A

Synthesising Non-Anomalous Event-Based Controllers for Liveness Goals

D'IPPOLITO R. NICOLÁS, Imperial College London
BRABERMAN VICTOR, Universidad de Buenos Aires
PITERMAN NIR, Leicester University
UCHITEL SEBASTIÁN, Universidad de Buenos Aires, Imperial College London

We present SGR(1), a novel synthesis technique and methodological guidelines for automatically constructing event-based behaviour models. Our approach works for an expressive subset of liveness properties, distinguishes between controlled and monitored actions, and differentiates system goals from environment assumptions. We show that assumptions must be modelled carefully in order to avoid synthesising anomalous behaviour models. We characterise non-anomalous models and propose assumption compatibility, a sufficient condition, as a methodological guideline.

Categories and Subject Descriptors: D.2 [**Software Engineering**]: Software Engineering

General Terms: Design, Algorithms

Additional Key Words and Phrases: controller synthesis, behavioural modelling

ACM Reference Format:
D'Ippolito, N., Braberman, V., Piterman, N., and Uchitel, S. 2012. Synthesising Non-Anomalous Event-Based Controllers for Liveness Goals. ACM Trans. Embedd. Comput. Syst. V, N, Article A (January YYYY), 36 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Automated construction of event-based operational models of intended system behaviour has been extensively studied in the software engineering community for some time. Synthesis of such models from scenario-based specifications (e.g. [Uchitel et al. 2009; Damas et al. 2006; Bontemps et al. 2004]) allows integrating a fragmented, example-based specification into a model which can be analysed via model checking, simulation, animation and inspection, the latter aided by automated slicing and abstraction techniques. Synthesis from formal declarative specification (e.g. temporal logics) has also been studied with the aim of providing an operational model on which to further support requirements elicitation and analysis [Letier et al. 2008; Kazhamiakin et al. 2004].

This article significantly revises and extends approach presented at FSE 2010 in Santa Fe, New Mexico ; the relationship with that previous work is discussed in the related work section of this article.
Author's addresses: N. D'Ippolito, Department of Computing, Imperial College, 180 Queen's Gate, London, SW7 2RH, UK; email: srdipi@doc.ic.ac.uk; V. Braberman, Departamento de Computación, Facultad de Ciencias Exactas y Naturales Universidad de Buenos Aires Pabellon 1, Ciudad Universitaria. C1428EHA; email: vbraber@dc.uba.ar; N. Piterman, Department of Computer Science, University of Leicester, University Road, Leicester, LE1 7RH. He can be reached at nir.piterman@leicester.ac.uk; S. Uchitel, Department of Computing, Imperial College, 180 Queen's Gate, London, SW7 2RH, UK; su2@doc.ic.ac.uk.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

Behaviour model synthesis is also used to automatically construct plans that are then straightforwardly enacted by some software component. For instance, synthesis of glue code and component adaptors has been studied in order to achieve safe composition at the architecture level [Autili et al. 2004; Inverardi and Tivoli 2007], and in particular in service oriented architectures [Bertolino et al. 2009].

In the domain of self-adaptive systems [Huang et al. 2004] there has also been an increasing interest in behaviour model synthesis as such systems must be capable of designing at run-time adaptation strategies. Hence, they rely heavily on automated synthesis of behaviour models that will guarantee the satisfaction of requirements under the constraints enforced by the environment and the capabilities offered by the self-adaptive system [Kramer and Magee 2007; Gat et al. 1997; Dalpiaz et al. 2009].

A limitation that existing behaviour model synthesis techniques have is that they are restricted to safety properties and do not support liveness. Hence, synthesis can be posed as a backward error propagation [Russell and Norvig 1995] variant where a behaviour model is pruned by disabling controllable actions that can lead to undesirable states.

In many domains, and particularly in the realm of reactive systems [Manna and Pnueli 1992], liveness requirements can be of importance and having synthesis techniques capable of dealing with them is desirable. However, very few approaches to behaviour model synthesis that support liveness have been proposed, notably [Giunchiglia and Traverso 2000; Bertoli et al. 2001; Sykes et al. 2007; Heaven et al. 2009] all of which have been applied in self-adaptive systems. The problem with these approaches is that the distinction between controlled and monitored actions [Parnas and Madey 1995], and between descriptive and prescriptive behaviour [Jackson 1995b] is not made explicit. As a consequence, the behaviour models they synthesise in order to enact self-adaptation, may not be realisable by the self-adaptive system or unexpected results may be obtained when the self-adaptive system interacts with its environment due to non-valid implicit assumptions.

Making assumptions explicit is crucial, and even more so with liveness system goals. Jackson [Jackson 1995b], and others (e.g., [van Lamsweerde and Letier 2000; Lamsweerde 2001; Parnas and Madey 1995]) have argued the importance of distinguishing between descriptive and prescriptive assertions and, more specifically, between software requirements (prescriptive statements to be achieved by the machine), system goals (prescriptive statements to be achieved by the machine and its environment) and environment assumptions (descriptive statements guaranteed or assumed to be guaranteed by the environment).

Environment assumptions play a key role in the validation process. Many system failures are due to invalid assumptions, many times related to an over-idealisation of the environments behaviour. In other words, statements regarding environment behaviour that are not realistic are used to demonstrate the correctness of the requirements with respect to the goals. However, given that the assumptions are invalid, the goals are not achieved. Explicit assumption modelling not only better supports validation but also makes explicit when system goals are guaranteed to be achieved, helping to set more realistic expectations.

Assumptions, and their relation with the synthesis problem has been studied recently [Chechik et al. 2007; Chatterjee et al. 2008]. When dealing with liveness, assumptions play an even more prominent role: Typically, reasoning about liveness in behaviour models is performed under specific assumptions which correspond to liveness properties themselves. For instance, it is common to reason under some general notion of fairness or some domain specific property regarding the responsiveness of the environment to certain stimuli. Given the central role that liveness assumptions have for reasoning about liveness requirements, the use of approaches to synthesis [Bertoli

et al. 2001; Sykes et al. 2007; Heaven et al. 2009] that leave such assumptions implicit and do not allow for user tailored liveness assumptions entail some important risks and limitations for users.

On the other hand, techniques that support explicit liveness assumptions such as [Piterman et al. 2006] do not provide the required methodological support to verify that the environment assumptions are indeed guaranteed by the environment.

The main contribution of this paper is a technique and methodological guidelines for synthesising event-based behaviour models. Our approach works for an expressive subset of liveness properties, distinguishes between controlled and monitored actions, and differentiates system goals from environment assumptions. Indeed, it is the assumptions that must be modelled carefully in order to avoid synthesising anomalous behaviour models. We propose the notion of assumption compatibility as a guideline and show that it guarantees non-anomalous models.

