathcal{K}}[i] = \langle\{s_1, \ldots, s_\ell\}, t\rangle$, where $\{s_1, \ldots, s_\ell\} = \mathsf{Pos}(\Gamma_{\mathcal{S}}, i)$ and $t = \tau_{\mathcal{T}}[i]$, is a legal state of $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$ for all $i \leq n$;
- $\tau_{\mathcal{K}}[0]$ is the initial state of $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$.

Then we proceed by induction on n.

- For $n = 0$, we have that the trace $\tau_{\mathcal{K}}[0]$ consisting only of the initial state is trivially legal.
- By inductive hypothesis let us assume that $q_0 \xrightarrow{a^1} \cdots \xrightarrow{a^i} q_i$ (for $i < n$) is a legal trace of $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$, and we show that also $q_0 \xrightarrow{a^1} \cdots \xrightarrow{a^{i+1}} q_{i+1}$ is a legal trace of $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$.

  Consider the transition $q_i \xrightarrow{a^{i+1}} q_{i+1}$ of $\tau_{\mathcal{K}}$. Let $\mathsf{Pos}(\Gamma_{\mathcal{S}}, i) = \{s_1, \ldots, s_\ell\}$. Since τ' is realizable, there exists $s_j \xrightarrow{a^{i+1}, k^{i+1}_j} s'_j$ in $\mathcal{E}_{\mathcal{S}}$ for $j \leq \ell$ and $t_i \xrightarrow{a^{i+1}} t_{i+1}$ in $\mathcal{T}$. Hence, there exists exactly one set of indices (see definition of Γ_S, condition 2), $\mathsf{Indx} = \{\langle s_1 : k_1\rangle, \ldots, \langle s_\ell : k_\ell\rangle\}$, one per each element in $\mathsf{Pos}(\Gamma_{\mathcal{S}}, i)$, such that $\langle s, \tau_{\mathcal{T}}[i]\rangle \xrightarrow{a^{i+1}, k^{i+1}} \langle s', \tau_{\mathcal{T}}[i{+}1]\rangle$ in $\mathcal{F}_{\langle\mathcal{S},\mathcal{T}\rangle}$ where $s \in \mathsf{Pos}(\Gamma_{\mathcal{S}}, i)$, $s' \in \mathsf{Pos}(\Gamma_{\mathcal{S}}, i+1)$ and $\langle s : k^{i+1}\rangle \in \mathsf{Indx}$. That is, $q_i \xrightarrow{a^{i+1}} q_{i+1}$ in $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$.

So, RTB $\mathcal{T}'$ is simulated by $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$ (once we project out the indexes $\{1, \ldots, n\}$), that is, $\mathcal{T}' \preceq \mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$. From theorem 26 we know that union of two RTB's is an RTB, therefore $\mathcal{T}^*$ is also a RTB. Consequently, $\mathcal{T}^* \preceq \mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$.
To proof $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle} \preceq \mathcal{T}^*$, we simply observe that $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$ is an RTB, by construction, we have $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle} \preceq \mathcal{T}$ and $\mathcal{K}_{\langle\mathcal{S},\mathcal{T}\rangle} \preceq_{\mathcal{C}} \mathcal{E}_{\mathcal{S}}$. Hence $\mathcal{K}_{\langle\mathcal{S},\mathcal{T}\rangle}$ by theorem 26 $\mathcal{K}^{\mathcal{C}}_{\langle\mathcal{S},\mathcal{T}\rangle}$ is included in, and thus simulated by, $\mathcal{T}^*$. ■

5.3.2 Conformant SRTBs

Conformant solutions guarantee realizability in absence of any observation over exogenous events. For example, the conformant solution $\tilde{\mathcal{T}}_1$ in Figure 5.2 contains a very restricted subset of the target as, if the bulb is fused then the projector cannot be operated again without a `repair`. Solutions of such type are stricter, promising execution irrespective of which uncontrollable events occurs. This provides robustness in modelling as one can still prevent unacceptable conditions under non-observability at runtime. We say a RTB to be conformant if it does not include any exogenous event, that is, $\mathcal{A}^U_T = \varnothing$. Note, the target specification (problem input) is allowed to have exogenous events, however, a conformant RTB must have compiled them away. More precisely, a transition system $\tilde{\mathcal{T}} = \langle\tilde{T}, \tilde{\mathcal{A}}^C_T, \tilde{t_0}, \tilde{\varrho_T}\rangle$ is a *conformant-RTB* for a target $\mathcal{T} = \langle T, \mathcal{A}^C_T \cup \mathcal{A}^U_T, t_0, \varrho_T\rangle$ in system $\mathcal{S}$ with enacted system $\mathcal{E}_{\mathcal{S}} = \langle S, \mathcal{A}^C_{\mathcal{S}} \cup \mathcal{A}^U_{\mathcal{S}}, s_0, \delta\rangle$ iff $\tilde{\mathcal{T}} \preceq \mathcal{T}$ and $\langle\tilde{t_0}, s_0\rangle \in \mathcal{Z}$ where $\mathcal{Z} \subseteq \tilde{T} \times S$ is the *conformant simulation relation* of $\tilde{\mathcal{T}}$ by $\mathcal{E}_{\mathcal{S}}$ such that $\langle\tilde{t}, s\rangle \in \mathcal{Z}$ iff:

1. $\forall\tilde{t}\ \forall a\ \exists k \forall s'(\langle\tilde{t}, a, \tilde{t'}\rangle \in \tilde{\varrho_T} \rightarrow \langle s, a, k, s'\rangle \in \delta)$ such that $\langle\tilde{t'}, s'\rangle \in \mathcal{Z}$;

2. $\forall \alpha \in \mathcal{A}^U_{\mathcal{S}},\ \forall k(\langle s, \alpha, k, s' \rangle \in \delta \rightarrow \langle \tilde{t}, s' \rangle \in \mathcal{Z})$; and
3. $\forall \alpha \in \mathcal{A}^U_{\mathcal{S}}, \forall k(\langle s, \alpha, k, s' \rangle \in \delta \wedge \langle \tilde{t}, t \rangle \in \preceq \rightarrow \langle t, \alpha, t' \rangle \in \varrho_T)$ such that $\langle \tilde{t}, t' \rangle \in \preceq$.

The first condition is analogous to the usual ND-simulation one. The second condition requires occurring of exogenous events should retain the simulation relation. The third condition enforces only permitted exogenous events to ever occur in the system. As usual, a conformant RTB is *supremal* iff it is not strictly simulated by any other conformant RTB.

Computing conformant SRTBs Conformant solutions require realizability guarantee irrespective of any nondeterministic or exogenous evolution. In order to include them in the belief-level system we first define what we call as the ε–closure of a state. That is, where all could the system be as a result of an exogenous event from that state. Formally, given a full enacted system $\mathcal{F}_{\langle \mathcal{S}, \mathcal{T} \rangle} = \langle F, \mathcal{A}^C_{\mathcal{F}} \cup \mathcal{A}^U_{\mathcal{F}}, f_0, \gamma \rangle$ and a state $f \in \mathcal{F}$, the *ε–closure* of f, denoted by $\varepsilon(f)$, is defined recursively as follows:

1. $f \in \varepsilon(f)$, that is, the state itself is in the closure;
2. $\forall \alpha \in \mathcal{A}^U_{\mathcal{F}},\ \forall f \in \varepsilon(f)\ (f \xrightarrow{\alpha, k} f' \in \gamma \rightarrow f' \in \varepsilon(f))$, that is, all exogenous event reachable states are included; and
3. Nothing else except for 1 and 2 should be in $\varepsilon(f)$.

Next, we re-define the belief level full enacted system to accommodate exogenous events. Here, we consider the ε–closure in both the initial state and the transition relation.

Definition 5.5. Given a full enacted system $\mathcal{F}_{\langle \mathcal{S}, \mathcal{T} \rangle} = \langle F, \mathcal{A}^C_{\mathcal{F}} \cup \mathcal{A}^U_{\mathcal{F}}, f_0, \gamma \rangle$ for a target $\mathcal{T} = \langle T, \mathcal{A}_T, t_0, \varrho_T \rangle$ and a system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n \rangle$, the *conformant belief-level full enacted system* is a tuple $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S}, \mathcal{T} \rangle} = \langle Q, \mathcal{A}^C_{\mathcal{F}}, q_0, \delta_K \rangle$, where:

- $Q = 2^{(B_1 \times \cdots \times B_n \times T)} \setminus \{S \mid s \in \Delta_{\langle \mathcal{S}, \mathcal{T} \rangle}, s \in S\}$;
- $q_0 = \varepsilon(f_0)$ is $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$'s initial state;
- $\delta_K \subseteq Q \times \mathcal{A} \times Q$, where $\langle S, a, S' \rangle \in \delta_K$ iff :
  - there exists a set $\mathsf{Indx} = \{\langle s_1 : k_1 \rangle, \ldots, \langle s_\ell : k_\ell \rangle\}$ such that $\{s_1, \ldots, s_\ell\} = S$; $s_i \xrightarrow{a, k_i} s'_i$ in $\mathcal{F}_{\langle \mathcal{S}, \mathcal{T} \rangle}$ for all $i \leq \ell$; and for all $i, j \leq \ell$ if $\mathsf{tgt}(s_i) = \mathsf{tgt}(s_j)$, then $\mathsf{tgt}(s'_i) = \mathsf{tgt}(s'_j)$; and
  - $S' = \bigcup_{\langle s:i \rangle \in \mathsf{Indx}} \{\varepsilon(s') | \langle s, a, i, s' \rangle \in \gamma\}$, that is, S' should contain the ε–closure of all successors of enacted system states in S resulting from action a.

△

Note, the belief level full enacted system is now exponential also on the Target states. Observe, if the target specification allows all exogenous events at any point then the complexity in regards to the target will no longer be exponential.

Theorem 31. Let $\mathcal{S}$ be an available system and $\mathcal{T}$ a target spec. Then, $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$ is the conformant-SRTB of $\mathcal{T}$ in $\mathcal{S}$.

Similar to proof sketch of theorem 29, we can construct a trace $\tau_{\mathcal{K}}$ of $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$ which simulates a trace τ' of a RTB $\mathcal{T}'$, contained in SRTB $\mathcal{T}^*$, of $\mathcal{T}$ in $\mathcal{S}$. Though, now the induced system traces, due to exogenous events, may be of variable length. So, in each state of $\tau_{\mathcal{K}}$ we include ε–closure of the states having exogenous transitions.

Proof. We will prove that $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$ and the SRTB $\mathcal{T}^*$ of $\mathcal{T}$ in $\mathcal{S}$ are simulation equivalent.
Proof for $\mathcal{T}^* \preceq \mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$: First, will show that all RTB's are simulated by $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$. Let $\mathcal{T}' = \langle T', \mathcal{A}', t'_0, \varrho'_T\rangle$ be a RTB of $\mathcal{T}$ in $\mathcal{S}$. Assume $\mathcal{T}' \not\preceq \mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$, that is, $\mathcal{T}'$ is not simulated by $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$. Let $\tau_{\mathcal{T}'} = t'_0 \xrightarrow{a^1} \cdots \xrightarrow{a^n} t'_n$ be a trace of $\mathcal{T}'$ such that $\tau_{\mathcal{T}'}$ cannot be simulated state-wise by any trace of $\mathcal{K}^{X}_{\langle \mathcal{S},\mathcal{T}\rangle}$ and the simulation breaks at a state t'_{n-1}. We show that this is impossible since, we can build a legal trace of $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$ which can simulate the entire τ'.
As $\mathcal{T}'$ is a RTB of $\mathcal{T}$ in $\mathcal{S}$, it holds that $\mathcal{T}' \preceq \mathcal{T}$ ($\mathcal{T}'$ is simulated by $\mathcal{T}$) and $\mathcal{T}' \preceq_{\mathcal{Z}} \mathcal{E}_{\mathcal{S}}$ ($\mathcal{T}'$ has an exact composition in $\mathcal{S}$). Note, this time since $\mathcal{T}'$ is a conformant RTB, it may be simulated by more than one trace of $\mathcal{T}$. Therefore, there exists a set of traces of $\mathcal{T}$ such that $\tau = t_0 \xrightarrow{a^1} \cdots \xrightarrow{a^\ell} t_\ell \in \Gamma_{\mathcal{T}}$, where $\ell \geq n$ iff:

1. $\mathsf{act\text{-}seq}(\tau'_{\mathcal{T}}, \mathcal{A}^C) = \mathsf{act\text{-}seq}(\tau_{\mathcal{T}}, \mathcal{A}^C)$, the sequence of controllable actions is same; and

2. if $t'_i \preceq t_j$, where $i \leq j, i \leq n, j \leq \ell$, then either $t'_i \preceq t_{j+1}$ or $t'_{i+1} \preceq t_{j+1}$; the simulation relation is maintained across exogenous events in the target spec.

Let us define $\Gamma_{\mathcal{S}}$ as the maximal set of traces $\tau_{\mathcal{S}} = s_0 \xrightarrow{a^1_{\mathcal{S}},k_1} \cdots \xrightarrow{a^m_{\mathcal{S}},k_m} s_m$, where $m \geq n$, of enacted system $\mathcal{E}_{\mathcal{S}}$ of $\mathcal{S}$, such that:

1. if $t'_i \preceq_{\mathcal{Z}} s_j$, where $i \leq j, i \leq n, j \leq m$, then either $t'_i \preceq_{\mathcal{Z}} s_{j+1}$ or $t'_{i+1} \preceq_{\mathcal{Z}} s_{j+1}$;

2. $\mathsf{act\text{-}seq}(\tau'_{\mathcal{T}}, \mathcal{A}^C) = \mathsf{act\text{-}seq}(\tau_{\mathcal{S}}, \mathcal{A}^C)$, the x-enacted system traces can copy the RTB trace τ';

3. they do so through transitions labelled by a_i, k_i for $i \leq n$ such that for any two traces $\tau_1, \tau_2 \in \Gamma_{\mathcal{S}}$ it is the case that if $\tau_1[i] = \tau_2[i]$, then $\tau_1\langle i\rangle = \tau_2\langle i\rangle$.

Note, since only allowed exogenous events occur, the induced system traces will correspond to the target spec traces in $\Gamma_{\mathcal{T}}$. Since, $\mathcal{T}'$ is realizable in $\mathcal{S}$ we know that at least one composition exists. Therefore, $\Gamma_{\mathcal{S}}$ will not be empty. Notice that, because of condition 3 above, there may be several such maximal sets. We nondeterministically take one.
We observe that due to exogenous events the enacted system traces may be longer in length than the target trace (due to exogenous events). Therefore, given a state

$\tau[i]$ of trace τ let $\varepsilon(\tau, i)$ be the set of states reachable from $\tau[i]$ by zero or more exogenous events in τ. Formally,

$$\varepsilon(\tau, i) = \{s \mid \tau[i] \xrightarrow{\alpha_{i+1}} \cdots \xrightarrow{\alpha_{i+\ell}} s, \alpha_{i+j} \in Act^X, 0 \leq j \leq \ell\}.$$

Hence, given an action sequence $\vec{a} = a_1 \ldots a_n$ and a trace τ_1, let $\tau_1^{\vec{a}}$ denote the shortest prefix of τ_1 such that $\mathsf{act\text{-}seq}(\tau_1^{\vec{a}}, \mathcal{A}^C) = \vec{a}$.

Now, consider a trace $\tau_{\mathcal{K}} = q_0 \xrightarrow{a^1} \cdots \xrightarrow{a^n} q_n$ such that $q_i = \langle \mathsf{Pos}^{\mathcal{Z}}(\Gamma_{\mathcal{S}}, \Gamma_{\mathcal{T}}, i) \rangle$ for all $i \leq n$ where:

$$\mathsf{Pos}^{Z}(\Gamma_{\mathcal{S}}, \Gamma_{\mathcal{T}}, i) = \\ \bigcup_{\tau_1 \in \Gamma_{\mathcal{F}}} \{\varepsilon(\tau_1, j) \mid j = |\tau_1^{\vec{a}}|, \vec{a} = \mathsf{act\text{-}seq}(\tau'[0, i], \mathcal{A}^C)\}$$

where,

$$\Gamma_{\mathcal{F}} = \{\langle s, t \rangle \xrightarrow{a^1,k_1} \cdots \xrightarrow{a^m,k_m} \langle s', t' \rangle \mid \\ s \xrightarrow{a^1,k_1} \cdots \xrightarrow{a^m,k_m} s' \in \Gamma_{\mathcal{S}}, t \xrightarrow{a^1} \cdots \xrightarrow{a^m} t' \in \Gamma_{\mathcal{T}}\}.$$

Observe, since the system evolutions have to match the original target specification, $\Gamma_{\mathcal{F}}$ is well defined. The idea behind Pos^Z is to return all states where the enacted system could be in either due to nondeterminism or exogenous events, after realizing a sequence of domain actions. We show $\tau_{\mathcal{K}}$ is a *legal trace of* $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$, that is, it consists of legal states and transitions. We start by observing that:

- $\tau_{\mathcal{K}}[i] = \langle \{s_1, \ldots, s_\ell\} \rangle$, where $\{s_1, \ldots, s_\ell\} = \mathsf{Pos}^{\mathcal{Z}}(\Gamma_{\mathcal{S}}, \Gamma_{\mathcal{T}}, i)$, is a legal state of $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$ for all $i \leq n$;
- $\tau_{\mathcal{K}}[0]$ is the initial state of $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$.

Then we proceed by induction on n.

- For $n = 0$, we have that the trace $\tau_{\mathcal{K}}[0]$ consisting only of the initial state is trivially legal.
- By inductive hypothesis let us assume that $q_0 \xrightarrow{a^1} \cdots \xrightarrow{a^i} q_i$ (for $i < n$) is a legal trace of $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$, and we show that also $q_0 \xrightarrow{a^1} \cdots \xrightarrow{a^{i+1}} q_{i+1}$ is a legal trace of $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$.

  Consider the transition $q_i \xrightarrow{a^{i+1}} q_{i+1}$ of $\tau_{\mathcal{K}}$. Let $\mathsf{Pos}^{\mathcal{Z}}(\Gamma_{\mathcal{S}}, \Gamma_{\mathcal{T}}, i) = \{s_1, \ldots, s_\ell\}$. Since τ' is realizable, there exists $s_j \xrightarrow{a^{p+1},k_j^{p+1}} s'_j$ in $\mathcal{E}_{\mathcal{S}}$ for $j \leq \ell, p \geq i$ and $t_i \xrightarrow{a^{p+1}} t_{p+1}$ in $\mathcal{T}$. Hence, there exists exactly one set of indices (see definition of Γ_S, condition 2), $\mathsf{Indx} = \{\langle s_1 : k_1 \rangle, \ldots, \langle s_\ell : k_\ell \rangle\}$, one per each element in $\mathsf{Pos}^{\mathcal{Z}}(\Gamma_{\mathcal{S}}, \Gamma_{\mathcal{T}}, i)$, such that $\langle s \rangle \xrightarrow{a^{p+1},k^{p+1}} \langle s' \rangle$ in $\mathcal{F}_{\langle \mathcal{S}, \mathcal{T} \rangle}$ where $s \in \mathsf{Pos}^X(\Gamma_{\mathcal{S}}, \Gamma_{\mathcal{T}}, i)$, $s' \in \mathsf{Pos}^X(\Gamma_{\mathcal{S}}, \Gamma_{\mathcal{T}}, i+1)$ and $\langle s : k^{p+1} \rangle \in \mathsf{Indx}$. Note, we consider ε–closure when evolving to successor belief state, in align with the definition of $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$. That is, $q_i \xrightarrow{a^{i+1}} q_{i+1}$ in $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S}, \mathcal{T} \rangle}$.

Note that by construction of $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$, the last condition of the conformant simulation definition is automatically satisfied.
So, RTB $\mathcal{T}'$ is simulated by $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$ (once we project out the indexes $\{1,\ldots,n\}$), that is, $\mathcal{T}' \preceq \mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$. From theorem 26 we know that union of two RTB's is an RTB, therefore $\mathcal{T}^*$ is also a RTB. Consequently, $\mathcal{T}^* \preceq \mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$.
To prove $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle} \preceq \mathcal{T}^*$, we simply observe that $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$ is an RTB, since by construction, we have $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle} \preceq \mathcal{T}$ and $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle} \preceq_{\mathcal{Z}} \mathcal{E}_{\mathcal{S}}$. Hence $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$ by theorem 26 $\mathcal{K}^{\mathcal{Z}}_{\langle \mathcal{S},\mathcal{T}\rangle}$ is included in, and thus simulated by, $\mathcal{T}^*$. ■

5.4 Discussion

We proved that *every* classical behavior composition problem instance has an optimal supremal solution (Theorem 27) and that such supremal can be effectively built using cross-product between transition systems and belief-level state construction operations combined (Theorem 29). What is more, borrowing notions from discrete-event systems and reasoning about action, we showed how to accommodate *exogenous uncontrollable events* to obtain a more expressive composition framework (Section 5.3). We demonstrated that the definitions and techniques for supremal fragments can be adapted to this new framework (Theorems 30 and 31).
Many issues remain to be investigated.First, we conjecture that our technique builds SRTBs that are optimal wrt worst-case complexity. This implies that synthesis of supremals for the general nondeterministic case is strictly harder than computing exact composition controllers or supremals for deterministic settings. Confirming this conjecture is our next step.
Second, our approach to realizing a target specification to the "best" possible is developed within a *strict* uncertainty context. This contrasts with the decision-theoretic approach of (Yadav and Sardina, 2011), where they proposed to optimize the expected reward of controllers. It would be interesting to adopt such a quantitative framework focusing on targets and devising a suitable decision-theoretic notion of SRTBs.
Finally, one may devise approaches that trade optimality for faster computation, such as restricting realizable target fragments to merely removing transitions from the original target specification, bounding its number of states, or computing it in anytime fashion.

Chapter 6

Generalized Agent Protocols for LTL

6.1 Generalized Planning for LTL

6.1.1 Planning in AI

Automated Planning (Ghallab et al., 2004) is the branch of AI that focuses on the deliberation process of building plans, i.e., an organized set of actions, in order to fulfill some objectives. Typically, these plans are executed by intelligent agents, and the solution amounts to synthesize agent's plans satisfying a goal specification, usually expressing a reachability requirement. In Section 3.2.3 sophisticated forms of planning for agent planning programs were briefly introduced (De Giacomo et al., 2010b): programs built only from achievement and maintenance goals, which merge two traditions in AI research, namely, Automated Planning and Agent Oriented Programming.
There exist indeed several forms of planning. For what follows, we need to distinguish between *classical planning* domains, *conditional planning* and *conformant planning*. Classical planning domains are fully observable, static, and deterministic, in which plans can be computed in advance and then applied unconditionally. Considering a planning domain as described by a finite-state automaton $A = \langle Act, S, s_0, \delta, F \rangle$, every input word in Act^* induces a run on A, which is accepted whether it terminates in a state in F. Hence, we can see a planning problem as to find a word leading from s_0 to a final state, and a *plan* as a simple sequence of actions in Act^*. Conditional (or contingency) planning deals instead with bounded indeterminacy by constructing a conditional plan with different branches for the different contingencies that may arise, and even though plans are precomputed, the agent finds out which part of the plan to execute by including sensing actions in the plan to test for the appropriate conditions. Therefore, conditional plans can be thought of as *tree-like* structures, in contrast with sequential plans that are instead action sequences. Finally, conformant planning aims to construct standard, sequential plans that are to be executed, in partially-observable settings, without perception. Namely, they are required to achieve the goal in all possible circumstances, regardless of the true initial state and the actual action outcomes.

As customary, we use nondeterministic state-transition systems as a *conceptual model*, i.e., a simple theoretical device for describing a dynamic system. Although such models may significantly depart from the computational concerns and algorithmic approaches for solving that problem, they allow to analyze requirements and assumptions as well as proving semantic properties.

6.1.2 Generalized Planning in AI

Informally, the problem of generalized planning is to find plans that can solve a set of problem instances. The problem of developing *generalized* plans which apply to classes of "similar" problem instances was identified almost immediately after the development of the STRIPS framework (Fikes and Nilsson, 1971) and the fundamental motivations behind it stem from classical planning itself and it has recently been drawing increasing attention in the AI community (Levesque, 2005; Srivastava et al., 2008; Bonet et al., 2009).
Indeed, while the vast majority of the work in AI planning today deals with sequential planning, thus generating a sequence of actions to achieve a goal, a smaller community is concerned with conditional planning where plans can be tree-like structures, and an even smaller community is concerned with iterative planning, where plans can be represented as finite-state structures *with loops*, i.e., a form of finite-state controllers. Informally, one of such controllers can be seen as a control structure with internal states (i.e. a *bounded* amount of memory) that is able to capture a class of plans that is more sophisticated than sequential and conditional plans. It is evident that such structures with loops are able to solve a class of iterative problems in which an unbounded number of object is processed. In (Levesque, 2005) authors identify the One-Dimensional planning problems as the class of problems that has a complete procedure to reason about the correctness of finite-state solutions.

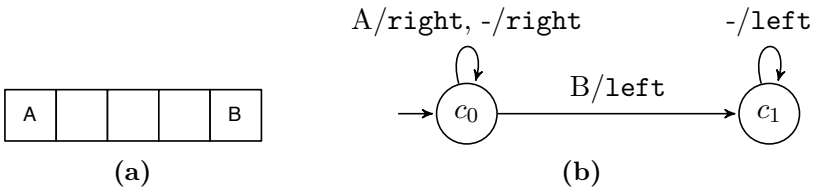


Figure 6.1. (a) A conditional problem where an agent initially is in one of the two leftmost positions has to get to B and then back to A. These two cell-marks are observable. (b) A 2-state controller that solves this problem and many variations.

As an example, consider the simple contingent planning problem as in (Bonet et al., 2009) –see Figure 6.1a– where an agent initially in one of the two extreme positions is assigned the (reachability) goal to get to B and then back to A. The solution is then a conditional plan for this specific problem instance. However, as we increase the number of cells in this problem, the complexity of solving it increases (even exponentially, e.g. in a blocks world), although the solutions address common subproblems and are remarkably similar to each other.

More generally, generalized planning can be though as the problem of finding a single plan works for a class of different environments that sharing a common interface or characteristics. Approaches for finding generalized plans thus attempt to extract, and subsequently make use of such common solutions and problem structures. The obvious advantage is that if a generalized solution is found for a problem class, then solving any particular instance in the class only requires the execution of the generalized plan, which is extremely efficient since no search is needed in the execution.
This raises the question whether one can find a general definition of generalized planning that is independent of any specific representation.

