Control and Discovery of Environment Behaviour

Maureen Keegan, Victor Braberman, Nicolás D'Ippolito, Nir Piterman, and Sebastián Uchitel, *Senior Member, IEEE*

Abstract—
An important ability of self-adaptive systems is to be able to autonomously understand the environment in which they operate and use this knowledge to control the environment behaviour in such a way that system goals are achieved. How can this be achieved when the environment is unknown? Two phase solutions that require a full discovery of environment behaviour before computing a strategy that can guarantee the goals or report the non-existence of such a strategy (i.e., unrealisability) are impractical as the environment may exhibit adversarial behaviour to avoid full discovery.
In this paper we formalise a control and discovery problem for reactive system environments. In our approach a strategy must be produced that will, for every environment, guarantee that unrealisablity will be correctly concluded or system goals will be achieved by controlling the environment behaviour. We present a solution applicable to environments characterisable as labeled transition systems (LTS). We use modal transition systems (MTS) to represent partial knowledge of environment behaviour, and rely on MTS controller synthesis to make exploration decisions. Each decision either contributes more knowledge about the environment's behaviour or contributes to achieving the system goals. We present an implementation restricted to GR(1) goals and show its viability.

Index Terms—Adaptive systems, reactive systems, discrete event controllers, environment control and discovery.

1 Introduction

An important ability of self-adaptive systems is to be able to autonomously understand the environment in which they operate and use this knowledge to control the environment in such a way that its goals are achieved([1], [2], [3], [4], [5], [6], etc.).

Consider a self-adaptive component that has to orchestrate available services whose protocols are completely or partially unknown at design time. How can the component gain sufficient knowledge about the behaviour of these services to come up with a strategy for correctly orchestrating them according to some system goal? Alternatively, how can the component conclude, should it be the case, that its environment is such that achieving system goals cannot be guaranteed, i.e., the goals are *unrealisable*? The component must explore its environment, stimulating the services and sensing their behaviour to acquire enough knowledge to guarantee its goals or declare unrealisability.

A key difficulty of controlling an unknown environment is that environment behaviour discovery is hindered by uncontrollable environment behaviour. For example, an external component or agent may exhibit different behaviours depending on some internal setting that cannot be changed by the system performing the exploration. Furthermore, to be on the safe side, it is reasonable to assume the environment can exhibit adversarial behaviour, hence, deliberately not exhibiting relevant potential responses to stimuli it receives. Thus, there is no guarantee that during the exploration all behaviour will be discovered. This renders *two phase* approaches inadequate: in such approaches the environment behaviour is expected to be fully discovered first (be it through automata learning [1], [3], [7], [8] or some sort of random exploration), and only then is a control strategy computed.

In this paper we present a formalisation of the problem of controlling the behaviour of a reactive system's unknown environment -which is assumed to be characterisable as a deterministic finite Labelled Transition System (LTS) supporting event-based communication- to achieve system goals expressed in Linear Temporal Logic. *The key insight we exploit to avoid a two phase strategy is to ensure that every action taken either increases the knowledge about the environment's behaviour or moves the system "closer" to achieving its goals.*

We show how Modal Transition Systems (MTS) [9] can be used to characterise the partial knowledge of the environment behaviour that the discovery process starts with and the knowledge that is accumulated during the discovery.

We also present a solution to the control and discovery problem restricted to GR(1) goals [10]. The solution is based on MTS controller synthesis [11]. We show how, assuming that the environment behaviour can be characterised as a deterministic finite Labelled Transition System (LTS), synthesis can be used to reason about control and discovery strategies that are guaranteed to either exhibit, in the long run, system behaviour that achieves the goal, or conclude that the goal cannot be realised.

- *M. Keegan was with FCEN, Universidad de Buenos Aires.*
- *V. Braberman, N. D'Ippolito, and S. Uchitel were with FCEN, Universidad de Buenos Aires and CONICET-ICC.*
- *N. Piterman was with the University of Gothenburg and the University of Leicester.*
- *S. Uchitel was also with Imperial College London.*

Manuscript received April 19, 2005; revised August 26, 2015.
This work is supported by the following grants: the ERC Consolidator grant D-SynMA (No. 772459), the Marie Skodowska-Curie BeHAPI (No. 778233), the grants ANPCYT PICT 2018-3835, ANPCYT PICT 2015-1718, CONICET PIP 2014/16 N11220130100688CO, 2015/2017 N11220150100931CO and UBACYT 20020170100419 BA

Indeed the control and discovery procedure we propose guarantees that given a realisable environment, even if the environment deliberately tries to prevent discovery, it can be controlled without necessarily achieving full behaviour knowledge (i.e., part of the LTS that characterises the behavior might remain unexplored/unknown).

The use of MTS controller synthesis allows guiding the control and discovery process through controlled actions that either are known to help achieve the goal or lead to undiscovered behaviour of the environment. Thus, synthesised controllers do not rely on achieving full exploration to either conclude that the goal is unrealisable or to actually achieve the goal. Should new behaviour be discovered, it is integrated into the current knowledge – we require the environment to be capable of consistently reporting its current state id, to allow loop detection. Should new behaviour lead to a state in which the goals are unrealisable, then control and discovery must be capable of resetting the environment to reattempt to control and discover but with the additional knowledge that has been acquired.

In addition to the need to reset the environment, as explained, our approach requires also to identify re-visits to the same state. These conditions, although restrictive, can be satisfied, for example, by service oriented systems (where sessions can be thought of as resets and state abstraction functions for state IDs). Discovery followed by control in this domain has been studied (e,g., [1], [3]) and is often based on automata learning techniques (e.g., MAT [7]) that have arguably stronger assumptions on what can be done with the environment (e.g., controlability of the environment responses and the ability to answer equivalence queries, typically approximated by using conformance testing [12], which, in particular, requires resetting the environment).

In the rest of the paper we provide motivation and an example in Section 2, preliminary definitions in Section 3 and then formalise the control and discovery problem and provide a solution in Sections 4-5. We report validation in Section 6 and finalise with related work and conclusions.

2 Motivation and Running Example

We are interested in developing a client that will interact with a book loan service that allows users to request and have books delivered to their homes. Having read the books, users return them by post. This example is taken from [13].

A book loan service and its interface may be discovered via a discovery service such as the Google APIs Discovery Service. However, in practice the protocol (the order in which the methods of the service should be called) to correctly interact with the service is usually at best only partially available. Consequently, the problem is to build a client that can deduce how to use the book loan service while attempting to satisfy its goals if it can.

The protocol applied by the service could be implemented in many different ways. For instance, in Figure 1 we depict the protocol of a very basic book loan service (interaction between user and client is not part of the protocol). Note that we use dotted lines to distinguish between event that the system can and cannot control. For example, the availability of a book is uncontrollable. When a user wants to borrow a book it performs a request $usrReq$, the client must then perform a search ($queryBook$). Then, the book loan service returns a $list$ of available copies from which the client can choose one ($select$) and get it delivered ($deliver$). Alternatively, if no copies are available ($unavailable$) the client can reattempt after some time or inform the user that the request cannot be fulfilled ($abort$).

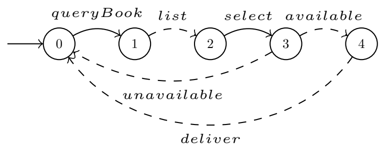

Fig. 1. Basic Book Loan service. No wait/hold functionality is provided ($usrReq$ and $abort$ are not depicted as they are user-client interactions). Dotted and continuous lines represent uncontrolled and controlled events

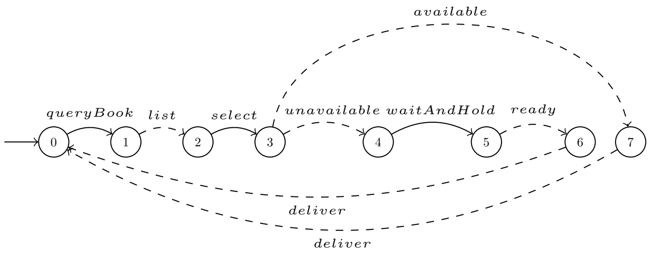

Fig. 2. A Book Loan service with a guaranteed wait/hold functionality.

Other services (e.g., Figure 2) allow users to reserve a copy of a currently unavailable book ($wait\&hold$). When a copy of the reserved book is returned, the copy is held and delivered. Some services (Figure 3) may provide the same functionality but in which finally, the book may finally be reported as $unavailable$.

Regardless of the protocol applied by the unknown service provider, the goals that the client is to achieve are the following. Firstly, that for every request a book is delivered. This goal can be formalised in linear temporal logic as follows $\varphi_1 = \Box(usrReq \rightarrow \Diamond deliver)$. Secondly, the client should not use the book loan service more than once per request ($\varphi_2 = \Box(queryBook \rightarrow \bigcirc(\neg queryBook\ W\, deliver)))$.

Note that goal realisability depends on how the available service is implemented. For instance, for the service depicted in Figure 1 there is no way of achieving the goals: To achieve φ_1, on $usrReq$ it must $queryBook$ in order to eventually get $available$ or $unavailable$. However, if the service reports the book unavailable, the client cannot query again (perhaps the day after) because it would violate φ_2.