The synthesis technique proposed this paper adapts and extends recent advances [Piterman et al. 2006] in synthesis of controllers for discrete event systems [Ramadge and Wonham 1989]. We adapt GR(1) [Piterman et al. 2006] to work in the context of event-based specifications using LTS semantics, parallel composition and to support safety properties as part of the specification. From a descriptive specification of the environment in the form of an LTS and a set of controllable actions, the synthesis procedure constructs a behaviour model that when composed with the environment satisfies a given FLTL [Giannakopoulou and Magee 2003] formula of the form $\Box\, I \wedge (\bigwedge_{i=1}^{n} \Box \diamondsuit A_i \rightarrow \bigwedge_{j=1}^{m} \Box \diamondsuit G_j)$ where $\Box\, I$ is a safety system goal, $\Box \diamondsuit A_i$ represents a liveness assumption on the behaviour of the environment, $\Box \diamondsuit G_j$ models a liveness goal for the system and A_i and G_j are non-temporal fluent expressions [Giannakopoulou and Magee 2003], while I is a system safety goal expressed as a Fluent Linear Temporal Logic formula [Giannakopoulou and Magee 2003].

We extend work on synthesis of controllers for discrete event systems [Ramadge and Wonham 1989] with two formal definitions of non-anomalous controllers (best effort and assumption preserving) that rule out behaviour models which attempt to satisfy their goals by preventing the environment in achieving its assumptions. We also propose assumption compatibility, a sufficient condition for avoiding anomalous behaviour models. Interestingly, and perhaps not surprisingly, the condition corresponds to following the methodological and theoretical guidelines dictated by the goal-oriented requirements engineering notion of realisability [Letier and van Lamsweerde 2002].

More specifically, technical contributions of this paper include *(i)* the presentation of the *event-based control problem* which gives a high level description of a certain kind of controller synthesis problems which aims to work under a theoretical framework adequate for event-based models; *(ii)* the grounding of the event-based control problem for Labelled Transitions Systems and parallel composition in the definition of the *LTS control problem*; *(iii)* the definition of a restricted LTS control problem, named *SGR(1) LTS* that supports safety and GR(1)-like properties, and provide a polynomial time solution which builds, from a theoretical perspective on GR(1) games and from an implementation perspective on *(iv)* a rank-based [Jurdziński 2000] algorithm which is suitable for explicit state space representation; *(v)* characterisation of non-anomalous controllers (best effort and assumption preserving) ;*(vi)* a sufficient condition, i.e. assumption compatibility, for an event-based setting to guarantee correctness of the synthesis procedure and to avoid anomalous controllers; *(vii)* evaluation through several case studies adapted and extended from the literature.

This paper is organised as follows. In Section 2 we present our running example and provide an overview of the approach from a black box perspective. We provide the necessary background in Section 3 to then present the LTS control problems in Section 4

where we also discuss anomalous controllers and links to the notion of realisability in requirements engineering. We show how SGR(1) LTS control can be solved in Section 5. Then, in Section 6, we present some case studies we ran to validate both, the applicability of our approach and the implementation we provide as an extension of the MTSA tool set [D'Ippolito et al. 2008]. We finish with discussion, related work and conclusions.

2. OVERVIEW

In this section we provide a black-box overview of our approach. Technical details are provided in the next sections.

Consider the following variation of the Production Cell case study [Lewerentz and Lindner 1995]: A factory manufactures several kinds of products each of which requires a production process which involves different tools applied in a specified order. The factory production system is expected to adapt its production process depending on a number of factors such as the available tools (which is subject to change for instance when a tool breaks or a new instance of an existing tool type is introduced), the specification of how to process each product type (which can change because the production requirements for a product type changes), and other constraints (for example, an energy consumption requirement that constrains the concurrent use of certain tools).

Given its potential for concurrent processing, the production should be scheduled in such a way that no product type is indefinitely postponed.

In addition to the tools, the factory has an in tray, an out tray and a robot arm. The robot arm is used to move products to and from tools and trays. Raw products arrive on the in tray, the robot arm must process them according to their specification and place the finished products on the out tray. The trays can hold products of any kind simultaneously.

To simplify the presentation, assume that the factory must produce two types of products, namely A and B, with three different tools: an oven, a drill and a press. Products of type A require using the oven, then the drill and finally the press, while products of type B are processed in the following order: drill, press, oven. In addition, there is a constraint on concurrent use of tools: the drill and the press cannot be used simultaneously. Finally, a liveness condition on the production of products of type A and B is also required, that is, the production of one kind of product cannot postpone indefinitely the production of products of the other kind.

We now describe how these requirements can be specified in our approach and comment on the production strategy automatically generated by our controller synthesis algorithm.

The environment model is the result of the parallel composition of LTSs modelling the robot arm, the tools, and the products being processed.

In Figure 1 we show a behaviour model, describing the drill tool: Any product (i.e. id from 0 to Max), can be *put* into the drill tool by the robot arm ($put.drill[id : 0..Max]$) and, subsequently, that product is processed ($drill.process[id]$) by the drill and can then be taken from the drill by the robot arm ($get.drill[id]$).

In Figure 2 we show a model that describes how raw products can be processed. A product is $idle$ until it appears in the in tray ($[id].inTray$), then it is picked up by the robot arm ($[id].getInTray$), subsequently, it can be freely placed and picked up from any tool (resp. $put.[t : Tools][id]$ and $get.[t : Tools][id]$) until the product processing is finished and the product is placed in the out tray ($[id].putOutTray$). For simplicity, we model that an instance of a product can be reprocessed, hence, once put on the out tray, the product model is at the initial state again. Note that id represents a particular product and $Tools$ the available tool set.

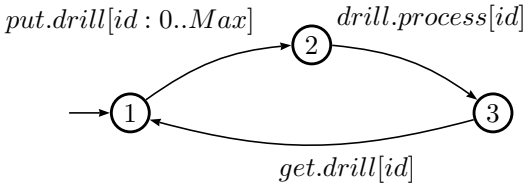


Fig. 1. LTS Model for the Drill.

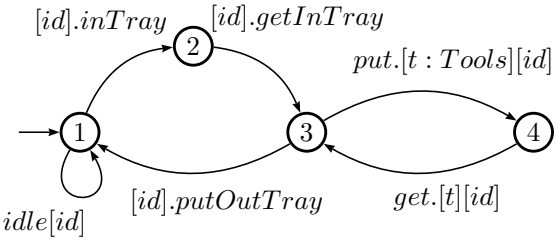


Fig. 2. Products.

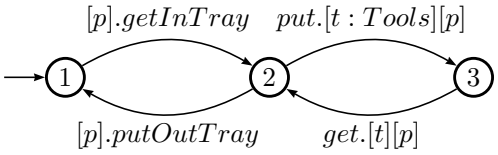


Fig. 3. Robot arm.

We do not include in the models of the products the requirements related to the order in which tools must be applied. This is because, as proposed in [Jackson 1995b], we avoid mixing the description of how the environment behaves with the prescription stating how the environment should behave once the controller is in place.

The model describing the robot arm (Figure 3) shows how the arm can pickup any product from any position (in tray, out tray, and tools) and then place that same product in another position. It can only hold one product at a time. To simplify we assume that the in and out trays are repositories of unbounded size and that the in tray does not enforce an ordering of products.