6.1.3 Planning for LTL

Up to now we considered the classical planning setting in which the goal condition captures a reachability requirement. The specification is thus met as soon as a state satisfying the condition is reached in the current finite run.
Instead, planning for *long-running goals* (or extended goals) asks the question of generating a plan that will enable the satisfaction of infinitely many goal conditions over infinite runs. Indeed, as we deal with LTL goals, plans can rely on *perfect recall*, i.e., be inherently infinite. Nonetheless, since we restrict to dynamic systems with finite number of states, we can employ effective planning techniques and show that these plans can be represented finitely. As a consequence observe that, differently from the classical planning, even sequential plans may now in fact involve finite-state controllers with loops.
An automata-based approach to planning for full-fledged LTL goals covering partial information was put forward in (De Giacomo and Vardi, 1999) in which authors presented an approach based on non-emptiness of Büchi-automata on infinite words. An assumption made in (De Giacomo and Vardi, 1999) is that the agent is interacting with a single deterministic environment which is only partially observable.
In Sections 6.1.3, 6.1.3 and 6.1.3 we briefly report the results of (De Giacomo and Vardi, 1999), as they are the base for the further development of a solution for generalized planning for LTL goals, that will be presented in Section 6.1.4. First, we define general notions that will be used throughout the remainder of the chapter.

Definition 6.1 (Dynamic System)**.** We model the *dynamic system* of interest as a transition system $\mathcal{D} = (D, D_0, Act, R, Obs, \pi)$ where:

- D is the finite set of possible states.
- $D_0 \subseteq D$ is the set of initial states.
- Act is the finite set of possible actions.
- $\delta : D \times Act \to D$ is the transition function that, given a state and an action, returns the successor states.
- Obs is the set of possible observations.
- $\pi : D \to Obs$ is the observation function that, given any system state, returns the corresponding observation.

△

A *plan* σ for $\mathcal{D}$ is an infinite sequence of actions $a_0, a_1, a_2, \ldots \in Act^\omega$. The *execution* of σ (starting from the initial state d_0) is the infinite sequence of states $d_0, d_1, d_2, \ldots \in D^\omega$ s.t. $d_0 \in D_0$ and $d_{i+1} = \delta(d_i, a_i)$. The *trace*, $\tau(\sigma, d_0)$, of σ (starting from the initial state d_0) is the infinite sequence $\pi(d_0), \pi(d_1), \pi(d_2), \ldots$.
Let $\mathcal{G}$ be a goal specification, i.e., a Büchi automaton specifying the (infinite) traces of the desired executions of the system. Namely, a plan σ *realizes a specification* $\mathcal{G}$ iff $\tau(\sigma, d_0) \in L(\mathcal{G})$. In fact, while any LTL formula can be translated into a Büchi automaton, the converse is not true. These results hold for any goal specified as a Büchi automaton, though for ease of exposition we give them as LTL.

Definition 6.2 (Specification)**.** Formally, $\mathcal{G} = (G, G_0, Obs, \rho, G^{acc})$ where:

- Obs plays the role of the alphabet of the automaton.
- G is the finite set of possible states of the automaton.
- $G_0 \subseteq S$ is the set of possible initial states.
- $\gamma : G \times Obs \to 2^G$ is the transition function of the automaton (the automaton need not to be deterministic).
- $G^{acc} \subseteq G$ is the set of accepting states.

△

Planning with complete information

Let us consider first a simplified scenario of classical planning, in which we have *complete information* on initial situation and that we have *full observability* on the state. The only kind of plans of interest in this case are sequential ones (sequences of actions), since a conditional plan exists iff a sequential plan does.
Observe that, as a consequence of the above mentioned assumptions, we are considering a system domain $\mathcal{D}$ such that:

1. D_0 is a singleton, as we are assuming complete information;
2. $Obs = D$ since we are assuming full observability;
3. $\pi : D \to Obs$ is the identity function.

To synthesize such a plan, we check for nonemptiness the following Büchi automaton $\mathcal{A}_\mathcal{D} = (W, W_0, Act, \rho, W^{acc})$ where:

- Act is the alphabet of the automaton
- $W = G \times D$
- $W_0 = G_0 \times D_0$
- $(g_j, d_j) \in \rho((g_i, d_i), a)$ iff $d_j = \delta(d_i, a)$ and $g_j \in \gamma(g_i, \pi(w_i))$
- $W^{acc} = G^{acc} \times D$

For $\mathcal{A}_\mathcal{D}$ we get the following result:

Theorem 32. (De Giacomo and Vardi, 1999) A plan σ for $\mathcal{D}$ realizing $\mathcal{G}$ exists iff $L(\mathcal{A}_{\mathcal{D}}) \neq \varnothing$.

Notably the nonemptiness algorithm can be easily modified to return a plan if a plan exists. The plan returned always consists of two parts: a sequence arriving to a certain state, and a second sequence that forms a cycle back into that state. Thus, such plans can be captured with finite controllers. As an immediate consequence of the construction we get that

Theorem 33. (De Giacomo and Vardi, 1999) Planning in the setting above is decidable in NLOGSPACE.

This result can be easily proved by noting that the automaton $\mathcal{A}_{\mathcal{D}}$ can be built on the fly, thus for checking nonemptiness using a nondeterministic algorithm we only need $O(\log(|W|) + \log(|S|))$ bits.
Observe that if we adopt a compact (i.e., logarithmic) representation of the transition system, for example by using propositions to denote states and computing the transitions directly on such propositionsthen planning in the above setting becomes PSPACE. This is the complexity of planning in STRIPS (Bylander, 1991), which can be seen as a special case of the setting considered here – reachability of a desired state of affairs is the only kind of goal considered in STRIPS; moreover, only certain transition systems are (compactly) representable.
Moreover considering that STRIPS is PSPACE-hard (Bylander, 1991), we can conclude that planning in the setting above is NLOGSPACE-complete (PSPACE-complete wrt a compact representation of $\mathcal{D}$).

Conformant Planning with incomplete information

Next we consider the more general case. We assume to have only partial information on the initial situation, and we assume that only part of the state is observable. In this section we consider generating sequential plans, in the next section we turn to conditional plans.
We model the dynamic system of interest as a general transition system $\mathcal{D} = (D, D_0, Act, R, Obs, \pi)$ as in the general definition. However, by assuming both partial information and observability, the three simplifications of before hold no more. Hence $\mathcal{D}$ has several initial states $D_0 = \{d_{00}, \ldots, d_{0n-1}\}$, for $n > 1$, reflecting the uncertainty about the initial situation.
As in the previous section we specify the behavior of the desired executions of the system by a Büchi automaton $\mathcal{G}$. This time, however, the construction of the product automaton $\mathcal{A}_{\mathcal{D}}$ is slightly more involved. We first build the *generalized* Büchi automaton $\mathcal{A}_{\mathcal{D}} = (W, W_0, Act, \rho, W^{acc})$ where:

- $W = G^n \times D^n$
- $W_0 = G_0^n \times \{(d_{00}, \ldots, d_{0n-1})\}$
- $(\vec{g}', \vec{d}') \in \rho((\vec{g}, \vec{d}), a)$ iff $d'_h = \delta(d_h, a)$ and $g'_h \in \gamma(g_h, \pi(d_h))$ for $h = 0, \ldots, n-1$.
- $W^{acc} = \{G^{acc} \times G^{n-1} \times D^n, \ldots, G^{n-1} \times G^{acc} \times D^n\}$

From such an automaton we obtain, by employing the "counting construction", an equivalent Büchi automaton $\mathcal{A}^b_{\mathcal{D}} = (W^b, W^b_0, Act, \rho^b, F^b)$ where:

- $W^b = W^n \times D^n \times \{0, \ldots, n-1\}$
- $W^b_0 = W^n_0 \times \{(d_{00}, \ldots, d_{0n-1})\} \times \{0\}$
- $(\vec{g}', \vec{d}', \ell') \in \rho^b((\vec{g}, \vec{d}, \ell), a)$ iff $d'_h = \delta(d_h, a)$ and $g'_h \in \gamma(g_h, \pi(d_h))$ for $h = 0, \ldots, n-1$, and $\ell' = (\ell+1) \bmod n$ if $g_\ell \in G^{acc}$ and $\ell' = \ell$ otherwise.
- $F^b = G^{acc} \times G^{n-1} \times D^n \times \{0\}$

Theorem 34. (De Giacomo and Vardi, 1999) A plan σ for $\mathcal{D}$ realizing the specification $\mathcal{G}$ exists iff $L(\mathcal{A}_{\mathcal{D}}) = L(\mathcal{A}^b_{\mathcal{D}}) \neq \varnothing$.

Again the nonemptiness algorithm can be easily modified to return a plan if a plan exists. Again, due to the Büchi acceptance condition, this plan has a *lazo* structure, and it can be captured as a finite-state controllers.
Building the automaton $\mathcal{A}_{\mathcal{D}}$ on the fly, we can check nonemptiness with a nondeterministic algorithm needing $O(n \cdot \log(|D|) + \log(|S|))$ bits, where n is bounded by the size of $|D|$. Considering that NPSPACE=PSPACE, we get:

Theorem 35. (De Giacomo and Vardi, 1999) Planning in the setting above is decidable in PSPACE.

If we adopt a compact representation of the transition system, then planning in the above setting becomes EXPSPACE. Moreover, the setting itself is proven (De Giacomo and Vardi, 1999) to be EXPSPACE-complete, hence the previous upper bound is tight.
Note that plan existence in STRIPS with incomplete information on the initial situation is PSPACE-complete (Bäckström, 1992) – polynomial reduction to the case where the initial situation is completely known. This means that we do pay a price this time in generalizing the setting wrt more traditional approaches.

Conditional planning with incomplete information

Now we turn to synthesis of conditional plans. A *vector plan* $\vec{\sigma}$ is an infinite sequence of *action vectors* $\vec{a}_0, \vec{a}_1, \vec{a}_2, \ldots \in (Act^n)^\omega$. The *execution*, $ex_h(\vec{\sigma}, d_{0h})$ of its h-component (starting from the initial state d_{0h}) is the infinite sequence of states $d_{0h}, d_{1h}, d_{2h}, \ldots \in D^\omega$ s.t. $d_{0h} \in D_0$ and $d_{i+1h} = \rho(d_{ih}, a_{ih})$. The *trace*, $\tau_h(\vec{\sigma}, d_{0h})$, of its h-component is the infinite sequence $\pi(d_{0h}), \pi(d_{1h}), \pi(d_{2h}), \ldots$. A vector plan $\vec{\sigma}$ *realizes* a specification $\mathcal{A}$ iff $\tau_h(\vec{\sigma}, d_{0h}) \in L(\mathcal{A})$ for $h = 0, \ldots n-1$.
A vector plan is not a conditional plan yet, it is simply the parallel compositions of n sequential plans, one for each initial state. *Conditional plans* are vector plans whose actions agree on executions with the same observations.
To formally define conditional plans, we introduce the following notion of equivalence on finite traces. Let $d_{0l}, \ldots, d_{nl}$ and $d_{0m}, \ldots, d_{nm}$ be two finite traces, then

$$\langle d_{0l}, \ldots, d_{nl} \rangle = \langle d_{0m}, \ldots, d_{nm} \rangle \text{ iff } \langle \pi(d_{0l}), \ldots, \pi(d_{nl}) \rangle = \langle \pi(d_{0m}), \ldots, \pi(d_{nm}) \rangle$$

In other words, two finite traces are said to be equivalent (and indistinguishable) iff they produce the same sequence of observations.
A *conditional plan* $\vec{\sigma}$ is a vector plan such that given the executions $d_{0l}, d_{1l}, d_{2l}, \ldots$ and $d_{0m}, d_{1m}, d_{2m}, \ldots$ of a pair of components l and m, we have that $a_{nl} = a_{nm}$ whenever $\langle d_{0l}, \ldots, d_{nl}\rangle \sim \langle d_{0m}, \ldots, d_{nm}\rangle$.
Intuitively a conditional plan can be though of as composed by an (infinite) sequence of *case* instructions that at each step on the base of the observations select how to proceed. However, these case conditions are "packed", i.e., hidden in the sequence of action vectors. The task of representing such plans in an explicit tree-like structure is not addressed here.
How can we synthesize a conditional plan? We follow the line of the construction in the previous section, checking nonemptiness of a Büchi automaton which this time has Act^n as alphabet. Specifically, we build the generalized Büchi automaton $\mathcal{A}_{\mathcal{D}} = (W, W_0, Act^n, \rho, W^{acc})$ where:

- $W = G^n \times D^n \times \mathcal{E}_n$, where $\mathcal{E}_n$ is the set of equivalence relations on the set $\{0, \ldots, n-1\}$,
- $W_0 = G_0^n \times \{(d_{00}, \ldots, d_{0n-1})\} \times \equiv_0$, where $i \equiv_0 j$ iff $\pi(d_{0i}) = \pi(d_{0j})$;
- $(\vec{s_j}, \vec{w_j}, \equiv') \in \rho((\vec{g_i}, \vec{d_i}), \vec{a}, \equiv)$ iff
  - $d_{jh} = \delta(d_{ih}, a_h)$ and $g_{jh} \in \gamma(g_{ih}, \pi(d_{ih}))$
  - if $l \equiv m$ then $a_l = a_m$
  - $l \equiv' m$ iff $l \equiv m$ and $\pi(d_{jl}) = \pi(d_{jm})$
- $W^{acc} = \{G^{acc} \times G^{n-1} \times D^n \times \mathcal{E}_n, \ldots, G^{n-1} \times G^{acc} \times D^n \times \mathcal{E}_n\}$

Such automaton can be transformed into a Büchi automaton $\mathcal{A}^b_{\mathcal{D}}$ as before.

Theorem 36. (De Giacomo and Vardi, 1999) A conditional plan $\vec{\sigma}$ for $\mathcal{D}$ realizing the specification $\mathcal{G}$ exists iff $L(\mathcal{A}_{\mathcal{D}}) = L(\mathcal{A}^b_{\mathcal{D}}) \neq \varnothing$.

The nonemptiness algorithm can again be immediately modified to return a plan if a plan exists. The plan returned again consists of two parts: a sequence arriving to a certain state and a second sequence that forms loop over that state (however, this time the element of the sequences are n-tuples of actions). Observe that even if formally we still deal with vectors of sequential plans, the conditional plan returned can be put in a more convenient form using *case* instructions and loops.
Finally, It is easy to verify that the same complexity bounds of the previous case still hold, as only n bits to represent an equivalence relation on $\{0, \ldots, n-1\}$ are needed.

Theorem 37. (De Giacomo and Vardi, 1999) Finding a conditional plan in the setting above is PSPACE-complete (EXPSPACE-complete wrt a compact representation of $\mathcal{D}$).

6.1.4 Generalized Planning for LTL

In the previous section we presented the automata-based approach for planning for full-fledged LTL goals that was put forward in (De Giacomo and Vardi, 1999). It is however natural to relax this assumption, and study the case where one has the capability of interacting with multiple domain systems.
A first contribution in this direction was made in (Hu and De Giacomo, 2011), which proposes a framework that captures and extends most generalized planning formalisms, introducing generalized planning under multiple domain systems for reachability goals. The solution is shown to be EXPSPACE-complete which, notably, this is the same complexity as conformant and conditional planning under partial observability with deterministic actions. Hence, the main technical results of this section is thus to give a sound and complete procedure for solving the generalized planning problem for LTL goals, within the same complexity bound.
We assume that a generalized plan has to work on a set of k deterministic domain systems $\{\mathcal{D}_k, \ldots, \mathcal{D}_k\}$, each with its own associated goal specification $\mathcal{G}_i$, whose state space may be completely unrelated, and which share only actions (used by plans) and observations (used as sensing tests in plans). Hence we assume

1. partial observability
2. complete information

What follows extends the techniques for automata on infinite strings presented in the previous sections (De Giacomo and Vardi, 1999), to the case of a finite set of domains, and constitutes the mathematical foundations of the agent-based technique of (Felli et al., 2012), as illustrated in Section 6.3.2.
This setting is that of generalized planning (Hu and De Giacomo, 2011), although such work is not dealing with long-running (LTL) goals but just reachability specifications.

Basic planning problem. Given dynamic domain $\mathcal{D} = \langle D, d_0, Act, R, Obs, \pi \rangle$ and a specification $\mathcal{G} = \langle Obs, G, g_0, \gamma, G^{acc} \rangle$ as in Section 6.1.3, a (basic) planning problem is the couple $P = \langle \mathcal{D}, \mathcal{G} \rangle$.

Observe that, for clarity of presentation, since we are assuming complete information –and hence unique initial state of $\mathcal{D}$– we will denote the singleton set of initial states with the initial state $d_0 \in D$ itself.
This basic setting matches the planning with complete information, and the same resolution technique of Section 6.1.3 applies.

Generalized planning problem. A generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_k\}$ is a *finite* set of k basic planning problems $P_i = \{\mathcal{D}_i, \mathcal{G}_i\}$.

Intuitively, we require that a plan satisfies, on every dynamic system $\mathcal{D}_i = \langle D_i, d_{0i}, Act, \delta_i, Obs, \pi_i \rangle$, its corresponding goal $\mathcal{G}_i = \langle G_i, g_{0i}, Obs, \gamma_i, G_i^{acc} \rangle$.
Again, note that observations Obs and actions Act are fixed, shared among all problems.

Conditional plan. A conditional plan $\vec{\sigma}$ is as in Section 2.2, i.e., it is a vector plan such that given the executions $d_{0l}, d_{1l}, d_{2l}, \ldots$ and $d_{0m}, d_{1m}, d_{2m}, \ldots$ of a pair of components l and m, we have that $a_{nl} = a_{nm}$ whenever $\langle d_{0l}, \ldots, d_{nl}\rangle \sim \langle d_{0m}, \ldots, d_{nm}\rangle$. In order to achieve this, we keep a simmetric equivalence relation $\equiv \subseteq \{1, \ldots, k\} \times \{1, \ldots, k\}$.

A conditional plan $\vec{\sigma}$ is a solution of the generalized planning problem iff it generates, for each domain $\mathcal{D}_i$, a run $\tau_i(\sigma_i, d_{0i})$ that is accepted by $\mathcal{G}_i$.
We now build an automaton capturing the generalized planning problem. Given a set of k (basic) planning problems $P_i = \langle \mathcal{D}_i, \mathcal{G}_i\rangle$ with $\mathcal{G}_i = \langle G_i, g_{i0}, Obs, \gamma_i, G_i^{acc}\rangle$ and $\mathcal{D}_i = \langle D_i, d_{0i}, Act, \delta_i, Obs, \pi_i\rangle$, $i \in \{1, \ldots, k\}$, we build the generalized automaton $\mathcal{A}_\mathcal{P} = \{Act^k, W, w_0, \rho, W^{acc}\}$ where:

- Act^k is the set of k-vectors of actions;
- $W = D_1 \times \ldots \times D_k \times G_1 \times \ldots \times G_k \times \mathcal{E}_k$, where $\mathcal{E}_k$ is the set of equivalence relations on the set $\{0, \ldots, k-1\}$;
- $w_0 = \langle d_{i0}, \ldots, d_{k0}, g_{i0}, \ldots, g_{k0}, \equiv_0\rangle$ where $i \equiv_0 j$ iff $\pi_i(d_{i0}) = \pi_j(d_{j0})$;
- $\langle \vec{d}', \vec{g}', \equiv'\rangle \in \rho(\langle \vec{d}, \vec{g}, \equiv\rangle, \vec{a})$ iff
  - if $i \equiv j$ then $a_i = a_j$;
  - $d_i' = \delta_i(d_i, a_i)$ and $g_i' = \gamma_i(g_i, \pi_i(d_i))$;
  - $i \equiv' j$ iff $i \equiv j$ and $\pi_i(d_i') = \pi_j(d_j')$.
- $W^{acc} = \{ \; D_1 \times \ldots \times D_k \times G_1^{acc} \times G_2 \times \ldots \times G_k \times \equiv \; , \; \ldots \; , \; D_1 \times \ldots \times D_k \times G_1 \times \ldots \times G_{k-1} \times G_k^{acc} \times \equiv \; \}$

Observe that there exists tight similarity between this automaton and the automaton built for solving the conditional planning problem with incomplete information, as in Section 6.1.3. Although structurally similar, the two generalized automata capture substantially different problems. First, in the previous case one is required to account for incomplete information about the initial state of the system; therefore every state $w \in W$ keeps track of n distinct possible states in which the system could be in. In the generalized case instead, we keep track of the current state in which each one of the k domain systems is. Second, we are now dealing with k distinct goal specifications, thus progressing each i-th component according to $\mathcal{G}_i$.

Theorem 38. The generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_k\}$ has a solution iff $L(\mathcal{A}_\mathcal{P}) \neq \varnothing$.

As before, the plan returned consists of two parts: a sequence of k-vectors reaching a certain state and a second sequence that forms loop over that state.
As for complexity, we thus obtain the same PSACE-complete results as for the previous case. Indeed, recall that for each generalized Büchi automaton $\mathcal{A}_g$ there exists a Büchi automaton $\mathcal{A}$ such that $L(\mathcal{A}_g) = L(\mathcal{A})$ and $|\mathcal{A}| = \mathcal{O}(|\mathcal{A}_g| \cdot |S^{acc}|)$ where S^{acc} denotes the set of accepting states of $\mathcal{A}$.

6.1.5 Discussion

In this chapter we first resumed a theoretical approach to planning for long-running (LTL) goals (De Giacomo and Vardi, 1999) in various settings, then adapted the approach also to Generalized Planning for LTL goals, showing that the solution of the generalized case is within the same complexity bound. In particular, we saw how solving the (generalized) planning problem for LTL goals amounts to synthesize a plan that can be represented finitely. However, we did not suggest any mean to transform such plans into finite-state controllers that can be "read" at runtime in order to retrieve the action to execute. In the next chapter we will also address this particular aspect. Moreover, we will ground the work on Interpreted Systems (Ronald Fagin and Vardi, 1995), a popular agent-based semantics. This will allow us to anchor the planning framework to the notion of agent protocol, i.e., a function returning the set of possible actions in a given internal state of the agent, thereby permitting implementations on top of existing model checkers.

6.2 Agents and Interpreted Systems

The study of *Multi-Agent Systems* (MAS) is concerned with the study of open, distributed systems, where the process entities (or *agents*) possess highly flexible and autonomous behaviour. Differently from disciplines such as distributed systems and software engineering, the emphasis here is on the prominence given to concepts such as knowledge, beliefs, protocols, obligations, etc. Since information technology is facing the task of delivering ever more complex distributed applications, MAS researchers argue that much is to be gained from an approach that focuses on high-level macroscopic characteristics of the entities, at least in the modeling stage. MAS theories involve the formal representation of agents' behaviour and attitudes. To this end various modal logics have been studied and developed, including logics for knowledge, beliefs, actions, obligations, intentions, as well as combinations of these with temporal operators. Researchers involved in the formalisation of intelligent agents have investigated temporal extensions of particular modal logics used to model intensional notions such as knowledge or belief. Several modal logics have been proposed to deal with these concepts, and properties such as completeness, decidability, and computational complexity have been explored (Ronald Fagin and Vardi, 1995).

These logics are seen as specifications of particular classes of MAS systems. Their aim is to offer a precise description of the mental or behavioural properties that a MAS should exhibit in a specific class of scenarios. The complexity of systems under analysis as well as the computational complexity of such logics are often so hard (Halpern and Vardi, 1986) that current theorem provers seem to offer no easy solution to the problem of verification of MAS. Hence, following a very influential paper by Halpern and Vardi (Halpern and Vardi, 1991), attempts have been made to use model checking techniques (Clarke et al., 1999d) to tackle the verification problem of MAS. Specifically, (Hoek and Wooldridge, 2002a;b) analyse respectively the application of SPIN and MOCHA to model checking of LTL and ATL extended by epistemic modalities, whereas (Meyden and Shilov, 1999) studies the complexity of the model checking problem for systems for knowledge and time. Formal frame-

works for verifying temporal and epistemic properties of multi-agent systems have been devised by means of bounded model checking (Penczek and Lomuscio, 2003) as well as unbounded (Kacprzak et al., 2004). More recent investigations concern deontic extensions.
Much of this literature on knowledge representation for MAS employs modal temporal-epistemic languages defined on semantic structures called *interpreted systems* (Ronald Fagin and Vardi, 1995; Harel and Pnueli, 1985b). These are refined versions of Kripke semantics in which the notions of computational states and actions are given prominence. Interpreted Systems can be seen as an evolution of temporal epistemic systems based on Kripke frames and they provide a modular framework for reasoning about distributed systems and their properties. They are commonly seen as a prime example of computationally grounded, modular models of distributed systems, and they also suggest an intuitive methodology for modeling epistemic capabilities.

6.2.1 Interpreted Systems

In this section we just provide some concepts about interpreted systems; however, as we will relate to them in the technical development that follows. Hence, this is not a formal tractation of this subject, for which we refer the reader to, e.g., (Ronald Fagin and Vardi, 1995). One of our major design concern is to preserve and exploit IS modularity and independence, decoupling design phases of the agent and the set of environments. In fact, an agent is in general supposed to interact with an environment only partially known in advance. As noted in, e.g., (Jamroga and Ågotnes, 2007), this leads to the need of a *modular* representation that is able to both allow and exploit such independence, such that the agent or the environment can be modified or replaced without affecting the design or representation of its counterpart. With respect to the usual MAS setting though, we don't need any interaction between different agents or coalitions of them. On the contrary, we need to independently specify both our agent and multiple environments the agent interacts with, without any communication channel between such environments.

Local and global states Specifically, all the information an agent has at its disposal (variables, facts of a knowledge base, observations from the environment, etc.) is captured in the *local state* relating to the agent in question. *Global states* are tuples of local states, each representing an agent in the system as well as the environment. The environment is used to represent information such as messages in transit and other components of the system that are not amenable to an intention-based representation.
Consider n agents in a system and n non-empty sets $L_1, \ldots, L_n$ of local states, one for every agent of the system, and a set of states for the environment L_e.
Formally, a *system of global states* for n agents $\mathcal{S}$ is a non-empty set

$$G \subseteq L_1 \times \cdots \times L_n \times L_e$$

When $g = \langle l_1, \ldots l_n, l_e \rangle$, each l_i denotes the local stare of agent i and l_e the current state of the environment.

An *interpreted* system of global states is a pair $IS = (\mathcal{S}, \mathcal{V})$, where $\mathcal{S}$ is a system of global states and $\mathcal{V} : G \to Pwr(P)$ is the interpetation function for some fixed set of propositions AP.