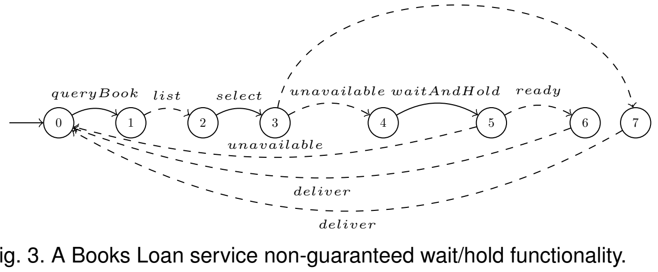

Fig. 3. A Books Loan service non-guaranteed wait/hold functionality.

On the other hand, the goals are achievable with the service in Figure 2 as in the event of an unavailable book, the hold and wait functionality can be used. Finally, for the service in Figure 3 the goals cannot be guaranteed because the hold and wait service may finally return $unavailable$.

How should the client behave to achieve its goals, or to conclude that it cannot do so if it knows the interface of the book loan service it is connected to but not the protocol and supported functionalities of the service? The client must interact with the service and accumulate sufficient knowledge about the service's behaviour. During the discovery process, due to lack of knowledge, some goals may be violated. In these cases, the client must be able to reset the service to try again (but this time with more knowledge). Thus, we assume an action $reset!$ that does so.

An example of how the client may try to control and discover the initially unknown service shown in Figure 1 (for which it is not possible to achieve φ_1 and φ_2) is as follows (in order to identify returning to the same state we assume the client can request a state id of the service): Having received $usrReq$ to reach state 0, the client must try some controllable action that may lead it to obtain $deliver$. Assume it tries $select$. The service will reject this call and its state will continue to be 0. If the client now tries $queryBook$, it succeeds leading to state 1. It then starts trying controlled actions only to discover that in state 1 none are possible. It will eventually receive event $list$ (possibly before trying all controlled events) leading to state 2. From state 2, it might try $select$ and reach state 3. Then event $available$ may happen followed by $deliver$ taking the service back to state 0. Should another $usrReq$ occur, the client knows how to achieve $deliver$ as long as it observes the same uncontrollable behaviour (i.e., it gets $available$ on 3).

If in state 3 it gets an $unavailable$ event, then it will land in state 1 from which it knows that $queryBook$ is available while $select$ is not. Performing $queryBook$ is not an option as it would violate φ_2 so the client would attempt the rest of its controlled actions only to discover that none are available in state 0. Clearly there is no strategy from this point on that can guarantee φ_1 as no controlled actions can move the service forward and the client has no guarantee that an uncontrolled action may occur. Indeed, we know this as a fact (see Figure 1) but the client does not.

Figure 4 depicts an MTS that encodes the knowledge accumulated by the client so far. Transitions with question marks indicate that it is currently unknown if such behaviour is available. States 1 through 6 encode behaviour observed so far and unknown behaviour goes to state 0 which models no knowledge at all about future behaviour.

Based on the MTS, the client can reason that the fact that it has reached a dead end does not mean that there is no hope of achieving the goals. The dead end may have been reached by mistakenly trying $select$ in state 3. Perhaps, had it tried some other controlled action ($queryBook$ or $abort$) the dead end would have been avoided. Thus, the client performs a reset! and starts again.

Of course, when the client reaches state 3 again it will discover that there are no other controlled actions that are accepted by the service. It will also eventually discover that it did not have other controllable options in state 2. As a result it will conclude that there is no hope of guaranteeing system goals and will halt, outputting a message that denotes unrealisability (which in this paper we refer to as none!). Note that for states 2, 4 and 5 knowing if controllable actions are available is irrelevant because even if they were, there is no guarantee that they would win a race condition against the uncontrollable actions that are known to be possible in those states.

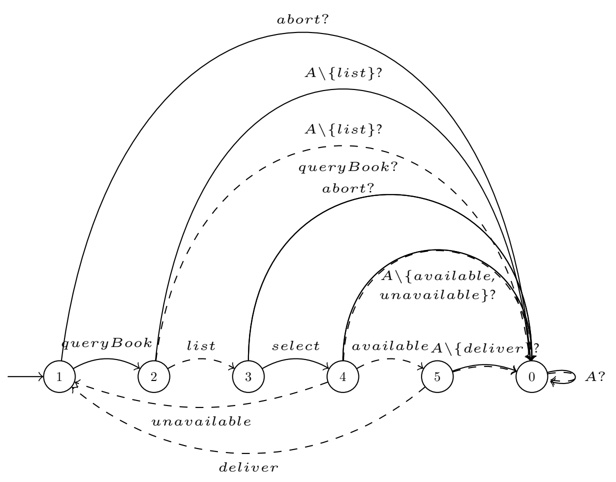

Fig. 4. MTS with partial knowledge of a Book Loan service's behaviour. A is the set of actions of its interface. $A \setminus B$ is set subtraction. Labels with question marks represent unknowns (i.e., possible but not required transitions).

The same client will, when connected to service in Figure 2, explore the available behaviour and eventually try the wait and hold functionality. It will discover that this functionality is crucial for guaranteeing delivery and thus will finally succeed in producing an execution that, since the last reset!, achieves all goals forever. On the other hand, for the service in Figure 3 as with the simplest service, the client will achieve its goals as long as $unavailable$ does not occur in either state 3 or 5. Once both occur, the client will report that the book loan service cannot be used with full guarantees of achieving system goals.

Note the tension between control and discovery: If the environment deliberately or by chance does not exhibit some of its behaviour then full knowledge cannot be achieved. Some of the behaviour not observed may be needed to conclude unrealisability of the goals and even allow achieving goals temporarily.

How can the reasoning described above be automated? In this paper we formalise the control and discover problem described above and show how to solve it using MTS controller synthesis for SGR(1) goals.

3 Preliminaries

We give a summarised background on controller synthesis.

3.1 Transition Systems

Definition 3.1. (Labelled Transition Systems) *A* Labelled Transition System *(LTS) is $W = (S_W, \Sigma_W, \Delta_W, s^0_W)$, where S_W is a finite set of states, $\Sigma_W \subseteq Act$ is its* communicating *alphabet, $\Delta_W \subseteq (S_W \times \Sigma_W \times S_W)$ is a transition relation, and $s^0_W \in S_W$ is the initial state. We use $\Delta_W(s)$ to denote $\{e \mid \exists s'.(s,e,s') \in \Delta_W\}$. We say an LTS is deterministic if (s,e,s') and (s,e,s'') are in Δ_W implies $s' = s''$. An execution ε of W is a finite or infinite word $\varepsilon = s^0, e^0, s^1, \ldots \in (S_W \times \Sigma_W)^\omega \cup (S_W \times \Sigma_W)^* \cdot S_W$, where $|\varepsilon|$ is the number of symbols in Σ_W appearing in ε and for every $i < |\varepsilon|$ we have $(s^i, e^i, s^{i+1}) \in \Delta_W$. An execution is* maximal *if it is infinite or ends in a state s such that $\Delta_W(s) = \emptyset$. A word $\pi \in \Sigma^\omega_W \cup \Sigma^*_W$ is a* trace *of W if there is an execution ε of W such that $\varepsilon|_{\Sigma_W} = \pi$, where $\varepsilon|_\Sigma$ is the projection of word ε over the alphabet Σ. A trace is* maximal *if it is the projection of a maximal execution. If $(s,e,s') \in \Delta_W$ we say that e is enabled from s.*

Modal Transition Systems (MTSs) [9] are abstract notions of LTSs. They extend LTSs by distinguishing between two sets of transitions. Intuitively an MTS describes a set of possible LTSs by describing an upper bound and a lower bound on the set of transitions from every state. Thus, an MTS defines required transitions, which must exist, and possible transitions, which may exist. By elimination, other transitions cannot exist.

Definition 3.2. (Modal Transition Systems [9]) *A* Modal Transition System (MTS) *is $K = (S_K, \Sigma_K, \Delta^r_K, \Delta^p_K, s^0_K)$, where S_K is a finite set of states, Σ_K is its* communicating alphabet, *$\Delta^r_K \subseteq \Delta^p_K \subseteq (S_K \times \Sigma_K \times S_K)$ are the required and possible transition relations, respectively, and $s^0_K \in S_K$ is the initial state.*

We denote by $\Delta^p_K(s)$ the set of possible actions enabled in s, namely $\Delta^p_K(s) = \{e \mid \exists s' \cdot (s,e,s') \in \Delta^p_K\}$. Similarly, $\Delta^r_K(s)$ denotes the set of required actions enabled in s.

We define a refinement relation between MTSs [9].

Definition 3.3. (Refinement [9]) *Let K and M be two MTSs, where $K = (S_K, \Sigma_K, \Delta^r_K, \Delta^p_K, s^0_K)$ and $M = (S_M, \Sigma_M, \Delta^r_M, \Delta^r_M, s^0_M)$. Relation $R \subseteq S_K \times S_M$ is a* refinement *between K and M if the following holds for every $e \in \Sigma_K \cup \Sigma_M$ and every $(s_K, s_M) \in R$.*

- *If $(s_K, e, s'_K) \in \Delta^r_K$ then there is s'_M such that $(s_M, e, s'_M) \in \Delta^r_M$ and $(s'_K, s'_M) \in R$.*
- *If $(s_M, e, s'_M) \in \Delta^p_M$ then there is s'_K such that $(s_K, e, s'_K) \in \Delta^p_K$ and $(s'_K, s'_M) \in R$.*

We say that M refines K if there is a refinement relation R between K and M such that $(s^0_K, s^0_M) \in R$, denoted $K \preceq M$.

