Interaction Models and Automated Control under Partial Observable Environments

Daniel Ciolek*, Victor Braberman*, Nicolás D'Ippolito†*, Nir Piterman+ and Sebastián Uchitel†*

Abstract—The problem of automatically constructing a software component such that when executed in a given environment satisfies a goal, is recurrent in software engineering. Controller synthesis is a field which fits into this vision. In this paper we study controller synthesis for partially observable LTS models. We exploit the link between partially observable control and non-determinism and show that, unlike fully observable LTS or Kripke structure control problems, in this setting the existence of a solution depends on the interaction model between the controller-to-be and its environment. We identify two interaction models, namely Interface Automata and Weak Interface Automata, define appropriate control problems and describe synthesis algorithms for each of them.

Index Terms—LTS, Controller Synthesis, Imperfect-Information Games.

1 Introduction

Component-based construction of software is based on the idea that each component operationally contributes to achieve a sub-goal and that the conjunction of these sub-goals achieves system requirements set out by stakeholders. Upon a specification change only affected components need to be updated in order to adapt to the new requirements. In this context, producing correct-by-construction components is expected to ease development and maintenance, and has been pursued in many guises and increasingly in adaptive systems (e.g, [1], [2], [3], [4], [5]).

Controller synthesis is a field which fits into this vision and that has been successful in hardware engineering (a highly componentised engineering discipline). Traditional controller synthesis (e.g. [6]) focuses on a component interaction model based on shared memory, hence Kripke structures are used as the formal setting for behaviour modelling. However, in software engineering, a common and relevant interaction model used to reason about system behaviour is event-based; including message passing and remote procedure calls among others. In particular we address control problems for behaviour models expressed as Labelled Transition Systems (LTS) and parallel composition defined broadly as synchronous product. A setting widely adopted in the software engineering literature.

We ground the general controller synthesis formulation: given a model of the assumed behaviour of the environment, produce an operational behaviour model for a component M such that when executing M in a consistent environment E is guaranteed to satisfy a goal G, namely $E\|M \models G$. We specify goals with a linear temporal logic based of fluents [7] (FLTL) and operational models with LTS, where the controller is able to control some events and only monitor others [2], [8].

* *Departamento de Computación, Universidad de Buenos Aires, Argentina*
† *Department of Computing, Imperial College, London, UK*
+ *Department of Computer Science, University of Leicester, Leicester, UK*
Manuscript received DATE; revised DATE.
This work was partially supported by grants ERC PBM-FIMBSE, ANPCYT PICT 2011-1774, ANPCYT PICT 2012-0724, ANPCYT PICT 2013-2341, UBACYT 20020130100384BA, UBACYT 20020130300036BA, CONICET PIP 112 201301 00688 CO, MEALS 295261.

In this paper we focus on partial observability with general goal specifications (including liveness), but restricting goal specifications to predicate on actions observable to the controller (which allows for a simpler technical solution). A partially observable environment has the capacity of altering its state through an internal action (τ) invisible to the controller. Internal actions naturally appear when there is limited visibility of the environment and can also arise from an abstraction of the behaviour of the environment. Thus, partial observability results in uncertainty on the controller's side regarding the exact state of the environment. Intuitively the behaviour of a partially observable system can be captured by means of non-determinism. We show that an adaptation of the classical notion of τ-closure (as presented in [9]) is required in order to preserve controllability while translating the effects of partial observation to non-determinism. On that account we exploit the link between control under partial observability and non-determinism for event-based control and devise algorithmic solutions for both settings.

Having fixed LTS and FLTL as languages for E, M and G, a key consideration is the interaction mechanism modelled by the parallel composition operator $\|$. In this paper we explore two settings. The first is based on Interface Automata (IA) [10] in which M must not, at any point in time, block uncontrollable actions; nor issue actions that are not enabled by E at that point in time. The second setting, Weak Interface Automata (WIA), relaxes the latter restriction and only expects M not to block uncontrollable actions that E may wish to perform. Indeed, this weaker setting allows M to offer more controlled actions than what the environment may accept. This is akin to external choice in CSP [9].

The IA setting is appropriate for problem domains in which a failure ensues when a controller attempts to perform an action at a point in time when it is proscribed by the environment. Such is commonly the case in robotics and cyber-physical systems in general, but also in software systems, for example in the use of APIs specified using typestates [11], [12]. The WIA setting is appropriate for

problem domains in which interactions between M and E are based on handshakes, rendezvous, and contexts in which the controller can sense the subset of actions enabled by the environment at a certain point in time.

The interaction model distinction is important because in the WIA setting, controllers have additional capabilities of reducing uncertainty about the system state and thus can realise goals that are not achievable otherwise. The WIA model is effectively a weaker version of the IA model that arises from abstracting a handshake communication mechanism. Hence, if there is a solution to the IA synthesis problem there is also a solution in the WIA setting, but the converse is not always true. Interestingly, the distinction between interaction models is irrelevant for deterministic control problems because the controller always has full information about the environment's state, and for this reason has never been studied for controller synthesis.

The first contribution of this paper is a reduction from partially observable LTS control problems to non-deterministic LTS control. We make clear the distinction between the problems and show that although non-determinism has been extensively studied for Kripke style models (e.g., [13]), its application to partially observable event-based models raises different issues depending on the specific interaction model chosen.

The second contribution of this paper is the definition of IA and WIA non-deterministic LTS control problems and algorithms for solving them. For this we use two different styles of reductions, one for each interaction model. We reduce the IA control problem to a deterministic LTS control problem and also show how naive determinisation (in the spirit of [14]) fails in the presence of liveness assumptions and goals. On the other hand we reduce the WIA control problem to an imperfect-information game [13] and comment on a more direct and optimised algorithmic solution.

The rest of the paper is organised as follows. In Section 2 we motivate and explain the difference between the interaction models. Section 3 provides background that serves as basis for the results herein. In Section 4 we present and define IA and WIA LTS control problems for non-deterministic domains. In Section 5 we present a reduction from partially observable problems to the non-deterministic case. Evaluation is provided in Section 6. Related work is presented in Section 7. We close with a discussion about future work and conclusions. Formal proofs for stated properties can be found as a supplemental Appendix.

2 MOTIVATION

In this section we provide an overview of what is meant by controller synthesis and motivate the need for considering the appropriate interaction model of the domain in which a controller is to be deployed. We make casual use of the LTS framework formally described in Section 3.

Consider the simple model depicted with an LTS in Fig. 1a, which details the sequences of requests a server can service without failing. The model stipulates that after opening a session the server may be ready to provide different services, namely x and y. At any point, the session may be closed. Note than on opening the session, the server may switch to maintenance mode, as a consequence clients

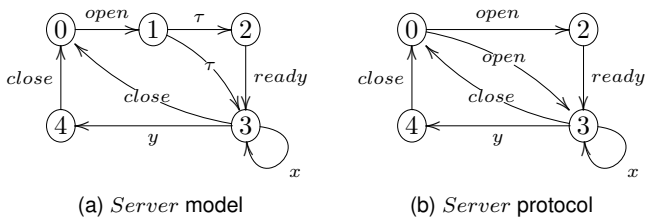


Fig. 1: Motivating Example

must wait for a $ready$ event before making further requests. Choosing between entering maintenance or not are internal actions invisible to the client, thus characterized with τ. A client connecting to this server must follow a valid sequence of requests; a sequence such as $open$, y, x, $close$ is guaranteed to fail as x cannot be served after y.

Observe that since switching to maintenance is an internal action, a client cannot determine the state the server is in after opening the session. Thus from a client point of view internal actions are absent and the protocol for communicating with the server exhibits non-deterministic behaviour, as depicted in Fig. 1b.

Controller synthesis can be used to automatically generate client behaviour that will guarantee a given goal. In its simplest (and naive) formulation, it is only a matter of automatically constructing a $Controller$ LTS that restricts occurrence of controlled actions (in this example all but $ready$) and does so in a way that when composed in parallel with the $Server$ LTS satisfies the goal.

Let G be the simple goal of achieving y infinitely often. In Fig. 2 we depict an LTS that satisfies this property when composed with the $Server$ LTS using standard LTS composition.

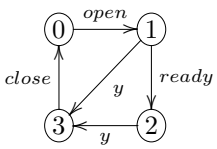


Fig. 2: $Controller$ for goal G

Although the LTS in Fig. 2 would seem to be an adequate controller for the $Server$, consider what happens when enacting such controller by message passing: the controller would first $open$ a session and then would either wait for the $ready$ event or decide to go ahead with requesting y. Should the environment be in a maintenance state (state 2 in Fig. 1a) when requesting y, a failure would ensue. Hence, requesting x or y is not safe until the server has come out of the maintenance state, i.e. wait for $ready$. However, it may be the case that the $Server$ did not enter in maintenance (and is in state 3 in Fig. 1a), thus checking for $ready$ leads to waiting forever (not achieving the goal).

The reason why the LTS in Fig. 2 is not guaranteed to achieve its goal when enacted against the server, while its composition against the $Server$ LTS does satisfy the goal, is that standard LTS product does not match the client-server interaction model of the domain where the controller is to be enacted.

The mismatch between parallel composition and the interaction model for our motivating example can be resolved by requesting controllers to be legal in the sense of Interface Automata [10]: the controller when executed in parallel with the server must, at any point in time, not block uncontrollable actions (i.e. $ready$) nor trigger actions that are not enabled by the environment. Interface automata were developed to capture precisely the interaction model of service provision through interfaces with no handshaking.

It is straightforward to see that with the requirement of being a legal interface automaton, there can be no controller that achieves the goal as the controller does not have sufficient information about the state of the server after the $open$ action. Had the controller been enacted in a context based on rendezvous communication (provided by languages such as ADA amongst others) then the requirements of being an interface automaton would not be necessary. In this setting, the controller would be prevented (by the environment) from executing a controlled action that the environment is not prepared to accept. Thus, an LTS such as the one depicted in Fig. 2 would actually achieve its goals: having opened a session, it could try to do y and would only succeed if the environment is not in maintenance mode; should the environment be in maintenance, the controller would have no option but to wait for the $ready$ event.

The possibility of relying on a handshaking mechanism is the difference between there being a controller that can achieve the goal or not. Indeed, this is because this interaction model allows the controller to gain some knowledge about the state of the environment and exploit this knowledge to achieve its goals. Clearly, should the controller have full knowledge of the state of the environment, one or the other interaction model provide no advantage.