The environment model can be built as the parallel composition of a model for each tool, a model for the robot arm and a model for each product: $(PRODUCT_A[1] \| ... \| PRODUCT_A[MAX] \| PRODUCT_B[1] \| ... \| PRODUCT_B[MAX] \| DRILL \| OVEN \| PRESS \| ARM)$. The LTS for this composition is too big to be shown, it can be constructed using the modified MTSA tool and data available at [D'Ippolito et al. 2008].

What remains now is to define the set of actions that the controller-to-be can control and the specification that it must satisfy when composed with the environment.

The controlled actions must be a subset of the actions of the environment model and we define them to be the actions of the robot arm. In other words, we aim to build a controller that restricts the behaviour of the arm so that the way the arm moves the products satisfies the production requirements.

The goals for the controller consist of a safety and a liveness part. The safety part is twofold. On one hand, the order in which tools will process raw products is encoded with a model describing the expected processing order for each type of product. In Fig-

ure 4 we show how to model the processing requirements for products of type A, a temporal logic representation of such requirement is also possible (and can be constructed automatically from Figure 4) but more cumbersome. We omit it here but assume that the FLTL formula ϱ is obtained by transforming the LTS in Figure 4 to FLTL. Hence, ϱ is an FLTL formula that captures the requirements for products of type A and B.

On the other hand, the drill and press cannot be used simultaneously. This can be easily encoded with the following temporal logic property: $\psi = \Box(\neg\exists x, y \in Products \cdot Processing(drill, x) \wedge Processing(press, y))$ where $\Box$ means "always in the future" and $Processing(t, p)$ is a predicate which is true when tool t is processing product p. Thus, the safety goal for the system is $I = \Box(\varrho \wedge \psi)$.

The liveness prescription for the controller must capture the requirement of not indefinitely postponing the production of any product type. Such requirement can be formalised in temporal logic as follows:
$G = \bigwedge_{t\in\{A,B\}} \Box \Diamond(\bigvee_{id=0}^{M} AddedToOutTray(t, id))$ where M is the maximum amount of products to process and $AddedToOutTray(t, i)$ is true if the product has just been added to the out tray.

If we attempt to build a controller for the arm such that it guarantees $I \wedge G$ when composed with the model of the environment, our approach will indicate that such controller is not possible. This is true, as there is no guarantee of producing an infinite number of products of type A and of type B if the environment does not guarantee that it will provide the raw products to be processed.

Consequently, we must assume that the environment will produce an infinite number of raw products of type A and B:
$As = \bigwedge_{t\in\{A,B\},0\leq id\leq M}(\Box \Diamond AddedToInTray(t, id))$ where given a product with id equal to i and of type t, $AddedToInTray(t, i)$ is true if the product has just been added to the in tray.

If we attempt to build a controller that guarantees $I \wedge (As \rightarrow G)$ our approach successfully builds one. In other words, we will obtain a controller that guarantees when composed with its environment that the products are processed by applying tools in the correct order (ϱ), that the drill and press are not used simultaneously (ψ) and that if the environment provides infinitely many raw products of both types (As) both types of products will be produced (G).

It is interesting to note that a controller for the robot arm that satisfies the specification above when composed with the model of the environment cannot be produced by simply pruning the environment model (as behaviour model synthesis techniques for safety properties do). This is because, in order to fulfill the liveness part of the specification, a controller must "remember" if it has been postponing one type of product for too long. Say products of type A have been postponed for too long, the controller must stop processing the other component type, B, giving way to the production of A products. How much the controller waits before switching type could vary from one

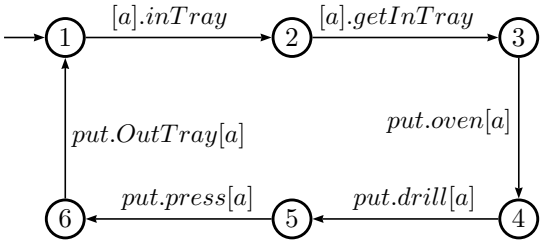


Fig. 4. Production process for products of type A.

controller to another, but all controllers must have some sort of memory in order to achieve the liveness condition. This memory is not encoded in the state space of the environment and hence a controller cannot be achieved through its pruning.

In Section 5 we describe the procedure for synthesising behaviour models that satisfy the specification described above, and hence capable of, among other things, identifying the model's need for memorising specific aspects of the system behaviour in order to satisfy liveness properties.

3. BACKGROUND

In this section we discuss the World-Machine model in addition to defining and fixing notation for Labelled Transition Systems, Fluent Linear Temporal Logic and Two-player Games.

3.1. The World and the Machine

We begin by providing an overview of the relevant requirements engineering notions. In particular, we present the requirements engineering view of Zave and Jackson [Jackson 1995a; 1995b; Zave and Jackson 1997] and of Letier and Van Lamsweerde [van Lamsweerde and Letier 2000; Lamsweerde 2001]. Both views agree that distinguishing between the problem *world* and the *machine* solution is central to understanding whether the machine correctly solves the problem in question. Indeed, the effect of the machine on the world and the assumptions we make about this world are central to the requirements engineering process. The problem *world* defines a part of the real world that we want to improve by constructing a machine solution. Typically, it embodies some components that interact following known rules and processes. For instance, a drill tool, a robot arm and rules for processing products that enter a production cell (see Figure 5). On the other hand, the *machine* solution is expected to solve the problem. The example in Figure 5 shows that the production cell should process products when they are available on the $InTray$. Consequently, the robot arm has to pick up products from the in tray if they are ready to be processed. In other words, the *machine* abstracts what is needed to be done in order to solve the problem. Finally, the shared phenomena is a portion of the problem world and the machine solution that is shared among them. Hence, it defines the interface through which the machine interacts with the world, represented as the intersection of the two sets in Figure 5. The *machine* is referred to in the context of synthesis as the *controller*, we shall either term depending on the context. Following [Parnas and Madey 1995], we may refer to the *problem world* as the *environment*.

Statements describing phenomena of both the problem world and the machine solution may differ in scope and mood [Jackson 1995a; Parnas and Madey 1995]. Statements may have indicative or optative mood. In [van Lamsweerde 2009], statements describing the system are characterised as *descriptive* and *prescriptive*. *Descriptive* statements represent properties about the system that hold independently of how the system behaves. *Descriptive* statements are in indicative mood. *Prescriptive* statements state desirable properties which may hold or not. Indeed, *prescriptive* statements must be enforced by system components. Naturally, *prescriptive* statements may be changed, strengthened/weakened or even removed while *descriptive* cannot.