Intuitively, M refines K if every required transition of K exists in M and every possible transition in M is possible also in K. An LTS can be viewed as an MTS where $\Delta^p_K = \Delta^r_K$. Thus, the definition generalises to when an LTS refines an MTS. LTSs that refine an MTS K are complete descriptions of the system behaviour and thus are called *implementations* of K. We denote by $\mathtt{I}^{det}[K]$ the set of deterministic implementations of K.

3.2 Fluent Linear Temporal Logic

We describe properties using Fluent Linear Temporal Logic [14] (FLTL). To simplify notations we do not include the next operator in FLTL. All our results can be easily generalised to include the next operator.

A *fluent Fl* is defined by a pair of sets and a Boolean value: $Fl = \langle I_{Fl}, T_{Fl}, Init_{Fl} \rangle$, where $I_{Fl} \subseteq Act$ is the set of initiating actions, $T_{Fl} \subseteq Act$ is the set of terminating actions and $I_{Fl} \cap T_{Fl} = \emptyset$. A fluent may be initially *true* or *false* as indicated by $Init_{Fl}$. Every action $e \in Act$ induces a fluent, namely $\dot{e} = \langle e, Act \setminus \{e\}, false \rangle$. To simplify presentation we interchangeably refer to action fluents as $\dot{e}$ or e.

Let $\mathcal{F}$ be the set of all possible fluents over Act. An FLTL formula is defined inductively using the standard Boolean connectives and the temporal operator U operator (strong until) as follows: $\varphi ::= Fl \mid \neg\varphi \mid \varphi \vee \psi \mid \varphi U \psi$, where $Fl \in \mathcal{F}$. As usual we introduce $\wedge$, $\rightarrow$, $\Diamond$ (eventually), and $\Box$ (always) as syntactic sugar. We do not include the next operator (although we have used it in the motivating example). All our results can be easily generalised to include the next operator.

Let Π be the set of infinite traces over Act. The trace $\pi = e_0, e_1, \ldots$ satisfies a fluent Fl at position i, denoted $\pi, i \models Fl$, if and only if one of the following conditions holds:

- $Init_{Fl} \wedge (\forall j \in \mathbb{N} \cdot 0 \leq j \leq i \rightarrow e_j \notin T_{Fl})$
- $\exists j \in \mathbb{N} \cdot (j \leq i \wedge e_j \in I_{Fl}) \wedge (\forall k \in \mathbb{N} \cdot j < k \leq i \rightarrow e_k \notin T_{Fl})$

In other words, a fluent holds at position i if and only if it holds initially or some initiating action has occurred, but no terminating action has yet occurred. The interval over which a fluent holds is *closed* on the left and *open* on the right, since actions have an immediate effect on the value of fluents.

Given an infinite trace π, the satisfaction of a formula φ at position i, denoted $\pi, i \models \varphi$, is defined as in standard LTL [15]. We say that φ holds in π, denoted $\pi \models \varphi$, if $\pi, 0 \models \varphi$. A formula $\varphi \in$ FLTL holds in an LTS W (denoted $W \models \varphi$) if it holds on every infinite trace produced by W.

We use FLTL to describe control goals as explained below. We restrict attention to SGR(1) formulas [16], which are formulas of the form $\Box\rho \wedge (\bigwedge^n_{i=1} \Box\Diamond A_i \rightarrow \bigwedge^m_{j=1} \Box\Diamond G_j)$, where ρ, A_i and G_j are Boolean combinations of fluents. We choose FLTL rather than LTL since it provides a convenient way of referring abstract states (defined by the fluents) in a model that is based on events. We concentrate on SGR(1) due to the good performance of SGR(1) control solvers and their availability.

3.3 LTS Controller Synthesis

We now define the controller synthesis problem.

Definition 3.4. (LTS Control Problem [17]) *The LTS control problem is a tuple $\mathcal{K} = \langle W, G, \Sigma^c \rangle$. It includes a domain model in the form of a deterministic LTS $W = (S, \Sigma, \Delta, s^0_W)$, where the alphabet Σ is partitioned in controllable and uncontrollable actions, i.e. $\Sigma = \Sigma^c \uplus \Sigma^u$, and an FLTL formula G.*

A controller *is a function $f : \Sigma^* \rightarrow \mathcal{P}(\Sigma^c)$ that associates a set of controllable actions with every sequence of actions.*

A trace $\sigma \in \Sigma^\omega$ is compatible with controller f if for every $i \geq 0$ we have $\sigma_i \in f(\sigma_{0..i-1}) \cup \Sigma^u$.

We say f is a solution for $\mathcal{K}$ if all maximal traces σ of W that are compatible with f are infinite and in addition we have $\sigma \models G$. When such a controller exists we say the problem is realisable, and is unrealisable otherwise.

LTL realisability is decidable [18] and thus LTS Control is decidable in consequence. If realisable, extracting a controller C can be done simultaneously. Efficient synthesis of reactive-design have been developed to treat important fragments of LTL (e.g., GR(1) [10]). In this paper we will build on the polynomial time solution for SGR(1) goals (i.e., GR(1) goals [16] plus safety goals) in [11].

The decision of whether an LTS control problem is realisable is translated to a two-player game and winning in this game for one of the players (the controller). We freely use the term *winning state* to signify states that are visited by a solution controller. In other words, winning states are those from which it is possible for a controller to guarantee the goals. By determinacy of the underlying games all states that are not winning are losing.

We note that the specification of the example in Section 2 is not directly expressible as a SGR(1) goal. However, standard techniques can be used to extend the model with extra parts that allow to express these specifications as SGR(1) goals [16].

3.4 MTS Controller Synthesis

The problem of control synthesis for MTS is to check whether all, some or none of the deterministic LTS implementations of a given deterministic MTS can be controlled by an LTS controller [11]. This is defined formally below.

Definition 3.5. (MTS Control Problem [11]) *Given a deterministic MTS K, an FLTL formula G and a set $\Sigma^c \subseteq \Sigma$ of controllable actions, to solve the* MTS control problem $\mathcal{K} = \langle K, G, \Sigma^c\rangle$ *is to answer:*

- ***All**, if for all LTS $W \in \mathcal{I}^{det}[K]$, the control problem $\langle W, G, \Sigma^c\rangle$ is realisable,*
- ***Some**, if for some LTS $W \in \mathcal{I}^{det}[K]$, the control problem $\langle W, G, \Sigma^c\rangle$ is realisable,*
- ***None**, otherwise.*

Note that, as in the case of LTS control problem, we restrict attention to deterministic domain models. This follows from the fact that our solution for MTS realisability is by a reduction to LTS realisability. In this paper we distinguish between two cases: (a) **None** and (b) **Some** or **All**. As **All** implies **Some** we refer to this generally as **Some**.

The MTS control problem can be solved by reducing it to LTS control. We provide an overview here of how to decide if the answer is **None** or **Some**. How to distinguish between **Some** and **All** is not relevant for this paper. A full description of the reduction of MTS to LTS control can be found in [11].

To distinguish the **None** and **Some** cases, we build an LTS control problem in which at each state of the MTS, the controller gets to decide which possible but not requires actions should be available. If under these conditions the controller can achieve its goal, then the picks of possible but not required actions determine an MTS implementation for which a controller exists.

Indeed, this approach models an LTS control problem in which the controller gets to pick the "easiest" implementation to control. Formally, given an LTS K, we construct from it an LTS, which we call K^I, that has additional transition controllable actions ($i \subseteq \Sigma_K$). The additional transitions model the choice that the controller can make regarding which possible but not required transitions are available.

Definition 3.6. *Given an MTS $K = (S_K, \Sigma_K, \Delta^r_K, \Delta^p_K, s^0_K)$. We define $K^I = (S_{K^I}, \Sigma_{K^I}, \Delta_{K^I}, s^0_K)$ as follows:*

$$\begin{aligned} S_{K^I} &= S_K \cup \{(s,i) \mid s \in S_K,\ i \subseteq \Sigma_K \text{ and } \Delta^r_K(s) \subseteq i\} \\ \Sigma_{K^I} &= \Sigma_K \cup \overline{\Sigma_K}, \text{ where } \overline{\Sigma_K} = \{e_i \mid i \subseteq \Sigma_K\} \\ \Delta_{K^I} &= \{(s, e_i, (s,i)) \mid s \in S_K \text{ and } \Delta^r_K(s) \subseteq i \subseteq \Delta^p_K(s)\} \\ &\quad \cup \quad \{((s,i), e, s') \mid (s,e,s') \in \Delta^p_K \text{ and } e \in i\} \end{aligned}$$

States in K^I are of two kinds. Either states $s \in S_K$ or pairs (s,i), where $s \in S_K$ and i is a set of actions. Those that are of the form s encode a choice of which subset of possible transitions the implementation of K implements from s. Choosing a subset $i \subseteq \Sigma$, where $\Delta^r_K(s) \subseteq i \subseteq \Delta^p_K(s)$, leads to a state (s,i). States of latter form (s,i) have outgoing transitions labelled with actions in i. A transition from (s,i) on an action $e \in i$ leads to the same state s' in K^I as taking e from s in K. Thus, for (s,i) and $e \in i$ the e-transition from (s,i) leads to the t such that $(s,e,t) \in \Delta^p_K$.