In summary, the example discussed in this section attempts to establish three points. The first is that the domain in which a controller is to be enacted determines an interaction model that must be appropriately captured when performing synthesis. The second is that standard LTS product is only appropriate for some interaction models (i.e., relying in some form of handshaking). The third is that some interaction models provide the controller with additional capabilities for reducing uncertainty about the state of the environment and thus allow resolving control problems that in other settings may be unrealisable.

In Section 4 we formalise two non-deterministic controller synthesis problems, one for each interaction model motivated above, and show how they can be resolved. In Section 5 we formalise the same problems for partially observable domains and provide reductions to the non-deterministic case. Each control problem requires different algorithmic treatment, so their specific synthesis algorithms are also reported.

3 Background

3.1 Labelled Transition Systems

Definition 1 (Labelled Transition Systems). A *Labelled Transition System* (LTS) is a tuple $E{=}(S,A,\Delta,s_0)$, where S is a finite set of states, A is its communicating alphabet, $\Delta \subseteq (S \times A \times S)$ is a transition relation, and $s_0 \in S$ is the initial state.

An LTS is *deterministic* if (s,ℓ,s') and (s,ℓ,s'') are in Δ implies $s' = s''$. An LTS is *partially observable* if the special label τ belongs to A.

Notation 1. Let E and M be LTSs such that:

- $E = (S_E, A_E, \Delta_E, s_0)$
- $M = (T_M, A_M, \Delta_M, t_0)$
- s and s' are states of S_E
- t and t' are states of T_M

Notation 2. We will denote $(s,\ell,s') \in \Delta_E$ by $s \xrightarrow{\ell}_E s'$ and call it a *step*.

Definition 2 (Parallel Composition). The *Parallel Composition* ($\|$) is a symmetric operator such that it yields an LTS $E\|M = (S_E \times T_M, A_E \cup A_M, \Delta, \langle s_0,t_0\rangle)$, where Δ is the smallest relation that satisfies the rules:

$$\frac{s \xrightarrow{\ell}_E s'}{\langle s,t\rangle \xrightarrow{\ell}_{E\|M} \langle s',t\rangle}\ \ell \in (A_E \setminus A_M) \cup \{\tau\}$$

$$\frac{t \xrightarrow{\ell}_M t'}{\langle s,t\rangle \xrightarrow{\ell}_{E\|M} \langle s,t'\rangle}\ \ell \in (A_M \setminus A_E) \cup \{\tau\}$$

$$\frac{s \xrightarrow{\ell}_E s', t \xrightarrow{\ell}_M t'}{\langle s,t\rangle \xrightarrow{\ell}_{E\|M} \langle s',t'\rangle}\ \ell \in (A_E \cap A_M) \setminus \{\tau\}$$

We restrict attention to states in $S_E \times T_M$ that are reachable from the initial state $\langle s_0,t_0\rangle$ using transitions in Δ. Note that the special transition τ does not cause an interaction in the parallel composition.

Notation 3. We denote by $s \overset{\ell}{\Rightarrow}_E s'$ that there exist a sequence of transitions starting in a state s and reaching a state s' after zero or more τ-transitions followed by exactly one ℓ transition (where ℓ could be τ) and again followed by zero or more τ-transitions; we call $s \overset{\ell}{\Rightarrow}_E s'$ a *walk*. More formally, there exists a sequence of states $s_0,\ldots,s_i,s_{i+1},\ldots,s_n$, such that $s_0 = s$, $s_n = s'$ and

$$s_0 \xrightarrow{\tau}_E \ldots \xrightarrow{\tau}_E s_i \xrightarrow{\ell}_E s_{i+1} \xrightarrow{\tau}_E \ldots \xrightarrow{\tau}_E s_n$$

A *walk* captures the notion of τ-closure as in [9].

Notation 4. We use the following notation to refer to set of transition labels:

- $\Delta_E(s) = \{\ell \mid s \xrightarrow{\ell}_E s'\}$, the set of outgoing transition labels from state s
- $\Delta^*_E(s) = \{\ell \mid s \overset{\ell}{\Rightarrow}_E s' \wedge \ell \neq \tau\}$, the set of τ-reachable transitions labels from state s

Note that when $\tau \notin A_E$, $\Delta_E(s) = \Delta^*_E(s)$. We say a state s of an LTS E is a *deadlock* if and only if $\Delta_E(s) = \emptyset$.

Given an LTS E, controller synthesis seeks for an LTS M that restricts E in order to achieve a given goal. Nevertheless further restrictions can be imposed on M. Naturally given a partition of the alphabet of E to *controllable* and *uncontrollable* actions (i.e. $A_E = A_C \dot\cup A_U$), we want M to only observe the actions in A_U and to control the occurrence of actions in A_C. Additionally we differentiate controllers with the capability of sensing the set of available controllable actions. Thus we identify two possible requirements for valid controllers.

Definition 3 (Weak Legal LTS). An LTS M is a *weak legal* LTS for E if for all reachable states $\langle s,t\rangle$ of $E\|M$ it holds that $\Delta_E^*(s)\cap A_U \subseteq \Delta_M(t)\cap A_U$, or informally reachable uncontrollable actions are not blocked.

Definition 4 (Legal LTS). An LTS M is a *legal* LTS for E if it is *weak legal* and for all reachable states $\langle s,t\rangle$ of $E\|M$ it holds that $\Delta_E^*(s)\cap A_C \supseteq \Delta_M(t)\cap A_C$, or informally in addition to not blocking uncontrollable actions M does not enable controllable actions not allowed by E.

3.2 Fluent Linear Temporal Logic

Definition 5 (Trace). A trace of an LTS E is a τ-free sequence of labels $\pi{=}\ell_0,\ell_1\ldots$, for which there exists a sequence of states $s_0,s_1,\ldots$ such that $\forall i{\geq}0\,.\,s_i \stackrel{\ell_i}{\Rightarrow}_E s_{i+1}$.

We fix Fluent Linear Temporal Logic (FLTL) [7] as the language for describing properties of LTS traces. FLTL is a linear-time temporal logic for reasoning about fluents. The logic has the same expressiveness as standard LTL [15]. However, as fluents can be used to overlay state-based propositions on an event-based model, FLTL allows for a more compact representation of properties. Intuitively fluents are boolean values that can change over time as a consequence of events.

Given a set of labels A, a *fluent* $Fl{=}(I_{Fl},T_{Fl},\mathit{Init}_{Fl})$ is defined by a pair of sets and a Boolean value, where $I_{Fl}\subseteq A$ is the set of initiating actions and $T_{Fl}\subseteq A$ is the set of terminating actions such that $I_{Fl}\cap T_{Fl}=\emptyset$. A fluent may be initially *true* or *false* as indicated by Init_{Fl}. Naturally, every label $\ell\in A$ induces a fluent, namely $\mathit{fl}_\ell=(\{\ell\},A\backslash\{\ell\},\mathit{false})$.

Let $\mathcal{F}$ be the set of all possible fluents over A. An FLTL formula is defined inductively using the standard Boolean connectives and temporal operators $\mathbf{X}$ (next), $\mathbf{U}$ (strong until) as follows:

$$\varphi ::= Fl \mid \neg\varphi \mid \varphi\vee\psi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\psi$$

where $Fl\in\mathcal{F}$. Additionally, as it is usual, we introduce $\wedge$, $\Diamond$ (eventually), $\Box$ (always) and $\mathbf{W}$ (weak until) as syntactic sugar.

Let Π be the set of infinite traces over A. The trace $\pi=\ell_0,\ell_1,\ldots$ satisfies a fluent Fl at position i, denoted $\pi,i\models Fl$, if and only if one of the following holds:

- $\mathit{Init}_{Fl}\wedge(\forall j\,.\,0\leq j\leq i\Rightarrow \ell_j\notin T_{Fl})$
- $\exists j\,.\,j\leq i\wedge\ell_j\in I_{Fl}\wedge(\forall k\,.\,j<k\leq i\Rightarrow \ell_k\notin T_{Fl})$

In other words, a fluent holds at position i if and only if it holds initially and has not been terminated; or if some initiating action has occurred, but no terminating action has yet occurred. Note that fluents are not influenced by τ happening.

For an infinite trace π, the semantics of a (composite) formula φ at position i, denoted $\pi,i\models\varphi$, is commonly defined as follows:

$$\begin{array}{lcl}
\pi,i\models\neg\varphi & \triangleq & \neg(\pi,i\models\varphi)\\
\pi,i\models\varphi\vee\psi & \triangleq & (\pi,i\models\varphi)\vee(\pi,i\models\psi)\\
\pi,i\models\mathbf{X}\varphi & \triangleq & \pi,i{+}1\models\varphi\\
\pi,i\models\varphi\mathbf{U}\psi & \triangleq & \exists j\,.\,i\leq j\wedge(\pi,j\models\psi)\wedge\\
 & & \forall k\,.\,i\leq k<j\Rightarrow(\pi,k\models\varphi)
\end{array}$$

We say that an FLTL formula φ holds in the trace π, denoted $\pi\models\varphi$, if $\pi,0\models\varphi$. Furthermore, φ holds in an LTS E (denoted $E\models\varphi$) if it holds on every infinite trace produced by E.

3.3 Imperfect-Information Games

The problem of synthesising a controller for a non-deterministic environment model can be translated to finding a winning strategy for an Imperfect-Information Game (IIG), a problem that has been studied thoroughly. Such strategy should take into account observations made in order to reduce uncertainty in the attempt to determine a course of action [13].

Definition 6 (Imperfect-Information Game). An *Imperfect-Information Game* is a tuple $G{=}(V,O,\Gamma,\delta,v_0,L,F)$, where V is a set of locations, O a set of observations, Γ a set of moves, $\delta: V\times\Gamma\to 2^V$ a transition function associating every location and move with a non-empty set of possible successor locations, $v_0\in V$ an initial location, $L: V\to O$ a labelling function, and $F\subseteq O^\omega$ a winning condition given as the set of desirable sequences of observations.

A game is said to be *full information* if $O=V$ and for every location we have $L(v)=v$.

A play in G starts by placing a token on v_0. When the token is on node v the play is extended by Player 1 choosing a move $\gamma\in\Gamma$ and by Player 2 choosing a successor $v'\in\delta(v,\gamma)$. An infinite play $p=v_0,v_1,\ldots$ is winning for Player 1 if $L(v_0),L(v_1),\ldots\in F$.