As mentioned above, statements may vary in scope. Both prescriptive and descriptive statements may refer to phenomena of the machine that is not shared with the world. Other statements may refer to phenomena shared by the machine and the world. More precisely, a *Domain property* is a descriptive statement about the problem world. It must hold regardless on how the system behaves. In this work we call *Environment Model*, the set of *domain properties* for a particular problem. An *Environmental Assumption* is a statement that may not hold and must be satisfied by

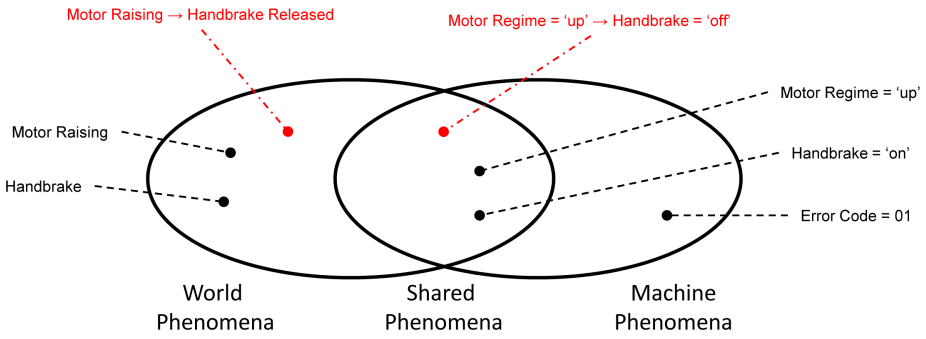


Fig. 5. World and Machine Phenomena.

the environment. A *Software Requirement*, or *Requirement* for short, is a prescriptive statement to be enforced by the machine regardless of how the problem world behaves and must be formulated in terms of the phenomena shared between the machine and the problem world. A *System Goal*, or *Goal* for short, is a prescriptive statement to be enforced by the machine. Some collaboration with the environment might be needed.

In this paper we specify descriptive and prescriptive statements of the world and machine using Fluent Linear Temporal Logic and Labelled Transitions Systems which we recall below.

3.2. Transition Systems

We describe and fix notation for labelled transition systems (LTSs) [Keller 1976], which are widely used for modelling and analysing the behaviour of concurrent and distributed systems. An LTS is a state transition system where transitions are labelled with actions. The set of actions of an LTS is called its communicating alphabet and constitutes the interactions that the modelled system can have with its environment. Model for the drill in the previous section, i.e. Figure 1, is an example of an LTS. Recall that states with an incoming transition with no source state represent initial states.

Definition 3.1. *(Labelled Transition Systems [Keller 1976])* Let $States$ be a universal set of states, Act be the universal set of observable action labels and $Act_\tau = Act \cup \{\tau\}$. A *Labelled Transition System* (LTS) is a tuple $P = (S, L, \Delta, s_0)$, where $S \subseteq States$ is a finite set of states, $L \subseteq Act_\tau$, $\Delta \subseteq (S \times L \times S)$ is a transition relation, and $s_0 \in S$ is the initial state. We use $\alpha P = L \setminus \{\tau\}$ to denote the *communicating alphabet* of P. We denote $\Delta(s) = \{s' \mid (s, a, s') \in \Delta\}$ and $traces(P)$ the set of traces $t = s, \ell, s', \ell', \ldots$ of P. We say an LTS is deterministic if (s, ℓ, s') and (s, ℓ, s'') are in Δ implies $s' = s''$.

Definition 3.2. *(Parallel Composition)* Let $M = (S_M, L_M, \Delta_M, s_{0M})$ and $N = (S_N, L_N, \Delta_N, s_{0N})$ be LTSs. *Parallel composition* $\|$ is a symmetric operator such that $M\|N$ is the LTS $P = (S_M \times S_N, L_M \cup L_N, \Delta, (s_{0M}, s_{0N}))$, where Δ is the smallest relation that satisfy the rules below, where $\ell \in L_M \cup L_N$:

$$\frac{M \xrightarrow{\ell} M'}{M\|N \xrightarrow{\ell} M'\|N}\ \ell \notin L_N \qquad \frac{M \xrightarrow{\ell} M',\ N \xrightarrow{\ell} N'}{M\|N \xrightarrow{\ell} M'\|N'}\ \ell \in L_M \cup L_N \qquad \frac{N \xrightarrow{\ell} N'}{M\|N \xrightarrow{\ell} M\|N'}\ \ell \notin L_M$$

The following definition is based on that of *Interface Automata* and *Legal Environment* presented in [de Alfaro and Henzinger 2001].

Definition 3.3. *(LTS Legal Environment)* Given $M = (S_M, L_M, \Delta_M, s_{M_0})$ and $P = (S_P, L_P, \Delta_P, s_{P_0})$ LTSs, where $L_M = L_{M_c} \sqcup L_{M_u}$ and $L_P = L_{P_c} \sqcup L_{P_u}$. We say that M is an ***LTS legal environment*** for P with controlled actions L_{M_c}, if the interface automaton $M' = \langle S_M, \{s_{M_0}\}, L_{M_u}, L_{M_c}, \emptyset, \Delta_M \rangle$ is a *legal environment* for the interface automaton $P' = \langle S_P, \{s_{P_0}\}, L_{P_u}, L_{P_c}, \emptyset, \Delta_P \rangle$.

3.3. Fluent Linear Temporal Logic

Linear temporal logics (LTL) are widely used to describe behaviour requirements [Giannakopoulou and Magee 2003; van Lamsweerde and Letier 2000; Letier and van Lamsweerde 2002; Kazhamiakin et al. 2004]. The motivation for choosing an LTL of fluents is that it provides a uniform framework for specifying state-based temporal properties in event-based models [Giannakopoulou and Magee 2003]. FLTL [Giannakopoulou and Magee 2003] is a linear-time temporal logic for reasoning about fluents. A *fluent* $\boldsymbol{Fl}$ is defined by a pair of sets and a boolean value: $\boldsymbol{Fl} = \langle I_{\boldsymbol{Fl}}, T_{\boldsymbol{Fl}}, \mathit{Init}_{\boldsymbol{Fl}} \rangle$. $I_{\boldsymbol{Fl}}$ is the set of initiating actions and $T_{\boldsymbol{Fl}}$ is the set of terminating actions. A fluent may be initially *true* or *false* as indicated by the $\mathit{Init}_{\boldsymbol{Fl}}$. Every action $\ell \in Act$ induces a fluent, namely $\dot{\ell} = \langle \ell, Act \setminus \{\ell\}, \bot \rangle$. Finally, the alphabet of a fluent is the union of its terminating and initiating actions.

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

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

where $\boldsymbol{Fl} \in \mathcal{F}$. As usual we introduce $\wedge$, $\Diamond$ (eventually), and $\Box$ (always) as syntactic sugar.

Let Π be the set of infinite traces over $\boldsymbol{Act}$. For $\pi \in \Pi$, we write π^i for the suffix of π starting at $\ell_i \in \boldsymbol{Act}$. π^i satisfies a fluent $\boldsymbol{Fl}$, denoted $\pi^i \models \boldsymbol{Fl}$, if and only if one of the following conditions holds:

- $\mathit{Init}_{\boldsymbol{Fl}} \wedge (\forall j \in \mathbb{N} \cdot 0 \leq j \leq i \rightarrow \ell_j \notin T_{\boldsymbol{Fl}})$
- $\exists j \in \mathbb{N} \cdot (j \leq i \wedge \ell_j \in I_f) \wedge (\forall k \in \mathbb{N} \cdot j < k \leq i \ \rightarrow \ell_k \notin T_{\boldsymbol{Fl}})$

3.4. Controller Synthesis

We consider now the problem of FLTL control over LTS. For that we distinguish between controllable and uncontrollable actions in an LTS (introduced formally later). Intuitively, of the actions possible in a state, the controller can choose to disable controllable actions. However, it cannot disable all possible actions as this would lead to a deadlock. This leads naturally to the concept of a game, where one player (the controller) chooses which actions to enable and the other player (environment) chooses which actions to follow. Formally, we have the following.