Temporal evolution Interpreted systems can be extended to deal with temporal evolution. Since the evolution of such systems is the result of interactions among agents, it can be described in terms of a joint transition function, as for explicit models. To this aim, consider the evolution function

$$\delta : G \times (Act_1 \times \cdots \times Act_n \times Act_e) \to G$$

Hence the evolution of agents' local states describes a set of runs and thus a set of reachable states, used to interpret temporal modal formulae. A *run* λ on IS is a function $\mathbb{N} \to G$ from time (ranging over natural numbers) to global states. Hence, any *point* t of λ identifies a state of the run, i.e., $(\lambda, t) \in G$. Hence one can verify whether a system complies with a temporal/epistemic specification Φ.
While the transition relation encapsulates possible evolutions of the system over time, the epistemic dimension is defined by the local fragments of a global state: $\langle l_1, \ldots, l_k \rangle \sim_i \langle l'_1, \ldots, l'_k \rangle$ iff $l_i = l'_i$, meaning that l and l' are indistinguishable for agent i. Hence global states can be used to interpret epistemic modalities K_i, one for each agent:

$$(IS, g) \vDash K_i\varphi \text{ if for all } g' \in G \text{ we have } g_i = g'_i \text{ implies } (IS, g') \vDash \varphi$$

It has been shown (Ronald Fagin and Vardi, 1995; Lomuscio and Ryan, 1997) that we can associate a Kripke structure $\mathcal{K}_{IS}$ to an interpreted system $IS = \langle G, \mathcal{V} \rangle$. The states of such a structure consist of the points in IS, hence one can use the model $\mathcal{K}_{IS}$ to verify properties of IS, but we won't address here such task.

Agent protocols We assume that for every agent of the system and for the environment there is a set Act_i and Act_e of actions (we can also assume all these set equal). A *protocol* protcl_i for an agent i is a function from the set L_i of local states to a non-empty set of actions Act_i:

$$\mathsf{protcl}_i : L_i \to Act_i$$

If instead we allow sets of actions, we allow nondeterminism in the protocol.

6.3 Embedding strategies into Agent protocols

A key component of any intelligent agent is its ability of reasoning about the actions it performs in order to achieve a certain goal. If we consider a single-agent interacting with an environment, a natural question to ask is the extent to which an agent can derive a plan to achieve a given goal. Under the assumption of full observability of the environment, the methodology of LTL synthesis enables the automatic generation, e.g., through a model checker, of a set of rules for the agent to achieve a goal expressed as an LTL specification. This is a well-known decidable setting

but one that is 2EXPTIME-complete (Pnueli and Rosner, 1989a; Kupferman and Vardi, 2000) due to the required determinisation of non-deterministic Büchi automata. Solutions that are not complete but computationally attractive have been put forward (Harding et al., 2005). Alternative approaches focus on subsets of LTL, e.g., GR(1) formulas as in (Piterman et al., 2006a; Bloem et al., 2012).
Work in AI on planning has of course also addressed this issue albeit from a rather different perspective. The emphasis here is most often on sound but incomplete heuristics that are capable of generating effective plans on average. Differently from main-stream synthesis approaches, a well-explored assumption here is that of partial information, i.e., the setting where the environment is not fully observable by the agent. Progress here has been facilitated by the relatively recent advances in the efficiency of computation of Bayesian expected utilities and Markov decision processes. While these approaches are attractive, differently from work in LTL-synthesis, they essentially focus on goal reachability (Bonet and Geffner, 2009).
An automata-based approach to planning for full-fledged LTL goals (De Giacomo and Vardi, 1999), covering partial information, has beed illustrated in Section 6.1.3. The approach is based on non-emptiness of Büchi-automata on infinite words, assuming that the agent is interacting with a single deterministic environment which is only partially observable. As noted in the previous chapter, it is however natural to relax this assumption, and study the case where an agent has the capability of interacting with multiple, partially-observable, environments sharing a common interface. In fact, Section 6.1.4 extends the technique based on non-emptiness of Büchi-automata to this case, following the contribution of (Hu and De Giacomo, 2011), which introduced generalized planning under multiple environments for reachability goals and showed that its complexity is EXPSPACE-complete. The main technical results was thus to give a sound and complete procedure for solving the generalized planning problem for LTL goals, within the same complexity bound.

6.3.1 Synthesizing Agent Protocols From LTL Specifications

In this Section we consider the problem of synthesizing an agent protocol to satisfy temporal specifications. In particular, the problem we consider here is composed on this ingredients:

- we assume LTL goals;
- we consider a single agent, for which we want to synthesize a protocol;
- we address the generalized planning problem, hence we consider an agent interacting with multiple environments;
- environments are partially observable.

We present a sound and complete procedure for solving the synthesis problem in this setting and show it is computationally optimal from a theoretical complexity standpoint. While this produces perfect-recall, hence unbounded, strategies we show how to transform these into agent protocols with bounded number of states. Differently from (Hu and De Giacomo, 2011) we here prove the complexity of optimal strategies based on perfect recall and, importantly, we show their reduction to

finite state controllers, which can be embedded in the agent protocol, i.e., the set of rules that constrain the actions executable by the agent at each point in time.
A further departure from (Hu and De Giacomo, 2011) is that we here ground the work on Interpreted Systems (Ronald Fagin and Vardi, 1995; Parikh and Ramanujam, 1985), a popular agent-based semantics. This enables us to anchor the framework to the notion of *agent protocol*, i.e., a function returning the set of possible actions in a given local state, thereby permitting implementations on top of existing model checkers (Gammie and van der Meyden, 2004; Lomuscio et al., 2009). As as already remarked in the literature (van der Meyden, 1996), interpreted systems are particularly amenable to incorporating observations and uncertainty in the environment. The model we pursue here differentiates between visibility of the environment states and observations made by the agent (given what is visible). Similar models have been used in the past, e.g, the VSK model discussed in (Wooldridge and Lomuscio, 2001). Differently from the VSK model, here we are not concerned in epistemic specifications nor do we wish to reason at specification level about what is visible, or observable. The primary aim, instead, is to give sound and complete procedures for solving the generalized planning problem for LTL goals.

6.3.2 State-based and History-based Solutions

Finite-state and memoryless controllers are simple action selection mechanisms widely used in domains such as videogames and mobile robotics. Memoryless controllers stand for functions that map observations into actions, while finite state controllers generalize memoryless ones with a finite amount of memory.
With the formal notion of agent protocol hand, it thus becomes natural to distinguish several ways to address generalized planning for LTL goals, namely:

- *State-based solutions*, where the agent has the ability of choosing the "right" action toward the achievement of the LTL goal, among those that its protocol allows, exploiting the current observation, but avoiding memorizing previous observations. So the resulting plan is essentially a *state-based strategy*, which can be encoded in the agent protocol itself. While this solution is particularly simple, is also often too restrictive.

- *History-based solutions*, where the agent has the ability of remembering all observations made in the past, and use them to decide the "right" action toward the achivement of the LTL goal, again among those allowed by its protocol. In this case we get a *perfect-recall strategy*. These are the most general solutions, though in principle such solutions could be infinite and hence unfeasible. One of the key results, however, is that if a perfect-recall strategy exists, then there exist one which can be represented with a finite number of states.

- *Bounded solutions*, where the agent decides the "right" course of actions, by taking into account only a fragment of the observation history. In this case the resulting strategies can be encoded as a new agent protocol, still compatible with the original one, though allowing the internal state space of the original agent to grow so as to incorporate the fragment of history to memorize. Since

we show that if a perfect-recall strategy exists, there exist one that is finite state (and hence makes use only of a fragment of the history), we essentially show that wlog we can restrict our attention to bounded solution (for a suitable bound) and incorporate such solutions in the agent protocol.

The remainder of this section is organized as follows. First we give the formal framework for our investigations, and we look at state-based solutions. Then, we move to history based solutions and prove our key technical results. We give a sound, complete, and computationally optimal (wrt worst case complexity) procedure for synthesizing perfect-recall strategies from LTL specification. Moreover, we observe that the resulting strategy can be represented with finite states. Then, we show how to encode such strategies (which are in fact bounded) into agent protocol. Finally, we briefly look at an interesting variant of the presented setting where the same results hold.

6.3.3 Framework

An interesting case is that of a single agent interacting with an environment. This is not only interesting in single-agent systems, or whenever we wish to study the single-agent interaction, but it is also a useful abstraction in loosely-coupled systems where all of the agents' interactions take place with the environment. Also note that the modular reasoning approach in verification focuses on the single agent case in which the environment encodes the rest of the system and its potential influence to the component under analysis. In this and other cases it is useful to model the system as being composed by a single agent only, but interacting with multiple environments, each possibly modelled by a different finite-state machine.
With this background we here pursue a design paradigm enabling the refinement of a generic agent program following a planning synthesis procedure given through an LTL specification, such that the overall design and encoding is not affected nor invalidated. We follow the interpreted system model and formalise an agent as being composed of *(i)* a set of local states, *(ii)* a set of actions, *(iii)* a *protocol* function specifying which actions are available in each local state, *(iv)* an observation function encoding what perceptions are being made by the agent, and *(v)* a local evolution function. An environment is modelled similarly to an agent; the global transition function is defined on the local transition functions of its components, i.e., the agent's and the environment's.
We refer to Figure 6.2 for a pictorial description of our setting in which the agent is executing actions on the environment, which, in turn, respond to these actions by changing its state. The observations of the agent depend on what perceived of the states of the environment. Notice that environments are only partially observable through what perceivable of their states. So two different environments may give rise to the same observations in the agent.

Environment. An *environment* is a tuple $Env = \langle E, Act_e, Perc, \mathsf{perc}, \delta, e_0 \rangle$ such that:

- $E = \{e_0, \ldots\}$ is a finite set of local states of the environment;

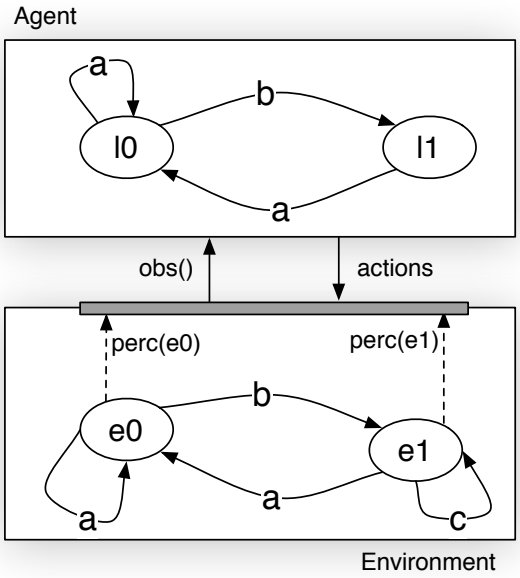


Figure 6.2. Interaction between Ag and Env

- Act_e is the alphabet of the environment's actions;
- $Perc$ is the alphabet of perceptions;
- $\mathsf{perc} : E \to Perc$ is the perceptions (output) function;
- $\delta : E \times Act_e \to E$ is a (partial) transition function;
- $e_o \in E$ is the initial state.

Notice that, as in classical planning, such an environment is deterministic.
A *trace* is a sequence $\tau = e_0\alpha_1 e_1\alpha_2 \ldots \alpha_n e_n$ of environment's states such that $e_{i+1} = \delta(e_i, \alpha_{i+1})$ for each $0 \leq i < n$. The corresponding *perceivable* trace is the trace obtained by applying the perception function: $\mathsf{perc}(\tau) = \mathsf{perc}(e_0), \ldots, \mathsf{perc}(e_n)$.
Similarly, an agent is represented as a finite machine, whose state space is obtained by considering the agent's internal states, called *configurations*, together with all additional information the agent can access, i.e., the observations it makes. We take the resulting state space as the agent's *local states*.

Agent. An *agent* is a tuple $Ag = \langle Conf, Act_a, Obs, \mathsf{obs}, L, \mathsf{poss}, \delta_a, c_0\rangle$ where:

- $Conf = \{c_0, \ldots\}$ is a finite set of the agent's configurations (internal states);
- Act_a is the finite set of agent's actions;
- Obs is the alphabet of observations the agent can make;
- $\mathsf{obs} : Perc \to Obs$ is the observation function;
- $L = Conf \times Obs$ is the set of agent's local states;
- $\mathsf{poss} : L \to \wp(Act_a)$ is a protocol function;
- $\delta_a : L \times Act_a \to Conf$ is the (partial) transition function;

- $c_0 \in Conf$ is the initial configuration. A local state $l = \langle c, o \rangle$ is called *initial* iff $c = c_0$.

Agents defined as above are deterministic: given a local state l and an action α, there exists a unique next configuration $c' = \delta_a(l, \alpha)$. Nonetheless, observations introduce non-determinism when computing the new local state resulting from action execution: executing action α in a given local state $l = \langle c, o \rangle$ results into a new local state $l' = \langle c', o' \rangle$ such that $c' = \delta_a(l, \alpha)$ and o' is the new observation, which can not be foreseen in advance.
Consider the agent and the environment depicted in Figure 6.2. Assume that the agent is in its initial configuration c_0 and that the current environment's state is e_0. Assume also that $\mathsf{obs}(\mathsf{perc}(e_0)) = o$, i.e., the agent receives observation o. Hence, the current local state is $l_0 = \langle c_0, o \rangle$. If the agent performs action b (with $b \in \mathsf{poss}(l_0)$), the agent moves from configuration c_0 to configuration $c_1 = \delta_a(\langle c_0, o \rangle, b)$. At the same time, the environment changed its state from e_0 to e_1, so that the new local state is $l_1 = \langle c_1, o' \rangle$, where $o' = \mathsf{obs}(\mathsf{perc}(e_1))$.
Notice that the protocol function is not defined with respect to the transition function, i.e., according to transitions available to the agent. In fact, we can imagine an agent having its own behaviours, in terms of transitions defined over configurations, that can be constrained according to some protocol, which can in principle be modified or substituted. Hence, we say that a protocol is *compatible* with an agent iff it is compatible with its transition function, i.e., $\alpha \in \mathsf{poss}(\langle c, o \rangle) \rightarrow \exists c' \in Conf \mid \delta_a(\langle c, o \rangle, \alpha) = c'$. Moreover, we say that a protocol poss is an *action-deterministic protocol* iff it always returns a singleton set, i.e., it allows only a single action to be executed for a given local state. Finally, an agent is *nonblocking* iff it is equipped with a compatible protocol function poss and for each sequence of local states $l_0\alpha_1 l_1 \alpha_2 \ldots \alpha_n l_n$ such that $l_i = \langle c_i, o_i \rangle$ and $\alpha_{i+1} \in \mathsf{poss}(\langle c_i, o_i \rangle)$ for each $0 \leq i < n$, we have $\mathsf{poss}(l_n) \neq \varnothing$. So, an agent is nonblocking iff it has a compatible protocol function which always provides a non-empty set of choices for each local state that is reachable according to the transition function and the protocol itself.
Finally, given a perceivable trace of an environment Env, the *observation history* of an agent Ag is defined as the sequence obtained by applying the observation function: $\mathsf{obs}(\mathsf{perc}(\tau)) = \mathsf{obs}(\mathsf{perc}(e_0)), \ldots, \mathsf{obs}(\mathsf{perc}(e_n))$. Given one such history $h \in Obs^*$, we denote with $last(h)$ its latest observation: $last(h) = \mathsf{obs}(\mathsf{perc}(e_n))$.

6.3.4 Problem

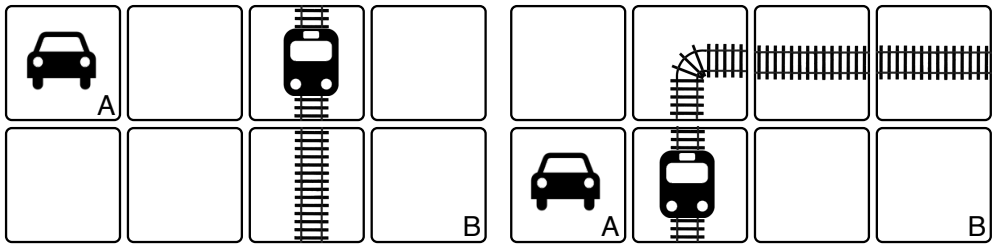
We consider an *agent* Ag and a *finite set* $\mathcal{E}$ *of environments*, all sharing common actions and the same perception alphabet. Such environments are indistinguishable by the agent, in the sense that the agent is not able to identify which environment it is actually interacting with, unless through observations. The problem we address is thus to synthesize a (or customize the) agent protocol so as to fulfill a given LTL specification (or goal) in all environments. A planning problem for an LTL goal is a triple $\mathcal{P} = \langle Ag, \mathcal{E}, \mathcal{G} \rangle$, where Ag is an agent, $\mathcal{E}$ an environment set, and $\mathcal{G}$ an LTL goal specification. This setting is that of *generalized planning* (Hu and De Giacomo, 2011), extended to deal with *long-running goals*, expressed as arbitrary LTL formulae. This is also related to *planning for* LTL *goals* under partial observability (De

Giacomo and Vardi, 1999).
Formally, we call *environment set* a finite set of environments $\mathcal{E} = \{Env_1, \ldots, Env_k\}$, with $Env_i = \langle E_i, Act_{ei}, Perc, \mathsf{perc}_i, \delta_i, e_{0i}\rangle$. Environments share the same alphabet $Perc$ of the agent Ag. Moreover the Ag's actions must be executable in the various environments: $Act_a \subseteq \bigcap_{i=1,\ldots,k} Act_{ei}$. This is because we are considering an agent acting in environments with similar "interface".
As customary in verification and synthesis (Baier and Katoen, 2008), we represent an LTL goal with a Büchi automaton $\mathcal{G} = \langle Perc, G, g_0, \gamma, G^{acc}\rangle$, describing the desired behaviour of the system in terms of perceivable traces, where:

- $Perc$ is the finite alphabet of perceptions, taken as input;
- G is a finite set of states;
- g_0 is the initial state;
- $\gamma : G \times Perc \to 2^G$ is the transition function;
- $G^{acc} \subseteq G$ is the set of accepting states.

A run of $\mathcal{G}$ on an input word $w = p_0, p_1, \ldots \in Perc^\omega$ is an infinite sequence of states $\rho = g_0, g_1, \ldots$ such that $g_i \in \gamma(g_{i-1}, p_i)$, for $i > 0$. Given a run ρ, let $inf(\rho) \subseteq G$ be the set of states occurring infinitely often, i.e., $inf(\rho) = \{g \in G \mid \forall i\ \exists j > i \text{ s.t. } g_j = g\}$. An infinite word $w \in Perc^\omega$ is accepted by $\mathcal{G}$ iff there exists a run ρ on w such that $inf(\rho) \cap G^{acc} \neq \varnothing$, i.e., at least one accepting state is visited infinitely often during the run. Notice that, since the alphabet $Perc$ is shared among environments, the same goal applies to all of them. As we will see later, we can characterize a variant of this problem by considering the environment's internal states as $\mathcal{G}$'s alphabet.

Example 21. *Consider a simple environment Env_1 constituted by a grid of 2x4 cells, each denoted by e_{ij}, a train, and a car. A railroad crosses the grid, passing on cells e_{13}, e_{23}. Initially, the car is in e_{11} and the train in e_{13}. The car is controlled by the agent, whereas the train is a moving obstacle moving from e_{13} to e_{23} to e_{13} again and so on. The set of actions is $Act_{e1} = \{goN, goS, goE, goW, \texttt{wait}\}$. The train and the car cannot leave the grid, so actions are allowed only when feasible. The state space is then $E_1 = \{e_{11}, \ldots, e_{24}\} \times \{e_{13}, e_{23}\}$, representing the positions of the car and the train. We consider a set of perceptions $Perc = \{\texttt{posA}, \texttt{posB}, \texttt{danger}, \texttt{dead}, \texttt{nil}\}$, and a function perc_1 defined as follows: $\mathsf{perc}_1(\langle e_{11}, e_t\rangle) = \texttt{posA}$, $\mathsf{perc}_1(\langle e_{24}, e_t\rangle) = \texttt{posB}$, $\mathsf{perc}_1(\langle e_{13}, e_{23}\rangle) = \mathsf{perc}_1(\langle e_{23}, e_{13}\rangle) = \texttt{danger}$ and $\mathsf{perc}_1(\langle e_{13}, e_{13}\rangle) = \mathsf{perc}_1(\langle e_{23}, e_{23}\rangle) = \texttt{dead}$. $\mathsf{perc}_1(\langle e_c, e_t\rangle) = \texttt{nil}$ for any other state.*
We consider a second environment Env_2 similar to Env_1 as depicted in Figure 6.3b. We skip details about its encoding, as it is analogous to Env_1.
Then, we consider a third environment Env_3 which is a variant of the Wumpus world (Russell and Norvig, 2010), though sharing the interface (in particular they all share perceptions and actions) with the other two. Following the same convention used before, the hero is initially in cell e_{11}, the Wumpus in e_{31}, gold is in e_{34} and the pit in e_{23}. The set of actions is $Act_{e3} = \{goN, goS, goE, goW, \texttt{wait}\}$. Recall that



(a) Env_1 **(b)** Env_2

Figure 6.3. Environments Env_1 and Env_2

the set of perceptions Perc is instead shared with Env_1 *and* Env_2*. The state space is* $E_3 = \{e_{11}, \ldots, e_{34}\}$*, and the function* perc_3 *is defined as follows:* $\mathsf{perc}_3(e_{11}) =$ `posA`*,* $\mathsf{perc}_3(e_{34}) =$ `posB`*,* $\mathsf{perc}_3(e_{23}) = \mathsf{perc}_3(e_{31}) =$ `dead`*,* $\mathsf{perc}_3(e) =$ `danger` *for* $e \in \{e_{13}, e_{21}, e_{22}, e_{24}, e_{32}, e_{33}\}$*, whereas* $\mathsf{perc}_3(e) =$ `nil` *for any other state. This example allows us to make some observation about our framework. Consider first the perceptions Perc. They are intended to represent signals coming from the environment, which is modeled as a "black box". If we could distinguish between perceptions (instead of having just a* `danger` *perception), we would be able to identify the current environment as* Env_3*, and solve such a problem separately. Instead, in our setting the perceptions are not informative enough to discriminate environments (or the agent is not able to observe them); so all environments need to be considered together. Indeed,* Env_3 *is similar to* Env_1 *and* Env_2 *at the interface level, and it is attractive to try to synthesize a single strategy to solve them all. In some sense, crashing in* Env_1 *or* Env_2 *corresponds to falling into the pit or being eaten by the Wumpus; the same holds for danger states with the difference that perceiving* `danger` *in* Env_1 *or* Env_2 *can not be used to prevent an accident.*

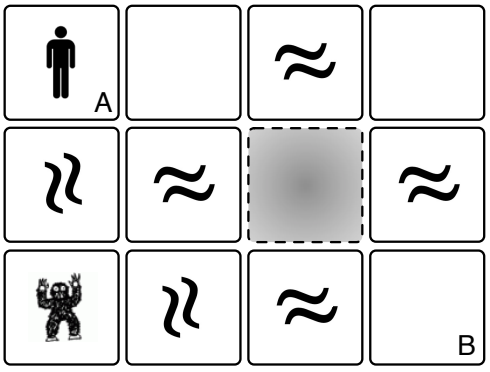


Figure 6.4. Environment Env_3

Notice that in our framework we design the environments without taking into account the agent Ag that will interact with them. Likewise, the same holds when designing

Ag. Indeed, agent Ag is encoded as an automaton with a single configuration c_0*, and all actions being loops. In particular, let* $Act_a = Act_{ei}$*,* $i = 1, 2, 3$*. Notice also that, by suitably defining the observation function* obs*, we can model the agent's sensing capabilities, i.e., its ability to observe perceptions coming from the environment. Suppose that Ag can sense perceptions* $\mathtt{posA}, \mathtt{danger}, \mathtt{dead}$*, but it is unable to sense* $\mathtt{posB}$*, i.e., its sensors can not detect such signal. To account for this, it is enough to consider the observation function* obs *as a "filter" (i.e.* $Obs \subseteq Perc$*), such that* $Obs = \{\mathtt{posA}, \mathtt{danger}, \mathtt{dead}, \mathtt{nil}\}$ *and* $\mathsf{obs}(\mathtt{posA}) = \mathtt{posA}$*,* $\mathsf{obs}(\mathtt{danger}) = \mathtt{danger}$*,* $\mathsf{obs}(\mathtt{dead}) = \mathtt{dead}$*, and* $\mathsf{obs}(\mathtt{nil}) = \mathsf{obs}(\mathtt{posB}) = \mathtt{nil}$*. Since Ag is filtering away perception* $\mathtt{posB}$*, the existence of a strategy does not imply that agent Ag is actually able to recognize that it has achieved its goal. Notice that this does not affect the solution, nor its executability, in any way. The goal is achieved irrespective of what the agent can observe.*

Moreover, Ag has a "safe" protocol function poss *that allows all moving actions to be executed in any possible local state, but prohibits it to perform wait if the agent is receiving the observation* $\mathtt{danger}$*:* $\mathsf{poss}(\langle c_0, o\rangle) = Act_a$ *if* $o \neq \mathtt{danger}$*,* $Act_a \setminus \{wait\}$ *otherwise.*

Finally, let $\mathcal{G}$ *be the automaton corresponding to the LTL formula* $\phi_{\mathcal{G}} = (\Box\Diamond\mathtt{posA}) \wedge (\Box\Diamond\mathtt{posB}) \wedge \Box\neg\mathtt{dead}$ *over the perception alphabet, constraining perceivable traces such that the controlled objects (the car / the hero) visit positions A and B infinitely often.* □

6.3.5 State-based solution

To solve the synthesis problem in the context above, the first solution that we analyze is based on customizing the agent to achieve the LTL goal, by *restricting the agent protocol* while keeping the same set of local states. We do this by considering so called state-based strategies (or plans) to achieve the goal. We call a strategy for an agent *Ag state-based* if it can be expressed as a (partial) function

$$\sigma_p : (Conf \times Obs) \to Act_a$$

For it to be acceptable, a strategy also needs to be *allowed* by the protocol: it can only select available actions, i.e., for each local state $l = \langle c, o\rangle$ we have to have $\sigma_p(l) \in \mathsf{poss}(l)$.

State-based strategies do not exploit an agent's memory, which, in principle, could be used to reduce its uncertainty about the environment by remembering observations from the past. Exploiting this memory requires having the ability of extending its configuration space, which at the moment we do not allow (see later). In return, these state-based strategies can be synthesized by just taking into account all allowed choices the agent has in each local state (e.g., by exhaustive search, possibly guided by heuristics). The advantage is that to meet its goal, the agent *Ag* does not need any modification to its configurations, local states and transition function, since only the protocol is affected. In fact, we can see a strategy σ_p as a restriction of an agent's protocol yielding an action-deterministic protocol $\overline{\mathsf{poss}}$ derived as

follows:

$$\overline{\mathsf{poss}}(\langle c, o\rangle) = \begin{cases} \{\alpha\}, & \text{iff } \sigma_p(c,o) = \alpha \\ \varnothing, & \text{if } \sigma_p(c,o) \text{ is undefined} \end{cases}$$

Notice that $\overline{\mathsf{poss}}$ is then a total function. Notice also that agent $\overline{Ag}$ obtained by substituting the protocol function maintains a protocol compatible with the agent transition function. Indeed, the new allowed behaviours are a subset of those permitted by original agent protocol.