Definition 7 (Observation-Based Strategy). An *Observation-Based Strategy* for Player 1 with memory Ω is a pair $\langle\alpha,\eta\rangle$, where $\alpha:\Omega\to\Gamma$ guides the next move based on the memory contents, and $\eta:\Omega\times O\to\Omega$ updates the memory based on an observation. With Ω there is an associated initial memory value ω_0.

An infinite play $p=v_0,v_1,\ldots$ is *consistent* with strategy $\langle\alpha,\eta\rangle$ if there is a sequence of memory values $\omega_0,\omega_1,\ldots$, such that for every $i{\geq}0$, $v_{i+1}\in\delta(v_i,\alpha(\omega_i))$ and $\omega_{i+1}=\eta(\omega_i,L(v_{i+1}))$. That is, the next location is chosen according to a move advised by α and the memory is updated according to η given the observation at location v_{i+1}. A strategy $\langle\alpha,\eta\rangle$ is *sure winning* for Player 1 if all plays consistent with $\langle\alpha,\eta\rangle$ are winning for her. Player 1 wins the game G if she has a *sure winning* strategy for G.

A partial information game is solved by establishing whether the game is won by Player 1. This is done by converting G to a full-information game obtained by a construction akin to the determinisation construction for finite automata. This construction is many times referred to as *information based determinisation* [16]. Essentially, the construction summarizes the information that Player 1 has regarding the location in the game that she could be at. If she can win based on this information then she can win all plays.

Note that the notion of imperfect information employed is not as general as it might be, since the specification represented by the FLTL formula is expressed solely in terms of information observable to the controller. This may seem too restrictive and is indeed one of the current limitations of

our technique. However, from a software engineering point of view, abstraction is an essential process used to deal with the complexity of systems and requirements and, as in a good API design, one makes a compromise about what to expose in order to allow the resolution of objectives. Furthermore, this simplification allows us to provide simpler constructions that the one needed to deal with the general problem that include the deduction of unobservable fluents (e.g. [17]).

4 Non-Deterministic control problems

In this section we define two control problems for environments described as non-deterministic LTS (with no internal actions). In Section 5 we extend this result for partially observable environments. First we present a control problem in which we require the solution to be a *legal* LTS. Then we present a control problem in which we require the solution to be a *weak legal* LTS.

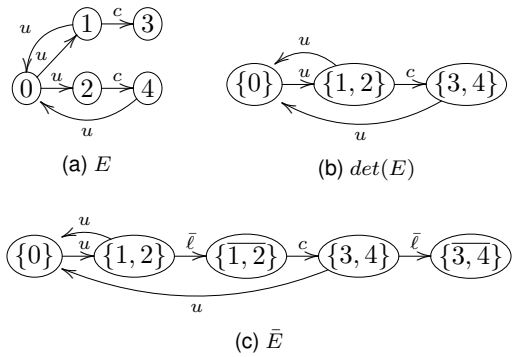
4.1 Interface Automata Control

Definition 8 (IA ND LTS Control). Given a τ-free LTS E, a set of controllable actions $A_C \subseteq A_E$, and an FLTL formula φ, a solution for the IA non-deterministic LTS control problem $\mathcal{E} = (E, \varphi, A_C)$ is an LTS M such that M is a *legal* LTS for E, $E\|M$ is deadlock-free, and $E\|M \models \varphi$.

We refer to the solution of an IA non-deterministic LTS control problem as an IA controller. If a controller exists we say that the problem is realisable. It is unrealisable otherwise.

Synthesis techniques for solving control problems for non-deterministic environment models in the form of Kripke structures have been studied for some time and are not trivial. There are a number of difficulties to be addressed. Dealing with the uncertainty of the exact state of the environment is one shared by all techniques. In general, this issue is addressed by modifying the environment model, possibly extending its alphabet, and considering some deterministic version of it. Typically, techniques that work for safety goals apply classical determinisation [14], considering the power set of states to capture all the possible states the environment could be in after a non-deterministic action. Accounting for the partition of the alphabet is usually done by including controllable actions allowed in all the states in which the environment could be in, while including uncontrollable actions available in at least one of those states.

Unfortunately, when dealing with a partition of the alphabet and more general goals, classical determinisation does not work. For instance, consider the IA LTS control problem $\mathcal{E} = (E, \Box\Diamond u, \{c\})$, where E is the environment model in Fig. 3a and we abuse notation and use u as the fluent induced by the label u – i.e. fl_u=($\{u\}$, $A\backslash\{u\}$,*false*). From state 0 in E there is a non-deterministic choice over the uncontrollable action u that leads either to state 1 or 2. As the goal for the controller is to guarantee infinitely many u actions, if state 1 is reached the controller should disable c to prevent the environment from reaching a deadlock. On the other hand, if state 2 is reached the controller must enable c to avoid deadlock, continuing to state 4 where the environment is forced to go back to state 0. As the controller (when composed with the environment) cannot identify the exact state the environment is in, it is impossible to determine whether to enable or disable c. Thus, there is no solution for $\mathcal{E}$. However, classical determinisation yields the LTS in Fig. 3b, which can be controlled by either disabling or enabling c from state $\{1, 2\}$. Consequently, determinisation must be revisited in the context of non-deterministic control problems.



(a) E (b) $det(E)$ (c) $\bar{E}$

Fig. 3: Determinisation Example

In order to synthesise IA controllers and preserve realisability a more sophisticated notion of determinisation is required. We have to take into account that disallowing all controlled actions may lead to a deadlock when composing the resulting controller with the environment. In particular, one needs to handle the case where no uncontrollable action is enabled in some reachable state – i.e. pure controllable states.

In Def. 9 we present the notion of *Controlled Determinisation*, which provides the basis for synthesising IA controllers that guarantee satisfaction of its goals even in the presence of non-determinism in the environment. Controlled determinisation not only preserves realisability but also allows for *legal* LTS controllers. As with classical determinisation, in controlled determinisation states are subsets of the states of the original LTS. However, the procedure determinises the model distinguishing states according to their outgoing actions (either controllable or uncontrollable) and discerns them by marking the states with controllable actions.

Definition 9 (Controlled Determinisation). Given an LTS $E = (S, A, \Delta, s_0)$, where $A_E = A_C \dot\cup A_U$, we define the controlled deterministic version of E, $\bar{E} = (\bar{Q}, \bar{A}, \bar{\Delta}, q_0)$ as follows:

- $\bar{Q} = 2^S \cup \{\bar{q} \mid q \in 2^S\}$
- $\bar{A} = A \cup \{\bar{\ell}\}$ and $\bar{A}_C = A_C$, with $\bar{\ell} \notin A$
- $q_0 = \{s_0\}$
- $\bar{\Delta}$ the transition relation:

$$\bar{\Delta} = \{\, q \xrightarrow{u}_{\bar{E}} \Delta(q,u) \mid \exists s{\in}q \;.\; u \in \Delta(s) \cap A_U \,\} \cup \quad (1)$$
$$\{\, q \xrightarrow{\bar{\ell}}_{\bar{E}} \bar{q} \mid \exists s{\in}q \;.\; \Delta(s) \cap A_U = \emptyset \,\} \cup \quad (2)$$
$$\{\, \bar{q} \xrightarrow{c}_{\bar{E}} \Delta(q,c) \mid \forall s{\in}q \;.\; c \in \Delta(s) \cap A_C \,\} \quad (3)$$

where $\Delta(q, \ell) = \{\, s' \mid s \in q \wedge s \xrightarrow{\ell}_E s' \}$ denotes the set of states reachable from q after an ℓ-transition.

Controlled determinisation yields a deterministic LTS revising the original by adding a fresh action $\bar{\ell}$ and states with transitions over that action (see Fig. 3c). Note that

a state $q \in \bar{Q}$ consists of subsets of S, which can be either marked or unmarked. By (1) uncontrollable actions are only possible from an unmarked state q, while by (3) controllable actions are only available from a marked state $\bar{q}$. By (2) an $\bar{\ell}$ transition exists between q and $\bar{q}$ if and only if there is at least one fully controllable state s in q, thus $\bar{q}$ is unreachable otherwise. This effectively partitions the states of $\bar{E}$ in fully controllable states (i.e. marked) and completely uncontrollable states (i.e. unmarked), and so if a marked state is reached the controller must enable at least one controllable action in order to avoid deadlock.

As a consequence of the separation of controllable and uncontrollable states, controllable actions are pruned from mixed states (states with both controllable and uncontrollable actions), still we show that controllability is preserved. Intuitively, mixed states represent a race condition between the controller and the environment, which in the worst case can always be won by the environment. Hence, if there is a solution for the control problem it cannot rely on the result of a race condition.

In order to synthesise controllers for the translated models it is necessary to change the FLTL formulas to ignore the transition label $\bar{\ell}$. We define the *alphabetised next* version of φ, denoted $\mathcal{X}_{\bar{\ell}}(\varphi)$, by replacing every sub-formula $\mathbf{X}\psi$ in φ by $\mathbf{X}(\bar{\ell}\,\mathbf{U}\,\mathcal{X}_{\bar{\ell}}(\psi))$. Thus, this transformation replaces every next operator occurring in the formula by an until operator that skips the uninteresting action $\bar{\ell}$.

Property 1. Given a trace $\pi = \ell_0, \ell_1, \ldots$ of an LTS E and an FLTL formula φ, we have that for every trace π' such that when removing every occurrence of the label $\bar{\ell}$ is equal to π, it holds that $\pi \models \varphi$ if and only if $\pi' \models \mathcal{X}_{\bar{\ell}}(\varphi)$.

Theorem 1 (Determinization Correctness). Given an IA non-deterministic LTS control problem $\mathcal{E}{=}(E, \varphi, A_C)$ and its controlled deterministic version $\bar{\mathcal{E}}{=}(\bar{E}, \mathcal{X}_{\bar{\ell}}(\varphi), A_C)$, $\mathcal{E}$ is realizable if and only if $\bar{\mathcal{E}}$ is realizable.

Note that $\bar{\mathcal{E}}$ is a deterministic control problem, hence event-based control synthesis algorithms can be applied. However, in the worst case the number of states of the deterministic problem can be exponentially larger than the original non-deterministic problem. Notably a simple modification to the solution for the deterministic problem is enough to control the analogous non-deterministic problem. As exhibited in the proof (in the Appendix) the modification consists of removing the spurious transition $\bar{\ell}$.