Having built K^I, the LTS control problem to be solved $\mathcal{K}^I_S = \langle K^I, G, \Sigma^c \cup \overline{\Sigma}\rangle$ extends the controllable actions to include the new labels in K^I that choose which actions to enable and includes special treatment of the controller goals. We note that the algorithm for solving MTS control is simplified by the removal of the next operator, which we do not use. For further details, refer to [11].

The algorithm in Listing 1 computes the solution for the MTS control problem.

```
procedure solveMTS($K, G, \Sigma^c$)
  $\mathcal{K}^I_S = \langle K^I, G, \Sigma^c \cup \overline{\Sigma}\rangle$
  if (isLTSRealisable($\mathcal{K}^I_S$))
    return Some
  else
    return None
```

Listing 1. Solution for Definition 3.5

Note that a controller for $\mathcal{K}^I_S$ encodes both the controllable actions Σ^c needed to achieve the goal G but also the actions in $\overline{\Sigma}$ that "pick" for which implementation of K the goal G could be achieved. By hiding transitions labelled with $\overline{\Sigma}$, the controller can be run in an unknown environment with the hope that it is the the environment consistent with K for which the controller achieves its goal G.

4 Control and Discovery Problem

We now formalise the Control and Discovery Problem.

The Control and Discovery (C&D) problem assumes that the actual environment behaviour (currently unknown to the controller) can be described with an LTS W (the world). Furthermore, it assumes that there is an MTS K that represents the initial (possibly null) partial knowledge available about the environment behaviour that is to be controlled and that initial knowledge available is consistent with the environment. Formally, this means that they have the same alphabet ($\Sigma = \Sigma_W = \Sigma_K$) and the environment behaviour is a refinement of the knowledge: ($K \preceq W$). Note that K can model the complete absence of information about the environment's behaviour (i.e., $(\{s\}, \Sigma_K, \emptyset, \{s\} \times \Sigma_K \times \{s\}, s)$). We also assume that the interface with the environment (i.e., controlled and monitored events) is known. The problem then is to find an approach that controls the environment by choosing which controlled actions to enable when.

We assume it is possible to reset ($reset!$) the environment, which means to set the current state of W to its original initial one (s^0_W). Finally, we assume that there is a controlled action ($none!$) that is used by the controller to report that the goal is not realisable in the current environment. We denote the complete set of events of the C&D problem as $\Sigma_! = \Sigma \uplus \{reset!, none!\}$ and its controlled events as $\Sigma^c_! = \Sigma^c \uplus \{reset!, none!\}$.

The C&D problem for an FLTL goal G is to find a controller (f: $\Sigma^*_! \rightarrow \mathcal{P}(\Sigma^c_!)$) that, based on the behaviour observed so far, chooses controllable actions to enable in such a way that it will eventually either satisfy G from the last $reset!$ or conclude unrealisability.

Definition 4.1. (Control and Discovery Problem)) *Let K be a deterministic MTS with alphabet partitioned into controllable and uncontrollable actions, i.e., $\Sigma = \Sigma^c \uplus \Sigma^u$, $\Sigma_! = \Sigma \uplus \{reset!, none!\}$, $\Sigma^c_! = \Sigma^c \uplus \{reset!, none!\}$ and G an FLTL formula. We define a Control and Discovery C&D Problem as a tuple $\mathcal{K} = \langle K, G, \Sigma^c_! \rangle$.*

A controller *is a function $f : \Sigma^*_! \rightarrow \mathcal{P}(\Sigma^c_!)$ that associates every sequence of actions with a set of controllable actions. A trace $\sigma \in \Sigma^\omega_!$ is compatible with controller f if for every $i \geq 0$ we have: if $f(\sigma_{0..i-1}) \in \{reset!, none!\}$ then $\sigma_i = f(\sigma_{0..i-1})$, otherwise $\sigma_i \in f(\sigma_{0..i-1}) \cup \Sigma^u$. To simplify some of the notations we assume that $f(\epsilon) = \{reset!\}$.*

We say f is a solution *for $\mathcal{K}$ if for all $W \in \mathcal{I}^{det}[K]$ and for every trace $\sigma \in \Sigma^\omega_!$ that is compatible with f such that each maximal subsequence with no $reset!$ symbols in σ, projected onto Σ is a (finite or infinite) trace of W, we have:*

1) $\sigma \models \Diamond\, none!$ implies $\langle W, G, \Sigma^c \rangle$ is unrealisable, and

2) $\sigma \models \Diamond\, none! \vee \Diamond(reset! \wedge \bigcirc(G \wedge \Box \neg(reset! \vee none!))$

According to the definition above, a solution f when used in an environment in which it is possible to guarantee G, the resulting system behaviour is guaranteed to eventually discover enough of the unknown environment behaviour to guarantee G. More precisely, to perform a final reset! of the environment and from then on guarantee G.

On the other hand, the same solution f if deployed in an environment in which it is impossible to guarantee G, the resulting system behaviour is guaranteed to either achieve G indefinitely after some last reset! or eventually report that G cannot be guaranteed, outputting $none!$. Recall that the former corresponds to the case in which the environment does not play its best strategy (i.e., it does not pick actions that would force a violation of G) or does not exhibit behaviour that would allow concluding unrealisability.

5 Control and Discovery Solution

We now present an algorithm that solves for C&D problems.

We restrict attention to SGR(1) goals (i.e., GR(1) goals [16] plus safety), which given the assumption of deterministic implementations, admit polynomial time solution of the MTS control problem [11]. Furthermore, realisability check allows for checking if an SGR(1) goal can be realised from a non-initial state of the MTS with no extra cost. We exploit this in each exploration step as we must query realisability both from the initial system state and the current exploration state. One can generalise these techniques to full FLTL by using an alphabetised next and adding the additional bookkeeping required for checking the full formula from the non-initial states.

The algorithm interacts with the environment by means of an API `env` that provides the following methods: `getCurrentStateID` returns a state ID from a finite unknown set of possible IDs (recall that we assume the environment behaviour can be characterised as a finite LTS). Method `reset` restarts the environment to its initial state. Method `try(C)` requests execution of one of the controllable actions in (the possibly empty) set `C` and returns the action that was actually executed. The return value may be an action not in `C` if an uncontrollable action occurred, i.e., a race condition. The return value may be `null` if none of the controllable actions in `C` was available for execution in the environment (and no uncontrollable action was available either). For simplicity, we assume that the execution of the algorithm is fast enough for only up to one uncontrolled event to occur between each call to `try`, dropping this assumption requires simply managing a queue of events.

If the environment exhibits behaviour that corresponds to that of an LTS W that is a refinement of the (possibly empty) initial knowledge captured by K then the algorithm forces the environment to exhibit a trace that is consistent with a controller that solves the C&D problem $\langle K, G, \Sigma^c_! \rangle$.

We introduce some additional methods to make the main algorithm more readable. Given an MTS $K = (S_K, \Sigma_K, \Delta^r_K, \Delta^p_K, s^0_K)$ we assume a method that solves the control problem appearing in Listing 1: Given a state $s \in S_K$, we will use `getController`(s) as the method that either (a) returns a set of actions enabled from s according to the controller for $\langle K^I, G, \Sigma^c \cup \overline{\Sigma} \rangle$, or (b) if there are no such actions returns `null`. Note that initiating to state s requires not only setting the initial state of K^I but also updating the initial value of fluents in G based on the trace observed up to s. We assume that the methods `solveMTS` and `getController` do not recompute the controller unless the knowledge changes. This affords both consistency and efficiency. This means that as long as the knowledge is not updated *one* controller is used ensuring that if this controller is used continuously the goal is satisfied.

We assume `copyState(`s`)`, which creates a fresh copy of the state s with the same *outgoing* transitions and target states. Note that this operation produces an equivalent MTS (i.e., does not change its set of implementations).

For states s and s', a set of actions A, and an action e, method `removeTransitions(`s, A`)` removes all the transitions from s with actions in A and method `makeRequired(`s, e, s'`)` sets transition (s, e, s') as required. Finally, methods `getInitialState`, `setInitialState`, `existsStateWithID`, and `getStateByID` work as expected, and we assume that states of K have get/set methods for their ID attribute.