Example 22. *Consider again Example 21. No state-based solution exists for this problem, since selecting the same action every time the agent is in a given local state does not solve the problem. Indeed, just by considering the local states we can not get any information about the train's position, and we would be also bound to move the car (the hero) in the same direction every time we get the same observation (agent Ag has only one configuration c_0). Nonetheless, observe that if we could keep track of past observations when selecting actions, then a solution can be found.* □

6.3.6 History-based solution

We turn to strategies that can take into account past observations. Specifically, we focus on strategies (plans) that may depend on the entire unbounded observation history. These are often called perfect-recall strategies.
A *(perfect-recall) strategy* for $\mathcal{P}$ is a (partial) function

$$\sigma : Obs^* \to Act_a$$

that, given a sequence of observations (the *observation history*), returns an action. A strategy σ is *allowed* by Ag's protocol iff, given any observation history $h \in Obs^*$, $\sigma(h) = \alpha$ implies $\alpha \in poss(\langle c, last(h)\rangle)$, where c is the current configuration of the agent. Notice that, given an observation history, the current configuration can be always reconstructed by applying the transition function of Ag, starting from initial configuration c_0. Hence, a strategy σ is a solution of the problem $\mathcal{P} = \langle Ag, \mathcal{E}, \mathcal{G}\rangle$ iff it is allowed by Ag and it generates, for each environment Env_i, an infinite trace $\tau = e_{0i}\alpha_1 e_{1i}\alpha_2 \ldots$ such that the corresponding perceivable trace $\mathsf{perc}(\tau)$ satisfies the LTL goal, i.e., it is accepted by the corresponding Büchi automaton.
The technique we propose is based on previous automata theoretic approaches. In particular, we extend the technique for automata on infinite strings presented in (De Giacomo and Vardi, 1999) for partial observability, to the case of a finite set of environments, along the lines of (Hu and De Giacomo, 2011). The crucial point is that we need both the ability of simultaneously dealing with LTL goals and with multiple environments. We build a generalized Büchi automaton that returns sequences of *action vectors* with one component for each environment in the environment set $\mathcal{E}$. Assuming $|\mathcal{E}| = k$, we arbitrarily order the k environments, and consider sequences of action vectors of the form $\vec{a} = \langle a_1, \ldots, a_k\rangle$, where each component specifies one operation for each environment. Such sequences of action vectors correspond to a strategy σ, which, however, must be *executable*: for any pair $i, j \in \{1, \ldots, k\}$ and observation history $h \in Obs^*$ such that both σ_i and σ_j are

defined, then $\sigma_i(h) = \sigma_j(h)$. In other words, if we received the same observation history, the function select the same action. In order to achieve this, we keep an *equivalence relation* $\equiv \subseteq \{1,\dots,k\} \times \{1,\dots,k\}$ in the states of our automaton. Observe that this equivalence relation has correspondences with the epistemic relations considered in epistemic approaches (Jamroga and van der Hoek, 2004; Lomuscio and Raimondi, 2006; Pacuit and Simon, 2011).

We are now ready to give the details of the automata construction. Given a set of k environments $\mathcal{E}$ with $Env_i = \langle E_i, Act_{ei}, Perc, \mathsf{perc}_i, \delta_i, e_{0i}\rangle$, an agent $Ag = \langle Conf, Act_a, Obs, \mathsf{obs}, L, \mathsf{poss}, \delta_a, c_0\rangle$ and goal $\mathcal{G} = \langle Perc, G, g_0, \gamma, G^{acc}\rangle$, we build the generalized Büchi automaton $\mathcal{A}_\mathcal{P} = \langle Act_a^k, W, w_0, \rho, W^{acc}\rangle$ as follows:

- $Act_a^k = (Act_a)^k$ is the set of k-vectors of actions;
- $W = E^k \times Conf^k \times G^k \times \wp(\equiv)$;[1]
- $w_0 = \langle e_{10},\dots,e_{k0}, c_0,\dots,c_0, g_0,\dots,g_0, \equiv_0\rangle$ where

  $$i \equiv_0 j \text{ iff } \mathsf{obs}(\mathsf{perc}_i(e_{i0})) = \mathsf{obs}(\mathsf{perc}_j(e_{j0}));$$

- $\langle \vec{e}', \vec{c}', \vec{g}', \equiv'\rangle \in \rho(\langle \vec{e}, \vec{c}, \vec{g}, \equiv\rangle, \vec{\alpha})$ iff
  - if $i \equiv j$ then $\alpha_i = \alpha_j$;
  - $e_i' = \delta_i(e_i, \alpha_i)$;
  - $c_i' = \delta_a(l_i, \alpha_i) \wedge \alpha_i \in \mathsf{poss}(l_i)$ where $l_i = \langle c_i, \mathsf{obs}(\mathsf{perc}_i(e_i))\rangle$;
  - $g_i' = \gamma(g_i, \mathsf{perc}_i(e_i))$;
  - $i \equiv' j$ iff $i \equiv j \wedge \mathsf{obs}(\mathsf{perc}_i(e_i')) = \mathsf{obs}(\mathsf{perc}_j(e_j'))$.
- $W^{acc} = \{$
  $E^k \times Conf^k \times G^{acc} \times G \times \ldots \times G \times \equiv\ ,\ \ldots\ ,$
  $E^k \times Conf^k \times G \times \ldots \times G \times G^{acc} \times \equiv\ \}$

Each automaton state $w \in W$ holds information about the internal state of each environment, the corresponding current goal state, the current configuration of the agent for each environment, and the equivalence relation. Notice that, even with fixed agent and goal, we need to duplicate their corresponding components in each state of $\mathcal{A}_\mathcal{P}$ in order to consider all possibilities for the k environments. In the initial state w_0, the environments, the agent and the goal automaton are in their respective initial state. The initial equivalence relation $\equiv_0$ is computed according to the observation provided by environments. The transition relation ρ is built by suitably composing the transition function of each state component, namely δ_a for agent, δ_i for the environments, and γ for the goal. Notice that we do not build transitions in which an action α is assigned to the agent when either it is in a configuration from which a transition with action α is not defined, or α is not allowed by the protocol poss for the current local state. The equivalence relation is updated at each step by considering the observations taken from each environment. Finally, each member of the accepting set W^{acc} contains a goal accepting state, in some environment.

[1] We denote by $\wp(\equiv)$ the set of all possible equivalence relations $\equiv \subseteq \{1,\dots,k\} \times \{1,\dots,k\}$.

Once this automaton is constructed, we check it for non-emptiness. If it is not empty, i.e., there exists a infinite sequence of action vectors accepted by the automaton, then from such an infinite sequence it is possible to build a strategy realizing the LTL goal. The non-emptiness check is done by resolving polynomially transforming the generalized Büchi automaton into standard Büchi one and solving *fair reachability* over the graph of the automaton, which (as standard reachability) can be solved in NLOGSPACE (Vardi, 1996). The non-emptiness algorithm itself can also be used to return a strategy, if it exists.
The following result guarantees that not only the technique is sound (the perfect-recall strategies do realize the LTL specification), but it is also complete (if a perfect-recall strategy exists, it will be constructed by the technique).
this technique is correct, in the sense that if a perfect-recall strategy exists then it will return one.

Theorem 39 (Soundness and Completeness)**.** A strategy σ that is a solution for problem $\mathcal{P} = \langle Ag, \mathcal{E}, \mathcal{G} \rangle$ exists iff $L(\mathcal{A}_{\mathcal{P}}) \neq \varnothing$.

Proof. ($\Leftarrow$) The proof is based on the fact that $L(\mathcal{A}_{\mathcal{P}}) = \varnothing$ implies that it holds the following persistence property: for all runs in $\mathcal{A}_{\mathcal{P}}$ of the form $r_\omega = w_0\vec{\alpha}_1 w_1 \vec{\alpha}_2 w_2 \ldots \in W^\omega$ there exists an index $i \geq 0$ such that $w_j \notin W^{acc}$ for any $j \geq i$. Conversely, if $L(\mathcal{A}_{\mathcal{P}}) \neq \varnothing$, there exists an infinite run $r_\omega = w_0\vec{\alpha}_1 w_1 \vec{\alpha}_2 w_2 \ldots$ on $\mathcal{A}_{\mathcal{P}}$ visiting at least one state for each accepting set in W^{acc} infinitely often (as required by its acceptance condition), thus satisfying the goal in each environment. First, we notice that such an infinite run r_ω is the form $r_\omega = r'(r'')^\omega$ where both r' and r'' are finite sequences. Hence such a run can be represented with a finite *lazo* shape representation: $r = w_0\vec{a}_1 w_1 \ldots \vec{a}_n w_n \vec{a}_{n+1} w_m$ with $m < n$ (Vardi, 1996). Hence we can synthesize the corresponding partial function σ by unpacking r (see later). Essentially, given one such r and any observable history $h = o_0, \ldots, o_\ell$ and denoting with α_{ji} the i-th component of $\vec{\alpha}_j$, σ is inductively defined as follows:

- if $\ell = 0$ then $\sigma(o_0) = \alpha_{1i}$ iff $o_0 = \mathsf{obs}(\mathsf{perc}_i(e_{0i}))$ in w_0.
- if $\sigma(o_0, \ldots, o_{\ell-1}) = \alpha$ then $\sigma(h) = \alpha_{ji}$ iff $o_\ell = \mathsf{obs}(\mathsf{perc}_i(e_{ji}))$ in $w_j = \langle \vec{e}_j, \vec{c}_j, \vec{g}_j, \equiv_j \rangle$ and α is such that $\alpha = \alpha_{\ell z}$ with $i \equiv_j z$ for some z, where $j = \ell$ if $\ell \leq m$, otherwise $j = m + \ell \bmod (n\text{-}m)$. If instead $o_\ell \neq \mathsf{obs}(\mathsf{perc}_i(e_{ji}))$ for any e_{ji} then $\sigma(h)$ is undefined.

Indeed, σ is a prefix-closed function of the whole history: we need to look at the choice made at previous step to keep track of it. In fact, we will see later how unpacking r will result into a sort of tree-structure representation. Moreover, it is trivial to notice that any strategy σ synthesized by emptiness checking $\mathcal{A}_{\mathcal{P}}$ is allowed by agent Ag. In fact, transition relation ρ is defined according to the agent's protocol function poss.
($\Rightarrow$) Assume that a strategy σ for $\mathcal{P}$ does exist. We prove that, given such σ, there exists in $\mathcal{A}_{\mathcal{P}}$ a corresponding accepting run r_ω as before. We prove that there exists in $\mathcal{A}_{\mathcal{P}}$ a run $r_\omega = w_0\vec{\alpha}_1 w_1 \vec{\alpha}_2 w_2 \ldots$, with $w_\ell = \langle \vec{e}_\ell, \vec{c}_\ell, \vec{g}_\ell, \equiv_\ell \rangle \in W$, such that:

1. ($\ell = 0$) $\sigma(\mathsf{obs}(\mathsf{perc}_i(e_{0i}))) = \alpha_{1i}$ for all $0 < i \leq k$;
2. ($\ell > 0$) if $\sigma(\mathsf{obs}(\mathsf{perc}_i(e_{(\ell-1)i}))) = \alpha$ for some $0 < i \leq k$, then $\alpha = \alpha_{\ell i}$ and $e_{\ell i} = \delta_i(e_{(\ell-1)i}, \alpha_{\ell i})$ and $\sigma(\mathsf{obs}(\mathsf{perc}_i(e_{\ell i})))$ is defined;
3. r_ω is accepting.

In other words, there exists in $\mathcal{A}_\mathcal{P}$ an accepting run that is induced by executing σ on $\mathcal{A}_\mathcal{P}$ itself. Point 1 holds since, in order to be a solution for $\mathcal{P}$, the function σ has to be defined for histories constituted by the sole observation $\mathsf{obs}(\mathsf{perc}_i(e_{0i}))$ of any environment initial state. According to the definition of the transition relation ρ, there exists in $\mathcal{A}_\mathcal{P}$ a transition from each e_{0i} for all available actions α such that $\delta_i(e_{0i}, \alpha)$ is defined for Env_i. In particular, the transition $\langle w, \vec{\alpha}, w' \rangle$ is not permitted in $\mathcal{A}_\mathcal{P}$ iff either some action component α_i is not allowed by agent's protocol poss or it is not available in the environment Env_i, $0 < i \leq k$. Since σ is a solution of $\mathcal{P}$ (and thus allowed by Ag) it cannot be one of such cases. Point 2 holds for the same reason: there exists a transition in ρ for all available actions of each environment. Point 3 is just a consequence of σ being a solution of $\mathcal{P}$. ■

Checking wether $L(\mathcal{A}_\mathcal{P}) \neq \varnothing$ can be done NLOGSPACE in the size of the automaton. Our automaton is exponential in the number of environments in $\mathcal{E}$, but its construction can be done on-the-fly while checking for non-emptiness. This give us a PSPACE upperbound in the size of the original specification with explicit states. If we have a compact representation of those, then we get an EXPSPACE upperbound. Considering that even the simple case of generalized planning for reachability goals in (Hu and De Giacomo, 2011) is PSPACE-complete (EXPSPACE-complete considering compact representation), we obtain a worst case complexity characterization of the problem at hand.

Theorem 40 (Complexity)**.** Solving the problem $\mathcal{P} = \langle Ag, \mathcal{E}, \mathcal{G} \rangle$ admitting perfect-recall solutions is PSPACE-complete (EXPSPACE-complete considering compact representation).

We conclude this section by remarking that, since the agent gets different observation histories from two environments Env_i and Env_j, then from that point on it will be always possible to distinguish these. More formally, denoting with r a run in $\mathcal{A}_P$ and with $r_\ell = \langle \vec{e}_\ell, \vec{c}_\ell, \vec{g}_\ell, \equiv_\ell \rangle$ its ℓ-th state, if $i \equiv_\ell j$, then $i \equiv_{\ell'} j$ for every state $r_{\ell' < \ell}$. Hence it follows that the equivalence relation $\equiv$ is indentical for each state belonging to the same strongly connected component of $\mathcal{A}_\mathcal{P}$. Indeed, assume by contradiction that there exists some index ℓ' violating the assumption above. This implies that $\equiv_{\ell'} \subset \equiv_{\ell'+1}$. So, there exists a tuple in $\equiv_{\ell'+1}$ that is not in $\equiv_{\ell'}$. But this is impossible since, by definition, we have that $i \equiv_{\ell'+1} j$ implies $i \equiv_{\ell'} j$.
Figure 6.5 shows a decision-tree like representation of a strategy. The diamond represents a decision point where the agent reduces its uncertainty about the environment. Each path ends in a loop thereby satisfying the automaton acceptance condition. The loop, which has no more decision point, represents also that the agent cannot reduce its uncertainty anymore and hence it has to act blindly as in

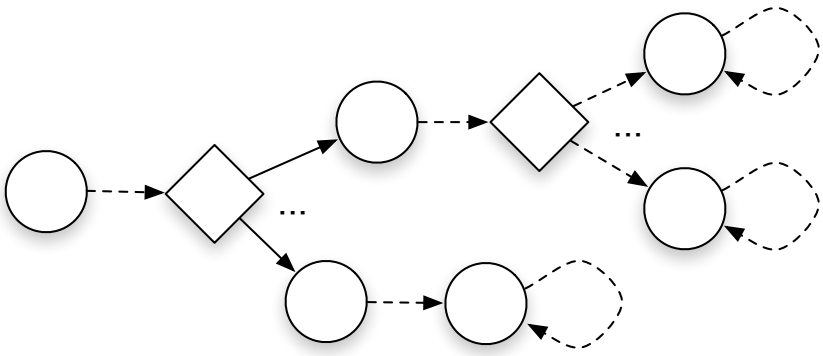


Figure 6.5. Decision-tree like representation of a strategy.

conformant planning. Notice that if our environment set includes only one environment, or if we have no observations to use to reduce uncertainty, then the strategy reduces to the structure in Figure 6.6, which reflects directly the general *lazo* shape of runs satisfying LTL properties: a sequence reaching a certain state and a second sequence consisting in a loop over that state.

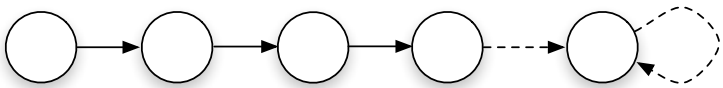


Figure 6.6. A resulting strategy execution.

6.3.7 Embedding strategies into protocols

Finally, we consider a variant of our problem where we specify LTL goals directly in terms of states of each environment in the environment set. In other words, instead of having a single goal specified over the percepts of the environments we have one LTL goal for each environment. More precisely, as before, we assume that a single strategy has to work on the whole set of deterministic environments. As previously, we require that $Act_a \subseteq \bigcap_{i=1,\ldots,k} Act_{ei}$ and that all environment share the same alphabet of perceptions $Perc$. Differently from before, we associate a distinct goal to each environment. We take as input alphabet of each goal specification $\mathcal{G}_i$ the set of environment's state E_i, i.e., $\mathcal{G}_i = \langle E_i, G_i, g_{i0}, \gamma_i, G_i^{acc} \rangle$. All goals are thus intimately different, as they are strictly related to the specific environment. Intuitively, we require that a strategy for agent Ag satisfies, in all environments Env_i, its corresponding goal $\mathcal{G}_i$. In other words, σ is a solution of the generalized planning problem $\mathcal{P} = \langle Ag, \mathcal{E}, \mathcal{G} \rangle$ iff it is allowed by Ag and it generates, for each environment Env_i, an infinite trace τ_i that is accepted by $\mathcal{G}_i$.
Devising perfect-recall strategies now requires only minimal changes to our original automata-based technique to take into account that we have now k distinct goals one for each environment. Given Ag and $\mathcal{E}$ as defined before, and k goals $\mathcal{G}_i = \langle E_i, G_i, g_{i0}, \gamma_i, G_i^{acc} \rangle$, we build the generalized Büchi automaton $\mathcal{A}_\mathcal{P} = \langle Act_a^k, W, w_0, \rho, W^{acc} \rangle$ as follows. Notice that each automaton $\mathcal{G}_i$ has E_i as input alphabet.

- $Act_a^k = (Act_a)^k$ is the set of k-vectors of actions;
- $W = E^k \times L^k \times G_1 \times \ldots \times G_k \times \wp(\equiv)$;
- $w_0 = \langle e_{i0}, \ldots, e_{k0}, c_{i0}, \ldots c_{k0}, g_{i0}, \ldots, g_{k0}, \equiv_0 \rangle$ where

  $$i \equiv_0 j \text{ iff } \mathsf{obs}(\mathsf{perc}_i(e_{i0})) = \mathsf{obs}(\mathsf{perc}_j(e_{j0}));$$

- $\langle \vec{e}', \vec{c}', \vec{g}', \equiv' \rangle \in \rho(\langle \vec{e}, \vec{c}, \vec{g}, \equiv \rangle, \vec{\alpha})$ iff
  - if $i \equiv j$ then $\alpha_i = \alpha_j$;
  - $e_i' = \delta_i(e_i, \alpha_i)$;
  - $c_i' = \delta_a(l_i, \alpha_i) \wedge \alpha_i \in \mathsf{poss}(l_i)$ with $l_i = \langle c_i, \mathsf{obs}(\mathsf{perc}_i(e_i)) \rangle$;
  - $g_i' = \gamma_i(g_i, e_i)$;
  - $i \equiv' j$ iff $i \equiv j \wedge \mathsf{obs}(\mathsf{perc}_i(e_i')) = \mathsf{obs}(\mathsf{perc}_j(e_j'))$.
- $W^{acc} = \{$
  $E^k \times Conf^k \times G_1^{acc} \times G_2 \times \ldots \times G_k \times \equiv \ , \ \ldots \ ,$
  $E^k \times Conf^k \times G_1 \times \ldots \times G_{k-1} \times G_k^{acc} \times \equiv \}$

The resulting automaton is similar to the previous one, and the same synthesis procedures apply, including the embedding of the strategy into an agent protocol. We also get the analogous soundness and completeness result and complexity characterization as before.

Example 23. *Consider again Example 21 but now we require, instead of having a single goal $\phi_{\mathcal{G}}$, three distinct goals over the three environments. In particular for the car-train environments Env_1 and Env_2, we adopt the same kind of goal as before, but avoiding certain cells for environments, e.g., $\phi_{\mathcal{G}_1} = (\Box\Diamond e_{11}) \wedge (\Box\Diamond e_{24}) \wedge \Box\neg(\langle c_{13}, c_{13}\rangle \vee \langle c_{23}, c_{23}\rangle) \wedge \Box\neg\langle e_{22}, e_t\rangle$ and $\phi_{\mathcal{G}_2} = (\Box\Diamond e_{21}) \wedge (\Box\Diamond e_{24}) \wedge \Box\neg(\langle c_{23}, c_{23}\rangle \vee \ldots \vee \langle c_{14}, c_{14}\rangle) \wedge \Box\neg\langle e_{11}, e_{14}\rangle$, whereas in the Wumpus world we only require to reach the gold after visiting initial position: $\phi_{\mathcal{G}_3} = \Box(e_{11} \rightarrow \Diamond e_{34}) \wedge \Box\neg(c_{23} \vee c_{31})$. It can be shown that a (perfect-recall) strategy for achieving such goals exists. In fact, there exists at least one strategy (e.g., one extending the prefix depicted in Figure 6.8 avoiding in Env_1 and Env_2 states mentioned above) that satisfies goal $\phi_{\mathcal{G}}$ over all environments as well as these three new goals (in particular, if a strategy satisfies $\phi_{\mathcal{G}}$ then it satisfies $\phi_{\mathcal{G}_3}$ too). Such a strategy can be transformed into an agent protocol, by enlarging the configuration space of the agent, as discussed in the previous section.* □

6.3.8 Representing strategies

The technique described in the previous section provides, if a solution does exist, the run of $\mathcal{A}_{\mathcal{P}}$ satisfying the given LTL specification. As discussed above such a run can be represented finitely. In this section, we exploit this possibility to generate a finite representation of the run that can be used directly as the strategy σ for the agent Ag. The strategy σ can be represented as a finite-state structure with nodes

labeled with agent's configuration and edges labeled with a *condition-action* rule $[o]\alpha$, where $o \in Obs$ and $\alpha \in Act_a$. The intended semantics is that a transition can be chosen to be carried on for environment Env_i only if the current observation of its internal state e_i is o, i.e. $o = \mathsf{obs}(\mathsf{perc}_i(e_i))$. Hence, notice that a strategy σ can be expressed by means of sequences, *case*-conditions and infinite iterations.

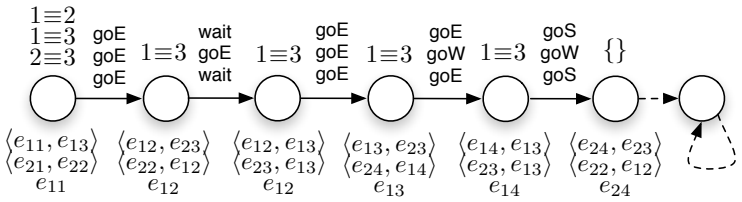


Figure 6.7. Accepting run r for Example 21.

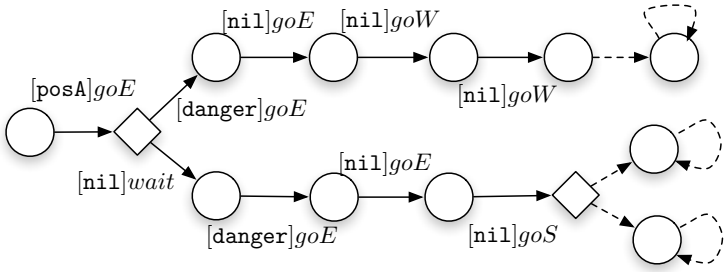


Figure 6.8. Corresponding G_r

In other words, it can be represented as a graph $G_r = \langle N, E \rangle$ where N is a set of nodes, $\lambda : N \to Conf$ its labeling, and $E \subseteq N \times \Phi \times N$ is the transition relation. G_r can be obtained by unpacking the run found as witness, exploiting the equivalence relation $\equiv$. More in details, let $r = w_0 \vec{a}_1 w_1 \ldots \vec{a}_n w_n \vec{a}_{n+1} w_m$ with $m \leq n$ be a finite representation of the infinite run returned as witness. Let r_ℓ be the ℓ-th state in r, whereas we denote with $r|_\ell$ the sub-run of r starting from state r_ℓ. A projection of r over a set $X \subseteq \{1, \ldots, k\}$ is the run $r(X)$ obtained by projecting away from each w_i all vector components and indexes not in X. We mark state w_m: $loop(w_\ell) = true$ if $\ell = m$, *false* otherwise.

$$G_r = \textsc{Unpack}(r, nil);$$

$\textsc{Unpack}(r, loopnode)$:

1: $N = E = \varnothing$;
2: be $r_0 = \langle \vec{e}, \vec{c}, \vec{g}, \equiv \rangle$;
3: **if** $loop(r_0) \wedge loopnode \neq nil$ **then**
4: **return** $\langle \{loopnode\}, E \rangle$;
5: **end if**
6: be $\vec{a}_1 = \langle \alpha_1, \ldots, \alpha_k \rangle$;
7: Let $X = \{X_1, \ldots, X_b\}$ be the partition induced by $\equiv$;
8: $node$ = new node;
9: **if** $loop(r_0)$ **then**
10: $loopnode = node$;

11: **end if**
12: **for** ($j = 1$; $j \leq b$; $j{+}{+}$) **do**
13: $G' = \textsc{Unpack}(r(X_j)|_1, loopnode)$;
14: choose i in X_j;
15: $\lambda(node) = c_i$;
16: $E = E' \cup \langle node, [\mathsf{obs}(\mathsf{perc}_i(e_i))]\alpha_i, root(G')\rangle$;
17: $N = N \cup N'$;
18: **end for**
19: **return** $\langle N \cup \{node\}, E\rangle$;

The algorithm above, presented in pseudocode, recursively processes run r until it completes the loop on w_m, then it returns. For each state, it computes the partition induced by relation $\equiv$ and, for each element in it, generates an edge in G_r labeled with the corresponding action α taken from the current action vector.
From G_r we can derive *finite state strategy* $\sigma_f = \langle N, succ, act, n_0\rangle$. where:

- $succ : N \times Obs \times Act_a \to N$ such that $succ(n, o, a) = n'$ iff $\langle n, [o]\alpha, n'\rangle \in E$;
- $act : N \times Conf \times Obs \to Act_a$ such that $\alpha = act(n, c, o)$ iff $\langle n, [o]\alpha, n'\rangle \in E$ for some $n' \in N$ and $c = \lambda(n)$;
- $n_0 = root(G_r)$, i.e., the initial node of G_r.

From σ_f we can derive an actual perfect-recall strategy $\sigma : Obs^* \to Act_a$ as follows. We extend the deterministic function $succ$ to observation histories $h \in Obs^*$ of length ℓ in the obvious manner. Then we define σ as the function: $\sigma(h) = act(n, c, last(h))$, where $n = succ(root(G_r), h^{n-1})$, $h^{\ell-1}$ is the prefix of h of length ℓ-1 and $c = \lambda(n)$ is the current configuration. Notice that such strategy is a partial function, dependent on the environment set $\mathcal{E}$: it is only defined for observation histories embodied by the run r.
It can be show that the procedure above, based on the algorithm Unpack, is correct, in the sense that the executions of the strategy it produces are the same as those of the strategy generated by the automaton constructed for Theorem 39.