4.2 Weak Interface Automata Control

Definition 10 (WIA ND LTS Control). Given a τ-free LTS E, a set of controllable actions $A_C \subseteq A_E$, and an FLTL formula φ, a solution for the WIA non-deterministic LTS control problem $\mathcal{E} = (E, \varphi, A_C)$ is an LTS M such that M is *weak legal* for E (cf. Def. 3), $E \| M$ is deadlock-free, and $E \| M \models \varphi$.

We may refer to the solution of a WIA non-deterministic LTS control problem as a WIA controller.

In Def. 11 we present a reduction from WIA control problems to imperfect-information games [13].

Definition 11 (WIA to IIG). Given a WIA LTS control problem $\mathcal{E} = (E, \varphi, A_C)$, where $E = (S, A, \Delta, s_0)$ and $A = A_C \dot{\cup} A_U$, we define the Imperfect-Information Game $G_E = (V, O, \Gamma, \delta, v_0, L, F)$ as follows:

- $O = A \cup \{\lambda\}$, observations
- $V = (S \times (A \cup \{\lambda\})) \cup \{s_e\}$, locations
- $\Gamma = 2^{A_C}$, moves
- δ is the transition function:

$$\delta = \begin{Bmatrix} (\langle s, \ell\rangle, \gamma, s_e) & \big| \gamma{\in}\Gamma \wedge \Delta(s) \subseteq A_C \backslash \gamma \\ (\langle s, \ell\rangle, \gamma, \langle s', \ell'\rangle) & \big| \gamma{\in}\Gamma \wedge s \xrightarrow{\ell'}_E s' \wedge \ell' {\in} A_U {\cup} \gamma \\ (s_e, \gamma, s_e) & \big| \gamma{\in}\Gamma \end{Bmatrix}$$

- $v_0 = \langle s_0, \lambda\rangle$, an initial location
- $L : V \to O$ is the labelling function:

$$L(v) = \begin{cases} \ell & \text{if } v = \langle s, \ell\rangle \\ \lambda & \text{if } v = s_e \end{cases}$$

- $F = \{\ell_0, \ell_1, \ell_2, \ldots \mid \ell_1, \ell_2, \ldots \models \varphi\}$, the winning condition. Observe that we remove the initial spurious label $\ell_0 = \lambda$ and consider whether the sequence of actions in the rest of the game satisfy the goal φ. In particular, F does not contain sequences with λ and hence forces not to reach the sink state s_e (avoiding to stay in s_e forever).

We now provide an intuition on how the reduction works, and why, although the resulting game may seem very different from the original LTS, it captures the essence of the original problem. An example of the transformation is given in Fig. 4, in which for ease of representation we omit transitions from deadlock locations to the sink location s_e.

Locations in G_E are states of S decorated with actions of A denoting the label of the transition that lead to them in E. Observations match the label contained in a given location. The transition function δ is designed to encode, from every location, all possible choices for a controller of E. Thus, moves in the game represent sets of controllable actions. Recall that turns in imperfect-information games proceed as follows: Player 1 (the controller) chooses a move, then Player 2 (the environment) gets to choose which of the possible target locations is reached – i.e. the environment resolves the non-determinism. Therefore, in G_E, Player 1 chooses some set of controllable actions γ to enable. Then, Player 2 chooses the next location based on where one of the offered controllable actions or some uncontrollable action would lead to in E. From the perspective of Player 1, this is non-deterministic as she offers a move and it is the other player who chooses the destination. The encoding also includes a fresh location s_e which symbolizes E reaching a deadlock, including but not exclusively, the case in which the controller decides to disable all controllable actions from a fully controllable state.

Having defined the translation from a WIA control problem $\mathcal{E}$ to the imperfect-information game G_E, we show that there is a solution to $\mathcal{E}$ if and only if Player 1 (the controller) can surely win G_E. Preliminarily, we describe how to translate a strategy to an LTS (Def. 12) and vice-versa (Def. 13).

Definition 12 (Strategy to LTS). Given a WIA LTS control problem $\mathcal{E}{=}(E, \varphi, A_C)$ and $\langle \alpha, \eta\rangle$ an observation-based strategy for G_E with memory Ω. We construct an LTS $M = (T_M, A_M, \Delta_M, t_0)$ solution for $\mathcal{E}$, such that:

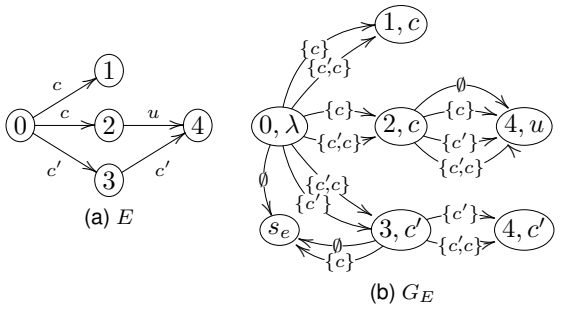


Fig. 4: Example: LTS to IIG

- $T_M = \Omega$
- $A_M = A_E$
- $t_0 = w_0$
- $\Delta_M = \{w \xrightarrow{\ell}_M w' \mid \ell \in \alpha(w) \cup A_U \wedge w' = \eta(w, \ell)\}$

Definition 13 (LTS to Strategy). Given a WIA LTS control problem $\mathcal{E}=(E, \varphi, A_C)$ and $M = (T_M, A_M, \Delta_M, t_0)$ a solution for $\mathcal{E}$. We construct an observation-based strategy $\langle \alpha, \eta \rangle$ with memory Ω for G_E, where:

- $\Omega = 2^{T_M}$
- $w_0 = \{t_0\}$
- $\alpha(w) = \bigcup_{t \in w} \Delta_M(t) \cap A_C$
- $\eta(w, \ell) = \begin{cases} \bigcup_{t \in w} \Delta_M(t, \ell) & \text{if } \exists t \in w \,.\, \Delta_M(t, \ell) \neq \emptyset \\ w & \text{otherwise} \end{cases}$

Theorem 2 (WIA to IIG Correctness). Given a WIA LTS control problem $\mathcal{E}=(E, \varphi, A_C)$ and its corresponding imperfect-information game G_E, $\mathcal{E}$ is realisable if and only if G_E is surely-won by Player 1.

4.2.1 Optimising realisability

Solving a WIA non-deterministic LTS control problem involves a sequence of reductions. First, the LTS environment is converted to an imperfect-information game, which is then converted to a full-information game. The first reduction causes an exponential explosion of the alphabet. The second reduction causes an exponential explosion of the number of states. We have found that in practice, the second exponential explosion is restricted by the structure of the LTS and is "well behaved". The first exponential explosion, on the other hand, occurs in all cases, and was a major bottleneck in our implementation. However, we have added an optimisation that circumvents the first exponential blow up. The optimisation computes the best successor location considering actions one by one and not exponentially many sets of actions.

Fix an LTS $E = (S, A, \Delta, s_0)$, with $A = A_C \dot\cup A_U$. Let G_E be the imperfect-information game obtained from E and G_D the full-information game obtained from G_E. A location of G_D is a set of states of E. A move of G_D is a subset of A_C. Given a location v of G_D, the set of possible successors using a move $\gamma \subseteq A_C$ is the set of locations $\delta(v, \gamma)$. Note that the successor of a location v of G_D after a move consisting of only one action ℓ is a single location v', hence we abuse notation and say that $\delta(v, \ell) = v'$.

Consider the question of whether from a location of G_D Player 1 can choose a move that forces Player 2 to choose a target location from a set $\mathcal{V}$. We call the set of such locations the *control predecessor* of $\mathcal{V}$. Straightforwardly, one can evaluate all moves in 2^{A_C}. However, this question can be answered by considering the moves in A_C one by one. Suppose that for some set γ it holds that $\delta(v, \gamma) \subseteq \mathcal{V}$. Consider that there exists an action $c \in A_C \backslash \gamma$ such that $\delta(v, c) \in \mathcal{V}$. Then, clearly $\delta(v, \gamma \cup \{c\}) \subseteq \mathcal{V}$ as well. Hence actions can be considered one by one until finding the maximal move $\gamma \subseteq A_C$ such that $\delta(v, \gamma) \subseteq \mathcal{V}$. Then, if for some state $s \in v$ we have $\Delta(s) \subseteq A_C \backslash \gamma$, i.e., there is some state in v for which no controllable action leads to $\mathcal{V}$ or there is at least one uncontrollable action; then we say that v is not in the control predecessor, otherwise it is. It follows that we can compute the control predecessor considering the actions in A_C one by one and not the (exponentially many) sets in 2^{A_C}.

Finally we consider the computation of the best/worst successor location. Let $<$ be a linear order between locations of G_D, such that s_e is maximum. Let $\ell_1 < \ell_2 < \ldots < \ell_n$ be an ordering on A with respect to a location v established by $\ell_i < \ell_j$ if and only if $\delta(v, \ell_i) < \delta(v, \ell_j)$. Take the maximum action ℓ_i in A_U and consider the set $A_i = \{\ell_1, \ldots, \ell_i\}$. If for every $s \in v$ there exists $1 \leq j \leq i$ such that $\ell_j \in \Delta(s)$, then $\delta(v, \ell_i)$ is the best possible worst successor for v. Otherwise, let $A_k = \{\ell_1, \ldots, \ell_k\}$ be the minimum set such that $A_i \subseteq A_k$ and for every $s \in v$ there is $1 \leq j \leq k$ such that $\ell_j \in \Delta(s)$. Then, $\delta(v, \ell_k)$ is the best possible worst successor for v. If no such set exists then there is no way to avoid s_e from v. It follows that we can compute the best possible worst successor for v by sorting the set A according to the order $<$ and then consider *linearly* many subsets of A_C.

5 Partially Observable Control Problems

In this section we present a reduction from control problems based on partially observable LTS environments to non-deterministic LTS control problems. The reduction in Def. 15 works by removing τ-transitions while capturing the same behaviour by propagating non-determinism to previous non-τ-transitions. This is necessary since a naive reduction that only guarantees trace equivalence may not preserve controllability, while stronger reductions may use an unnecessarily large number of states.