Definition 3.4. *(Two-player Game)* A *Two-player Game* (Game) is $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi)$, where S is a finite set of states, $\Gamma^-, \Gamma^+ \subseteq S \times S$ are transition relations of uncontrollable and controllable transitions, respectively, $s_{g_0} \in S$ is the initial state, and $\varphi \subseteq S^\omega$ is a winning condition. We denote $\Gamma^-(s) = \{s' \mid (s, s') \in \Gamma^-\}$ and similarly for Γ^+. A state s is *uncontrollable* if $\Gamma^-(s) \neq \emptyset$ and *controllable* otherwise. A *play* on G is a sequence $p = s_{g_0}, s_{g_1}, \ldots$. A play p ending in s_{g_n} is extended by the controller choosing a subset $\gamma \subseteq \Gamma^+(s_{g_n})$. Then, the environment chooses a state $s_{g_{n+1}} \in \gamma \cup \Gamma^-(s_{g_n})$ and adds $s_{g_{n+1}}$ to p.

Notice that if in a controllable state γ is empty the choice of controller may lead to a deadlock. This is prohibited later by defining this as a losing choice for the controller.

From an uncontrollable state the controller may decide to disable all controllable actions. The choices of the controller are formalised in the form of a strategy. This is the policy that the controller applies. In general, the strategy may depend on the history. This is reflected in the strategy depending on a memory value in the domain Ω and updating this value according to the evolvement of the play.

Recall that this game is different from the one defined in [Piterman et al. 2006]. Piterman et al. define a game in which the environment chooses its next valuation and only then, the controller gets to choose what to do next.

Definition 3.5. *(Strategy with memory)* A *strategy with memory* Ω for the controller is a pair of functions (σ, u), where Ω is some memory domain with designated start value ω_0, $\sigma : \Omega \times S \to 2^S$ such that $\sigma(\omega, s) \subseteq \Gamma^+(s)$ and $u : \Omega \times S \to \Omega$.

Intuitively, σ tells controller which states to enable as possible successors and u tells controller how to update its memory. If Ω is finite, we say that the strategy uses finite memory.

Definition 3.6. *(Consistency and Winning Strategy)* A finite or infinite play $p = s_0, s_1, \ldots$ is *consistent* with (σ, u) if for every n we have $s_{n+1} \in \sigma(\omega_n, s_n)$, where $\omega_{i+1} = u(\omega_i, s_{i+1})$ for all $i \geq 0$. A strategy (σ, u) for controller is *winning* if every maximal play consistent with (σ, u) is infinite and in φ. We say that controller *wins* the game G if it has a winning strategy.

As the controller is defined as losing on all finite plays it follows that it cannot disable all controllable actions from a controllable state. We refer to checking whether controller wins a game G as *solving* the game G. The *controller synthesis* problem is to produce a winning strategy for controller. If such winning strategy for controller exists we say that the control problem is *realisable* [Ramadge and Wonham 1989; Maler et al. 1995]. It is well known that if controller wins a game G and φ is ω-regular it can win using a finite memory strategy [Pnueli and Rosner 1989]. We now define the class of winning conditions φ that is of our interest.

Definition 3.7. *(Generalised Reactivity(1) [Piterman et al. 2006])* Given an infinite sequence of states p, let $inf(p)$ denote the states that occur infinitely often in p. Let $\phi_1, \ldots, \phi_n$ and $\gamma_1, \ldots, \gamma_m$ be subsets of S. Let $gr((\phi_1, \ldots, \phi_n), (\gamma_1, \ldots, \gamma_m))$ denote the set of infinite sequences p such that either for some i we have $inf(p) \cap \phi_i = \emptyset$ or for all j we have $inf(p) \cap \gamma_j \neq \emptyset$. A GR(1) game is a game where the winning condition φ is $gr((\phi_1, \ldots, \phi_n), (\gamma_1, \ldots, \gamma_m))$.

We refer to games (as defined in Definition 3.4) with Piterman's GR(1) winning conditions (Recalled in Definition 3.7) as GR(1) games.

THEOREM 3.8. *Given a game $G = (S, \Gamma^-, \Gamma^+, s_{g_0}, \varphi)$ the complexity of deciding whether G is winning and computing a winning strategy for controller is as follows.*

— *The complexity is 2EXPTIME-complete for general φ [Pnueli and Rosner 1989].*
— *The complexity is $O(nm|S|(|\Gamma^- \cup \Gamma^+|))$ for φ a generalised reactivity(1) formula [Kesten et al. 2005; Juvekar and Piterman 2006].*

4. EVENT BASED CONTROL SYNTHESIS

4.1. Control Problems

We now present a high level description of an event-based control problem following the world-machine model [Jackson 1995b]. We distinguish between software requirements, system goals and environment assumptions. We then define the LTS control problem which grounds the event-based control problem by fixing a specific formal

specification framework: Labelled Transition Systems and the Linear Temporal Logic of Fluents. Finally, given the computational complexity of the general LTS control problem, we define SGR(1) LTS Control, a restricted LTS control problem for expressive subset of temporal properties that includes liveness and allows for a polynomial solution. In the next section, we show how such polynomial solution can be achieved.

The problem of control synthesis is to automatically produce a machine that restricts the occurrence of events it controls based on its observation of the events that have occurred. When deployed in a suitable environment such a machine will ensure the satisfaction of a given set of system goals. Satisfaction of these goals depends on the satisfaction of the assumptions by the environment. In other words, we are given a specification of an environment, assumptions, system goals, and a set of controllable actions.

A solution for the *Event-Based control problem* is to find a machine whose concurrent behaviour with an environment that satisfies the assumptions satisfies the goals.

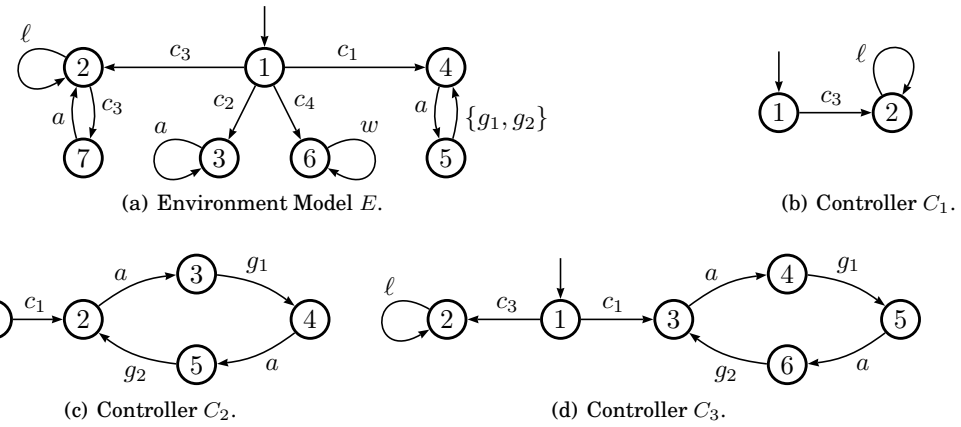
We adopt labelled transition systems (LTS) and parallel composition in the style of CSP [Hoare 1978] as the formal basis for modelling the environment and for representing the synthesised controller, and FLTL, with its corresponding satisfiability notion, as a declarative specification language to describe both environment assumptions and system goals.