The algorithm that solves the C&D problem is given in Listing 2. The main procedure `Control&Discover` treats the `env` API as implicit. It receives am MTS parameter K, which we assume is refined by `env`. Its other parameters are the goal G and the set of controllable actions Σ^c. Initially, the procedure creates a copy of the initial state of K and sets its ID to the ID of the initial state of the environment `env`. It then sets the initial state of K to the newly created

```
1   procedure Control&Discover($K, G, \Sigma^c$):
2     output("$reset!$")
3     $s^c_K =$ $K$.copyState($K$.getInitialState)
4     $s^c_K$.setID(env.getCurrentStateID)
5     $K$.setInitialState($s^c_K$)
6
7     while (solveMTS($K, G, \Sigma^c$) $\neq$ None)
8         $C^S =$ getController($s^c_K$)
9         if ($C^S ==$ null)
10            env.reset
11            $s^c_K = K$.getInitialState
12            output("$reset!$")
13        else
14            $A^c = C^S \cap \Sigma^c$
15            $e =$ env.try($A^c$)
16            if ($e ==$ null)
17                $K$.removeTransitions($s^c_K, A^c \cup \Sigma^u$)
18            else
19                updateKnowledge($K, e, s^c_K$)
20                output($e$)
21            endif
22        endif
23    endwhile
24    output("$none!$")
25  endprocedure
26
27
28  procedure getController($s$)
29      ${K^I}_s = (S_{K^I}, \Sigma_{K^I}, \Delta_{K^I}, s)$
30      ${}_s\mathcal{K}^I_S = \langle {K^I}_s, G, \Sigma^c \cup \overline{\Sigma} \rangle$
31      if (isLTSRealisable(${}_s\mathcal{K}^I_S$))
32          win = winningStatesOf(${}_s\mathcal{K}^I_S$)
33          $succ(s) = \{(s,i) \in$ win $\mid (s,i,(s,i)) \in \Delta_{K^I}\}$
34          $\Delta^{\text{win}}(succ(s)) = \{((s,i),e,s') \in \Delta_{K^I} \mid (s,i), s' \in$ win$\}$
35          $C^S = \{e \mid ((s,i),e,s') \in \Delta^{\text{win}}(succ(s))\}$
36          return $C^S$
37      else
38          return null
39  endprocedure
40
41  procedure updateKnowledge($K, e, s^c_K$)
42    if ($K$.existsStateWithID(env.getCurrentStateID))
43        $s' = K$.getStateByID(env.getCurrentStateID)
44    else
45        $s' = K$.copyState($\Delta_K(s^c_K, e)$)
46        $s'$.setID(env.getCurrentStateID)
47    endif
48    $K$.removeTransitions($s^c_K, \{e\}$)
49    $K$.makeRequired($s^c_K, e, s'$)
50    $s^c_K = s'$
51  endprocedure
```

Listing 2. Algorithm for solving C&D Problems

state. The procedure continues as long as the answer to the MTS control problem is not **None** from the *initial state* of the current knowledge. If this is not the case, it signals $none!$ and terminates. Otherwise, the main loop proceeds by first, checking if from the *current state* the answer to the MTS control problem is **None** (inside `getController`). If that is the case, the C&D algorithm needs to restart from the initial state (lines 10-12). Note that this is not a case of unrealisability as from the initial state of K we just confirmed that the answer is not **None**. Reaching **None** is due to a bad (uninformed) decision of the controller at some earlier point.

By line 8, C^S is a set of actions that are possible from the current state of K and are part of the controller for the current stage of the knowledge. This set of actions is then used to offer its subset of controllable actions A^c that are guaranteed to move towards achieving G for some implementation that refines K.

We then call `env`'s `try` method to attempt to execute some controllable action in A^c. Method `try` returns information (e) of what actually happens: If e is `null`, then in the current state of the environment no uncontrollable actions are possible and all actions in A^c are not possible either. Accordingly, we update K by making $A^c \cup \Sigma^u$ prohibited from s^c_K in Line 17. Otherwise, we update our MTS by calling `updateKnowledge` (Line 19) informing e and s^c_K. Finally, the algorithm outputs the executed action e.

Procedure `updateKnowledge` gets the action that was executed by the environment e and current state s^c_K, which it updates. The procedure must decide if the current state of the environment (having executed e) has been visited before, and decides whether it must update the knowledge K. If the state has been visited before, we use the same knowledge state. Otherwise, we create a fresh copy of the e successor of s^c_K and set its ID. As action e was just taken, we know that it is required. Finally, we update the current state of K before returning. Notice that the procedure does not necessarily *refine* the knowledge. That is, the set of implementations of the updated knowledge is not necessarily a subset of the set of implementations before updating. However, the *sensed environment* remains an implementation of the updated knowledge.

Choice of Controllable Actions

Method `getController` in Line 8, which is elaborated in Line 28, encapsulates the choice of controllable actions to offer in Line 15. This choice requires a more detailed explanation of how the set of actions $C^S$is chosen.

Let $\mathcal{K} = \langle K, G, \Sigma^c \rangle$ be the MTS control problem, the derived LTS $K^I = (S_{K^I}, \Sigma_{K^I}, \Delta_{K^I}, s^0_K)$, and the control problem *some* $\mathcal{K}^I_S$ as in Definition 3.6. Recall that the states of K^I are either states s of K or pairs (s, i), where s is a state of K and i is a set of actions.

The solution of the ${}_{s^c_K}\mathcal{K}^I_S$ control problem classifies states of ${K^I}_{s^c_K}$ as winning or losing. A state s is winning if for some set of actions i we have that (s, i) is winning. However, due to the transition structure of $\mathcal{K}^I_S$ it follows that regardless of the set of actions i, there is some set of successors of s that are also winning. Hence, there is a unique *maximal* set of actions i such that all $l \in i$ successors of s are winning. In our implementation, method `getController` returns the set i. Later, set $i \cap \Sigma^c$ of controllable actions that lead to winning states is offered to `env.try`. Notice that every non-empty subset of this maximal set i would be suitable for our purposes. For example, we could choose to explore more or less based on including more/less possible (but not known to be required) transitions.

Statement of Correctness

We now state and prove the correctness of the algorithm in Listing 2.

Theorem 5.1. *The algorithm* `Control&Discover` *enacts a controller that solves the Control and Discovery problem for K and G.*

The proof of correctness relies on the following lemmata.

Lemma 5.1. *The current state of K always has the ID of the current state of the environment.*

Proof. This is established initially (in line 5) by creating a copy of the initial state of K and giving it the identifier of the initial state of W. Whenever in W there is a non `null` transition, this invariant is re-established by either getting the state of K that has the identifier of the current state of W or creating such a state. □

Lemma 5.2. *If initially W is an implementation of K, then after every knowledge update W is an implementation of K.*

Proof. We prove a stronger claim, the current state of K is refined by the current state of W and that for every pair of states t of W and s of K such that s has t's identifier then t refines s. This is the loop invariant.

Our proof is by induction on the progress of the algorithm. Initially, we assume that K is refined by W. Hence, the initial state of K is refined by the initial state of W. We create a copy of the initial state of K and give it the identifier of the initial state of W (line 5). It follows that the lemma holds when the loop condition is evaluated the first time.

Suppose by induction that K is refined by W, that the current state t of W is refined by the current state s of K, and that for every pair t' and s' such that t' and s' have the same identifier we have t' refines s'. There are three possible runs through the loop. Either C^S is `null` or not. In the second case, either the action returned in line 15 is `null` or not `null`. If C^S is `null`, then nothing in the refinement mapping is changed and both the environment and the knowledge return to their initial states, which, by induction, are in a refinement relation.

Suppose that C^S is not `null`. If the returned action in line 15 is `null`, then both the environment and K remain in the same state. Every possible transition that is removed in K is known not to exist in the environment maintaining the refinement relation.

The last case is when the action returned in line 15 is not `null`. Let t denote the state of W before taking action e and t' denote the state of W after the action. By assumption s^c_K refines t. Procedure updateKnowledge then re-establishes the invariant as follows.

In case t' was visited previously, there is a state s' of K such that t' refines s'. The first branch of the `if` in updateKnowledge is taken. We set s' as a required e successor of s^c_K. The transition (t, e, t') is required in W and hence possible as well. As required by the refinement relation, transition (s^c_K, e, s') is possible in K. Transition (s^c_K, e, s') is required in K. As needed by the refinement relation, transition (t, e, t') is required in W. It follows that this change re-establishes the refinement relation.

In case t was not visited previously, there is no state of K with the same ID. The second branch of the `if` in updateKnowledge is taken. As (t, e, t') is a required (and hence possible as well) transition of W and by induction s^c_K is refined by t, it follows that for some state s' we have (s^c_K, e, s') is a possible transition in K and t' refines s'. We create a copy of s' and set the ID of the new copy to that of t'. Let s'' denote the copy of s'. Clearly, s'' is still refined by t'. We make the transition (s^c_K, e, s'') required in K. In order to re-establish the refinement, we have to show that there is a matching required transition in W. However, the transition (t, e, t') is a required transition in W supplying this requirement. □

We are now ready to prove Theorem 5.1.

Proof. Sequences of actions taken between $reset!$ or $none!$ are sequences of actions taken by W. It follows that maximal subsequences with no $reset!$ symbols are traces of W.

We show the algorithm outputs $none!$ only when $\langle W, G, \Sigma^c \rangle$ is unrealisable. The only case where the algorithm outputs $none!$ is after concluding that the result of the MTS control problem $\langle K, G, \Sigma^c \rangle$ is none. By the soundness of the MTS control it follows that in every implementation of K the goal G cannot be realised. By Lemma 5.2 W implements K implying that $\langle W, G, \Sigma^c \rangle$ is unrealisable.

Finally, we show that the trace satisfies either eventually $none!$ or eventually after the last reset the trace satisfies the goal. Suppose that the algorithm does not output $none!$. We have to show that there is a final $reset!$ and that after that $reset!$ the trace satisfies G.