Example 24. *Let us consider again the three environments, the agent and goal as in Example 21. Several strategies do exist. In particular, an accepting run r for $\mathcal{A}_\mathcal{P}$ is depicted in Figure 6.7, from which a strategy σ can be unpacked. Strategy σ can be equivalently represented as in Figure 6.8 as a function from observation histories to actions. For instance, $\sigma(c_0, \{\texttt{posA}, \texttt{nil}, \texttt{danger}, \texttt{nil}\}) = goE$. In particular, being all environments indistinguishable in the initial state (the agent receives the same observation* `posA`*), this strategy prescribes action goE for the three of them. Resulting states are such that both Env_1 and Env_3 provide perception* `nil`*, whereas Env_2 provides perception* `danger`*. Having received different observation histories so far, strategy σ is allowed to select different action for Env_2: goE for Env_2 and* `wait` *for Env_1 and Env_3. In fact, according to protocol* poss*, action* `wait` *is not an option for Env_2, whereas action goE is not significant for Env_3, though it avoids an accident in Env_1. In this example, by executing the strategy, the agent eventually*

receives different observation histories from each environment, but this does not necessary hold in general: different environments could also remain indistinguishable forever. □

There is still no link between synthesized strategies and agents. The main idea is that a strategy can be easily seen as a sort of an agents' protocol refinement where the states used by the agents are extended to store the (part of the) relevant history. This is done in the next section.

6.3.9 A notable variant

We have seen how it is possible to synthesize perfect-recall strategies that are function of the observation history the agent received from the environment. Computing such strategies in general results into a function that requires an unbounded amount of memory. Nonetheless, the technique used to solve the problem shows that *(i)* if a strategy does exist, there exists a bound on the information actually required to compute and execute it and *(ii)* such strategies are finite-state. More precisely, from the run satisfying the LTL specification, it is possible to build the finite-state strategy $\sigma_f = \langle N, succ, act, n_0 \rangle$. We now incorporate such a finite-state strategy into the agent protocol, by suitably expanding the configuration space of the agent to store in the configuration information needed to execute the finite state strategy. This amounts to define a new configuration space $\overline{Conf} = Conf \times N$ (hence a new local state space $\overline{L}$).
Formally, given the original agent $Ag = \langle Conf, Act_a, Obs, L, \mathsf{poss}, \delta_a, c_0 \rangle$ and the finite state strategy $\sigma_f = \langle N, succ, act, n_0 \rangle$, we construct a new agent $\overline{Ag} = \langle \overline{Conf}, Act_a, Obs, \overline{L}, \overline{\mathsf{poss}}, \overline{\delta}_a, \overline{c}_0 \rangle$ where :

- Act_a and Obs are as in Ag;
- $\overline{Conf} = Conf \times N$ is the new set of configurations;
- $\overline{L} = \overline{Conf} \times Obs$ is the new local state space;
- $\overline{\mathsf{poss}} : \overline{L} \to Act_a$ is an action-deterministic protocol defined as:

$$\overline{\mathsf{poss}}(\langle c, n \rangle, o) = \begin{cases} \{\alpha\}, & \text{iff } act(n, c, o) = \alpha \\ \varnothing, & \text{if } act(n, c, o) \text{ is undefined;} \end{cases}$$

- $\overline{\delta}_a : \overline{L} \times Act_a \to \overline{Conf}$ is the transition function, defined as:

$$\overline{\delta}_a(\langle \langle c, n \rangle, o \rangle, a) = \langle \delta_a(c, o), succ(n, o, a) \rangle;$$

- $\overline{c_0} = \langle c_0, n_0 \rangle$.

On this new agent the original strategy can be phrased as a state-base strategy:

$$\overline{\sigma} : \overline{Conf} \times Obs \to Act_a$$

simply defined as: $\overline{\sigma}(\langle c, n \rangle, o) = \overline{\mathsf{poss}}(\langle c, n \rangle, o)$.

It remains to understand in what sense we can think the agent $\overline{Ag}$ as a refinement or customization of the agent Ag. To do so we need to show that the executions allowed by the new protocol are also allowed by the original protocol, in spite of the fact that the configuration spaces of the two agents are different. We show this by relaying on the theoretical notion of *simulation*, which formally captures the ability of one agent (Ag) to simulate, i.e., copy move by move, another agent ($\overline{Ag}$).
Given the two agents Ag_1 and Ag_2, a *simulation relation* is a relation $\mathcal{S} \subseteq L_1 \times L_2$ such that $\langle l_1, l_2 \rangle \in \mathcal{S}$ implies that:

> if $l_2 \xrightarrow{\alpha} l_2'$ and $\alpha \in \mathsf{poss}_2(l_2)$ then there exists l_1' such that $l_1 \xrightarrow{\alpha} l_1'$ and $\alpha \in \mathsf{poss}_1(l_1)$ and $\langle l_1', l_2' \rangle \in \mathcal{S}$.

where $l_i \xrightarrow{\alpha} l_i'$ iff c_i' is the agent configuration in l_i' and $c_i' = \delta_a(l_i, \alpha)$. We say that agent Ag_1 *simulates* agent Ag_2 iff there exists a simulation relation $\mathcal{S}$ such that $\langle l_1^0, l_0^2 \rangle \in \mathcal{S}$ for each couple of initial local states $\langle l_1^0, l_2^0 \rangle$ with the same initial observation.

Theorem 41. Agent Ag simulates $\overline{Ag}$.

Proof. First, we notice that $\overline{\mathsf{poss}}(\langle c, n \rangle, o) \subseteq \mathsf{poss}(\langle c, o \rangle)$ for any $c \in Conf, o \in Obs$. In fact, since we are only considering allowed strategies, the resulting protocol $\overline{poss}$ is compatible with agent Ag. The result follows from the fact that original configurations are kept as fragment of both L and $\overline{L}$. Second, being both the agent and environments deterministic, the result of applying the same action α from states $\langle \langle c, n \rangle, o \rangle \in \overline{L}$ and $\langle c, o \rangle \in L$ are states $\langle \langle c', n' \rangle, o' \rangle$ and $\langle c', o' \rangle$, respectively.
Finally, assume towards contradiction that $\overline{Ag}$ is not simulated by Ag. This implies that there exists a sequence of length $n \geq 0$ of local states $l_0 \xrightarrow{\alpha_1} l_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_k} l_k$ of $\overline{Ag}$, where l_0 is some initial local state, and a corresponding sequence $\bar{l}_0 \xrightarrow{\alpha_1} \bar{l}_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_k} \bar{l}_k$ of $\overline{Ag}$, starting from a local state $\bar{l}_0$ sharing the same observation of l_0, such that $\alpha \in \overline{\mathsf{poss}}(\bar{l}_k)$ but $\alpha \notin \mathsf{poss}(l_k)$ for some α. For what observed before, l_k and $\bar{l}_k$ share the same agent's configuration; in particular, they are of the form $\bar{l}_k = \langle \langle c_k, n_k \rangle, o_k \rangle$ and $l_k = \langle c_k, o_k \rangle$. Hence $\overline{\mathsf{poss}}(\langle \langle c_k, n_k \rangle, o_k \rangle) \subseteq \mathsf{poss}(\langle c_k, o_k \rangle)$ and we get a contradiction. ∎

Theorem 42. $\overline{Ag}$ is nonblocking.

It follows from the fact that a strategy σ that is a solution for problem $\mathcal{P}$ is a prefix-closed function and it is allowed by Ag. Hence, for any $\bar{l} \in \overline{L}$ reachable from any initial local state by applying σ, we have $\overline{\mathsf{poss}}(\bar{l}) \neq \varnothing$.
From Theorem 1 and results in previous sections we have:

Theorem 43. Any execution of agent $\overline{Ag}$ over each environment Env_i satisfies the original agent specification Ag and the goal specification.

6.4 Discussion

We have investigated the synthesis of an agent's protocol to satisfy LTL specifications while dealing with multiple, partially-observable environments. In addition to the

computationally optimal procedure here introduced, we explored an automata-based protocol refinement for a perfect-recall strategy that requires only finite states.
There are several lines we wish to pursue in the future. Firstly, we would like to implement the procedure here described and benchmark the results obtained in explicit and symbolic settings against planning problems from the literature. We note that current model checkers such as MCMAS (Lomuscio et al., 2009) and MCK (Gammie and van der Meyden, 2004) support interpreted systems, the semantics here employed.
It is also of interest to explore whether the procedures here discussed can be adapted to other agent-inspired logics, such as epistemic logic (Ronald Fagin and Vardi, 1995). Epistemic planning (van der Hoek and Wooldridge, 2002), i.e., planning for epistemic goals, has been previously discussed in the agents-literature before, but synthesis methodologies have not, to our knowledge, been used in this context.
When dealing with LTL goals we need to consider that the agent cannot really monitor the achievement of the specification. Indeed every linear temporal specification can be split into a "liveness" part which can be checked only considering the entire run and a "safety" part that can be checked on finite prefixes of such runs (Baier and Katoen, 2008). Obviously the agent can look only at the finite history of observations it got so far, so being aware of achievement of LTL properties is quite problematic in general. This issue is related to runtime verification and monitoring (Eisner et al., 2003; Bauer et al., 2011), and in the context of AI, it makes particularly attractive to include in the specification of the dynamic property aspects related to the knowledge that the agent acquires, as allowed by interpreted systems.

Chapter 7

Synthesis via Model Checking for BDI agents

The formal verification of agent-oriented programs requires logic frameworks capable of representing and reasoning about agents' abilities and capabilities, and the goals they can feasibly achieve. In particular, we are interested here in programs written in the the family of Belief-Desire-Intention (BDI) agent programming systems (Bratman et al., 1988; Rao and Georgeff, 1992; Bordini et al., 2006), a popular paradigm for building multi-agent systems.
This is often related to the philosophical notion "intentional stance": it is a term coined by philosopher Daniel Dennett for the level of abstraction in which we view the behavior of a thing in terms of mental properties. It is part of a theory of mental content proposed by Dennett, which provides the underpinnings of his later works on free will, consciousness, folk psychology, and evolution:

> *"First you decide to treat the object whose behavior is to be predicted as a rational agent; then you figure out what beliefs that agent ought to have, given its place in the world and its purpose. Then you figure out what desires it ought to have, on the same considerations, and finally you predict that this rational agent will act to further its goals in the light of its beliefs."*

Traditional BDI logics based on CTL (e.g., (Rao and Georgeff, 1991)) are generally too weak for representing ability. They primarily allow to define constraints on rational behaviour, whereas they do not encode capabilities of agents and thereby leave a sizable gap between agent-oriented programs and their formal verification. Recent work (e.g., (Alechina et al., 2007; 2008; Dastani and Jamroga, 2010)) has better bridged the gap between formal logic and practical programming by providing an axiomatisation of a class of models that is designed to closely model a programming framework (in the cited case, Simple-APL). However, this is done by restricting the logic's models to those that satisfy transition relations corresponding to the agents' plans, as defined by the semantics of the programming language itself. In such a framework, it is not possible to reason about the agent's know-how capabilities and what the agent could achieve *if she had* specific capabilities and/or goals.

Our aim is to define a framework that will allow us to speculate about a group of agents' capabilities, i.e. to reason about what they can achieve under the BDI agent paradigm. Observe how these capabilities can be see as external, shared components that to which agents have access. Indeed, this approach allows us to abstract plans written by different programmers to be combined and used in real-time, under the principles of practical reasoning (Bratman et al., 1988).

Considering then a set of BDI agents, each with a certain specified set of capabilities and goals, we want to verify whether they can achieve (have a collective strategy for) a given goal and, possibly, synthesize it. In particular, we want to reason about what could that coalition achieve (regardless of what other agents outside the coalition may do). The prototypical strategic logic for reasoning about such coalitions' capability is ATL (see, e.g., Chapters 3). Still, since we are considering BDI agents, also desires and capabilities have to be taken into account.

This requires the ability to be able to express capabilities directly in our logic. As pointed out by (Walther et al., 2007), standard ATL does not allow agents' strategies to be explicitly represented in the syntax of the logic.

In (Walther et al., 2007) authors introduce ATLES, i.e., an extension to ATL with Explicit Strategies. ATLES extends ATL by allowing strategy terms in the language: $\langle\langle A\rangle\rangle_{\rho}\varphi$ holds if coalition A has a joint strategy for ensuring φ assuming that some agents are committed to specific strategies as specified by commitment function ρ.

In (Yadav and Sardiña, 2012) authors go further and develop a framework—called BDI-ATLES—so that the strategy terms are tied directly to the plans available to agents under the notion of practical reasoning embodied by BDI paradigm (Bratman et al., 1988; Rao and Georgeff, 1992):

> the only strategies that can be employed by a BDI agent are those that ensue by the rational execution of its predefined plans, given its goals and beliefs

The key construct $\langle\langle A\rangle\rangle_{\omega,\varrho}\varphi$ in the new framework states that coalition A has a joint strategy for ensuring φ, under the assumptions that some agents in the system are BDI-style agents with capabilities and (initial) goals as specified by assignments ω and ϱ, respectively. For instance, in the Gold Mining domain from the International Multi-Agent Contest, one may want to verify if two miner agents programmed in a BDI-style language can successfully collect gold pieces when equipped with capabilities (i.e., plans) for navigation and communication and want to win the game, while the opponent agents can perform any physically legal action. More interesting, a formula like $\langle\langle A\rangle\rangle_{\varnothing,\varnothing}\varphi \supset \langle\langle A\rangle\rangle_{\omega,\varrho}\varphi$ can be used to check whether a coalition A has enough know-how (and motivations) to carry out a task φ that is indeed (physically) feasible for the coalition.

In the reminder of this chapter, we first briefly introduce the BDI programming paradigm, then extend the BDI-ATLES logic (Yadav and Sardiña, 2012) to a more general setting, then present a technique for verifying and synthesizing rational strategies for BDI agents.

7.0.1 BDI Programming

The BDI agent-oriented programming paradigm is a popular and successful approach for building agent systems, with roots in philosophical work on rational action (Bratman et al., 1988). There are a number of programming languages and platforms in the BDI tradition, such as AgentSpeak/Jason, PRS, Jack, JADEX, and 3APL/2APL (Bordini et al., 2006).
In particular, we envision BDI agents defined with a set of *goals* and so-called *capabilities* (Busetta et al., 1999; Padgham and Lambrix, 2005). Generally speaking, a capability is a set/module of procedural knowledge (i.e., plans) for some functional requirement. An agent may have, for instance, the Nav capability encoding all plans for navigating an environment. Equipped with a set of capabilities, a BDI agent executes actions as per plans available so as to achieve her goals, e.g., exploring the environment.
Indeed, an agent in a BDI system continually tries to achieve its goals/desires by selecting an adequate plan from the *plan library* given its current beliefs, and placing it into the *intention base* for execution. The agent's plan library Π encodes the standard operational knowledge of the domain by means of a set of *plan-rules* (or "recipes") of the form $\phi[\vec{\beta}]\psi$: *plan* $\vec{\beta}$ *is a reasonable plan for achieving* ψ *when (context) condition* ϕ *is believed true*. Though different BDI languages offer different constructs for crafting plans, most allow for sequences of domain actions that are meant to be directly executed in the world (e.g., lifting an aircraft's flaps), and the posting of (intermediate) *sub-goals* $!\varphi$ (e.g., obtain landing permission) to be resolved. The intention base, in turn, contains the current, partially executed, plans that the agent has already *committed* to for achieving certain goals. Current intentions being executed provide a screen of admissibility for attention focus (Bratman et al., 1988).
Important, also, is the usual plan/goal *failure mechanism* typical of BDI architectures, in which alternative plans for a goal are tried upon failure of the current plan. Plan failure could happen due to an action precondition not holding or the non-availability of plans for a sub-goal. If alternative plans for a goal are not available, then failure is propagated towards higher-level goals/ and plans.
Though we do not present it here for lack of space, most BDI-style programming languages come with a clear single-step semantics basically realizing (Rao and Georgeff, 1992)'s execution model in which (rational) behavior arises due to the execution of plans from the agent's plan library so as to achieve certain goals relative to the agent's beliefs.

7.0.2 ATL and ATLES Logics of Coalitions

Alternating-time Temporal Logic (ATL) (Alur et al., 2002) is a logic for reasoning about the ability of agent coalitions in *multi-agent game structures*. This logic was already introduced in Section 2.4, so refer to that section for a detailed introduction. The main point of ATL, in essence, is constituted by the coalition modalities. Recall that, given a path formula ϕ and an ATL transition system $\mathcal{M} = \langle \mathcal{A}, Q, P, \mathcal{V}, \delta \rangle$ (ATS) (refer to Section 2.4), we say that

$\mathcal{M}, q \vDash \langle\langle A \rangle\rangle \phi$ *iff* there is a collective strategy F_A such that for all computations

$\lambda \in \mathsf{out}(q, F_A)$, we have $\mathcal{M}, \lambda \vDash \phi$.

The coalition modality only allows for implicit (existential) quantification over strategies. In some contexts, though, it is convenient and even necessary to refer to strategies explicitly in the language (e.g., can a player win the game if the opponent plays a specified strategy?). To address this limitation, (Walther et al., 2007) proposed ATLES, an extension of ATL where the coalition modality is extended with $\langle\langle A \rangle\rangle_\rho$, where ρ is a *commitment function*, that is, a partial function mapping each agent $a \in \mathcal{A}$ to a member of a set of so-called *strategy terms* Υ_a. Hence, considering the set of all strategy terms $\Upsilon = (\bigcup_{a \in \mathcal{A}} \Upsilon_a)$, a commitment is a function $\rho : \mathcal{A} \to \Upsilon$. The idea is that each agent $a \in \mathcal{A}$, for which $\rho(a)$ is defined, commits to the specific strategy $\rho(a)$. As for standard ATL, a strategy is a function of the form $f : Q^+ \to 2^Q$ (or, equivalently, $f : Q^+ \to Act$ for concurrent game structures). To give meaning to the extended coalition modality, the semantics of ATL is extended with a mapping $\| \cdot \| : \Upsilon \to (Q^+ \to 2^Q)$ from strategy terms to actual ATL strategies. Hence a collective strategy f_A for $A \subseteq \mathcal{A}$ *agrees* with ρ if $f_a = \rho(a)$ for each $a \in \mathcal{A}$ such that $\rho(a)$ is defined.
The set of ATLES formulas is generated by the following grammar, where $p \in P$ and A ranges over coalitions (Walther et al., 2007):

$$\varphi := \ p \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle\langle A \rangle\rangle_\rho \bigcirc \varphi \mid \langle\langle A \rangle\rangle_\rho \Box \varphi \mid \langle\langle A \rangle\rangle_\rho \, \varphi \mathcal{U} \varphi.$$

Hence we say that:

$\mathcal{M}, q \vDash \langle\langle A \rangle\rangle_\rho \varphi$ *iff* there is a collective strategy F_A, agreeing with ρ, such that for all computations $\lambda \in \mathsf{out}(q, F_A)$, we have $\mathcal{M}, \lambda \vDash \varphi$.

7.0.3 BDI-ATLES: a logic for BDI Agents

In this section, we introduce the BDI-ATLES logic which was first presented in (Yadav and Sardiña, 2012). It pushes forward the aim of ATLES of making strategies explicit in the language, assuming that the agent system is built adopting the BDI agent-oriented paradigm. The main intent is to bridge the gap between verification frameworks and BDI agent-oriented programming languages, so that BDI programmers can encode ATL models for a given BDI application in a principled manner and reason about what agents can achieve at the level of goals and capabilities. Also, as ATLES, this logic has the ability to reason both about what is feasible and what is possible relative to agents' plan libraries. Indeed, recall that ATLES uses strategies to denote the agent's choices among possible actions. For BDI agents these strategies are *implicit* in her know-how from the plan library.
Given BDI plan of the form $\phi[\vec{\alpha}]\psi$, we use α_i to denote the i-th step of α and $|\alpha|$ the length of α. In (Yadav and Sardiña, 2012) authors introduce a technique for synthesizing rational strategies for agents; however, they restrict to the case in which BDI plans α are composed of primitive actions only, and do not consider subgoals. In this chapter instead, we generalize the approach of (Yadav and Sardiña, 2012) to the case of action sequences.

BDI-ATL syntax

The language of BDI-ATLES is defined over a finite set of atomic propositions P, a finite set of agents $\mathcal{A}$, and a finite set of capability terms $\mathcal{C}$ available in the BDI application of concern.

Plan libraries We denote with $\mathbf{\Pi}^P_{Act}$ the (infinite) set of all possible plan-rules given a set of actions Act and a set of domain propositions P. A plan library Π is one of its finite subsets.

Capability terms Consider now a set $\mathcal{C}$ of capability terms. Intuitively, each capability term $c \in \mathcal{C}$ (e.g., NAV) stands for a plan library $\Pi^c \subset \mathbf{\Pi}^P_{Act}$ (e.g., Π^{NAV}). This mapping will be formalized later.

Capability assignments A *capability assignment* ω consists of a set of pairs of agents with their capabilities of the form $\langle agt : \{c_1, \ldots, c_\ell\}\rangle$, where $agt \in \mathcal{A}$ and $c_i \in \mathcal{C}$.

Goal assignments A *goal assignment* ϱ, in turn, defines the goal base (i.e., set of propositional formulas) for some agents, and is a set of tuples of the form $\langle agt : \{\gamma_1, \ldots, \gamma_\ell\}\rangle$, where $agt \in \mathcal{A}$ and γ_i are boolean formulas over P.

Given a capability assignment ω (resp., goal assignment ϱ), we denote $\mathcal{A}_\omega \subseteq \mathcal{A}$ (resp., $\mathcal{A}_\varrho \subseteq \mathcal{A}$) the set of agents for which their capabilities (resp., goal bases) are defined by assignment ω (resp., ϱ), that is, $\mathcal{A}_\omega = \{agt \mid \langle agt : \{c_1, \ldots, c_n\}\rangle \in \omega\}$ (resp., $\mathcal{A}_\varrho = \{agt \mid \langle agt : \{\gamma_1, \ldots, \gamma_n\}\rangle \in \varrho\}$). As espected, we require that ω and ϱ are such that $\mathcal{A}_\omega = \mathcal{A}_\varrho$ and, Informally, we will ofter refer to such agents as "BDI agents", in contrast with standard agents that are not specified in the BDI paradigm.

Formulae The set of BDI-ATLES formulas follow the same grammar as in ATL(ES), except that coalition formulas are now of the form $\langle\langle A\rangle\rangle_{\omega,\varrho}\varphi$, where φ is a path formula (i.e., it is preceded by $\bigcirc$, $\Box$, or $\mathcal{U}$), A is a coalition and ω and ϱ are, respectively, a capability and a goal assignment. Its intended meaning is as follows:

> $\langle\langle A\rangle\rangle_{\omega,\varrho}\varphi$ expresses that coalition of agents A can jointly force temporal condition φ to hold when (rational) BDI agents in $\mathcal{A}_\omega$ (or $\mathcal{A}_\varrho$) are equipped with the capabilities as per assignment ω and their (initial) goals are as described by the goal assignment ϱ.

Notice that we require, in each coalition (sub)formula, that the agents for which capabilities and goals are assigned to be the same.

BDI-ATL semantics

Next, we provide the semantics of the logic as in (Yadav and Sardiña, 2012). For clarity, as customary, we will use actions in Act instead of move indexes: wlog we can associate some enumeration of its elements, hence given a game structure

$\mathcal{M} = \langle \mathcal{A}, Q, P, \mathcal{V}, d, \delta, \Theta \rangle$, we will assume $d(q) \subseteq Act$ for each $q \in Q$. Accordingly, we will consider strategy functions of the form $f : Q^+ \to Act$.
Given as input a set of capability terms $\mathcal{C}$ and a plan library $\Pi \subset \mathbf{\Pi}^P_{Act}$, a game structure for BDI-ATLES is a concurrent game structure $\mathcal{M} = \langle \mathcal{A}, Q, P, \mathcal{V}, d, \delta, \Theta \rangle$ where:

- $\mathcal{A}$, Q, P, d, $\mathcal{V}$ and δ are, respectively, the standard sets of agents, states, propositions, labeling function, available moves and transition function as in ATL.
- There is a distinguished dummy action $\textsc{noOp} \in Act$ available to all agents in all states (i.e., $\textsc{noOp} \in d_{agt}(q)$, for all $agt \in \mathcal{A}$ and $q \in Q$), and such that the system remains still when all agents perform such action (i.e., $\delta(q, \langle \textsc{noOp}, \ldots, \textsc{noOp} \rangle) = q$, for all $q \in Q$);
- The capability function $\Theta : \mathcal{C} \mapsto \mathbf{\Pi}^P_{Act}$ maps capability terms to their set of plans.

Thus, BDI-ATLES models are similar to those of ATLES, except that capability interpretations are used instead of strategy term interpretations. Generally speaking, our task then will be to characterize what the (low-level) strategies for rational agents with certain capabilities and goals are. We call such strategies *rational strategies*, in that they are compatible with the standard BDI rational execution model (Rao and Georgeff, 1992). So, given an agent $agt \in \mathcal{A}$, a plan-library Π, and a goal base $\mathcal{G}$, let $\Sigma^{agt}_{\Pi,\mathcal{G}}$ be the set of standard ATL strategies for agent agt in $\mathcal{M}$ that are *rational* strategies when the agent is equipped with plan-library Π and has $\mathcal{G}$ as initial goals. What exactly this set should be is, of course, far from trivial and it is addressed at the end of this section. Assuming then that such set has been suitably defined, we shall develop next the semantics for formulas of the form $\langle\langle A \rangle\rangle_{\omega,\varrho} \varphi$.
So, first we extend—following ATLES—the notion of a joint strategy for a coalition to that of joint strategy under a given capability and goal assignments. Given a capability (goal) assignment ω (ϱ) and an agent $agt \in \mathcal{A}_\omega$ ($agt \in \mathcal{A}_\varrho$), we denote agt's capabilities (goals) under ω (ϱ) by $\omega[agt]$ ($\varrho[agt]$).

BDI-ATLES strategies Intuitively, an $\langle \omega, \varrho \rangle$-*strategy for* A is a joint strategy for coalition A such that agents in $A \cap \mathcal{A}_\omega$ only follow "rational" (plan-goal compatible) strategies as per their ω-capabilities and ϱ-goals, and agents in $A \backslash \mathcal{A}_\omega$ follow arbitrary strategies. Formally, an $\langle \omega, \varrho \rangle$-strategy for coalition A (with $\mathcal{A}_\omega = \mathcal{A}_\varrho$) is a collective strategy F_A for agents A such that for all $f_{agt} \in F_A$ with $agt \in A \cap \mathcal{A}_\omega$, it is the case that $f_{agt} \in \Sigma^{agt}_{\Pi,\mathcal{G}}$, where $\Pi = \cup_{c \in \omega[agt]} \Theta(c)$ and $\mathcal{G} = \varrho[agt]$. Note no requirements are asked on the strategies for the remaining agents $A \backslash \mathcal{A}_\omega$, besides of course being legal (ATL) strategies.