We ground the problem of control synthesis in event-based models as follows: Given an LTS that describes the behaviour of the environment, a set of controllable actions, a set of FLTL formulas as the environment assumptions and a set of FLTL formulas as the system goals, the LTS control problem is to find an LTS that only restricts the occurrence of controllable actions and guarantees that the parallel composition between the environment and the LTS is deadlock free and that if the environment assumptions are satisfied then the system goals will be satisfied too.

Definition 4.1. *(LTS Control)* Given a specification for an environment in the form of an LTS E, a set of controllable actions A_c, and a set H of pairs (As_i, G_i) where As_i and G_i are FLTL formulas specifying assumptions and goals respectively, the solution for the LTS control problem $\mathcal{E} = \langle E, H, A_c \rangle$ is to find an LTS M such that M with controlled actions A_c and uncontrolled $\overline{A_c}$ is a legal environment for E, $E||M$ is deadlock free, and for every pair $(As_i, G_i) \in H$ and for every trace π in $M||E$ the following holds: if $\pi \models As_i$ then $\pi \models G_i$.

The problem with using FLTL as the specification language for assumptions and goals is that, just like in traditional (i.e. state-based) controller synthesis, the synthesis problem is 2EXPTIME complete [Pnueli and Rosner 1989]. Nevertheless, restrictions on the form of the goal and assumptions specification have been studied and found to be solvable in polynomial time. For example, goal specifications consisting uniquely of safety requirements can be solved in polynomial time, and so can particular styles of liveness properties such as [Asarin et al. 1998] and GR(1) under the assumption of full observability. The latter can be seen as an extension of [Asarin et al. 1998] to a more expressive liveness fragment of LTL.

We now define the SGR(1) control problem which is computable in polynomial time. It builds on the GR(1) and safety control problems but is set in the context of event-based modelling. We require the model of the environment E to be a deterministic LTS to ensure that the controller will have full observability of the environment's state. We require H to be $\{(\emptyset, I), (As, G)\}$, where I is a safety invariant of the form $\Box\, \rho$, the assumptions As are a conjunction of FLTL sub-formulas of the form $\Box \diamondsuit \phi$, the goal G a conjunction of FLTL sub-formulas of the form $\Box \diamondsuit \gamma$, and ϕ, ρ and γ are Boolean combinations of fluents.



(a) Environment Model E.

(b) Controller C_1.

(c) Controller C_2.

(d) Controller C_3.

Fig. 6.

Definition 4.2. *(SGR(1) LTS Control)* An LTS control problem $\mathcal{E} = \langle E, H, A_c \rangle$ is SGR(1) if E is deterministic, and $H = \{(\emptyset, I), (As, G)\}$, where $I = \Box\, \rho$, $As = \bigwedge_{i=1}^{n} \Box \Diamond \phi_i$, $G = \bigwedge_{j=1}^{m} \Box \Diamond \gamma_j$, and ϕ_i, ρ and γ_j are Boolean combinations of fluents.

Consider the SGR(1) LTS control problem $\mathcal{R} = \langle E, H, A_c \rangle$, where E is the LTS in Figure 6(a), $A_c = \{c_1, c_2, c_3, c_4, g_1, g_2\}$, $H = \{(\emptyset, I), (As, G)\}$, $I = \Box\, \neg \dot{w}$, $As = \Box \Diamond \dot{a}$ and $G = \Box \Diamond \dot{g_1} \wedge \Box \Diamond \dot{g_2}$. Recall that for all ℓ in the alphabet of E, the fluent $\dot{\ell}$ is defined as the fluent that becomes true when ℓ occurs and becomes false when any other action occurs.

The LTS C_1, C_2 and C_3 of Figures 6(b) to 6(d) are some of the possible solutions to $\mathcal{R}$: $C_1 \| E$ has no traces satisfying the assumptions As, hence it is not obligated to satisfy G; all traces in $C_2 \| E$ satisfy As and also G; and traces in $C_3 \| E$ either do not satisfy As or satisfy both As and G. We will discuss in the next subsection the differences between these solutions. For now, it is interesting to note that neither C_2 nor C_3 can be obtained only by pruning E. Both models introduce new states which allow the controller to "remember" which is the next goal that must be achieved ($g1$ or $g2$). The automated construction of these "memory" states will be described in detail in section 5.3.

The SGR(1) control problem restricts the form of the environment assumptions and system goals. Thus, a valid concern is the impact of this restriction on expressiveness in practice. A closer look at the family of liveness formulas reveals it is not arbitrary: they are designed to capture a Büchi acceptance condition. More concretely, any liveness property specifiable by a deterministic Büchi automaton can be handled by the proposed approach. The trick is, basically, to compose the Büchi automaton structure with the original plant LTS and then use assumptions and goals to express that their acceptance conditions will/should (respectively) be visited infinitely often. Typical responsiveness assumptions and goals (e.g. $\Box(\phi \rightarrow \Diamond\, \psi)$) could be treated in this way [Piterman et al. 2006]. In the context of LTS and FLTL this kind of assumptions can be handled without explicitly generating the deterministic Büchi automaton. In many cases, this can be done by encoding the responsiveness with fluents and assumptions in GR(1) form. An example of a responsiveness goal that does not fit the syntactic requirements of SGR(1) but could be dealt with by means of this encoding is that if a product is waiting to be processed by the cell (i.e. it has been placed on the in tray and not yet picked up by the arm), then it will eventually be put onto the out tray $\bigwedge_{id=0}^{MAX} \bigwedge_{t \in \{A,B\}} \Box(WaitingForProcessing(t, id) \rightarrow \Diamond\, [t].[id].\dot{put}.OutTray)$ where $WaitingForProcessing(t, id)$ is a fluent initiated by event $[t].[id].inTray$ and ter-

minated by event $[t].[id].getInTray$. In this case the trick to encode the assumption as a GR(1)-like formula is to encode the responsiveness as follows, define a fluent with the initiating action of the antecedent as the initiating action and the initiating action of the consequent as the terminating action. Consequently, we first define the fluent $\mathit{fl} = \langle \{[t].[id].inTray\}, \{[t].[id].put.OutTray\}, \bot \rangle$ and then we include the formula $\Box \Diamond \neg \mathit{fl}$ as an environmental assumption in the GR(1) specification.