We note that a state can be found to be losing only once. A visited state of K has an ID and the same ID cannot be given to two different states of knowledge. Hence, this can happen a finite number of times. Once a state of K is found to be losing w.r.t. the goal, it can only be visited again by exploring a new transition. Indeed, otherwise, the controller in $\langle K^I, G, \Sigma^c \cup \overline{\Sigma} \rangle$ never visits states that are losing. More formally, we say a state s of K is said to be visited if at some point of the procedure it is stored as value of s^c_K. Note that those states also have been assigned an ID and, as said before, that, for a given ID, there is at most one state of K with that ID assigned. Revisiting a losing state s of K can only happen after executing an env.Try (line 15), and finding that the target state of the e transition has an ID that has been already assigned to s (line 30) and is losing. Given that there is a refinement relation between K and W that is consistent with the IDs assigned so far, there must be at least a possible transition labeled as e from state stored in s^c_K. Note that line 15 was executed knowing that the current state (s^c_K) of K is not a losing state implying that (a) all required uncontrollable transitions lead to winning states and (b) all controllable transitions that are enabled lead to winning states. It follows that one of the following two options happened: (a) s^c_K has an uncontrollable permissible (but not required) transition to the losing s and by execution of the env.Try we are learning that this transition is required or (b) s^c_K has a (controllable or uncontrollable) permissible or required transition to some other state s' that has not been visited. However, upon taking the e-transition we learn that the e successor in the environment actually has the ID of s. In both cases, a new transition is explored, which can happen a finite number of times as there are finitely many transitions.

Hence, there is a finite number of $reset!$ events.

For similar reasons, after the last $reset!$, the number of updates to K is finite. We conclude that an infinite suffix is played according to a controller obtained from the same knowledge. However, this controller guarantees the goal according to $\langle K^I, G, \Sigma^c \cup \overline{\Sigma} \rangle$. It follows that the suffix satisfies G from the last $reset!$. □

6 Validation

The goal of this section is to explore feasibility and in particular the behaviour that the C&D exhibits for different case studies and how it varies depending on two factors:

Realisability of the goals in the environment to be controlled and discovered, and the degree to which the environment facilitates behaviour discovery by exhibiting behaviour unknown to the controller. We use seven case studies from the literature in two different versions, realisable and unrealisable, and for each one we run the C&D seven times against three different kinds of environments.

Briefly, each case study includes a third-party developed behaviour model of the environment and goals that a controller is to achieve. We treat this model as the unknown W to be discovered and controlled. We define ourselves some (very basic) initial knowledge K. We then show how the algorithm either controls the (unknown) environment and achieves the goals or discovers enough behaviour to declare that the (unknown) environment cannot be controlled.

All the material for replicating the validation can be found at [19].

6.1 Subjects

All case studies were run using an extension of the MTSA tool, which natively supports specification of LTS and properties using a textual process-algebraic notation (FSP) and temporal logic. The tool also supports synthesis of controllers for SGR(1) control problems. The tool was extended to support defining and solving C&D problems, and also for executing solutions against environments. The extended version of the tool and case studies can be found at [19].

To avoid bias and manual manipulation, we used case studies where environments to be controlled are described as LTS and goals are given as temporal logic formulas.

We use the case study from [4] taken from the Global Monitoring for Environment and Security (***GMES***) European Programme emergency management service, which covers catastrophic events such as floods, earthquakes, and fires. The scenario involves coordinating services from two different countries. In one country there is a Command and Control center in charge of forest monitoring and forest fire management. The other country is to provide source reinforcement services including UAVs and weather services.

We used the specification of the ***Production Cell*** case study in in [20], which was originally presented in [21] and studied extensively: a robotic arm coordinates the application of various tools to construct a product fulfilling some safety and liveness requirements. The liveness requirement is to construct infinitely many products. The assumption is that if the controller is waiting for pieces to construct a new product, it eventually receives them. Safety rules describe how and when tools can be used and the requirements to be fulfilled before placing products in the outgoing conveyor belt. The specification assumes that tools may fail but include a liveness assumption stating, in essence, that these failures are of a probabilistic nature and that retrying them sufficiently will eventually lead to a successful use of the tool. The problem here is to synthesise a controller for the robotic arm that satisfies the specification based on the knowledge of the tool APIs.

We use a ***search and rescue*** case study originally presented in [22]. Here a robot must explore within a collapsed house taking supplies to people trapped in one of the rooms. In addition a number of obstacles may intermittently

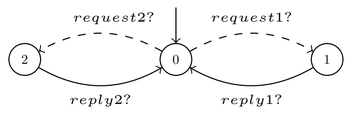

Fig. 5. MTS for request/response patterns.

impede movement of the robot. The controller for the robot must be synthesised automatically based on the behaviour of the robot API and mission requirements.

We also adapted the ***Purchase and Delivery*** case study from [5], which involves synthesising a controller for composing and monitoring of distributed web services. We used the ***Bookstore*** from [6] that has similarities with the Purchase and Deliver case study, services are to be coordinated to provide a more complex book querying, purchasing and delivery system. For all the case studies mentioned above we used the formalisation published in [20]. We use the ***service mediation*** case study from [23] which involves mediating two services with different interfaces, and a travel aggregator service (***Travel Agency***) [20]) which involves an orchestration scenario of multiple booking services to provide holiday packages.

All of the case studies above present realisable control problems. To include unrealisable scenarios, and due to the unavailability of published unrealisable control problems expressed as LTS + FLTL, we manually modified each case study to make unrealisable versions of them. We modified the GMES case study to allow for unrecoverable UAV landing failures, the Production Cell was changed similarly to allow for unrecoverable tool failures. In the search and rescue case study we removed transitions in which the environment allowed the door to be opened once the robot has picked up but not delivered packages, thus impeding it from achieving its goals. Service mediation was changed to allow for different services to time-out and not return the expected results. In Travel Agency and Bookstore case studies we removed a fairness assumption on the successful termination of services and for Purchase and Delivery we added the possibility of failing purchases. Unrealisable versions of the case study are referred to with an "UR" suffix.

6.2 Experiments

For each case study we chose an MTS to represent the initial knowledge the controller has of its environment. We used two different types of initial knowledge. For Production Cell, Disaster Recovery, Service Mediation, Travel Agency, and Purchase and Delivery we assumed that there was no initial knowledge whatsoever. In other words we used an MTS with one state and all model actions enabled with looping maybe transitions. For the rest of the case studies (GMES and Bookstore) we assumed knowledge of request response nature of some events, resulting in MTS following a pattern depicted in Figure 5.

To run the algorithm we need to provide an interface to the concrete execution environment that is to be controlled and discovered. We built an enactor that given the description of the environment behaviour as an LTS implements the World API. A key decision is how the enactor chooses the next action to take (i.e., `try(`A^c`)`). We chose three kinds of enactors: a *random* environment that picks the next action to occur randomly, an environment that *facilitates* discovery by choosing transitions that have not been taken before, and an environment that *hinders* discovery by deliberately taking transitions that have been taken before.

As the algorithm is designed to control the environment, if possible, *ad infinitum* while achieving the goal, we introduced a stopping point to heuristically detect if the process has converged (i.e., no more reset! occur in the future). Note that the algorithm cannot necessarily recognise this point as it cannot distinguish between behaviour that may occur but the environment decides not to exhibit for some (unbounded) time from behaviour that cannot occur. Thus, we use a criterion that stops the C&D process when the number of actions of the execution trace is 11 times the number of events up to the last time the knowledge about the environment was refined. After stopping, we checked if the entire behaviour of W was covered or if the resulting behaviour since the last reset! corresponds to behaviour that a controller built directly with full knowledge of W would have performed. Such an inspection contributes to ensuring that indeed the behaviour of the controller has stabilised. Indeed, in all cases at the stopping point the C&D process had stabilised for all realisable case studies.

6.3 Results

In Table 1 we show results obtained from running the algorithm against the case studies for the random (R) environment and for the environments that facilitate (F) and hinder (H) discovery. We report the number of refinements and resets performed while discovering behaviour. This provides information on how many times new information was incorporated into the knowledge that the algorithm is building up, and the number of times the algorithm ran into a dead end (making it inevitable that it will violate a property) and had to start again.

We also report on the moment (measured in number of actions) when the last refinement was performed. This shows how soon sufficient knowledge to control or declare unrealisability was achieved. We also report on the occurrence of the last reset which, for executions that do gives an indication as to from when the system goal is being achieved. Note that no information available for last reset means that the behaviour from the very beginning satisfies the system goals (i.e., the goals were never violated). Finally, we report on the degree of coverage of the actual environment behaviour in terms of states and transitions.

The table reports coverages with significant variability amongst the various case studies. Variation can be due to different reasons, for instance the degree of controllability of the environment and the extent to which certain uncontrolled actions do not contribute to achieving the goal or lead to violations. The latter can allow the controller to avoid exploration of large portions of the state space. Of course, the use of an environment that hinders discovery (H) typically leads to less exploration than one that facilitates discovery (F) or a random (R) one. As expected, F tends to produce higher coverage than R. There is also significant variability between case studies in terms of the number of resets and refinements required to stabilise behaviour or actually declare non-realisability. Also, as can be expected, unrealizable environments tend to have more resets than their realisable counterparts.