Using the notion of $\langle \omega, \varrho \rangle$-strategies and that of possible outcomes for a given collective strategy from ATL (refer to function $\mathsf{out}(\cdot, \cdot)$ from the second section), we are now able to state the meaning of formulas in BDI-ATLES, in particular of coalition formulas, e.g.:

$$\mathcal{M}, q \vDash \langle\langle A\rangle\rangle_{\omega,\varrho}\varphi \textit{ iff } \text{there is a } \langle\omega,\varrho\rangle\text{-strategy } F_A \text{ for } A \text{ such that for all } \langle\omega,\varrho\rangle\text{-strategies } F_{\mathcal{A}_\omega \setminus A} \text{ for } \mathcal{A}_\omega \setminus A, \text{ it is the case that } \mathcal{M}, \lambda \vDash \varphi, \text{ for all paths } \lambda \in \mathsf{out}(q, F_A \cup F_{\mathcal{A}_\omega \setminus A}).$$

where φ ought to be a path formula. Intuitively, F_A stands for the collective strategy of agents A that guarantees the satisfaction of formula φ. Because F_A is a $\langle\omega,\varrho\rangle$-strategy, some agents in A, namely, those whose capabilities and goals are defined by ω and ϱ, respectively, are to follow strategies that correspond to a rational execution of its capabilities. At the same time, because other agents outside the coalition A could have also been assigned capabilities and goals, the chosen collective strategy F_A needs to work no matter how those agents (namely, agents $\mathcal{A}_\omega \setminus A$) behave, as long as they do it rationally given their plans and goals. That is, F_A has to work with *any* rational collective strategy $F_{\mathcal{A}_\omega \setminus A}$. Finally, the behavior of all remaining agents—namely those in in $\mathcal{A} \setminus (A \cup \mathcal{A}_\omega)$—are taken into account when considering all possible outcomes, after all strategies for agents in $A \cup \mathcal{A}_\omega$ have been settled.

While similar to ATLES coalition formulas $\langle\langle A\rangle\rangle_\rho\varphi$, BDI-ATLES coalition formulas $\langle\langle A\rangle\rangle_{\omega,\varrho}\varphi$ differ in one important aspect that makes its semantics more involved. Specifically, whereas commitment functions ρ prescribe *deterministic* behaviors for agents, capabilities and goals assignments yield multiple potential behaviors for the agents of interest. This nondeterministic behavior steams from the fact that BDI agents can choose what goals to work on at each point and what available plans to use for achieving such goals. Technically, this is reflected in that the strategies for each agent in $(\mathcal{A}_\omega \setminus A)$—that is, those agents with assigned capabilities and goals but are not part of the coalition—cannot be (existentially) considered together with those of agents in A or (universally) accounted for via the possible outcomes function $\mathsf{out}(\cdot,\cdot)$, as such function puts no rationality constraints on the remaining (non-committed) agents. Thus, whereas agents in $A \cap \mathcal{A}_\omega$ are allowed to select one possible rational behavior, all rational behaviors for agents in $(A_\omega \setminus A)$ need to be taken into considered.

This basically concludes the semantics of BDI-ATLES, though predicated in the set $\Sigma^{agt}_{\Pi,\mathcal{G}}$ of rational strategies. Let us next formally characterize such important set.

BDI-ATLES Rational Strategies $\Sigma^{agt}_{\Pi,\mathcal{G}}$. $\Sigma^{agt}_{\Pi,\mathcal{G}}$ is defined as the set of rational strategies for an agent *agt* when equipped with plan-library Π and has $\mathcal{G}$ as initial goals.

The formal definition is detailed in Appendix A, which is the authors' extension to sequences of the corresponding definitions in (Yadav and Sardiña, 2012), where only atomic plans were considered. It defines when, in the context of this logic, a strategy is *rational*. The tractation is quite technical and not essential for understanding the approach here proposed. Informally, the first step is to define *rational traces*, i.e., sequences of states and action of a concurrent game structure that "agree" with the goal and capacity assignments ρ, ω specified for the subset of BDI agents $\mathcal{A}_\omega = \mathcal{A}_\rho$. Informally, a trace $\lambda^+ = q_0\alpha_1 q_1 \cdots \alpha_\ell q_\ell$ is rational when every agent in $\mathcal{A}_\omega$ acts rationally, i.e., according to such assignments. This implies that every action α_i at location i can be "explained" by an plan-rule $\phi[\vec{\alpha}]\psi$. Moreover, such plan rule has to meet the two core notions in BDI programming; intuitively:

- it is *relevant* at every location $\lambda^+[i]$, i.e., the agent currently has a goal γ such that $\psi \vDash \gamma$.
- it is *applicable* at every location $\lambda^+[i]$, that it, $\mathcal{V}(\lambda^+[i]) \vDash \phi$.

Hence an $\langle \omega, \varrho \rangle$-strategy is a *rational strategy* in $\Sigma^{agt}_{\Pi,\mathcal{G}}$ (when the agent is equipped with plan-library Π and has $\mathcal{G}$ as initial goals) iff it induces only rational traces. For more details, refer to Appendix A.

7.1 Model Checking BDI-ATLES

In this section we present a sound and complete technique for synthesizing rational strategies for achieving a BDI-ATLES coalition formula $\langle\!\langle A \rangle\!\rangle_{\omega,\varrho}\ \varphi$, where φ is a path formula. This approach can be summarized in two steps: first, build a *rational* extended model $\mathcal{M}_{\omega,\rho}$ corresponding to $\mathcal{M}$, then perform standard ATL model-checking.

7.1.1 Extended Model $\mathcal{M}_{\omega,\rho}$

States of such an extended model hold information about the original state, current execution of plans (*program counter*), and the current goal set. Given a plan library Π where m is the number of plans and ℓ is the maximum plan length, we denote with $PlanPos$ the set of all possible couples in $\{0,\dots,m\} \times \{1,\dots,\ell\}$. Assuming a total order among plans, a couple $\langle \pi, p \rangle \in PlanPos$ means that plan $\pi = \phi[\vec{\alpha}]\psi \in \Pi$ is at step p, i.e., action α_i have been executed for each $i \leq p \leq |\vec{\alpha}|$. We will make use of these program counters to store possible "*explanations*" for the execution of actions. Intuitively, given a physical state of $\mathcal{M}$ and an action vector, we want to scan the plan library Π in order to find all applicable, relevant plans for each agent, so as to "explain" the execution of those actions. The successor state will be marked with such explanations. Iterating this procedure, we will check whether such explanations are still plausible for the next action or have to be dropped. If all explanations are dropped and new ones can't be found (an empty program counter for some agent), then we can conclude that the agent did not act rationally. The key point of this approach is that we don't need to store more information than the current program counter for every agent, i.e., the rationality check can be performed step-by-step. Notably, we are not committed to *all* explanations as long as we can find at least one for the last action.
To this end, we define an extended state space: $W = Q \times PC^{|\mathcal{A}_\omega|} \times (2^\Gamma)^{|\mathcal{A}_\omega|}$ where Q is original game state-space, Γ is the goal base, and $PC = 2^{PlanPos}$. Finally, given $\mathcal{M} = \langle \mathcal{A}, Q, P, \mathcal{V}, d, \delta, \Theta \rangle$, we define the corresponding rational model $\mathcal{M}_{\omega,\rho}$ as follows. We will also make use, with respect to such a model, of the notions of path, trace and strategy as for $\mathcal{M}$.
$\mathcal{M}_{\omega,\rho} = \langle \mathcal{A}, W, P, \mathcal{V}, d', succ, \Theta \rangle$ where:

- $\mathcal{A}, \mathcal{P}, Act, \mathcal{V}, d'$ are as in $\mathcal{M}$;
- W is the extended state space as above;
- $succ : W \times Act^n \to W$ is the transition function, defined below.

For readability, given an extended state $w = \langle q, pc_1, \ldots, pc_{|\mathcal{A}_\omega|}, \Gamma_1, \ldots, \Gamma_{|\mathcal{A}_\omega|}\rangle$, we define the following range restricted functions:

- $\mathsf{state}(w) = q$,
- $\mathsf{pc}_{agt}(w) = pc_{agt}$,
- $\mathsf{plans}_{agt}(w) = \{\pi \mid \langle \pi, p\rangle \in \mathsf{pc}_{agt} \text{ for some } p\}$
- $\mathsf{step}_{\pi,agt}(w) = \{p \mid \langle \pi, p\rangle \in \mathsf{pc}_{agt}\}$
- $\mathsf{goals}_{agt}(w) = \Gamma_{agt}$

Hence a plan $\pi = \phi[\vec{\beta}]\psi$ is relevant in w for agt iff it satisfies the conditions:

1. $\exists \gamma \in \mathsf{goals}_{agt}(w)$ s.t. $\gamma \vDash \psi$ — an active goal justifies π;
2. $\mathcal{V}(\mathsf{state}(w)) \nvDash \psi$ — the effects of π are not yet true.

Observe that the second condition extends the original definition of relevant plan by requiring the so-called "belief-goal consistency": we require no agent ever wants something already true.
Finally, we can formalize the transition function. Observe that this definition implies that we can always compute successor states "locally", i.e., we don't need to look at any other state in $\mathcal{M}_{\omega,\rho}$. Hence, $w' = succ(w, \vec{\alpha})$ is the (unique) state satisfying the following conditions:

1. $\alpha_{agt} \in d(\mathsf{state}(w))$ for any $agt \in \mathcal{A}$, i.e., the action vector $\vec{\alpha}$ is legal: each α_{agt} is a feasible move for agt in state $\mathsf{state}(w)$ according to $\mathcal{M}$. If the agent does not have any active goal, then NOOP is the only allowed action:

$$\forall agt \in \mathcal{A}_\omega \ (\mathsf{goals}_{agt}(w) = \varnothing \rightarrow \alpha_{agt} = \text{NOOP})$$

2. $\mathsf{state}(w') = \delta(\mathsf{state}(w), \vec{\alpha})$, i.e., the update of the state original component conforms to $\mathcal{M}$;
3. $\mathsf{goals}_{agt}(w') = \mathsf{goals}_{agt}(w) \backslash \{\gamma \mid \mathcal{V}(\mathsf{state}(w')) \vDash \gamma\}$ for each $agt \in \mathcal{A}_\omega$: we update the goal set of each agent agt by removing achieved goals;
4. For each BDI agent having at least a relevant goal and at least an active plan (explanation for it), then update the plan's program counter (step) if the plan is not completed, otherwise find a new plan to explain the current action.

   $\forall agt \in \mathcal{A}_\omega$ s.t. $\mathsf{plans}_{agt}(w) \neq \varnothing$ and $\mathsf{goals}_{agt}(w) \neq \varnothing$, then

$$\langle \pi, p\rangle \in \mathsf{pc}_{agt}(w')$$

   iff (exactly) one of these conditions holds (below $\pi = \phi[\vec{\beta}]\psi$):

   (a) π is the same plan as the previous step, it is still relevant and not completed – the step is progressed:
   $\mathsf{step}_{\pi,agt}(w) = (p-1) \wedge \alpha_{agt} = \vec{\beta}_p \wedge p \leq |\vec{\beta}|$ and π is relevant;

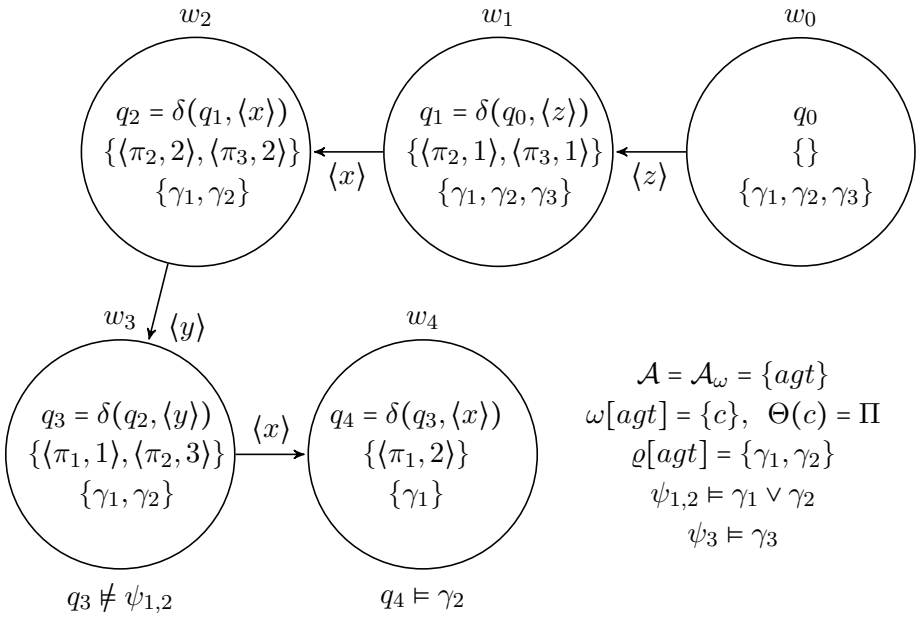


Figure 7.1. a computation fragment for a single agent agt equipped with plan library $\Pi = \{\pi_1 : \phi_1[yxz]\psi_1, \pi_2 : \phi_2[zxyzy]\psi_2, \pi_3 : \phi_3[zx]\psi_3\}$

(b) π is a new plan: some other plan π' was either completed or not relevant anymore in w:

i. $p = 1$ and $\alpha_{agt} = \vec{\beta}_1$, i.e., π is started;

ii. $\pi \in \bigcup_{c \in \omega[agt]} \Theta(c)$, i.e. agt can execute π;

iii. π is relevant and applicable in $\mathsf{state}(w)$;

iv. $\exists \pi' = \phi'[\vec{\beta}]\psi' \in \mathsf{plans}_{agt}(w)$ s.t. either:

- $\mathsf{step}_{\pi', agt}(w) = |\vec{\beta}|$, i.e. π' was completed, or
- $\pi' \neq \pi$ and π' is no more relevant (dropped);

Items 1 – 3 take into account, respectively, action feasibility, physical evolution in the original model $\mathcal{M}$, and the update of the goal $q \not\models \psi_{1,2}$ set. Item 4.*a* considers the case of plan progression: since we are not allowing for interleaving of plan executions, we check whether possible explanations found in w are still plausible (hence progressing the program counter) or they have to be dropped. 4.*b* checks whether some new plan π can be started in w to explain action α_{agt}. Indeed, this is possible only if some plan π' was just finished or dropped: we rule out the case of a new applicable plan, under these conditions, since we do not allow for interleaving.

Notice that in the same state w we can have different couples $\langle \pi, p_1 \rangle$, $\langle \pi, p_2 \rangle \in \mathsf{pc}_{agt}(w)$, with $p_1 \neq p_2$. This means that we are taking into account more than one explanation.

7.1.2 Model checking BDI-ATLES

Since, as discussed above, checking rationality while looking for the existence of winning strategies does not require any additional memory (but it can be done in a step-wise fashion), we can then perform ATL model-checking of $\Phi = \langle\langle A \rangle\rangle_{\omega,\varrho} \varphi$ over $\mathcal{M}_{\omega,\rho}$. Indeed, it is well known that memory plays no role for standard ATL. The model checking returns the set of states from which there exists a winning strategy for Φ, denoted with $[\Phi]_{\mathcal{M}_{\omega,\rho}}$. The algorithm can be easily modified so as to return the collective strategy found as witnesses, as memory is not required. We denote with F_A^{rat} one such strategy, i.e., F_A^{rat} is a set of strategies f_{agt}^{rat}, one for each agent $agt \in A$. Obviously, $[\Phi]_{\mathcal{M}_{\omega,\rho}} \neq \varnothing$ iff F_A^{rat} exists.
Finally, we relate the strategies found for $\mathcal{M}_{\omega,\rho}$ to strategies for $\mathcal{M}$: to each strategy F_A^{rat} corresponds a collective strategy $F_A = \{f_{agt} \mid agt \in A\}$ for $\mathcal{M}$, where each $f_{agt} : Q^+ \mapsto Act$ is a prefix-closed function built as follows:

- $f_{agt}(q) = f_{agt}^{rat}(w)$ iff $q = \mathsf{state}(w)$ and $\mathsf{pc}_{agt}(w) = \varnothing \wedge \mathsf{goals}_{agt}(w) = \Gamma_{agt}$ for any $agt \in \mathcal{A}_\omega$
- $f_{agt}(q_0, \ldots, q_k) = f_{agt}^{rat}(w_0, \ldots, w_k)$ iff $q_i = \mathsf{state}(w_i)$ and $w_k = succ(w_{k-1}, \vec{\alpha})$ for some action vector $\vec{\alpha}$ with $\alpha_{agt} = f_{agt}(w_0, \ldots, w_{k-1})$ $\forall agt \in \mathcal{A}$, $k > 0$.

Hence, given $[\Phi]_{\mathcal{M}_{\omega,\rho}} \subseteq W$, we denote with $[\Phi]_{\mathcal{M}}$ the set of states $q \in Q$ for which there exists a strategy F_A^{rat} such that F_A is defined, i.e., $f_{agt}(q)$ is defined for each $agt \in A$.
Finally, we relate each trace $\lambda_2^+ = GetTrace(\lambda_2, f_{agt}^{rat})$, for each $\lambda_2 \in \mathsf{out}(w, f_{agt}^{rat}) \in W^*$ in $\mathcal{M}_{\omega,\rho}$, to a trace $\lambda_1^+ = GetTrace(\lambda_1, f_{agt})$, for each $\lambda_1 \in \mathsf{out}(q, F_A) \in Q^*$ in $\mathcal{M}$. We will denote this mapping $\lambda_2^+ \rightsquigarrow \lambda_1^+$. Notice that it implies that $\lambda_1^+\langle i \rangle = \lambda_2^+\langle i \rangle$ and $\lambda_1[i] = \mathsf{state}_{agt}(\lambda_2[i])$ for all $0 \leq i \leq |\lambda_2^+|$.

Theorem 44 (Soundness and completeness)**.** $f_{agt}^{rat} \in F_A^{rat}$ iff the corresponding f_{agt} as above is such that *(i)* it is rational and *(ii)* $F_A = \{f_{agt} \mid agt \in A\}$ satisfies the ATL formula φ in $\mathcal{M}$, i.e., $\mathcal{M}, \lambda \vDash \varphi$ for each $\lambda \in \mathsf{out}(q, F_A)$ and $q \in [\Phi]$.

Proof. ($\Rightarrow$) Consider soundness. First, we prove that f_{agt}^{rat} is rational wrt $\mathcal{M}_{\omega,\rho}$ ($agt \in \mathcal{A}_\omega$). It implies that for every state $w \in [\Phi]$ we have that every $\lambda^+ = GetTrace(\lambda, f_{agt}^{rat})$, for $\lambda \in \mathsf{out}(w, f_{agt}^{rat})$, is a rational trace. For λ^+ to be rational, we need to build, for each $agt \in \mathcal{A}_\omega$, a marking function g_Γ, under which the trace can be broken down into ℓ segments that are all goal-complete for some active goal, except the last one that may just be a goal-working segment for a goal. $\lambda^+[s:e]$ can be divided into segments $\langle s_1 : e_1 \rangle \cdots \langle s_\ell : e_\ell \rangle$ such that $s_1 = s$ and $e_\ell \leq e$, and for all $i \in \{1, \ldots, \ell\}$. We show now that is always possible to find such indexes. Intuitively, we know that λ^+ is such that for each state in it we have a non-empty program counter $\mathsf{pc}_{agt}(w)$, i.e., an explanation for reaching that state. Formally, if $\mathsf{goals}_{agt}(\lambda^+[i \geq 0]) \neq \varnothing$ then for each $w = \lambda^+[i > 0]$ we have a non-empty program counter $\mathsf{pc}_{agt}(w)$, i.e., an explanation for reaching that state executing action $\lambda^+\langle i-1 \rangle$. Indeed, according to the definition of function *succ*, if $\mathsf{goals}_{agt}(\lambda^+[i]) \neq \varnothing$ then every successor $w' = \lambda^+[i+1]$ is such that either some relevant plan in $\mathsf{pc}_{agt}(w')$

can be progressed $(4.b)$ or started $(4.a)$, otherwise no such transition exists (w' is not a successor). Hence, two case are possible for each goal $\gamma \in \mathsf{goals}_{agt}(w)$: either (a) it is eventually achieved or (b) it remains active forever. We now procede separately for each possible goal γ. If (a) is the case, then it means that there exists a state w' traversed by λ^+ such that $\gamma \notin \mathsf{goals}_{agt}(w')$. Hence, we can divide the trace segment from w to w' into a set of couples $\langle s_1 : e_1\rangle \cdots \langle s_\ell : e_\ell\rangle$ such that $\lambda^+[s_1] = w$, $\lambda^+[e_\ell] = w'$ and each $\langle s_i : e_i\rangle$ is such that for all states $w_j = \lambda^+[j]$ with $s_i \leq j \leq e_i$ we have $\pi \in \mathsf{plans}_{agt}(w_j)$ for some plan π, which implies that π remains applicable and relevant. In other words, $\langle s_i : e_i\rangle \in Exec_{agt}(\pi, \mathsf{goals}_{agt}, \lambda^+)$ and $\langle s : e\rangle \in GComplete_{agt}(\gamma, \Pi, \mathsf{goals}_{agt}, \lambda^+)$. Indeed, the goal-marking function is $g_\Gamma = \mathsf{goals}_{agt}$ and the plan we are possibly executing is exactly one of those marking the trace segment. Imagine by contradiction that $\lambda^+[s, e]$ is not a complete goal-execution for γ. Then, for some $j < e$, there is no applicable plan for any goal and then each $\pi \in \mathsf{plans}_{agt}(w_j)$ is not in $\mathsf{plans}_{agt}(w_{j+1})$ with $w_{j+1} = succ(w, \vec{\alpha})$ for all $\vec{\alpha}$ with $\alpha_{agt} = \lambda^+\langle j\rangle$. Hence the transition does not conform to case (4) as in definition of $succ$, and then w_{j+1} is not a successor of w_j in $\mathcal{M}_{\omega,\rho}$ unless $\mathcal{V}(\mathsf{state}(w_j)) \vDash \gamma$ and hence $j = e$. This means that $\lambda^+[s, e]$ is a goal-complete execution for γ. As for case (b), it is easy to follow the same reasoning as for (a) concluding that $\langle s : e\rangle \in GWork_{agt}(\gamma, \Pi, \mathsf{goals}_{agt}, \lambda^+)$.

We have proven that f^{rat}_{agt} is rational wrt ω, ρ. Recalling the definition of rational strategy, to prove that also f_{agt} is rational wrt $\mathcal{M}$, it is enough to show that for every $\lambda^+ \in GetTrace(\lambda, f_{agt})$, where $\lambda \in \mathsf{out}(q, f_{agt})$ and $q \in [\Phi]$, we can find a goal-marking function g_Γ such that λ^+ is a rational trace. This is trivially done exploiting the mapping between traces in $\mathcal{M}_{\omega,\rho}$ and traces in $\mathcal{M}$. In particular, recall that if $\lambda^+_2 \rightsquigarrow \lambda^+_1$, then to each $\lambda^+_2[i]$ corresponds the state $\lambda^+_1[i]$, with $0 \leq i \leq |\lambda^+_2|$. Also, $\lambda^+_1\langle i\rangle = \lambda^+_2\langle i\rangle$. Rationality can be showed repeating for $\mathcal{M}$ the same reasoning done for f^{rat}_{agt} wrt $\mathcal{M}_{\omega,\rho}$. This concludes the completeness proof for point *(i)*.

As for *(ii)*, it is enough to notice that, if $\lambda^+_2 = GetTrace(\lambda_2, f^{rat}_{agt}) \rightsquigarrow \lambda^+_1 = GetTrace(\lambda_1, f_{agt})$, with $\lambda_2 \in \mathsf{out}(w, f^{rat}_{agt})$, $w \in [\Phi]$, and $\lambda_1 \in \mathsf{out}(q, f_{agt})$, then $\lambda^+_1\langle i\rangle = \lambda^+_2\langle i\rangle$ for $0 \leq i \leq |\lambda^+_2|$. Hence, $\mathcal{M}_{\omega,\rho}, \lambda_2 \vDash \varphi$ iff $\mathcal{M}, \lambda_1 \vDash \varphi$.

($\Leftarrow$) Consider completeness. It remains to prove that if f_{agt} is a rational strategy and $\lambda^+_1 = GetTrace(\lambda_1, f_{agt})$ for each $\lambda_1 \in \mathsf{out}(q, f_{agt})$, then all traces $\lambda^+_2 = GetTrace(\lambda_2, f^{rat}_{agt})$ for each $\lambda_2 \in \mathsf{out}(w, f^{rat}_{agt})$, with $\lambda^+_2 \rightsquigarrow \lambda^+_1$, are rational, i.e., f^{rat}_{agt} is rational. Assume by contradiction that there exists a trace λ^+_1 induced by f_{agt} such that λ^+_2, with $\lambda^+_2 \rightsquigarrow \lambda^+_1$, is not rational. Since $\lambda^+_2 \rightsquigarrow \lambda^+_1$ implies that $\lambda^+_1\langle i\rangle = \lambda^+_2\langle i\rangle$ and $\lambda_1[i] = \mathsf{state}_{agt}(\lambda_2[i])$ for all $0 \leq i \leq |\lambda^+_2|$, this is not possible. ∎

Note that we can use this approach to model-check any arbitrary BDI-ATLES formula. Moreover, we can actually compute rational strategies from the witness returned for the corresponding ATL-formula.

Theorem 45 (Complexity)**.** Synthesizing $\langle\omega, \varrho\rangle$-strategies, achieving BDI-ATLES formula Φ over $\mathcal{M}$ by model checking ATL formula Φ over $\mathcal{M}_{\omega,\rho}$ is EXPTIME in the size of Γ, Π and $\mathcal{A}$.

The worst case complexity comes from the fact that the state space W is exponentially large, and ATL model-checking is PTIME-complete in the size of the model. As for hardness, that is still an open issue.