Consider the LTS E depicted in Fig. 5a and a trace equivalent reduction $E^\dagger$ shown in Fig. 5b, i.e. a trace belongs to E if and only if it also belongs to $E^\dagger$. For the control problem $\mathcal{E}=(E^\dagger, \Box\Diamond c_2, \{c_1, c_2\})$, we note that in $E^\dagger$ there is a choice between the controllable actions c_1 and c_2 in state 1, hence we may produce the controller $C^\dagger$ presented in Fig. 5c which is valid for $E^\dagger$ but not for E since disabling c_1 has the potential of creating a deadlock in E if the internal action reaches state 2. Considering $E^\dagger$ can make us reach the conclusion that the problem is controllable when it is not.

On the other hand, if we apply standard τ-closure we may obtain an LTS with unnecessary states, as in our example $E^\star$ shown in Fig. 5d, which has the unnecessary state 1. While this translation preserves weak bisimilarity with the original LTS, such restriction is too strong for controllability purposes. We seek for a reduction producing an equivalent τ-free LTS suitable for control. Thus, our reduction removes states irrelevant to the target non-deterministic problem. In

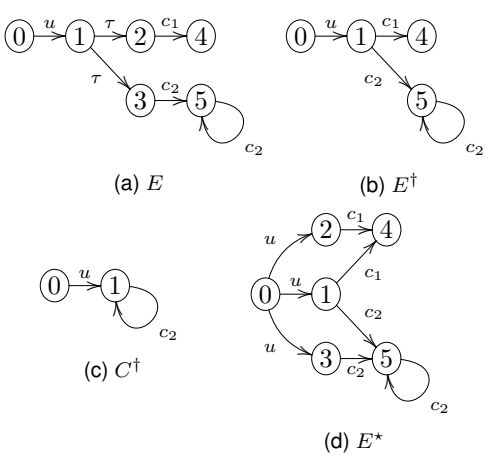


Fig. 5: Reduction variations

Def. 14 we identify *conducing states*, which are the states that are relevant. Removing non-conducing states results in a potentially smaller LTS better suited for further processing (e.g. effectively dropping state 1 of $E^\star$).

Furthermore, standard τ-closure cannot eliminate τ-transitions originating from the initial state. A special treatment is needed in this case, since there are no previous transition with which to capture the same behaviour via non-determinism. In such cases we add fresh states and labels in order to obtain a non-deterministic LTS compatible for control (see Def. 16).

Finally, we also handle the cases where an LTS can enter an infinite τ-loop, which could affect controllability. A τ-loop models an environment capable of closing itself both to observation and interaction. Such a scenario is considered undesirable in this context, since it is indistinguishable from a deadlock state. Thus, a fresh sink state is added by the translation in order to create a deadlock in the presence of τ-loops. In spite of that, if additional assumptions were considered (e.g. fairness) some τ-loops could be benign. This would give place to a different setting with potentially other acceptable interaction models. Our approach could be easily adjusted to accommodate to such requirements.

Definition 14 (Conducing State). Given an LTS E we say that a state $s \in S_E$ is a *conducing state* – denoted $cond(s)$ – if and only if $\Delta_E(s) \neq \{\tau\}$, or informally if it does not have only τ-transitions. Observe that states with no τ-transitions and deadlock states are *conducing states*.

By a slight abuse of notation we write $cond(S)$ for the subset of conducing states contained in S, that is, we also use it as $cond(S) = \{s \in S \mid cond(s)\}$.

Definition 15 (τ-removal). Given an LTS E, we define its τ-removed version $\hat{E} = (\hat{S}, \hat{A}, \hat{\Delta}, s_0)$ where:

- $\hat{S} = S_E \cup \{s_e\}$, the states of E plus the fresh state s_e used to create deadlocks in the presence of τ-loops
- $\hat{A} = A_E \setminus \{\tau\}$
- $\hat{\Delta}$ a transition relation that only reaches the *conducing states* (for all $\ell \in \hat{A}$):

$$\hat{\Delta} = \left\{ s \xrightarrow{\ell}_{\hat{E}} s' \;\middle|\; s \stackrel{\ell}{\Rightarrow}_E s' \wedge cond(s) \wedge cond(s') \right\} \cup \left\{ s \xrightarrow{\ell}_{\hat{E}} s_e \;\middle|\; s \stackrel{\ell}{\Rightarrow}_E s' \wedge s' \stackrel{\tau}{\Rightarrow}_E s' \right\}$$

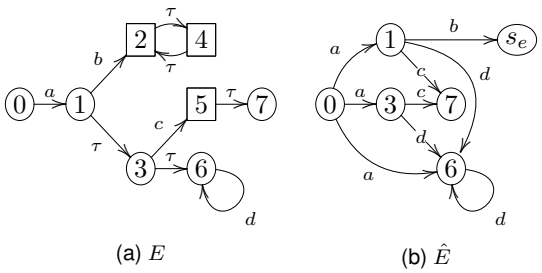


Fig. 6: τ-removal Example.

The definition of τ-removal produces a τ-free non-deterministic LTS by introducing a *step* between *conducing states* if there is a *walk* between them in the original LTS. Additionally, a *step* into the sink state s_e is introduced from a state that can *walk* into a τ-loop in the original LTS.

In Fig. 6 we show an example of the construction of the translation for an LTS E (Fig. 6a). We denote *conducing states* with round nodes and *non-conducing states* with square nodes. To build $\hat{E}$ (Fig. 6b) we connect the *walk*-reachable *conducing states* and add the state s_e to translate τ-loops into deadlocks.

Observe that τ-removal exhibits two useful properties formalised in following lemmas. Lemma 1 expresses that a *walk* in an LTS E is substituted by a *step* in $\hat{E}$. Lemma 2 extends this property for composed LTSs.

Lemma 1 (State Connectivity). A *conducing state* is reachable through a *walk* from another *conducing state* in an LTS E if and only if it is reachable in a *step* in $\hat{E}$, that is:

$$\forall s, s' \in cond(S_E), \ell \in A_E \backslash \{\tau\} \,.\, s \stackrel{\ell}{\Rightarrow}_E s' \Leftrightarrow s \xrightarrow{\ell}_{\hat{E}} s'$$

Lemma 2 (Composition State Connectivity). A state $\langle s, t \rangle$ of $E \| M$ with s and t *conducing*, is reachable through a *walk* from another such state if and only if it is also reachable in a *step* in $\hat{E} \| \hat{M}$, that is:

$$\forall s, s' \in cond(S_E), t, t' \in S_M, \ell \in A_E \backslash \{\tau\} \,. \\ \langle s, t \rangle \stackrel{\ell}{\Rightarrow}_{E\|M} \langle s', t' \rangle \Leftrightarrow \langle s, t \rangle \xrightarrow{l}_{\hat{E}\|\hat{M}} \langle s', t' \rangle$$

To handle the case where there are τ-transitions in the initial state we add a fresh action (μ) to capture the initial non-deterministic behaviour, lost otherwise.

Definition 16 (μ-addition). Given an LTS E we define its μ-added version $\breve{E}$ as follows

$$\breve{E} = (S_E \cup \{s_\mu\}, A_E \cup \{\mu\}, \Delta_E \cup \{(s_\mu, \mu, s_0)\}, s_\mu)$$

Note that if $\pi = \ell_0, \ell_1, \ldots$ is a trace of E then $\pi' = \mu, \ell_0, \ell_1, \ldots$ is a trace of $\breve{E}$. Furthermore, $\pi \models \varphi$ if and only if $\pi' \models \mathbf{X}\varphi$, that is, the next operator effectively ignores the initial action μ.

5.1 Interface Automata Control

Here we define a control problem for settings with partial observability and the IA interaction model.

Definition 17 (WIA PO LTS Control). Given an LTS E, a set of controllable actions $A_C \subseteq A_E \backslash \tau$, and an FLTL formula φ, a solution for the IA partially observable LTS control

problem $\mathcal{E}{=}(E, \varphi, A_C)$ is an LTS M such that M is a *legal* LTS for E (cf. Def. 4), $E\|M$ is deadlock-free, and $E\|M \models \varphi$ with no possible τ-loop.

Theorem 3 (IA Compatibility under τ-removal). An IA partially observable LTS control problem $\mathcal{E}{=}(E, \varphi, A_C)$ is realizable if and only if the IA non-deterministic LTS control problem $\hat{\mathcal{E}}{=}(\hat{\hat{E}}, \mathbf{X}\varphi, A_C)$ is realizable.

We now present the intuition of the proof. We first establish the compatibility under the simplifying assumption that there are no τ-transitions from the initial state, and then relax the assumption.

For the first part we use Lemmas 1 and 2 in order to show that a controller M is a solution for a problem with an environment E and its τ-removed version $\hat{E}$. In essence, using Lemma 2 we can replace *steps* in $E\|M$ with *walks* in $\hat{E}\|M$ and vice versa. Hence a trace is in $E\|M$ if and only if is also in $\hat{E}\|M$. We show that for similar reasons M avoids deadlocks, has no possible τ-loop and is *legal* with respect to E and $\hat{E}$.

For the second part we show how the fresh initial action μ, added in order to guarantee that the initial state has no τ-transitions (c.f. Def. 16), can be safely ignored. This is achieved with a slight modification of the goal to accept the initial μ. Note that the environment considered in the control problem $\hat{\mathcal{E}}$ is extended with the fresh action μ and then stripped from τ labels (i.e. $\hat{\hat{E}}$).

5.2 Weak Interface Automata Control

We definite a control problem for a setting with partial observability and the WIA interaction model.

Definition 18 (WIA PO LTS Control). Given an LTS E, a set of controllable actions $A_C \subseteq A_E \backslash \tau$, and an FLTL formula φ, a solution for the WIA partially observable LTS control problem $\mathcal{E}{=}(E, \varphi, A_C)$ is an LTS M such that M is *weak legal* for E (cf. Def. 3), $E\|M$ is deadlock-free, and $E\|M \models \varphi$ with no possible τ-loop.

Theorem 4 (WIA compatibility under τ-removal). A WIA partially observable LTS control problem $\mathcal{E}{=}(E, \varphi, A_C)$ is realizable if and only if the WIA non-deterministic LTS control problem $\hat{\mathcal{E}}{=}(\hat{\hat{E}}, \mathbf{X}\varphi, A_C)$ is realizable.

The proof follows that of Theorem 3 considering that the controller must be *weak legal* instead of *legal*.

5.3 Algorithm

Now we present an algorithm for computing the τ-removal of an LTS $E = (S, A, \Delta, s_0)$ to a τ-free non-deterministic LTS. The computation of *walks* in E has the flavour of computing the transitive closure of a transition relation. However, we show that it is simpler as τ-loops lead to uncontrollability and thus report this optimization. The procedure is presented in three simple routines and assumes that there are no τ-transitions from the initial state, which can be satisfied by μ-addition (cf. Def. 16).