We do not report on execution times as these are irrelevant in our experiments using synthetic environments. Execution time in a real setting will depend on aspects such as the speed at which the service responds to controlled actions or the time it takes to produce an uncontrolled event that is expected by the controller. The C&D algorithm's impact on execution time depends on the computation time for the next action to control. The average time for this operation was under 1 second for 10 of the 14 case studies, the GMES and Travel Agency were significantly more complex with average times of 18 and 32 seconds.

Additional data that is relevant but not in the table is which of the C&D executions for unrealisable specifications achieved their goals and did not terminate because the controller concluded none!. As expected, when using an environment that facilitates discovery (F), all executions concluded none!. This is because the C&D controller succeeds in exploring sufficient environment behaviour and refine its knowledge to conclude that realising the goal is impossible. The same happened for random environments (R) because with sufficient attempts, all uncontrolled choices are taken at least once, hence eventually sufficient knowledge to declare none! is acquired. For environments that hinder discovery, 30% of executions did not declare none!, rather they satisfied the goals from the last reset. These cases correspond to executions in which the first time the environment picked randomly the actions that the controller needed to achieve its goals. Environment H then sticks to this choice, helping the controller achieve its goals.

In contra-position to our approach (in which goals may be achieved in unrealisable settings depending on the behaviour of the environment), consider an approach where the controller selects randomly what action to take until it has enough information to either produce a strategy that wins or declare unrealisability. Such a naive approach with an unrealisable environment runs the risk of not achieving either. Indeed, we experimented with such an approach and found executions in which the controlled system continuously reset! and never declared none!. An example is the unrealisable Production Cell case study which we ran for 20 times longer than any other execution and in which reset! was periodically performed.

In conclusion, we applied C&D to realisable and unrealisable versions of seven case studies taken from the literature. For each case study we used three different kinds of environments that vary the degree to which their behaviour is amenable to discovery. For each of these 42 scenarios we observed that the algorithm succeeds in its C&D task but that the degree to which behaviour is actually discovered and the number of resets varies significantly depending on the environment's strategy.

7 Related Work and Discussion

Interacting with unknown environments and trying to achieve system goals has been the focus of significant study in the context of service oriented systems. Service discovery is readily available in many environments (e.g., Google APIs Discovery Service, WS-Discovery, Bonjour) and interface descriptions can be described using languages such as WSDL.

	Refinements			Resets			Last Refinement			Last Reset (#Actions)			State Cvg.(%)			Transition Cvg. (%)		
Case study	R	F	H	R	F	H	R	F	H	R	F	H	R	F	H	R	F	H
Bookstore	19	19	8	0	0	0	60	54	8	-	-	-	100	100	69	100	100	71
Bookstore UR	17	17	9	1	1	0	54	30	9	-	24	-	98	100	70	96	96	76
GMES	45	45	6	0	0	0	318	350	7	-	-	-	100	100	32	98	98	36
GMES UR	33	39	8	5	6	0	69	73	8	104	150	-	85	91	34	85	91	39
ProductionCell	54	58	4	6	6	0	176	171	5	137	148	-	83	85	9	59	62	4
ProductionCell UR	38	29	15	4	3	3	89	56	54	81	-	-	67	49	27	42	31	16
Purchase & Delivery	10	10	9	0	0	0	13	12	9	-	-	-	25	25	24	9	9	7
Purchase & Delivery UR	19	20	11	0	0	0	37	38	13	-	-	-	35	35	26	15	16	9
Search & Rescue	36	48	30	0	0	0	132	388	238	-	-	-	66	80	56	48	65	41
Search & Rescue UR	9	11	4	1	1	0	12	13	5	-	-	-	16	22	9	9	12	5
Service Mediation	27	38	29	0	0	0	1276	1354	3059	-	-	-	19	23	20	8	11	9
Service Mediation UR	40	40	37	3	2	1	71	69	60	62	-	50	26	26	23	9	9	8
Travel Agency	142	143	5	0	0	0	2767	1868	10	-	-	-	36	36	2	16	16	0
Travel Agency UR	38	37	10	2	2	1	81	58	18	57	52	-	14	14	4	4	4	1

TABLE 1
Control and discovery for three types environments: One that chooses randomly (R), one that facilitates discovery (F) and one that hinders discovery (H). Numbers are rounded to the nearest integer.

Verification of orchestration strategies of such environments typically assumes the existence of behaviour models of the services to be coordinated (e.g., [24]).

The automated construction of connectors, orchestrators and mediators can be understood as a controller synthesis problem. Connector synthesis approaches like [2] exploit ontology reasoning and constraint programming to automatically infer mappings between component interfaces. When behavior models are unavailable, those approaches suggest, as a pre-process, the use of automata learning techniques (e.g., the MAT framework [7]) for the extraction of behavioural models [1] and for the construction of emergent middleware [3]. Unlike our approach, MAT framework requires the ability to answer equivalence queries (typically approximated by using conformance testing (e.g., [12])). Moreover, it is assumed that the produced abstract output symbol is uniquely determined by the preceding sequence of abstract input symbols [8]. This assumption does not hold in general when the environment exhibits uncontrollable behaviour. Our approach is designed to overcome this limitation by integrating control strategy with the discovery process (i.e., it is not a two phase approach).

The community studying motion and high-level mission planning (e.g., [25], [26]) has recently addressed the problem of recovery from inaccurate description of the environment [27], [28], [29], [30], [31], [32], partial models of the environment [33] and model inference [34], [35]. In the first group of works, synthesis procedures are adapted to tolerate modelling assumption violations. The work presented in [27], considers uncertainties in open finite transition systems due to unmodeled transitions. In the same vein, [28] proposes a way to synthesise robust controllers that will be able to win even if certain unexpected transitions that occur a finite number of times. Similarly, [29] solves the problem of synthesising error resilient systems from specifications in temporal logic. Error resilience means tolerate arbitrarily many violations of safety assumptions. The work in [30] presents a tiered framework for combining behaviour models, each with different associated assumptions and risks due to modelling inaccuracies. In [31], [32] the controller for an originally realisable specification is extended with actions, if any exist, that preserve the robot's safety requirements and make sure that the robot can make progress towards its goals when the environment resumes the expected behaviour. In this line of work one should start with a realisable specification and the actual environment can just add (unexpected) behaviour. This is not adequate when behaviour to achieve goals needs to be discovered: in our approach, actual environment can either exhibit or even eliminate potential behaviour and there is no need to start with realisable specifications. MTS [9] have been used to reason about partial knowledge of system behaviour (e.g., [36]) and also for synthesising controllers with incomplete knowledge about the environment [11], [37]. However, progressive refinement of MTS models based on runtime observations of the system is not considered.

Recently [33] proposes an ad-hoc method, inspired on MTS [9], to produce plans for multi-robot missions with partial knowledge. Authors hard-code three sources of uncertainty: partial knowledge about the actions execution, unknown service provisioning, and unknown meeting capabilities of robots. Each type of uncertainty is managed in a specialised way. That approach in not meant to handle general partial models of reactive systems in a uniform way, as we propose in this work. In particular, the partial planning approach used is not able to deal with general uncertainty regarding uncontrollable actions: They only handle cases of uncertain responses to controllable actions and cannot, for instance, deal with uncertain occurrence of interrupt-like uncontrolled behaviour. Another difference is that in [33] a single non-occurrence of an uncertain transition is considered enough evidence to conclude, during execution, that the transition is not present. In the motivating example (Section 2) this means assuming that if the first query resulted in an available book, all future queries will do so too.

Another collection of results, on adaptive planning (e.g., [34], [35]), deal with a problem close to ours, namely when the precise shape of the model is (partially) unknown. In [38] a framework is used that integrates grammatical inference with symbolic control on finite-state transition systems interacting with partially unknown, adversarial, rule-governed environments (i.e., states are observable by means of the propositional assignment that rule precondition/postcondition of actions). However, there are a couple of limitations that are not present in our work. Firstly, the class of grammar representing the environment must be known in advance [39], secondly, a positive enumeration of the game's language (a function that enumerates all prefixes of the game runs) is given as hypothesis of the theoretical results on playing an equivalent game. Positive enumeration can be potentially hindered by an adversarial environment and while our approach explicitly deals with the environment not collaborating to exhibit its full behaviour, which may imply contingently but perpetually winning the game.

Given a unrealizable control problem, mining assumptions (e.g., [40], [41]) attempts to find an assumption for which a control problem is realisable. This process does not include finding an assumption that is consistent with an actual environment. Indeed, the authors argue that the goal is "to discover the designer's intent" as opposed to discover the assumptions that hold in the environment.

To the best of our knowledge, *the use of controller synthesis to develop strategies that also discover is a novel idea.* Synthesis based on partial knowledge guarantees that when the controller chooses a controlled action it either continues to progress towards achieving the goal or eventually reveals unknown environment behaviour. Put differently, the chosen action is guaranteed not to be losing (the environment can prevent the goal from being achieved) based on the current knowledge. A random exploration may pick actions that are losing, thus leading to behaviour that never acquires new knowledge nor gets closer to achieving its goals. A discovery strategy that aims to only discover (e.g., shortest path to a maybe transition in K) may never succeed in fully discovering while not achieving the goal either.