4.2. Assumptions and Anomalous Controllers

A valid concern is if there are semantic restrictions for what is called an assumption in a control problem. In other words, can any assertion be provided as an assumption? or the fact that it is deemed assumption implies that it should have specific semantic properties? This question can also be posed for the specific case of SGR(1) LTS control: are further semantic restrictions needed to ensure that the formula $As = \bigwedge_{i=1}^{n} \Box \Diamond \phi_i$ can be interpreted as an assumption on the environment? We now answer this question.

Consider the LTS controller C_1 discussed in the previous section. C_1 solves the SGR(1) control problem $\mathcal{R}$ by simply ensuring that for all trace $\pi \in traces(E \| C_1)$ $\pi \not\models As$. Such a solution, from an engineering perspective is unsatisfactory: C_1 should "play fair" by trying to achieve G when As holds rather than trying to avoid As. In this sense, C_2 and C_3 are more satisfactory. The best effort controller definition provided below formalises this preference by requiring the following: if the controller forces As not to hold after a sequence σ, no other controller that achieves G could have allowed As after σ.

Definition 4.3. *(Best Effort Controller)* Given an SGR(1) LTS control problem $\mathcal{E}$ with assumptions As and an LTS M such that M is a solution for $\mathcal{E}$, we say that M is a *best effort controller* for $\mathcal{E}$ if for all finite traces $\sigma \in traces(E \| M)$ if there is no σ' where $\sigma.\sigma' \in traces(E \| M)$ and $\sigma.\sigma' \models As$ then there is no other solution M' to $\mathcal{E}$ such that $\sigma \in traces(E \| M')$ and there exists σ' such that $\sigma.\sigma' \in traces(E \| M')$ and $\sigma.\sigma' \models As$

Controller C_1 is not a best effort controller as ϵ, the empty trace in $E \| C_1$ cannot be extended in $E \| C_1$ to satisfy As, yet it can be extended by $\sigma' = c_1, 2, a, 3, g_1, 4, a, 5, g_2, \cdots$ in $E \| C_2$ such that $\epsilon.\sigma'$ satisfies $\Box \Diamond As$. On the other hand, given that there are no traces in $E \| C_2$ violating As, C_2 is a *Best Effort* controller for $\mathcal{R}$. C_3 is also a *Best Effort* controller as the only finite trace violating As in C_3 is $\sigma = c_3, \cdots$ and there are no extension of σ satisfying As and G.

Note that controller C_3 also could be argued to be anomalous from an engineering perspective: Although C_3 does play fair when choosing action c_1 to state 3, it can also choose action c_3 to state 2 taking $E \| C_3$ to a state in which assumptions are no longer possible. This can motivate a stronger criterion than Best Effort: the controller should never prevent the environment from achieving its assumptions.

Definition 4.4. *(Assumption Preserving Controller)* Given an SGR(1) LTS control problem $\mathcal{E}$ with assumptions As and an LTS M such that M is a solution for $\mathcal{E}$, we say that M is an *assumption preserving controller* for $\mathcal{E}$ if for all finite traces $\sigma \in traces(E \| M)$ if there is no σ' where $\sigma.\sigma' \in traces(E \| M)$ and $\sigma.\sigma' \models As$ then there does not exist σ' such that $\sigma.\sigma' \in traces(E)$ and $\sigma.\sigma' \models (I \wedge As)$

THEOREM 4.5. *Given an SGR(1) LTS control problem $\mathcal{R}$ and M an LTS controller for $\mathcal{R}$, if M is a* Assumption Preserving *controller then M is a* Best Effort *controller.*

PROOF. If M is not *best effort* there must be a trace $\sigma \in traces(E \| M)$ such that there is no σ' where $\sigma.\sigma' \in traces(E \| M)$, $\sigma.\sigma' \models As$ and then there exists another solution M' to $\mathcal{E}$ such that $\sigma \in traces(E \| M')$ and there exists σ' such that $\sigma.\sigma' \in traces(E \| M')$

and $\sigma.\sigma' \models As$. By definition of LTS parallel composition $\sigma.\sigma' \in traces(E)$, therefore M cannot be *assumption preserving*, which contradicts our hypothesis. □

Theorem 4.5 and the fact that C_1 is not best effort it follows that C_2 is not an *assumption preserving* controller. Although C_3 is best effort, it is not an *assumption preserving* controller because the trace $\sigma = c_3, c_3, a, c_3, \ldots$ in E is a valid extension to $\sigma = c_3, \ldots$ in $C_3 \| E$ which satisfies As while violating G. On the other hand, given that every infinite trace in C_2 satisfies both As and G, C_2 is an *assumption preserving* controller.

Note that the *Best Effort* criterion compares two controllers while *Assumption Preserving* compares the behaviour of the controlled environment against the environment. Consequently, it is easy to see that *Assumption Preserving* and *Best Effort* controllers are related through logical implication. In other words, if a controller is *Assumption Preserving* then it is also *Best Effort*. It could be argued that *Assumption Preserving* is sufficient and *Best Effort* is somehow non desired. However, they are both relevant in different ways. There are situations in which if the goals are to be fulfilled by a controlled environment, the controller must take decisions that, at some point, might forbid the environment to satisfy its assumptions. In such cases, *Assumption Preserving* controllers cannot be achieved while *Best Effort* can.

Now, given an SGR(1) problem it is useful to know whether all solutions of an SGR(1) LTS control problem are assumption preserving or best effort. Interestingly, a sufficient condition for this can be achieved by restricting the relation between the assumptions As and the environment E. The essence of this relation is based on the notion of realisability and the fact that the environment is the agent responsible for achieving the assumptions as introduced in [Letier and van Lamsweerde 2002].

The notion of realisability requires that an agent responsible for an assertion be capable of achieving it based on its controlled actions regardless of what happens with the actions is does not control. In our setting, this notion can be used to formalise a sufficient condition for guaranteeing assumption preserving and best effort controllers.

The condition requires the environment be capable of achieving As regardless of the behaviour of any controller that it might be composed with. This is ensured by checking that for every state in E there is no strategy for the controller to falsify As. This adds no computational complexity to the control problem.

Definition 4.6. *(Assumption compatibility)* Given an SGR(1) LTS control problem $\mathcal{E} = \langle E, H, A_c \rangle$ and $H = \{(\emptyset, I), (As, G)\}$, we say that the As is compatible with E if for every state s in E there is no solution for the SGR(1) LTS control problem $\mathcal{E}_s = \langle E_s, H', A_c \rangle$, where $H' = \{(\emptyset, I), (As, \text{false})\}$ and E_s is the result of changing the initial state of E to s.

Hence, when the assumptions of an SGR(1) LTS control problem are compatible with the environment, it is guaranteed that anomalous controllers (such as those that are not best effort and assumption preserving) will not be produced.