The procedure `τ-reach` computes the set of *conducing states* reachable after zero or more τ-transitions from each state. The function uses a table (named $taus$) to store and reuse the results of the computation. Each iteration only considers states reachable in one step and works recursively over them. A special symbol (called `incomplete`) is introduced into the table to indicate that the computation for a given state is in progress, this is used to detect τ-loops in which case the τ-reachable set is replaced with the special state s_e. This is an optimization that allows avoiding the computation of the complete transitive closure, since valid controllers must avoid deadlock and hence disregard non-deterministic transitions that reach at least one deadlock state.

```
τ-reach()
    taus = empty-map
    foreach s in S
        τ-reach-recursive(s, taus)
    return taus

where τ-reach-recursive(s, taus)
    result = taus[s]
    if result ≠ null
        return result
    taus[s] = {incomplete}
    result = ∅
    foreach s' such that s ⟶τ_E s'
        reachable = τ-reach-recursive(s', taus)
        if incomplete ∈ reachable ∨ s_e ∈ reachable
            result = {s_e}
            break
        else
            result = result ∪ reachable
    if conducing(s) ∧ s_e ∉ result
        result = result ∪ {s}
    taus[s] = result
    return result
```

Observe that thanks to the storage of the results for every state and the fact that τ-loops are cut out as soon as they are found, computing `τ-reach` for every state of E turns out to be linear on the number of transitions. In other words, the worst-case complexity for computing `τ-reach` for all the states of E together is $O(|\Delta|)$.

The `walk-reach` routine builds a table with all the reachable *conducing states* after a *walk* (i.e.; zero or more τ-transitions, followed by exactly one non-τ-step and ending again with zero or more τ-transitions). The routine first invokes `τ-reach` and then computes the walk-reachable states in $O(|\Delta| \times |S|)$.

Finally the `τ-removal` function returns the translated non-deterministic LTS with no τ-transitions. The algorithm works by first building the table of *walks* and then creating a one step transition for every valid *walk* in the original LTS. The procedure `τ-removal` has an overall worst-case complexity of $O(|\Delta| \times |S|)$.

6 Evaluation

In this section we report on two case studies taken from domains in which controller synthesis techniques have been previously applied with fully observable deterministic environments. Case studies were selected to exemplify synthesis for both interaction models discussed in this paper. The first case study is taken from a service oriented architecture

```
walk-reach()
    taus = τ-reach()
    steps = empty-map
    foreach s in S
        steps[s] = ∅
        foreach ℓ in Δ(s)\{τ}
            foreach s' such that s -ℓ->_E s'
                foreach d in taus[s']
                    steps[s] = steps[s] ∪ {⟨ℓ, d⟩}
    walks = empty-map
    foreach s in S such that conducing(s)
        walks[s] = steps[s]
        foreach s* in taus[s]
            walks[s] = walks[s] ∪ steps[s*]
    return walks
```

```
τ-removal()
    Δ' = ∅
    walks = walk-reach()
    foreach s in S such that conducing(s)
        foreach ⟨ℓ, s'⟩ in walks[s]
            Δ' = Δ' ∪ {⟨s, ℓ, s'⟩}
    return (S, A, Δ', s0)
```

setting inspired by [5], [18], [19], where it is reasonable to assume that there is a handshake mechanism between the intervening components. We extend this case study to account for service variability, thus introducing partial observability. The second case study is a physical intrusion detection system, where the whereabouts of an intruder may be unknown. Note that, in general, in cyber-physical systems it is not sensible to presume a handshake between environment and actuators.

We present these case studies as proof of concept that shows the flexibility of the technique for problem domains with different interaction models. In other words, we aim to highlight the advantage that controller synthesis provides as a generic and fully automated method for constructing components that guarantee system goals even in the presence of non-determinism and partial observability. The generality and full automation, of course, comes at a cost in complexity which has been discussed above. In that respect, scalability is not the focus of the current validation.

Both case studies were analysed using our tool – MTSA[1] [20] – which accepts a textual representation of the LTS model and FLTL goals given in the Finite State Processes modelling language (FSP). Technically, FSP is a process calculus designed to be easily machine and human readable that includes standard constructs such as action prefix (->), external choice (|), label prefixing (:), hiding (\) and parallel composition (||). A thorough explanation can be found in the MTSA documentation. The computer used for the evaluation is an Intel i7-3770 with 8GB of RAM.

6.1 Travel Agency Service Orchestration

This case study deals with an implementation in a service oriented architecture for a travel agency. The agency sells vacation packages on-line by orchestrating existing third-party individual web-services for car rental, flight purchase and hotel booking. The aim is to synthesize an orchestrator based on the requirements for the provision of vacation packages and the protocol specification of the web-services.

Service oriented architectures provide a number of abstraction layers which can support a communication model based on handshakes. For instance, the Business Process Execution Language (BPEL) offers a rendezvous interaction mechanism when request-response operations are used. Indeed, CSP style composition semantics is commonly used as an abstraction for web-service orchestration (e.g. [21]). Thus, it is reasonable to interpret the composition of the controller and the web-services in a handshake communication model for which WIA control synthesis is adequate.

We now continue with the details of the case study, explaining the agency's requirements and the behaviour of the independent services the agency may interact with. The agency receives requests for vacation packages and interacts with different providers of cars, flights and hotel reservations to fulfil them. The protocols for the services may vary, one variant is the number of steps required for querying availability; in some cases the process requires a second step (e.g. querying for flight destination and dates, and if successful following with a selection for class). Another variant in service protocols is that some services may require a reservation step which guarantees a purchase order for a short period, while others do not, and hence the purchase may fail (e.g. on low availability reservation may be disabled in order to maximize concurrency between clients. As a result a race condition between two purchase orders may arise, and therefore one order will fail).

Protocol variability depends on internal decisions of each independent service, hence invisible to the coordinator. However, the underlying handshake mechanism allows for valid interactions. Since the LTS of a service is not easy to depict, in Fig. 7 we show the FSP specification of a generic service. Note that internal actions are hidden through the use of the operator (\). That is, we use internal actions in order to create a fitting specification, and later we hide these actions to prevent the controller from monitoring them.

In summary, the $Service$ waits for a query and continues according to the result. If unavailable it reports a failure and becomes idle again. If a single result is found it reports success and continues with the booking step. If multiple results are found it reports success and move towards a more detailed selection. On the booking step the $Service$ decides between committing to allow reservations or requiring a direct purchase order. Furthermore, the session can be terminated through the cancel action. From the controller's point of view all the non-hidden actions are controllable except for the reporting of failures and successes which are only observable.

In FSP the expression that builds an environment with services for car, flight and hotel booking can be seen in Fig. 8. Observe that we prefix the generic $Service$ with the names of particular services which prefixes every action of the LTS. Thus, $car.query$ represents a query for car rental.

The agency receives clients requests ($agency.request$) and must interact with the web-services in order to build the vacation package if possible. In the case that all services are

1. Available at https://bitbucket.org/dciolek/mtsa

```
Service = (
  query -> (
    unavailable-> query.failure -> Service  |
    single     -> query.success -> Booking  |
    multiple   -> query.success -> Selection)),
  Selection = (select -> Booking),
  Booking  = (committed   -> Reserve |
              uncommitted -> Direct  ),
  Reserve  = (reserve -> (
               cancel   -> Service |
               purchase ->
               purchase.success -> Service )),
  Direct   = (order -> (
              cancel   -> Service |
              purchase -> (
               purchase.success -> Service |
               purchase.failure -> Service )))
  \{unavailable, single, multiple,
    committed, uncommitted}.
```

Fig. 7: Generic $Service$ model

```
||Services = {car,hotel,flight}:Service.
```

Fig. 8: $Services$ environment

available for the required date the agency should hire them and report success ($agency.success$). Otherwise it should report failure ($agency.failure$). Additionally, on a failure no service should be paid for. Hence, given that direct purchase orders may fail, we must also allow the agency to report a failure when it is uncertain about two or more purchase orders (i.e. reservation not offered by at least two services).

To formalize the goals we use the following fluents:

(a) $agency.response$ that captures if the agency has issued a response for the last request, set to true on either $agency.success$ or $agency.failure$ and to false on $agency.request$

(b) $service.hire$ that captures if a given service has been paid for the current request, set to true on $service.purchase.success$ and to false on $agency.request$

(c) $service.unavailable$ that captures if a given service is unavailable for the current request, set to true on $service.query.failure$ or $service.purchase.failure$ and to false on $agency.request$

(d) $service.uncertain$ that captures if a given service does not provide a reservation step and thus its purchase may fail, set to true on $service.order$ and to false on $agency.request$

We now formalize in FLTL the agency's requirements needed to guarantee the desired goal:

- If successful, services of all kinds are hired:
  $\forall s \in Services\ .\ \Box(agency.success \Rightarrow s.hired)$
- If unsuccessful no service is purchased:
  $\Box(agency.failure \Rightarrow \neg\exists s \in Services\ .\ s.hired)$
- If unsuccessful at least one service is unavailable or more than one purchase is uncertain:
  $\Box(agency.failure \Rightarrow$
  $\exists s, s' \in Services\ .\ s.unavailable \vee$
  $(s \neq s' \wedge s.uncertain \wedge s'.uncertain))$
- Perform only one query to each service per request:
  $\forall s \in Services\ .$
  $s.query \Rightarrow \mathbf{X}(\Box(\neg s.query\ \mathbf{W}\ agency.response))$
- Every request must be replied to:
  $\Box\Diamond agency.request \Rightarrow \Box\Diamond agency.response$

Observe that the last requirement requires the system to be live, while the rest only require the system to be safe. In particular the liveness goal belongs to the General Reactivity 1 class (GR(1) [22]), hence the resulting game can be solved in quadratic time. Notably the specification is succinct and declarative. The model captures the desired behaviour in part thanks to an abstraction of the communication mechanism. We have synthesised controllers for several different versions of this case study simply by changing the services considered in the environment.

For the services considered in Fig. 8 the composition of the environment gives an LTS with 4394 states. Then the translation to an imperfect information game yields a game with 5194 states. Note that this is, in the worst case, an exponential step. The states of the game are sets of states of the original LTS. However, the number of game states grow moderately. Finally, a winning strategy for the game is translated to a controller with 301 states. The whole process takes approximately one minute.