7.1 Limitations of Our Approach

The approach presented in this paper has two assumptions that can be limiting in various contexts. Both assumptions are related to expectations about the interface with the environment. We assume that it is possible to ask the environment for an ID that uniquely identifies the current state of the environment. We also assume that it is possible to reset the environment back to its initial state. Although this may considered a tall order, it is arguably less restrictive than the assumptions used in widespread automata learning approaches inspired by the MAT framework [7]. In these approaches the use of equivalence queries means answering a more complex question (behaviour equivalence, which implies knowing the current state and its behavioral capabilities) and is typically implemented via conformance testing, which requires the ability to reset the environment.

Note that service oriented systems (where sessions can be thought of as resets and hash functions could implement state IDs) are an example of a class of systems that could be made compatible with our assumptions on environments. In addition note that for 6 out of 7 case studies (in their realisable versions) resetting was not needed to achieve the desired goal.

Nonetheless, our assumptions do limit the applicability of the approach. Beyond service-oriented systems, cyber-physical systems are significantly more challenging. Considering how to extend our approach to cope with partial knowledge regarding state ID (e.g., knowledge about location but not other features) is of interest.

Although this paper goes beyond a purely theoretic solution by providing an implementation and showing is can handle literature case studies, there are major technological challenges for actually deploying this approach. One of these is to perform an API wrapping of the system under control. Indeed, a self adaptation approach could include wrapping the managed system with the components that implement a MAPE loop [42] including a module that observes the state of the environment and recognises that it has been there before. This module could feed state ids to the approach described herein.

We envision a large spectrum of alternatives depending on how one could interact with the API/interface and underlying development technology of the system-to-be-controlled (we believe it would be easier to design or alter a component to provide the suggested API than altering it to yield an always up-to-date protocol model). Ultimately, if an external component has uncontrollable behavior, traditional Mealy machine learning approaches are not applicable, some form of exploration is required.

8 Conclusions

In this paper we define the control and discovery problem that supports controlling environments with initial unknown behaviour. The problem allows (but does not require) starting with partial knowledge, represented as an MTS, about the environment behaviour. We also present a solution to the control and discovery problem for FLTL goals by building on MTS control. The algorithm assumes that the discovery process can identify when the environment returns to a previously visited state and that environment's behaviour corresponds to that of a regular language. The solution to an MTS control problem provides a controller that either incrementally builds up more knowledge about environment behaviour or guarantees the systems goals. We discuss an implementation of the control and discovery algorithm in the MTSA tool and report on its use on various case studies taken from the literature.

References

[1] A. Bennaceur, V. Issarny, D. Sykes, F. Howar, M. Isberner, B. Steffen, R. Johansson, and A. Moschitti, "Machine learning for emergent middleware," in *Intl. Conf. on Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 16–29.

[2] A. Bennaceur and V. Issarny, "Automated synthesis of mediators to support component interoperability," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 221–240, March 2015.

[3] P. Inverardi, V. Issarny, and R. Spalazzese, "A theory of mediators for eternal connectors," in *Intl. Conf. on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part II*, ser. ISoLA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 236–250.

[4] N. Nostro, R. Spalazzese, F. D. Giandomenico, and P. Inverardi, "Achieving functional and non functional interoperability through synthesized connectors," *Journal of Systems and Software*, vol. 111, pp. 185 – 199, 2016.

[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. LNCS. Springer-Verlag, 2004, vol. 3192, pp. 106–115.

[6] P. Inverardi and M. Tivoli, "A reuse-based approach to the correct and automatic composition of web-services," in *Engineering of software services for pervasive environments*. ACM, 2007.

[7] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

[8] F. Aarts, B. Jonsson, J. Uijen, and F. Vaandrager, "Generating models of infinite-state communication protocols using regular inference with abstraction," *Form. Methods Syst. Des.*, vol. 46, no. 1, pp. 1–41, Feb. 2015.

[9] K. Larsen and B. Thomsen, ""A Modal Process Logic"," in *Proceedings of 3rd Annual Symposium on Logic in Computer Science (LICS'88)*. IEEE Computer Society Press, 1988, pp. 203–210.

[10] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive (1) designs," in *Intl. Conf. on Verification, Model Checking and Abstract Interpretation*. Springer, 2006, pp. 364–380.

[11] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "The modal transition system control problem," in *FM 2012*, D. Giannakopoulou and D. Méry, Eds., 2012, pp. 155–170.

[12] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Combining model learning and model checking to analyze tcp implementations," in *CAV*, 2016, pp. 454–471.

[13] A. van Lamsweerde, *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[14] D. Giannakopoulou and J. Magee, "Fluent model checking for event-based systems," in *ESEC/FSE-11*. ACM, 2003, pp. 257–266.

[15] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE, 1977, pp. 46–57.

[16] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *J. Comput. Syst. Sci.*, vol. 78, no. 3, pp. 911–938, 2012.

[17] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "Synthesising non-anomalous event-based controllers for liveness goals," *ACM Tran. Softw. Eng. Methodol.*, vol. 22, 2013.

[18] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Principles of programming languages*. ACM, 1989, pp. 179–190.

[19] M. Keegan, V. Braberman, N. D'Ippolito, N. Piterman, and S. Uchitel, "MTSA Tool and data for Control and Discovery of Reactive System Environmnets." 2020, http://mtsa.dc.uba.ar/.

[20] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel, "Synthesis of live behaviour models for fallible domains," in *Intl. Conf. on Software Engineering*. ACM, 2011, pp. 211–220.

[21] C. Lewerentz and T. Lindner, Eds., *Formal Development of Reactive Systems - Case Study Production Cell*, ser. LNCS, vol. 891, 1995.

[22] W. Heaven, D. Sykes, J. Magee, and J. Kramer, "A case study in goal-driven architectural adaptation," in *Software Engineering for Self-Adaptive Systems*. Springer-Verlag, 2009, pp. 109–127.

[23] A. Bennaceur and V. Issarny, "Mics: Mediator synthesis to connect components website," 2019. [Online]. Available: https://www.rocq.inria.fr/arles/index.php/software/219-mics

[24] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Model-based verification of web service compositions," in *ASE*, 2003.

[25] G. E. Fainekos, A. Girard, H. Kress-Gazit, and G. J. Pappas, "Temporal logic motion planning for dynamic robots," *Automatica*, vol. 45, no. 2, pp. 343–352, Feb. 2009.

[26] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. J. Pappas, "Symbolic planning and control of robot motion," *IEEE Robotics Automation Magazine*, vol. 14, no. 1, pp. 61–70, 2007.

[27] U. Topcu, N. Ozay, J. Liu, and R. M. Murray, "On synthesizing robust discrete controllers under modeling uncertainty," in *HSCC*. New York, NY, USA: ACM, 2012, pp. 85–94.

[28] S. Dathathri, S. C. Livingston, and R. M. Murray, "Enhancing tolerance to unexpected jumps in gr(1) games," in *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPS)*, April 2017, pp. 37–48.

[29] R. Ehlers and U. Topcu, "Resilience to intermittent assumption violations in reactive synthesis," in *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '14. New York, NY, USA: ACM, 2014, pp. 203–212. [Online]. Available: http://doi.acm.org/10.1145/2562059.2562128

[30] N. D'Ippolito, V. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel, "Hope for the best, prepare for the worst: Multi-tier control for adaptive systems," in *ICSE 2014*. New York, NY, USA: ACM, 2014, pp. 688–699.

[31] K. W. Wong, R. Ehlers, and H. Kress-Gazit, "Resilient, provably-correct, and high-level robot behaviors," *IEEE Transactions on Robotics*, vol. 34, no. 4, pp. 936–952, Aug 2018.

[32] ——, "Correct high-level robot behavior in environments with unexpected events," in *Proceedings of Robotics: Science and Systems*, Berkeley, USA, July 2014.

[33] C. Menghi, S. Garcia, P. Pelliccione, and J. Tumova, "Multi-robot ltl planning under uncertainty," in *Formal Methods*, 2018, pp. 399–417.

[34] J. Fu, H. G. Tanner, and J. Heinz, "Adaptive planning in unknown environments using grammatical inference," in *52nd IEEE Conference on Decision and Control*, Dec 2013, pp. 5357–5363.

[35] J. Fu, H. G. Tanner, J. N. Heinz, K. Karydis, J. Chandlee, and C. Koirala, "Symbolic planning and control using game theory and grammatical inference," *Engineering Applications of Artificial Intelligence*, vol. 37, pp. 378 – 391, 2015.

[36] S. Uchitel, G. Brunet, and M. Chechik, "Synthesis of partial behavior models from properties and scenarios," *IEEE Transactions on Software Engineering*, vol. 35, pp. 384–406, May 2009.

[37] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel, "Controllability in partial and uncertain environments," in *ACSD 2014*. IEEE, 2014, pp. 52–61.

[38] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. New York, NY, USA: Cambridge University Press, 2010.

[39] J. Heinz, "String extension learning," in *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ser. ACL '10. Stroudsburg, PA, USA: Association for Computational Linguistics, 2010, pp. 897–906. [Online]. Available: http://dl.acm.org/citation.cfm?id=1858681.1858773

[40] K. Chatterjee, T. A. Henzinger, and B. Jobstmann, "Environment assumptions for synthesis," in *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, ser. Lecture Notes in Computer Science, F. van Breugel and M. Chechik, Eds., vol. 5201. Springer, 2008, pp. 147–161. [Online]. Available: https://doi.org/10.1007/978-3-540-85361-9_14

[41] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011)*, 2011, pp. 43–50.

[42] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.