Proof. Given a BDI-ATLES model $\mathcal{M}$ for $\mathcal{C}$ and Π, be $\mathcal{M}_{\omega,\rho}$ as above. Let us denote with $\mathsf{copies}(q)$ the set of states $w \in \mathcal{M}_{\omega,\rho}$ such that $\mathsf{state}(w) = q$. Hence, for any $q \in Q$, $\mathsf{copies}(q)$ is bounded by $|2^Q| \times 2^{|\Pi|}$. Indeed, each element in $\mathsf{copies}(q)$ corresponds to a different goal set $\mathsf{goals}_{agt}(w)$ and program counters $\mathsf{pc}_{agt}(w)$, as the original state component is exactly q. Consider now –with little abuse of notation– the set $\mathsf{state}(W')$, for $W' \subseteq W$, to be the set of original states $q \in Q$ corresponding to a set of extended states: $\mathsf{state}(W') = \cup_{w \in W}\{q \mid q = \mathsf{state}(w)\}$. Then notice that, given any two computations $w_0, w_1 \dots w_{\bar{\ell}}$ and $w_0, w'_1 \dots w'_{\ell}$ in $\mathcal{M}_{\omega,\rho}$ such that $\mathsf{state}(\{w_0, w_1 \dots w_{\bar{\ell}}\}) = \mathsf{state}(\{w_0, w'_1 \dots w'_{\ell}\})$, we have that $\mathsf{goals}(agt, w_{\bar{\ell}}) = \mathsf{goals}(agt, w'_{\ell})$. Hence, there are at most $|2^Q|$ distinct possible computations λ such that $q \in \mathsf{state}(\lambda)$. As for the state fragment keeping information about program counters, it is enough to notice that there are at most $2^{n \times m}$ tuples in $PlanPos$, where n is the number of plans and m their length, assumed fixed.
Hence, we get that $|\mathcal{M}_{\omega,\rho}| \leq \bigcup_{q \in Q} \mathsf{copies}(q) \leq |Q| \times |2^Q| \times 2^{|\Pi|}$. By construction, we get the following complexity characterization as well: $|\mathcal{M}_{\omega,\rho}| \leq |Q| \times |2^G| \times 2^{|\Pi|}$ and we conclude that $|\mathcal{M}_{\omega,\rho}| \leq \textsc{min}(|2^Q| \times 2^{|\Pi|}, |2^G| \times 2^{|\Pi|})$ which, interestingly, exposes the fact that goals and plan libraries equally (but separately) concur to the size of the extended model. ■

7.2 Discussion

We have presented a sound and complete approach to compute rational strategies satisfying BDI-ATLES formulas. This can be seen as a form of synthesis in which agents have access to shared components (capabilities, hence plan libraries): BDI agents try to satisfy the specification selecting capabilities to use among those allowed by the assignment ω. However, in contrast with other settings in which shared functionalities are available (e.g., the composition setting), the access to such resources is constrained by the BDI-ATLES formula, and not by the system.

Chapter 8

Towards adding data to processes

This short chapter is a brief discussion about the work done, during my Ph.D., in the context of verification for data-aware processes, what does this mean and which are the major difficulties involved when we turn to the synthesis task. Since this work is (at this stage) focused on decidability issues of verification, and since a complete, thoughtful exposition would require an extensive tractation, we won't report here details. The reader is referred to the list of related publications, i.e., (Bagheri Hariri et al., 2011; Hariri et al., 2012; Bagheri Hariri et al., 2013).

Business Processes. A *business process* consists of a set of activities that are performed in coordination in an organizational and technical environment, which jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations. Business process management (BPM) (Weske, 2007) includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes. In operational business processes, the activities and their relationships are specified, but implementation aspects of the business process are disregarded. Operational business processes are specified by business process models.

By leveraging a precise descriptions of tasks, resources and data involved in the process (as well as their dependencies), the goal of business process management solutions are the automation, adaptation and analysis of processes. In particular, the goal is to automatize non-human tasks, adapt the process to react in real-time to environmental changes and also perform verification tasks, i.e., check whether the process model satisfies high-level specifications.

Therefore, the usual business process approach comprise the following fundamental phases (Weske, 2007): (i) *design and analysis*, where process models are designed from specifications, using suitable modeling languages; (ii) *configuration*, where an implementation platform is chosen and the system is configured according to the specific organizational environment of the enterprise; (iii) *enactment*, where process instances are then initiated, executed and monitored by the run-time environment;

(iv) *evaluation*, where process models and their implementation are evaluated, typically by processing execution logs with monitoring and process mining. In particular, we will address here the first phase by devising a verification framework able to deal with temporal specifications of processes, and we will comment on the possibility of performing the synthesis task also in this setting.

8.1 The setting

Formerly, most BPM frameworks were mostly organized around activity flows ("activity-centric" process models), i.e., the focused mainly on the modeling and analysis of the activities of which the processes were composed (see, e.g., modeling approaches base on Petri Nets (Aalst, 1998), on Workflow Nets (Aalst et al., 2002), BPMN (Object Management Group (OMG), 2011)). However, business processes operate on data. Explicitly representing such data and the dependencies between activities of a business process puts a business process management system in a position to control the transfer of relevant data as generated and processed during processes enactment. Data modelling is also the basis for the integration of heterogeneous data.

Business Artifacts. Therefore, recent work in business processes, services and databases is bringing forward the need of considering both data and processes as first-class citizens in process and service design (Nigam and Caswell, 2003; Bhattacharya et al., 2007; Deutsch et al., 2009; Vianu, 2009). In particular, the so called *business artifacts* (and, in general, artifact-centric approaches), which advocate a sort of middle ground between a conceptual formalization of dynamic systems and their actual implementation, are promising to be effective in practice (Cohn and Hull, 2009).
The research efforts on this kind of approaches merges database and knowledge representation, that has focused largely on data aspects and from research on programming languages, software engineering, workflow, and verification, that has centered on mostly on process aspects.
The business artifact framework (Nigam and Caswell, 2003; Bhattacharya et al., 2007; Deutsch et al., 2009; Vianu, 2009) provides a process design methodology that focuses on business relevant entities, called artifacts, rather than on activities. An artifact is thus characterized by two basic components:

- the information model
- the lifecycle model

The former holds all the business relevant information of interest ad it can be modified by tasks, whereas the latter contraints the application of tasks, end hence, it specifies the possible way the artifact can evolve over time.
Artifact instances evolves in an environment that includes users, external data sources, services and events. Moreover, relationships captures links between artifacts and events. The general framework for artifact-centric systems does not restrict the way to specify the information models and lifecycles. However, while the

information model is usually modeled as a set of (possibly nested) attributes, the lifecycle can be specified in several ways.

Verification of temporal properties on infinite-state systems. Irrespective of how the information model and the lifecycle are represented, the key point is that any verification of temporal properties in the presence of data represents a significant research challenge, since data makes the system infinite-state, and neither finite-state model checking (Clarke et al., 1999b) nor most of the current techniques for infinite-state model checking, which mostly tackle recursion (Burkart et al., 2001), apply to this case.

8.2 A Framework for Artifact-centric Processes

We are going to briefly discuss here our approach for verification of temporal properties of data-aware (business) processes (Bagheri Hariri et al., 2013).
Recently, there have been some advancements on this issue (Cangialosi et al., 2010; Damaggio et al., 2011; Belardinelli et al., 2011) in the context of suitably constrained relational database settings. While most of this work is based on maintaining information in a relational database (e.g. see (Bagheri Hariri et al., 2011)), for more sophisticated applications it is foreseen to enrich data-intensive business processes with a semantic level, where information can be maintained in a "semantically rich" knowledge base that allows for operating with incomplete information (Calvanese et al., 2012; Limonad et al., 2012). This leads us to look into how to combine *first-order data*, *ontologies*, and processes, while maintaining basic inference tasks decidable.
In particular, we capture the domain of interest in terms of semantically rich formalisms as those provided by ontological languages based on Description Logics (DLs) (Baader et al., 2003). These languages natively deal with incomplete knowledge in the modeled domain. This additional flexibility comes with an added cost, however: differently from relational databases, to evaluate queries we need to resort to logical implication. Moreover incomplete information combined with the ability of evolving the system through actions results in a notoriously fragile setting w.r.t. decidability (Wolter and Zakharyaschev, 1999; Gabbay et al., 2003). In particular, due to the nature of DL assertions (which in general are not definitions but constraints on models), we get one of the most difficult kinds of domain descriptions for reasoning about actions (Reiter, 2001a), which amounts to dealing with complex forms of state constraints (Lin and Reiter, 1994).
To overcome this difficulty, virtually all solutions that aim at robustness are based on a so-called "functional view of knowledge bases" (Levesque, 1984): a knowledge base (KB) provides the ability of querying based on logical implication, and the ability of progressing it to a "new" KB through forms of updates (Baader et al., 2012; Calvanese et al., 2011).
In our approach (Bagheri Hariri et al., 2013), we follow this functional approach, hence states are KBs, and we use *actions* to move from one state to the other. We thus define a description logic *Knowledge Action Base* (KAB) to be composed of these two fundamental components. Therefore, KABs are regarded as a mechanism

for providing both a semantically rich representation of the information on the domain of interest in terms of a description logic knowledge base and actions to change such information over time, possibly introducing new objects. Hence some key-points are:

- a KAB can be seen as a stateful device that stores the information of interest into a KB which evolves by executing those actions according to a process;
- the knowledge description formalism is decoupled from the formalism that describes the progression;
- we can define the dynamics through a *transition system* whose states are KBs, and transitions are labeled by the action (with object parameters) that causes the transition;
- the process is seen as the collection of action specifications, i.e., preconditions (local properties), parameters and effects;
- we allow for arbitrary introduction of new terms, i.e., each execution step external information is incorporated into the system;
- as verification formalism, we adopt a variant of *first-order* μ-calculus (Stirling, 2001; Park, 1976) with quantification across states.

μ-calculus is virtually the most powerful temporal logic used for model checking of finite-state transition systems, and is able to express both linear time logics such as LTL and PSL, and branching time logics such as CTL and CTL* (Emerson, 1996; Clarke et al., 1999a). The main characteristic of μ-calculus is its ability of expressing directly least and greatest fixpoints of (predicate-transformer) operators formed using formulae relating the current state to the next one. By using such fixpoint constructs one can easily express sophisticated properties defined by induction or co-induction. This is the reason why virtually all logics used in verification can be considered as fragments of μ-calculus.
In this work, we use a first-order variant of μ-calculus, where we allow local properties to be expressed as ECQs, and at the same time we allow for arbitrary first-order quantification across states. Hence, ECQs, i.e., *epistemic query language*, allows to reason about what is "known" by the current KB (Calvanese et al., 2007), whereas the first-order variant of the μ-calculus allows for temporal requirements on the "transition systems of KBs".

Decidability of Verification. Unsurprisingly, even for very simple KABs and temporal properties, verification is undecidable. However, we show that for a rich class of KABs, verification is in fact decidable and reducible to finite-state model checking. To obtain this result, following (Cangialosi et al., 2010; Bagheri Hariri et al., 2011), we rely on recent results in data exchange on the finiteness of the chase of tuple-generating dependencies (Fagin et al., 2005), though, in our case, we need to extend the approach to deal with *(i)* incomplete information, *(ii)* inference on equality, and *(iii)* quantification across states in the verification language.

About Synthesis. What about synthesis? As already done throughout this dissertation, we turn to adversarial synthesis, i.e., we consider a setting in which two agents act in turn as adversaries. The first agent, called environment, acts autonomously, whereas we control the second agent, called system. The joint behavior of the two agents gives rise to a so-called two-player game structure ($2GS$), inspired to (Piterman et al., 2006a) and introduced here in Section 3.2. Indeed, in Chapter 3 we have shown how, making use of $2GS$ and a variants of the μ-calculus, we can capture what the system should obtain in spite of the adversarial moves of the environment. This specification can be considered the goal of the game for the system. The synthesis problem amounts to synthesizing a strategy, i.e., a suitable refined behavior for the system that guarantees to the system the fulfillment of the specification (for more details refer to Section 3.2). The point is that, technically, μ-calculus separates local properties, asserted on the current state or on states that are immediate successors of the current one, from properties talking about states that are arbitrarily far away from the current one (Stirling, 2001). The latter are expressed through the use of fixpoints. Indeed, the need of quantifying separately on environment and controller moves, requires the use of μ-calculus, and not simply CTL or LTL (Clarke et al., 1999c) for which model checking tools are much more mature.
Hence, we can deal with adversarial synthesis in our framework by building a $2GS$ in which we encode explicitly in the DL KB (the state) the alternation of the moves of the two players by means of a fresh concept. We can thus accomodate a suitable version of the FO μ-calculus to reason about such games associated to KAB transition systems, by rewriting FO μ-calculus formulas as to as make the alternation of quantifiers explicit. Indeed, since the system is finite-state (under the above-mentioned restrictions), it is always possible to build a finite tow-player game of this sort, and thus extract winning strategies via model-checking the appropriate formula. However, this synthesis approach is not subject of study here, but can be seen, somehow, as a natural path to undertake.

Appendix A

BDI-ATLES Rational Strategies

We formally define here the notion of rational traces and rational strategies for ATLES. This is an authors' extension (to plan sequences) of the work published in (Yadav and Sardiña, 2012).

Given a plan library Π and an initial goal base $\mathcal{G}$ for an agent *agt* in structure $\mathcal{M}$, we are to characterize those strategies for *agt* within $\mathcal{M}$ that represent rational behaviors: *the agent tries plans from* Π *in order to bring about its goals* $\mathcal{G}$ *given its beliefs* (Bratman et al., 1988; Rao and Georgeff, 1992).
While technically involved, the idea to define set $\Sigma^{agt}_{\Pi,\mathcal{G}}$ is simple: first identify those paths in $\mathcal{M}$ that could be showing rational behavior for the agent; second consider rational strategies those that will always yield rational paths. We do this in three steps. First, we identify minimal constraints on how the goals of an agent can evolve in a path. Second, we define what it means for a plan to be tried by an agent in a path. Third, we define rational paths that result from an agent's deliberation process.
Before we start, we extend the notion of paths to account for the actions performed. Recall that, as for standard ATL models, a trace is a finite sequence of alternating states and actions $\lambda^+ = q_0\alpha_1 q_1 \cdots \alpha_\ell q_\ell$ such that $q_0 \cdots q_\ell$ is a (finite) path in $\mathcal{M}$. In particular, we use $\lambda^+[i]$ and $\lambda^+\langle i\rangle$ to denote the i-th state q_i and the i-th action α_i, respectively, in trace λ^+. The length $|\lambda^+|$ of a trace is the number of actions on it; hence it matches the length of the underlying path.

Goal evolution in traces

As standard, we assume BDI agents to have a *single-minded* type of commitment level to goals (Rao and Georgeff, 1991), that is, an agent does not drop a goal until it achieves it or believes it cannot be achieved. Here, we state what the possible goals an agent can have at each moment in a path may be.
To that end, we make use of so-called *goal-marking* functions $g_{\mathcal{G}}(\lambda^+, i)$, for an agent with an initial goal base $\mathcal{G}$, which outcomes a possible goal base of the agent at $\lambda^+[i]$. As expected, a goal-marking function must obey some basic rationality constraints in terms of goal persistence:

- $g_{\mathcal{G}}(\lambda^+, 0) = \mathcal{G}$, that is, $\mathcal{G}$ is the agent's initial goal base;

- for all $i \leq |\lambda^+|$ and $\gamma \in g_{\mathcal{G}}(\lambda^+, i)$, $\mathcal{V}(\lambda^+[i]) \not\models \gamma$, that is, the agent does not have (already) achieved goals.

Observe that these two constraints only capture half of the single-minded notion of commitment. Indeed, they do not detail on how an agent may abandon goals besides when achieved or how it may adopt new goals. To complete the picture, we need to take plans into consideration.

Plan executions in traces

Agents developed under the BDI paradigm are meant to execute actions as per the plans available to them. We shall next define what it means for a trace to include an execution of a plan.
When it comes to selecting plans for execution, there are generally two core notions in BDI programming. A plan is relevant if its intended effects are enough to bring about some of the agent's goals. Technically, given a trace λ^+ and a goal-marking function g, we say that a plan-rule $\phi[\vec{\alpha}]\psi$ is *relevant* at location $\lambda^+[i]$ in the trace, where $0 \leq i \leq |\lambda^+|$, if there exists $\gamma \in g(\lambda^+, i)$ such that $\psi \models \gamma$. Furthermore, the plan is *applicable* at location $\lambda^+[i]$ if besides being relevant, its context condition holds true, that is, $\mathcal{V}(\lambda^+[i]) \models \phi$.
The function $Exec_{agt}(\phi[\vec{\alpha}]\psi, g, \lambda^+)$ denotes the fragments in trace λ^+ that may stand for a, possibly partial, execution of plan $\phi[\vec{\alpha}]\psi$ by agent $agt \in \mathcal{A}$. We use pairs of indexes $\langle s : e \rangle$ to denote the start $0 \leq s < |\lambda^+|$ and terminating $s < e \leq |\lambda^+|$ positions for each execution fragment. Formally, $Exec_{agt}(\phi[\vec{\alpha}]\psi, g, \lambda^+)$ is the set of pairs $\langle s : e \rangle$ such that (below, $k = e - s$):

1. $\lambda^+\langle s + i \rangle = \alpha_i$, for all $1 \leq i \leq k$;
2. $\phi[\vec{\alpha}]\psi$ is applicable at $\lambda^+[s]$ relative to goal-marking g;
3. $\phi[\vec{\alpha}]\psi$ remains relevant from $\lambda^+[s]$ to $\lambda^+[e]$ relative to g;
4. if $k < |\alpha|$, then either $\phi[\vec{\alpha}]\psi$ is no longer relevant at state $\lambda^+[e]$ in the trace or $\alpha_{k+1} \notin d(agt, \lambda^+[e])$.

The first condition states that the first k (action) steps of plan α has been executed in trace λ^+ from position s. The second one implies that the plan was adequately selected—under the BDI programming paradigm—given the agent's goals and beliefs. The third requirement states that the plan does remain relevant during its execution (otherwise, the agent has no reason to insist with it). The last constraint characterizes the conditions under which the plan may be aborted before its completion: either it stops being relevant (e.g., the motivating goal was abandoned) or the next step was not physically possible in the world (i.e., the plan *failed*).

Rational traces and rational strategies

The last step involves defining the so-called rational traces, those that can be explained by the agent acting as per its available plan in order to achieve its goals relative to its beliefs.

Initially, an agent has a set of goals (initial goal base) that she wants to bring about. The agent chooses one goal to work on, and selects an applicable plan for such goal from its plan library for execution. If the plan successfully brings about the goal, then the agent deems the goal achieved and the plan finished. On the other hand, if the plan fails to achieve the goal, then the agent executes another applicable plan for the *same* goal (even the same failed plan if still applicable), thus realizing its commitment to the goal. When the agent has no applicable plan for a goal, the agent deems the goal impossible and drops it. Traces that can be "explained" in this way are referred to as *rational traces*.

To capture all this, we define the notions of *goal-working* and *goal-complete* segments of a trace. Informally, a goal-working segment of a trace is one in which the agent is working to achieve a particular goal, by rationally executing plans for it. Given a trace λ^+, a plan-library Π, a goal-marking function g, and a goal γ, the set $GWork_{agt}(\gamma, \Pi, g, \lambda^+)$ of all *goal-working segments for goal γ* in trace λ^+ (for agent $agt \in \mathcal{A}$) is the set of index pairs $\langle s : e \rangle$ such that:

- $\gamma \in g(\lambda^+, i)$, for all $s \leq i < e$;
- there exists a sequence of segment indexes $\Lambda = \langle s_1 : e_1 \rangle \cdots \langle s_\ell : e_\ell \rangle$ such that $s_1 = s$ and $e_\ell \leq e$, and for all $i \in \{1, \ldots, \ell\}$:
  - $e_i \leq s_{i+1}$ when $i < \ell$ (and of course $s_i \leq e_i$);
  - there is an applicable plan $\phi_i[\alpha_i]\psi_i \in \Pi$ for goal γ, such that $\langle s_i : e_i \rangle \in Exec_{agt}(\phi_i[\alpha_i]\psi_i, g, \lambda^+)$;
  - if $i < \ell$ and $e_i < s_{i+1}$, then $\lambda^+\langle e_i + k \rangle = \text{NOOP}$ and there is no applicable plan in Π for goal γ at $\lambda^+[e_i + k - 1]$, for all $1 \leq k \leq s_{i+1} - e_i$; and
  - if $e_\ell < e$, then $\lambda^+\langle e_\ell + k \rangle = \text{NOOP}$ and there is no applicable plan in Π for goal γ at (state) $\lambda^+[e_\ell + k - 1]$, for all $1 \leq k \leq e - e_\ell$;

In words, a trace segment $\lambda^+[s, e]$ is goal-working for goal γ if it can be divided into ℓ sub-segments in which the agent consecutively tries one applicable plan $\phi_i[\alpha_i]\psi_i$ in the i-th sub-segment for the goal γ. Observe that it could be the case that between segments, and at the very end, the agent remains still, by simply performing NOOP actions. This would only happen when the agent has a certain (strong) level of commitment towards the goal that causes her to wait for some plan to become applicable.

A goal-complete segment is one in which the agent has tried as much as possible to bring about the goal. Formally, the set of all *goal-complete segments for goal γ* in trace λ^+, denoted $GComplete_{agt}(\gamma, \Pi, g, \lambda^+)$, is defined by further restricting the notion of goal-working segments as follows:

- $\gamma \notin g(\lambda^+, e)$, that is, γ is not a goal anymore at the end;
- either $\mathcal{V}(\lambda^+[e]) \vDash \gamma$ or there is no applicable plan in library Π for goal γ at location $\lambda^+[e]$ in the trace, that is, either the goal has been achieved or the agent deems it impossible due to lack of useful plans; and
- $e_\ell = e$, that is, Λ covers the whole segment $\langle s : e \rangle$.

It follows then that trace segments that are goal-working but not goal-complete for γ are those in which the agent is still working on the goal.
We now have all the technical machinery to define the set of rational traces and the set $\Sigma^{agt}_{\Pi,\mathcal{G}}$ of rational strategies used to define the semantics of BDI-ATLES in the previous section. Roughly speaking, a trace is rational if there is a goal-marking function under which the trace can be broken down into segments that are all goal-complete for some goal, except the last one that may just be a goal-working segment for a goal or a sequence of noOp actions if the agent has in fact no goals. Technically, λ^+ is a *rational trace* for an agent $agt \in \mathcal{A}$ equipped with a plan-library Π and having an initial goal base $\mathcal{G}$, if there exists a goal-marking function $g_{\mathcal{G}}$ for an agent with goal base $\mathcal{G}$ and a sequence of indexes $k_1 \cdots k_\ell$, with $k_1 = 0$ and $k_\ell = |\lambda^+|$, such that:

- for all $i \in \{1, \ldots, \ell-2\}$, there exists formula γ_i such that $\langle k_i : k_{i+1} \rangle \in GComplete_{agt}(\gamma_i, \Pi, g_{\mathcal{G}}, \lambda^+)$; and
- either $\langle k_{\ell-1} : k_\ell \rangle \in GWork_{agt}(\gamma, \Pi, g_{\mathcal{G}}, \lambda^+)$, for some goal formula γ; or $g_{\mathcal{G}}(\lambda^+, i) = \varnothing$ and $\lambda^+\langle i+1 \rangle = \text{noOp}$, for all $i \in \{k_{\ell-1}, \ldots, k_\ell - 1\}$.

Let us denote by $\zeta^{agt}_{\Pi,\mathcal{G}}$ the set of all rational traces for agent agt with library Π and goal base $\mathcal{G}$.
So, a rational strategy is one that only yields rational traces. To link traces and strategies, we define $GetTrace(\lambda, f_{agt})$ to be the partial function that returns the trace induced by a path λ and a strategy f_{agt}, if any. Formally, $GetTrace(\lambda, f_{agt}) = q_0\alpha_1 q_1 \ldots \alpha_{|\lambda|} q_{|\lambda|}$ iff for all $k \leq |\lambda|$: *(i)* $q_k = \lambda[k]$; and *(ii)* there exist a joint-move $\vec{\alpha} \in \mathcal{D}(q_k)$ such that $\delta(q_k, \vec{\alpha}) = q_{k+1}$ and $f_{agt}(\lambda[0,k]) = \vec{\alpha}[agt] = \alpha_{k+1}$ (where $\vec{\alpha}[agt]$ denotes agent agt's move in joint-move $\vec{\alpha}$).
Finally, the set of *rational strategies* is defined as follows:

$$\Sigma^{agt}_{\Pi,\mathcal{G}} = \{f_{agt} \mid \bigcup_{\lambda \in \overline{\Lambda}} GetTrace(\lambda, f_{agt}) \subseteq \zeta^{agt}_{\Pi,\mathcal{G}}\},$$

where $\overline{\Lambda} \subseteq \Lambda$ is the set of finite paths in $\mathcal{M}$.

Bibliography

Van Der Aalst. The application of petri nets to workflow management, 1998.

Wil Van Der Aalst, Kees Van Hee, Prof. Dr. Kees, Max Hee, Remmert Remmerts De Vries, Jaap Rigter, Eric Verbeek, and Marc Voorhoeve. Workflow management: Models, methods, and systems, 2002.

N. Alechina, M. Dastani, B. S. Logan, and Ch. A logic of agent programs. In *Proc. of AAAI*, pages 795–800. AAAI Press, 2007. ISBN 978-1-57735-323-2.

N. Alechina, M. Dastani, B. S. Logan, and John-Jules Meyer. Reasoning about agent deliberation. In *Proc. of KR*, pages 16–26, 2008.

Gustavo Alonso, Fabio Casati, Harumi A. Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer, 2004.

Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, September 2002. ISSN 0004-5411.

Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

Franz Baader, Silvio Ghilardi, and Carsten Lutz. LTL over description logic axioms. *ACM Trans. on Computational Logic*, 13(3), 2012.

Christer Bäckström. Equivalence and tractability results for sas+ planning. In *Proceedings of the 3rd International Conference on Principles on Knowledge Representation and Reasoning (KR-92*, pages 126–137. Morgan Kaufmann, 1992.

Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Riccardo De Masellis, and Paolo Felli. Foundations of relational artifacts verification. In *Proc. of the 9th Int. Conference on Business Process Management (BPM 2011)*, volume 6896 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2011.

Babak Bagheri Hariri, Diego Calvanese, Marco Montali, Giuseppe De Giacomo, Riccardo De Masellis, and Paolo Felli. Description Logic Knowledge and Action Bases. *J. of Artificial Intelligence Research*, 2013. To appear.

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN 026202649X, 9780262026499.

Philippe Balbiani, Fahima Cheikh, and Guillaume Feuillade. Composition of interactive web services based on controller synthesis. In *Proc. of SERVICES*, pages 521–528, 2008.

Philippe Balbiani, Fahima Cheikh, and Guillaume Feuillade. Algorithms and complexity of automata synthesis by asynhcronous orchestration with applications to web services composition. *Electronic Notes in Theoretical Computer Science*, 229 (3):3–18, July 2009.

Fabien Baligand, Nicolas Rivierre, and Thomas Ledoux. A declarative approach for qos-aware web service compositions. In *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 422–428. Springer, 2007. ISBN 978-3-540-74973-8.

Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):14, 2011.