The synthesized controller is not trivial. It deals with the non-determinism that arises from partial observability. Specifically, after a successful query, the controller enables selection, reservation and direct order. Depending on the queried service some actions may be enabled and the resulting composition only allows interaction on those options. At the controller enactment level this would be akin to attempting a rendezvous style communication. Interestingly, the controller delays payment until all services are assured, that is, it has a reservation. In the case that it reserved two services but the third requires direct purchase it first attempts to secure the uncertain order. If the purchase fails, it cancels the reservations and reports a failure to the user. Furthermore, in the case that more than two services require direct purchase, it does not attempt any and returns a failure. Indeed, should the controller succeed in paying the first service but fail to pay the second the goal would not be achieved.

Summarising, we have discussed how the orchestration for a service oriented architecture setting that includes partial observability can be constructed automatically through controller synthesis. The advantage that controller synthesis provides is a generic and fully automated method for constructing the orchestrations from declarative specifications. The generality and full automation comes at a cost in complexity that may hinder scalability, for example if more types of services were needed.

6.2 Intrusion Detection System

In a physical security system for a warehouse, robotic sentries that can observe and move along corridors need to use a sophisticated search strategy to pinpoint, surround and capture a moving intruder that attempts to hide behind warehouse racks. The strategy requires reducing the possible locations of the intruder by observing down corridors and moving into the area where the intruder is suspected

```
Sentry =(alarm -> Active[0]),
  Active[s:Area] = (
   foreach [d:Area]
    when (Adjacent(s,d)||s==d) patrol[d] ->
     sense[d][Hint] -> (
      guard -> Active[d] |
      capture[d] -> Sentry)).
```

Fig. 9: *Sentry* model

```
Intruder =(enter[i:Area] -> alarm -> Evade[i]),
  Evade[i:Area] = (
   sense[s:Area][InLine(s,i)] -> Evade[i] |
   capture[i] -> Intruder |
   guard -> (foreach [d:Area]
    when (Adjacent(i,d)) move[d] -> Evade[d]))
  \{enter[Area],move[Area]}.
```

Fig. 10: *Intruder* model

to be without letting it move back into a secured area. In this scenario, partial observability is introduced not only because of the unknown initial position of the intruder but also because its location can change to an unknown location after moving.

The warehouse is divided into nine areas organized as a square grid, we use an adjacency matrix to model the valid movements for the to-be-controlled sentry. An intruder is expected to break into the facility and move as well. For simplicity, we assume that intruder and sentry move at similar speeds. The goal is to keep the warehouse secured, that is to capture the intruder by positioning the sentry close enough.

In Fig. 9 we present the FSP fragment that models the behaviour of a sentry. Upon activation of an $alarm$ the sentry starts working from an initial location. The sentry can $patrol$ to an adjacent area where it can $sense$ the direction of the intruder, if it is in the same area, or nothing if out of sight (i.e. six possible hints). Afterwards it can continue to $guard$ or attempt to $capture$ the intruder. In the figure $Adjacent$ is an expression that returns whether two locations are adjacent or not, areas and hints are represented with consecutive numbers starting in zero.

Fig. 10 models the behaviour of the intruder. The intruder can $enter$ the warehouse at some location, doing so activates an $alarm$. Once inside it $moves$ through the corridors until $captured$. When in line with the sentry it is detected by its sensors. Observe that the $enter$ and $move$ actions are hidden, thus introducing partial observability. In the figure $InLine$ is an expression that returns the direction of a location s with respect to another one i, or whether s is equals to i, or out-of-sight if they are not in the same row or column.

The goal in this case study can be expressed with the following FLTL formula:

$$\square\Diamond alarm \Rightarrow \square\Diamond capture$$

Before attempting to synthesise a controller for the sentry, it is necessary to determine the proper interaction model. To this end, a relevant question is whether it is correct for the controller to attempt to $patrol$ or $capture$ when uncertain about the current state of the warehouse. Can we assume that when enacting the sentry's controller, there will be a negotiation or handshake with the environment on whether the action is allowed or not? Indeed, this cannot be assumed. Commanding the sentry to move will execute code that will attempt to move the sentry despite whether this attempt will succeed or not, whether the sentry will end up crashing into a wall and breaking, or not. Similarly, we cannot assume a cordial protocol between the sentry and the intruder to decide if the capture action will be successful. Hence, we can conclude that the IA interaction model better captures the real mechanics of this system (as it is the case for most cyber-physical domains).

Armed with the partially observable model, the goal specification and the interaction model, we seek for a controller applying the corresponding algorithm. Surprisingly, executing the synthesis procedure returns that no controller exists. Closer inspection reveals that there is a strategy for the intruder such that it can evade the sentry indefinitely, thus no sure winning strategy exists.

The reason there is no strategy for the sentry is threefold:

i) the sentry cannot reduce uncertainty regarding where the intruder is. Even if the sentry while moving around happens to detect the intruder along a corridor (e.g. column i), the intruder can move out of sight into corridor $i-1$ or $i+1$, thus the sentry cannot start establishing a secure area (an area it knows that the intruder cannot be in)
ii) the sentry cannot run the intruder down as they move at the same speed
iii) the sentry cannot (alone) trap the intruder because every location has at least two escape routes (the corners have two, other location have more).

However, it is simple to check if adding more sentries makes the goal realisable. In fact, just adding a second sentry in the model allows synthesising a controller which can be thought of as a centralised controller that coordinates both sentries in order to keep the area secured. The difference with the single sentry case is that two collaborative sentries can corner the intruder independently of its initial position and movements. The synthesized strategy for this setting is far from trivial: The sentries must coordinate viewing down adjacent corridors moving from one corner to the opposite, this ensures building up a secure area while closing down on the intruder. The fact that for each movement of the intruder there are two movements of the sentries (one each) is also key.

The composition of $Intruder$ and $Sentries$ for the warehouse with nine areas gives an LTS with 56278 states. However, applying a standard LTS minimization procedure the number of states reduces to 2926, which is better suited for synthesis given that it is exponential on the number of states. After applying *controlled determinization* we obtain a deterministic LTS control problem with 7084 states. A sure winning strategy for the corresponding game is translated to a controller with 208 states. The whole process takes approximately three minutes.

Note that the generality of the technique allows modifying the model and goals as needed. In contrast to hand made controllers that need to be rewritten manually when the

context or requirements change. Indeed, migrating a manually crafted controller for a single sentry to a centralized controller for the coordination of two sentries would require major modifications. In fact, the flexibility of the model is even greater, since it accepts as input every possible topology (varying the $Valid$ and $InLine$ functions). However, not every layout can be secured by two sentries.

Summarising, we have discussed how the strategy for a cyber-physical system that must deal with partial observability can be constructed automatically. Notably, the same algorithm can be used to generate controllers for different variations of warehouse structures and number of sentries. As with the previous case study, the generality and flexibility of the approach comes at a cost in complexity. Controllers could also be built by hand for a specific layout, but such controllers would have to be manually tweaked upon a change to the specification. We believe that in general the difficulty of adapting the model to a change in the specification is significantly lower than manually devising a new controller to deal with the same change.

7 Discussion and Related Work

Synthesis of operational strategies in the form of LTS for achieving declaratively specified goals has been studied and applied in various forms (e.g. [3], [4], [5], [18], [19]). However, few techniques can handle general liveness goals and partially observable environments. In this paper we study this combination and show that the underlying interaction model is key for appropriately choosing the synthesis technique. We based this approach on our previous work [23] in which determinism and full-observability were required.

As shown in Section 5 partial observation and nondeterminism are closely related challenges. Most of the work on this aspect is based on Kripke structures ([24], [25]). In [24] the authors focus on the compositional synthesis of a reactive controller under partial observability, but only for safety properties. Technically speaking the approach boils down into safety imperfect-information games ([25]).

A noteworthy line of work is that of [1]. Safety properties and bounded-liveness are used as goals in an LTS-like framework. The technique assumes input-enabledness, still the resulting controller is legal in the sense of interface automata. A standard determinisation algorithm is used and hence, as discussed in Section 4, it is not adequate for general liveness goals.

In the area of supervisory control for Discrete Event Systems only an interaction model similar to the weak interface automata setting is studied. For treating liveness goals, supervisory controller synthesis literature commonly represents the "legal" specification language as a deterministic Rabin automaton. The problem of synthesis boils down to force that automata to accepts its language [26]. This disallows the use of adhoc procedures for logic fragments like we do for GR [22].

A compositional approach for dealing with safety and co-safety goals in a partially observed environment is presented in [27], the approach assumes as input nondeterministic automata. The technique looks for "most general" strategies for each component, such that when composed the crossed restrictions do not prevent the realization of the goals. Since a compositional treatment is performed the technique promises to attain superior scalability compared to our approach. However, additional constraints (e.g. fairness) need to be assumed to guarantee completeness.

The confluence of partial observability and liveness in supervisory control synthesis can be found in [17]; there the problem of synthesis under partial observability is reduced to that of complete observation for general ω-regular specifications, that may depend on actions not observable by the controller. Then procedures like the ones presented in [26] are invoked. This construction circumvents one limitation of our technique allowing goals to be expressed in terms of hidden actions. Unfortunately, there is no tool available and the end-to-end complexity and completeness is unclear given that the formal constructions are rather involved.

In [28] supervisors for CTL* specifications are generated assuming complete observability. Whereas in [29] CTL* synthesis (considering particular cases for CLT and LTL) is solved considering partial observability by using alternating tree automata. The approaches are based on a reduction to satisfiability of a CTL* specification which may produce a non-deterministic supervisor. In [30] a similar approach based on alternating tree automata is presented considering a partially observable version of μ-calculus. These techniques do not accept LTS-like models, but instead work on a shared signal setting for which the impact of interaction models has not been analyzed.

Supervisory control has recently been restated in a process-algebraic setting (e.g., [31]). In this setting the control requirements are given as the satisfaction of a bisimulation relation against some process that represents the expected behaviour. We point out that the notion of bisimulation is, in general, too strong. In fact, the reduction presented in this paper does not produce a bisimilar model, yet we prove it preserves controllability. Actually, our reduction abides to the weaker notion of testing equivalence [32].

Summarizing, the synthesis of operational strategies with partially observable environments and general liveness goals is a problem with many facets. In this paper we present two novel contributions absent in the related literature. The first is a simple reduction from partially observable LTS control to non-deterministic LTS control, proven to preserve controllability. The second is the identification of different interaction models that may impact on the realizability of the goals. However, our technique also exhibits two appreciable limitations. The first is that we cannot express goals in terms of hidden actions. The second, one shared by most approaches, is scalability.