Sandrine Beauche and Pascal Poizat. Automated Service Composition with Adaptive Planning. In *Proc. ICSOC 2008*, pages 530–537, 2008.

Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. Verification of deployed artifact systems via data abstraction. In *Proc. of the 9th Int. Joint Conf. on Service Oriented Computing (ICSOC 2011)*, volume 7084 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2011.

Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic Composition of e-Services that Export their Behavior. In *Proc. of ICSOC 2003*, pages 43–58, 2003.

Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic service composition based on behavioural descriptions. *International Journal of Cooperative Information Systems*, 14(4):333–376, 2005a.

Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic service composition based on behavioural descriptions. *International Journal of Cooperative Information Systems*, 14(4):333–376, 2005b.

Daniela Berardi, Fahima Cheikh, Giuseppe De Giacomo, and Fabio Patrizi. Automatic service composition via simulation. *International Journal of Foundations of Computer Science*, 19(2):429–452, 2008.

K. Bhattacharya, C. Gerede, R. Hull, R. Liu, and J. Su. Towards formal analysis of artifact-centric business process models. In *Proc. of the 5th Int. Conference on Business Process Management (BPM 2007)*, volume 4714 of *Lecture Notes in Computer Science*, pages 288–234. Springer, 2007.

Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

J. Blythe and J.L. Ambite, editors. *Proc. of ICAPS 2004 Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

Blai Bonet and Hector Geffner. Solving pomdps: Rtdp-bel vs. point-based algorithms. In *IJCAI*, pages 1641–1646, 2009.

Blai Bonet, Héctor Palacios, and Hector Geffner. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *ICAPS*, 2009.

M. Bordignon, J. Rashid, M. Broxvall, and Alessandro Saffiotti. Seamless integration of robots and tiny embedded devices in a PEIS-ecology. In *IROS*, pages 3101–3106, San Diego, CA, 2007.

Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El, Fallah Seghrouchni, Jorge J. Gomez-sanz, JoÃčo Leite, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for Multi-Agent systems. *Informatica (Slovenia)*, 30(1):33–44, 2006.

Michael E. Bratman, David J. Israel, and Martha E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(3):349–355, 1988.

Julius R. Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138: 295–311, April 1969.

O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification of infinite structures. In *Handbook of Process Algebra*. Elsevier Science, 2001.

Paolo Busetta, Ralph Rönnquist, Andew Hodgson, and Andrew Lucas. JACK intelligent agents: Components for intelligent agents in Java. *AgentLink Newsletter*, 2, January 1999. Agent Oriented Software Pty. Ltd.

Tom Bylander. Tractability and artificial intelligence. *JETAI*, 3:171–178, 1991.

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. EQL-Lite: Effective first-order query processing in description logics. In *Proc. of the 20th Int. Joint Conf. on Artificial Intelligence (IJCAI 2007)*, pages 274–279, 2007.

Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Actions and programs over description logic knowledge bases: A functional approach. In Gerhard Lakemeyer and Sheila A. McIlraith, editors, *Knowing, Reasoning, and Acting: Essays in Honour of Hector Levesque*. College Publications, 2011.

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Marco Montali, and Ario Santoso. Ontology-based governance of data-aware processes. In *Proc. of the 6th Int. Conf. on Web Reasoning and Rule Systems (RR 2012)*, volume 7497 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2012.

Piero Cangialosi, Giuseppe De Giacomo, Riccardo De Masellis, and Riccardo Rosati. Conjunctive artifact-centric services. In *Proc. of the 8th Int. Joint Conf. on Service Oriented Computing (ICSOC 2010)*, volume 6470 of *Lecture Notes in Computer Science*, pages 318–333. Springer, 2010.

J. Cardose and A.P. Sheth. Introduction to semantic web services and web process composition. In *Proc. of SWSWPC 2004*, 2004.

Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. SPRINGERP, Secaucus, NJ, USA, 2006.

A. Church. Logic, arithmetics, and automata. In *In Proc. International Congress of Mathematicians*, pages 23–35. Institut Mittag-Leffler, 1963.

Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence Journal*, 147(1-2):35–84, 2003. ISSN 0004-3702.

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag. ISBN 3-540-11212-X.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. The MIT Press, Cambridge, MA, USA, 1999a.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. The MIT Press, Cambridge, MA, USA, 1999b.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. The MIT Press, Cambridge, MA, USA, 1999c. ISBN 0-262-03270-8.

E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:279–287, 1999d. ISSN 1433-2779.

David Cohn and Richard Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *Bull. of the IEEE Computer Society Technical Committee on Data Engineering*, 32(3):3–9, 2009.

F. Curbera, A.P. Sheth, and K. Verma. Services oriented architecture and semantic web processes. In *Proc. of ICWS 2004*, 2004.

Elio Damaggio, Alin Deutsch, and Victor Vianu. Artifact systems with data dependencies and arithmetic. In *Proc. of the 14th Int. Conf. on Database Theory (ICDT 2011)*, pages 66–77, 2011.

Mehdi Dastani and Wojciech Jamroga. Reasoning about strategies of multi-agent programs. In *Proc. of AAMAS*, pages 997–1004, Richland, SC, 2010. IFAAMAS. ISBN 978-0-9826571-1-9.

Luca de Alfaro, Thomas A. Henzinger, and Rupak Majumdar. From verification to control: Dynamic programs for omega-regular objectives. In *Proc. of LICS'01*, pages 279–290, 2001.

Giuseppe De Giacomo and Sebastian Sardina. Automatic synthesis of new behaviors from a library of available behaviors. In *Proc. of IJCAI*, pages 1866–1871, 2007.

Giuseppe De Giacomo and Moshe Y. Vardi. Automata-theoretic approach to planning for temporally extended goals. In *ECP*, pages 226–238, 1999.

Giuseppe De Giacomo, Riccardo De Masellis, and Fabio Patrizi. Composition of partially observable services exporting their behaviour. In *ICAPS*, 2009.

Giuseppe De Giacomo, Paolo Felli, Fabio Patrizi, and Sebastian Sardina. Two-player game structures for generalized planning and agent composition. In *AAAI*, 2010a.

Giuseppe De Giacomo, Fabio Patrizi, and Sebastian Sardiña. Agent programming via planning programs. In *AAMAS*, pages 491–498, 2010b.

Giuseppe De Giacomo, Claudio Di Ciccio, Paolo Felli, Yuxiao Hu, and Massimo Mecella. Goal-based composition of stateful services for smart homes. In *OTM Conferences (1)*, pages 194–211, 2012.

Giuseppe De Giacomo, Fabio Patrizi, and Sebastian Sardina. Automatic behavior composition synthesis. *Artificial Intelligence Journal*, 2013.

Guiseppe De Giacomo and Paolo Felli. Agent composition synthesis based on ATL. In *Proc. of AAMAS*, pages 499–506, 2010.

Alin Deutsch, Richard Hull, Fabio Patrizi, and Victor Vianu. Automatic verification of data-centric business processes. In *Proc. of the 12th Int. Conf. on Database Theory (ICDT 2009)*, pages 252–267, 2009.

Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony Mcisaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. *Computer Aided Verification*, pages 27–39, 2003.

E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier, 1995.

E. Allen Emerson. Model checking and the mu-calculus. In *Descriptive Complexity and Finite Models*, pages 185–214, 1996.

E. Allen Emerson and Joseph Y. Halpern. 'sometimes' and 'not never' revisited: on branching versus linear time temporal logic. *J. ACM*, 33:151–178, January 1986. ISSN 0004-5411.

Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1): 89 – 124, 2005. ISSN 0304-3975.

Paolo Felli, Giuseppe De Giacomo, and Alessio Lomuscio. Synthesizing Agent Protocols From LTL Specifications Against Multiple Partially-Observable Environments. In *KR*, 2012.

Paolo Felli, Sebastian Sardiña, and Nitin Yadav. Reasoning about Agent Programs in BDI-ATLES. (submitted), 2013a.

Paolo Felli, Sebastian Sardiña, and Nitin Yadav. Supervisory Control for Behavior Composition with Constraints. (submitted), 2013b.

R. E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence Journal*, 2:189–208, 1971.

Christian Fritz, Jorge A. Baier, and Sheila A. McIlraith. ConGolog, Sin Trans: Compiling ConGolog into Basic Action Theories for Planning and Beyond. In *Proc. of KR'08*, pages 600–610, 2008.

Dov Gabbay, Agnes Kurusz, Frank Wolter, and Michael Zakharyaschev. *Many-dimensional Modal Logics: Theory and Applications*. Elsevier Science Publishers, 2003.

P. Gammie and R. van der Meyden. MCK: Model checking the logic of knowledge. In *Proceedings of 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 479–483. Springer-Verlag, 2004.

Michael R. Genesereth and Nils J. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987. ISBN 0-934613-31-1.

Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers Inc., 2004.

P. Gohari and W. M. Wonham. On the complexity of supervisory control design in the rw framework. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 30(5):643–652, October 2000. ISSN 1083-4419.

V. Goranko and W.J. Jamroga. Comparing semantics of logics for multi-agent systems. *Synthese*, 139(2):241–280, 2004. Imported from HMI.

Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*, 2002. Springer.

J Y Halpern and M Y Vardi. The complexity of reasoning about knowledge and time. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, STOC '86, pages 304–315, New York, NY, USA, 1986. ACM. ISBN 0-89791-193-8.

Joseph Y. Halpern and Moshe Y. Vardi. Model checking vs. theorem proving: A manifesto. In *KR*, pages 325–334, 1991.

A. Harding, M. Ryan, and P.-Y. Schobbens. A new algorithm for strategy synthesis in ltl games. In *TACAS*, pages 477–492, 2005.

D. Harel and A. Pnueli. *On the development of reactive systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985a. ISBN 0-387-15181-8.

D. Harel and A. Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and models of concurrent systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985b. ISBN 0-387-15181-8.

David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, 2000.

Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Riccardo De Masellis, Paolo Felli, and Marco Montali. Verification of description logic knowledge and action bases. In *ECAI*, pages 103–108, 2012.

Ramy Ragab Hassen, Lhouari Nourine, and Farouk Toumani. Protocol-Based Web Service Composition. In *Proc. ICSOC 2008*, pages 38–53, 2008.

Wiebe Van Der Hoek and Michael Wooldridge. Model checking knowledge and time. In *Model Checking Software, Proceedings of SPIN 2002 (LNCS Volume 2318*, pages 95–111. Springer-Verlag, 2002a.

Wiebe Van Der Hoek and Michael Wooldridge. Tractable multiagent planning for epistemic goals. In *In Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002*, pages 1167–1174. ACM Press, 2002b.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

Yuxiao Hu and Giuseppe De Giacomo. Generalized planning: Synthesizing plans that work for multiple environments. In *IJCAI*, pages 918–923, 2011.

Wojciech Jamroga. Some remarks on alternating temporal epistemic logic. In *Proceedings of Formal Approaches to Multi-Agent Systems (FAMAS 2003*, pages 133–140, 2004.

Wojciech Jamroga and Thomas Ågotnes. Modular interpreted systems. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, AAMAS '07, pages 131:1–131:8, New York, NY, USA, 2007. ACM. ISBN 978-81-904262-7-5.

Wojciech Jamroga and Wiebe van der Hoek. Agents that know how to play. *Fundam. Inform.*, 63(2-3):185–219, 2004.

Magdalena Kacprzak, Alessio Lomuscio, and Wojciech Penczek. From bounded to unbounded model checking for temporal epistemic logic. *Fundam. Inform.*, 63 (2-3):221–240, 2004.

Eirini Kaldeli, Ehsan Ullah Warriach, Jaap Bresser, Alexander Lazovik, and Marco Aiello. Interoperation, composition and simulation of services at home. In *Service-Oriented Computing - 8th International Conference, ICSOC 2010, San Francisco, CA, USA, December 7-10, 2010. Proceedings*, volume 6470 of *Lecture Notes in Computer Science*, pages 167–181, 2010. ISBN 978-3-642-17357-8.

Henry A. Kautz, Wolfgang Thomas, and Moshe Y. Vardi, editors. *Synthesis and Planning, 12.-17. June 2005*, volume 05241 of *Dagstuhl Seminar Proceedings*, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

Sylvain Kerjean, Froduald Kabanza, Richard St.-Denis, and Sylvie Thiébaux. Analyzing LTL model checking techniques for plan synthesis and controller synthesis (work in progress). *Electronic Notes in Theoretical Computer Science (ENTCS)*, 149(2):91–104, 2006.

Adrian Klein, Fuyuki Ishikawa, and Shinichi Honiden. Efficient qos-aware service composition with a probabilistic service selection policy. In *Service-Oriented Computing - 8th International Conference, ICSOC 2010, San Francisco, CA, USA, December 7-10, 2010. Proceedings*, volume 6470 of *Lecture Notes in Computer Science*, pages 182–196, 2010. ISBN 978-3-642-17357-8.

Joachim Klein and Christel Baier. Experiments with deterministic ω-automata for formulas of linear temporal logic. *Theor. Comput. Sci.*, 363(2):182–195, 2006.

Orna Kupferman and Moshe Y. Vardi. Synthesis with incomplete information. In *In Advances in Temporal Logic*, pages 109–127. Kluwer Academic Publishers, 2000.

Orna Kupferman and Moshe Y. Vardi. Safraless decision procedures. In *FOCS*, pages 531–542, 2005.

Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Safraless compositional synthesis. In *CAV*, pages 31–44, 2006.

Leslie Lamport. "sometime" is sometimes "not never" - on the temporal logic of programs. In *POPL*, pages 174–185, 1980.

Yves Lespérance, Giuseppe De Giacomo, and Atalay Nafi Ozgovde. A model of contingent planning for agent programming languages. In *Proc. of AAMAS'08*, pages 477–484, 2008.

Hector J. Levesque. Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23:155–212, 1984.

Hector J. Levesque. Planning with loops. In *IJCAI*, pages 509–515, 2005.

Lior Limonad, Pieter De Leenheer, Mark Linehan, Rick Hull, and Roman Vaculin. Ontology of dynamic entities. In *Proc. of the 31st Int. Conf. on Conceptual Modeling (ER 2012)*, 2012.

F. Lin and W. Murray Wonham. Decentralized supervisory control of discrete-event systems. *Inf. Sci.*, 44(3):199–224, 1988a.

F. Lin and W.M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44(3):173 – 198, 1988b.

Fangzhen Lin and Ray Reiter. State constraints revisited. *J. of Logic Programming*, 4(5):655–678, 1994.

Alessio Lomuscio and Franco Raimondi. Model checking knowledge, strategies, and games in multi-agent systems. In *AAMAS*, pages 161–168, 2006.

Alessio Lomuscio and Mark Ryan. On the relation between interpreted systems and kripke models. In *Agents and Multi-Agent Systems Formalisms, Methodologies, and Applications*, pages 46–59, 1997.

Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. Mcmas: A model checker for the verification of multi-agent systems. In *CAV*, pages 682–688, 2009.

Robert Lundh, Lars Karlsson, and Alessandro Saffiotti. Automatic configuration of multi-robot systems: Planning for multiple steps. In *Proc. of ECAI'08*, pages 616–620, Patras, Greece, 2008.

Yoad Lustig and Moshe Y. Vardi. Synthesis from component libraries. In *FOSSACS*, pages 395–409, 2009.

Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, January 1980. ISSN 0164-0925.

S.A. McIlraith and T.C. Son. Adapting golog for composition of semantic web services. In *Proc. KR 2002*, 2002.

B. Medjahed, A. Bouguettaya, and A.K. Elmagarmid. Composing web services on the semantic web. *Very Large Data Base Journal*, 12(4):333–351, 2003.

Tim Menzies, Adrian Pearce, Clinton Heinze, and Simon Goss. What is an agent and why should i care? In MichaelG. Hinchey, JamesL. Rash, WalterF. Truszkowski, Christopher Rouff, and Diana Gordon-Spears, editors, *Formal Approaches to Agent-Based Systems*, volume 2699 of *Lecture Notes in Computer Science*, pages 1–14. Springer Berlin Heidelberg, 2003.

R. Van Der Meyden and N. V. Shilov. Model checking knowledge and time in systems with perfect recall (extended abstract). In *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, pages 432–445. Springer-Verlag, 1999.

Robin Milner. An Algebraic Definition of Simulation Between Programs. In *Proc. of IJCAI 1971*, 1971a.

Robin Milner. An algebraic definition of simulation between programs. In *Proc. of IJCAI*, pages 481–489, 1971b.

Robin Milner. An algebraic definition of simulation between programs. Technical report, Stanford University, Stanford, CA, USA, 1971c.

A. Muscholl and I. Walukiewicz. A lower bound on web services composition. *Logical Methods in Computer Science*, 4(2), 2008.

A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.

Object Management Group (OMG). Business process model and notation (bpmn) ver. 2.0, 2011. URL http://www.omg.org/spec/BPMN/2.0.

Eric Pacuit and Sunil Simon. Reasoning with protocols under imperfect information. *Review of Symbolic Logic*, 4(3):412–444, 2011.

Lin Padgham and Patrick Lambrix. Formalisations of capabilities for BDI-agents. *Autonomous Agents and Multi-Agent Systems*, 10(3):249–271, May 2005.

Flavio De Paoli, G. Lulli, and Andrea Maurino. Design of quality-based composite web services. In *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, volume 4294 of *Lecture Notes in Computer Science*, pages 153–164. Springer, 2006. ISBN 3-540-68147-7.

R. Parikh and R. Ramanujam. Distributed processes and the logic of knowledge. In *Logic of Programs*, pages 256–268, 1985.

David Michael Ritchie Park. Finiteness is Mu-ineffable. *Theoretical Computer Science*, 3(2):173–181, 1976.

Fabio Patrizi. *Simulation-based Techniques for Automated Service Composition*. PhD thesis, DIS, Sapienza Univ. Roma, 2009.

Wojciech Penczek and Alessio Lomuscio. Verifying epistemic properties of multi-agent systems via bounded model checking. *Fundam. Inform.*, 55(2):167–185, 2003.

Marco Pistore, Annapaola Marconi, Piergiorgio Bertoli, and Paolo Traverso. Automated composition of web services by planning at the knowledge level. In *IJCAI*, pages 1252–1259, 2005.

N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380, 2006a.

Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Proc. of VMCAI*, volume 3855 of *LNCS*, pages 364–380, Charleston, SC, USA, January 2006b. Springer.

Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proc. of POPL*, pages 179–190, 1989a.

Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989b.

Amir Pnueli and Elad Shahar. The TLV system and its applications. Technical report, Department of Computer Science, Weizmann Institute, Rehovot, Israel, 1996.

Michael O. Rabin. Decidability of Second Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

Knut Åkesson, Martin Fabian, Hugo Flordal, and Arash Vahidi. Supremica – a tool for verification and synthesis of discrete event supervisors. In *Proceedings of the 11th Mediterranean Conference on Control and Automation*, Rhodos, Greece, 2003.

P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25:206–230, January 1987. ISSN 0363-0129.

Peter J. Ramadge. Observability of discrete event systems. In *Decision and Control, 1986 25th IEEE Conference on*, volume 25, pages 1108 –1112, dec. 1986.

Peter J. Ramadge and W. M. Wonham. The control of discrete event systems. *IEEE Trans. on Control Theory*, 77(1):81–98, 1989a.

P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989b.

Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proc. of KR*, pages 473–484, 1991.

Anand S. Rao and Michael P. Georgeff. An abstract architecture for rational agents. In *Proc. of KR*, pages 438–449, San Mateo, CA, 1992.

C. Reiser, A.E.C. da Cunha, and J.E.R. Cury. The environment GRAIL for supervisory control of discrete event systems. In *Proc. of 8th International Workshop on Discrete Event Systems workshop*, pages 390 –391. IEEE Computer Society Press, july 2006.

Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001a.

Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Cambridge, MA, 2001b.

L. Ricker, S. Lafortune, and S. Gene. DESUMA: A tool integrating GIDDES and UMDES. In *Proc. of 8th International Workshop on Discrete Event Systems workshop*, pages 392 –393. IEEE Computer Society Press, 2006.

Jussi Rintanen. Complexity of Planning with Partial Observability. In *Proc. of ICAPS'04*, pages 345–354, 2004a.

Jussi Rintanen. Complexity of planning with partial observability. In *ICAPS*, pages 345–354, 2004b.

Yoram Moses Ronald Fagin, Joseph Y. Halpern and Moshe Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, MA, 1995.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010. ISBN 978-0-13-207148-2.

Shmuel Safra. On the complexity of ω-automata. In *FOCS*, pages 319–327, 1988.

Sebastian Sardiña, Fabio Patrizi, and Giuseppe De Giacomo. Automatic synthesis of a global behavior from multiple distributed behaviors. In *AAAI*, AAAI'07, pages 1063–1069, 2007.

Sebastian Sardina and Giuseppe De Giacomo. Realizing multiple autonomous agents through scheduling of shared devices. In *Proc. of ICAPS*, pages 304–312, 2008.

Sebastian Sardina, Giuseppe De Giacomo, and Fabio Patrizi. Behavior Composition in the Presence of Failure. In *Proc. of KR'08*, pages 640–650, 2008.

Sebastian Sardiña, Fabio Patrizi, and Giuseppe De Giacomo. Behavior composition in the presence of failure. In *KR*, pages 640–650, 2008.

Sebastian Sardina, Fabio Patrizi, and Giuseppe De Giacomo. Behavior composition in the presence of failure. In *Proc. of KR*, pages 640–650, 2008.

Dieter Schuller, André Miede, Julian Eckert, Ulrich Lampe, Apostolos Papageorgiou, and Ralf Steinmetz. Qos-based optimization of service compositions for complex workflows. In *Service-Oriented Computing - 8th International Conference, ICSOC 2010, San Francisco, CA, USA, December 7-10, 2010. Proceedings*, volume 6470 of *Lecture Notes in Computer Science*, pages 641–648, 2010. ISBN 978-3-642-17357-8.

Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. Learning generalized plans using abstract counting. In *AAAI*, pages 991–997, 2008.

Colin Stirling. *Modal and Temporal Properties of Processes*. Springer, 2001.

Colin Stirling and David Walker. Local model checking in the modal mu-calculus. *Theor. Comput. Sci.*, 89(1):161–177, 1991.

Thomas Stroeder and Maurice Pagnucco. Realising deterministic behaviour from multiple non-deterministic behaviours. In *Proc. of IJCAI*, pages 936–941, Pasadena, CA, USA, July 2009. AAAI Press.

Jianwen Su, editor. *IEEE Data Engineering Bulletin: Special Issue on Semantic Web Services*, volume 31:2, 2008a.

Jianwen Su, editor. *IEEE Data Engineering Bulletin: Special Issue on Semantic Web Services*, volume 31:2, 2008b.

Jianwen Su, editor. *Semantic Web Services: Composition and Analysis. IEEE Data Eng. Bull.*, volume 31. IEEE Comp. Society, 2008c.

Maurice H. ter Beek, Antonio Bucchiarone, and Stefania Gnesi. Formal Methods for Service Composition. *Annals of Mathematics, Computing and Teleinformatics*, 1 (5):1–10, 2007.

J. Thistle and W. Wonham. Supervision of infinite behavior of discrete-event systems. *SIAM Journal on Control and Optimization*, 32(4):1098–1113, 1994.

Wiebe van der Hoek and Michael Wooldridge. Tractable multiagent planning for epistemic goals. In *AAMAS*, pages 1167–1174, 2002.

Ron van der Meyden. Finite state implementations of knowledge-based programs. In *FSTTCS*, pages 262–273, 1996.

Moshe Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham Birtwistle, editors, *Logics for Concurrency*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer Berlin / Heidelberg, 1996.

Victor Vianu. Automatic verification of database-driven systems: a new frontier. In *Proc. of the 12th Int. Conf. on Database Theory (ICDT 2009)*, pages 1–13, 2009.

Nico Wallmeier, Patrick Hütten, and Wolfgang Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *In CIAA, LNCS*, pages 11–22. Springer, 2003.

Dirk Walther, Wiebe van der Hoek, and Michael Wooldridge. Alternating-time temporal logic with explicit strategies. In *Proc. of the Conference on Theoretical Aspects of Rationality and Knowledge*, pages 269–278, New York, NY, USA, 2007. ACM Press. doi: 10.1145/1324249.1324285.

Hongbing Wang, Xuan Zhou, Xiang Zhou, Weihong Liu, Wenya Li, and Athman Bouguettaya. Adaptive service composition based on reinforcement learning. In *Service-Oriented Computing - 8th International Conference, ICSOC 2010, San Francisco, CA, USA, December 7-10, 2010. Proceedings*, volume 6470 of *Lecture Notes in Computer Science*, pages 92–107, 2010. ISBN 978-3-642-17357-8.

Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.

Frank Wolter and Michael Zakharyaschev. Temporalizing description logic. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems*, pages 379–402. Studies Press/Wiley, 1999.

K.C. Wong and W.M. Wonham. Hierarchical control of discrete-event systems. *Discrete Event Dynamic Systems*, 6:241–273, 1996. ISSN 0924-6703.

W. M. Wonham. Supervisory control of discrete-event systems. Technical Report ECE 1636F/1637S 2011-12, University of Toronto, Toronto, Canada, 2012.

W. M. Wonham and P. J. Ramadge. On the supremal controllable sub-language of a given language. *SIAM Journal on Control and Optimization*, 25(3):637–659, 1987.

M. Wooldridge and A. Lomuscio. A computationally grounded logic of visibility, perception, and knowledge. *Logic Journal of the IGPL*, 9(2):273–288, 2001.

Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2nd edition, 2009a.

Michael Wooldridge. *Introduction to Multi-Agent Systems*. John Wiley & Sons, second edition, 2009b. ISBN 978-0470519462.

Michael J. Wooldridge. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2009c.

D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating daml-s web services composition using shop2. In *Proc. of ISWC-03*, 2003.

N. Yadav and S. Sardina. Qualitative approximate behavior composition. *Logics in Artificial Intelligence*, pages 450–462, 2012.

Nitin Yadav and Sebastian Sardina. Decision theoretic behavior composition. In Tumer, Yolum, Sonenberg, and Stone, editors, *Proc. of AAMAS*, pages 575–582, Taipei, Taiwan, May 2011. ACM Press.

Nitin Yadav and Sebastian Sardiña. Reasoning about agent programs using atl-like logics. In *JELIA*, pages 437–449, 2012.

Nitin Yadav, Paolo Felli, Giuseppe De Giacomo, and Sebastian Sardiña. Supremal realizability of behaviors with uncontrollable exogenous events. In *IJCAI*, 2013.

J. Yang and M.P. Papazoglou. Service components for managing the life-cycle of service compositions. *Information Systems*, 29(2):97–125, 2004.

Zhonghua Zhang and W. M. Wonham. STCT: An efficient algorithm for supervisory control design. In *Symposium on Supervisory Control of Discrete Event Systems*, pages 249–6399, 2001.

Haibo Zhao and Prashant Doshi. A Hierarchical Framework for Composing Nested Web Processes. In *Proc. ICSOC 2006*, 2006.