8 Conclusions and Future Work

In this paper we present a solution to the automatic synthesis of controllers under non-deterministic and partially observable LTS environments for general FLTL goals. The problem differs from fully observable LTS control in that the exact state of the environment may be unknown. Our solution consist of a careful reduction to non-deterministic LTS environments, proven to preserve controllability.

Our technique admits general FLTL goals and assumptions restricted to fluents defined for visible actions, which allows for a simpler technical solution to that of related

works. The question of how the approach can be extended in order to support goals and assumptions over hidden actions has no immediate answer. The resulting control problem would need to, not only reduce uncertainty about the state of the environment, but also infer the value of unobservable fluents if possible. We leave this point open and intend to provide an answer in forthcoming studies.

We also report how different event-based interaction models impact on the LTS controller synthesis problem in the contexts of partial observability and non-determinism. We show that IA control problems can be solved by means of a carefully crafted determinisation – controlled determinisation. On the other hand, WIA control is solved by a translation to Imperfect-Information Games. These settings have different capabilities for reducing uncertainty about the state of the environment, which affects realizability. Hence it is important to determine the precise setting for a given problem and utilize a suitable algorithmic solution. Another avenue for future research includes exploring algorithmic solutions with better scalability.

The impact of interaction models on control problems remains a vastly unexplored area that we believe will lead to diverse synthesis techniques and relevant software engineering insights. In particular, the WIA interaction model is a weaker version of the IA interaction model that naturally arises while abstracting a handshaking mechanism. Other potential sources of interaction models could be found both in coordination models such as session types or contracts [33], [34] and in software architectures models like connector types (e.g. [35]).

REFERENCES

[1] E. Letier and W. Heaven, "Requirements Modelling by Synthesis of Deontic Input-output Automata," in *Proc. of the 2013 Int. Conf. on Software Engineering*, ser. ICSE '13. IEEE Press, 2013.

[2] M. Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*. ACM Press/Addison-Wesley Publishing Co., 1995.

[3] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "Synthesizing Nonanomalous Event-based Controllers for Liveness Goals," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, 2013.

[4] W. Heaven, D. Sykes, J. Magee, and J. Kramer, "Software Engineering for Self-Adaptive Systems," B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer-Verlag, 2009, ch. A Case Study in Goal-Driven Architectural Adaptation.

[5] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso, "Planning and Monitoring Web Service Composition," in *Artificial Intelligence: Methodology, Systems, and Applications*, ser. Lecture Notes in Computer Science, C. Bussler and D. Fensel, Eds. Springer Berlin, 2004, vol. 3192.

[6] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Saar, "Synthesis of Reactive(1) Designs," *J. Comput. Syst. Sci.*, vol. 78, no. 3, 2012.

[7] D. Giannakopoulou and J. Magee, "Fluent Model Checking for Event-based Systems," in *Proc. of the 9th European Software Engineering Conf. Held Jointly with 11th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, ser. ESEC/FSE-11. ACM, 2003.

[8] A. Van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour," in *Proc. of the 5th IEEE Int. Symp. on Requirements Engineering*, ser. RE '01. IEEE Computer Society, 2001.

[9] C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, no. 8, 1978.

[10] L. de Alfaro and T. A. Henzinger, "Interface Automata," in *Proc. of the 8th European Software Engineering Conf. Held Jointly with 9th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, ser. ESEC/FSE-9. ACM, 2001.

[11] K. Bierhoff and J. Aldrich, "Lightweight Object Specification with Typestates," in *Proc. of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, ser. ESEC/FSE-13. ACM, 2005.

[12] G. D. Caso, V. Braberman, D. Garbervetsky, and S. Uchitel, "Enabledness-based Program Abstractions for Behavior Validation," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, 2013.

[13] D. Berwanger, K. Chatterjee, M. De Wulf, L. Doyen, and T. A. Henzinger, "Strategy Construction for Parity Games with Imperfect Information," *Inf. Comput.*, vol. 208, no. 10, 2010.

[14] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[15] R. M. Keller, "Formal Verification of Parallel Programs," *Commun. ACM*, vol. 19, no. 7, 1976.

[16] J. H. Reif, "Universal Games of Incomplete Information," in *Proc. of the 11th Annual ACM Symp. on Theory of Computing*, ser. STOC '79. ACM, 1979.

[17] J. G. Thistle and H. M. Lamouchi, "Effective Control Synthesis for Partially Observed Discrete-Event Systems," *SIAM J. Control Optim.*, vol. 48, no. 3, 2009.

[18] P. Inverardi and M. Tivoli, "A Reuse-based Approach to the Correct and Automatic Composition of Web-services," in *Int. Workshop on Engineering of Software Services for Pervasive Environments: In Conjunction with the 6th ESEC/FSE Joint Meeting*, ser. ESSPE '07. ACM, 2007.

[19] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "Synthesis of Live Behaviour Models for Fallible Domains," in *Proc. of the 33rd Int. Conf. on Software Engineering*, ser. ICSE '11. ACM, 2011.

[20] N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel, "MTSA: The Modal Transition System Analyser," in *Proc. of the 23rd IEEE/ACM Int. Conf. on Automated Software Engineering*, ser. ASE '08. IEEE Computer Society, 2008.

[21] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Model-based Verification of Web Service Compositions," in *ASE*, 2003.

[22] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of Reactive(1) Designs," in *Proc. of the 7th Int. Conf. on Verification, Model Checking and Abstract Interpretation*, vol. 3855. Springer-Verlag, 2006.

[23] N. R. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "Synthesis of Live Behaviour Models," in *Proc. of the 18th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, ser. FSE '10. ACM, 2010.

[24] W. Kuijper and J. Pol, "Compositional Control Synthesis for Partially Observable Systems," in *Proc. of the 20th Int. Conf. on Concurrency Theory*, ser. CONCUR 2009. Springer-Verlag, 2009.

[25] K. Chatterjee, T. A. Henzinger, and B. Jobstmann, "Environment Assumptions for Synthesis," in *Proc. of the 19th Int. Conf. on Concurrency Theory*, ser. CONCUR '08. Springer-Verlag, 2008.

[26] J. G. Thistle and W. M. Wonham, "Control of Infinite Behavior of Finite Automata," *SIAM J. Control Optim.*, vol. 32, no. 4, 1994.

[27] J. Klein, C. Baier, and S. Klüppelholz, "Compositional Construction of Most General Controllers," *Acta Inf.*, vol. 52, no. 4-5, 2015.

[28] S. Jiang and R. Kumar, "Supervisory Control of Discrete Event Systems with CTL* Temporal Logic Specifications," *SIAM J. Control Optim.*, vol. 44, no. 6, 2006.

[29] O. Kupferman and M. Y. Vardi, *Advances in Temporal Logic*. Springer Netherlands, 2000, ch. Synthesis with Incomplete Informatio.

[30] A. Arnold, A. Vincent, and I. Walukiewicz, "Games for Synthesis of Controllers with Partial Observation," *Theoretical computer science*, vol. 303, no. 1, 2003.

[31] J. C. M. Baeten, B. van Beek, A. van Hulst, and J. Markovski, "A Process Algebra for Supervisory Coordination," in *PACO*, 2011.

[32] R. De Nicola and M. C. Hennessy, "Testing equivalences for processes," *Theoretical computer science*, vol. 34, no. 1, 1984.

[33] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language Primitives and Type Discipline for Structured Communication-Based Programming," in *Proc. of the 7th European Symp. on Programming: Programming Languages and Systems*, ser. ESOP '98. Springer-Verlag, 1998.

[34] G. Castagna, N. Gesbert, and L. Padovani, "A Theory of Contracts for Web Services," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 5, 2009.

[35] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a Taxonomy of Software Connectors," in *Proc. of the 22nd Int. Conf. on Software Engineering*, ser. ICSE '00. ACM, 2000.

Daniel Ciolek is a Phd. Student in Computer Science at the Department of Computing, FCEN, Universidad de Buenos Aires, where he also recieved his undergraduate Computer Science degree. His research is focused towards the application of artificial intelligence techniques to software engineering problems.

Victor Braberman holds a Professorship at the Department of Computing, FCEN, Universidad de Buenos Aires and he is a CONICET researcher working in the area of Software Engineering. He has headed the development of tools for the modelling and analysis of software intensive systems. His publication track record includes regular publications in the top Software Engineering conferences and journals. He has served as PC member for flagship conferences several times during the past 10 years. He acted as coordinator for the CS grant proposal evaluation process at the Argentinean research funding agency. He has a significant industrial experience as a consultant and leads R&D projects for local software companies.

Nicolás D'Ippolito is a Professor at Universidad de Buenos Aires and a CONICET researcher. He received his undergraduate Computer Science degree from University of Buenos Aires and his PhD in Computing from Imperial College London. His research interests fall in multiple areas Control Theory, Software Engineering, Adaptive Systems, Model Checking and Robotics. Specifically, he is interested in behaviour modelling, analysis and synthesis applied to requirements engineering, adaptive and autonomous systems, software architectures design, validation and verification. Dr. D'Ippolito regularly publishes in top conferences and journals in many areas. He has served as PC member for flagship conferences a number of times and has organised many events in top venues.

Nir Piterman received his Ph.D. in Computer Science from the Weizmann Institute of Science. He is currently Reader at the department of Computer Science in the University of Leicester. His research focuses on formal verification and automata theory. In particular he works on model checking, temporal logic, synthesis, game solving, and applications of formal methods to biological modelling.

Sebastián Uchitel is a Professor at University of Buenos Aires, a CONICET researcher and holds a Readership at Imperial College London. He received his undergraduate Computer Science degree from University of Buenos Aires and his Phd in Computing from Imperial College London. His research interests are in behaviour modelling, analysis and synthesis applied to requirements engineering, software architecture and design, validation and verification, and adaptive systems. Dr. Uchitel was associate editor of the Transactions on Software Engineering and is currently associate editor of the Requirements Engineering Journal and the Science of Computer Programming Journal. He was program co-chair of ASE06 and ICSE10, and will be General Chair of ICSE17 to be held in Buenos Aires. Dr Uchitel has been distinguished with the Philip Leverhulme Prize, an ERC StG Award, the Konex Foundation Prize and the Houssay Prize.