THEOREM 4.7. *Given an SGR(1) LTS control problem $\mathcal{E}$ with assumptions As and environment E, if As is compatible with E then all solutions to $\mathcal{E}$ are best effort and assumption preserving.*

PROOF. Since As is compatible with E then for all M and for every trace $\sigma \in traces(E \| M)$ there exists σ' such that $\sigma.\sigma' \in traces(E \| M)$ and $\sigma.\sigma' \models As$. By vacuity of the antecedent, it follows that M is best effort and assumption preserving controller. □

Note that the running example $\mathcal{R}$ violates Definition 4.6 and hence, has anomalous controllers such as C_1, which is not *Best Effort* nor *Assumption Preserving*, or C_3 which is *Best Effort* but not *Assumption Preserving*.

Also note that the assumptions for the example in Section 2 are not compatible with the environment described in the same section. This is because we modelled the environment so to "reuse" products once they have been processed. In other words, rather than modelling an infinite number of products to be processed (which would lead to an infinite state environment) we modelled that a product, once it has been fully processed becomes available once again to be put as raw on the in tray. As the assumptions require $AddedToInTray(t, i)$ infinitely often, the environment needs the robot to cooperate by processing the products infinitely often. Hence, the environment cannot guarantee the assumptions independently and a solution to the running example could be a robot that does absolutely nothing. A more appropriate assumption, which would guarantee non-anomalous controllers, is one that states that the environment reacts to products being placed in the out tray by eventually placing them back on the in tray: $\Box(AddedToOutTray(t, i) \rightarrow \Diamond\, AddedToInTray(t, i))$. Note that in this case it is also possible to apply the trick presented above.

Our notion of anomalous controllers is tightly coupled to the problem of properties which are trivially satisfied in system model [Beatty and Bryant 1994]. A typical pattern of vacuity [Beer et al. 1997] is one in which the left hand side of an implication is never fulfilled by the system model. A controller that that achieves its goals by falsifying assumptions can be thought of as the cast of the vacuity problem in controller synthesis.

Summarising the latter part of this section, best effort and assumption preserving controllers explain technically the sort of anomalies that might arise if requirement engineering practices such as ensuring realisability of assumptions by the environment are violated.

In the next section we present how to solve SGR(1) LTS problems. The synthesis algorithm we implemented does not require environment-assumption compatibility. However, as explained above, such a condition is desirable.

5. SOLVING SGR(1) CONTROL

In this section we explain how a solution for the SGR(1) control problem can be achieved by building on existing (state-based) controller synthesis techniques, namely GR(1).

The construction of the machine for an SGR(1) LTS control problem has two steps. Firstly, a GR(1) game G is created from the environment model E, the assumptions As, the goals G and the set of controllable actions A_c (Section 5.1). Secondly, a solution (σ, u) to the GR(1) game is used to build a solution M (i.e. an LTS controller) for $\mathcal{E}$ (Section 5.2). We also show that our approach is sound and complete. That is, a solution to the SGR(1) LTS control problem $\mathcal{E}$ exists if and only if a solution to the GR(1) game G exists. Furthermore, the LTS controller M built from (σ, u) is a solution to $\mathcal{E}$.

The reader not interested details of the mapping of SGR(1) into GR(1) can skip directly to Section 5.3 where we comment on the implementation of the synthesis technique and show a controller for a reduced version of the Production Cell case study.

5.1. SGR(1) LTS control to GR(1) games

We convert the SGR(1) LTS control problem into a GR(1) game. Given a SGR(1) LTS control problem $\mathcal{E} = \langle E, H, A_c\rangle$ we construct a GR(1) game $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ such that every state in S_g encodes a state in E and a valuation of all fluents appearing in As and G.

More precisely, consider an SGR(1) LTS control problem $\mathcal{E} = \langle E, H, A_c\rangle$, where, $H = \{(\emptyset, I), (As, G)\}$, $E = (S_e, L, \Delta_e, s_{e_0})$, $As = \bigwedge_{i=1}^{n} \Box \Diamond \phi_i$, $I = \Box\, \rho$ and $G = \bigwedge_{j=1}^{m} \Box \Diamond \gamma_j$.

Let $\mathit{fl} = \{\dot{1}, \ldots, \dot{k}\}$ be the set of fluents used in As and G and $\dot{i} = \langle I_i, T_i, \mathit{Init}_i \rangle$. The game $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ is constructed as follows.

We build S_g from E such that states encode a state in E and truth values for all fluents in φ: Let $S_g = S_e \times \prod_{i=1}^{k} \{\mathit{true}, \mathit{false}\}$. Consider a state $s_g = (s_e, \alpha_1, \ldots, \alpha_k)$. Given fluent fl_i, we say that s_g satisfies fl_i if α_i is *true* and s_g does not satisfy fl_i otherwise. We generalise satisfaction to Boolean combination of fluents in the natural way.

We build transition relations Γ^- and Γ^+ using the following rules. Consider a state $s_g = (s_e, \alpha_1, \ldots, \alpha_k)$. If s_g does not satisfy ρ (i.e., s_g is unsafe) we do not add successors to s_g. Otherwise, for every transition $(s_e, \ell, s'_e) \in \Delta_e$ we include $(s_g, (s'_e, \alpha'_1, \ldots, \alpha'_k))$ in Γ^β, where β is $+$ if $\ell \in A_c$, β is $-$ if $a \notin A_c$ and (1) α'_i is α_i if $\ell \notin I_{\mathit{fl}_i} \cup T_{\mathit{fl}_i}$, (2) α'_i is *true* if $\ell \in I_{\mathit{fl}_i}$ and (3) α'_i is *false* if $\ell \in T_{\mathit{fl}_i}$. The initial state s_{g_0} is $(s_{e_0}, \mathit{initially}_1, \ldots, \mathit{initially}_k)$.

We build the winning condition φ_g, defined to be a set of infinite traces, from AS and G as follows: We abuse notation and denote by ϕ_i the set of states s_g such that s_g satisfies the assumptions ϕ_i and by γ_i the set of states s_g such that s_g satisfies the goal γ_i. Let $\varphi_g \subseteq S_g^\omega$ be the set of sequences that satisfy $gr((\phi_1, \ldots, \phi_n), (\gamma_1, \ldots, \gamma_m))$. It follows that $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ is a GR(1) game.

It can be shown that if there is a solution to a SGR(1) LTS control problem then there is a winning strategy for a controller in the constructed GR(1) game (refer to Theorem 5.1).

Note that the safety part of the specification is not encoded as part of the wining condition φ_g of the GR(1) game, rather it is encoded as a deadlock avoidance problem when constructing Γ^- and Γ^+. Consequently, the winning condition we realise is $\Box\, \rho \wedge (\bigwedge_{i=1}^{n} \Box \Diamond \phi_i \Rightarrow \bigwedge_{j=1}^{m} \Box \Diamond \gamma_j)$

Figure 7(a) shows the transition relations Γ^- and Γ^+ for G_R, the game obtained by applying to $\mathcal{R}$ the procedure described above. Transitions in Γ^- and Γ^+ are marked as γ^- and γ^+ respectively. States are labelled with a state in the original LTS model (i.e. model E in Figure 6(a)) and the set of fluents holding in the state of the LTS model.

5.2. Translating strategies to LTS Controllers

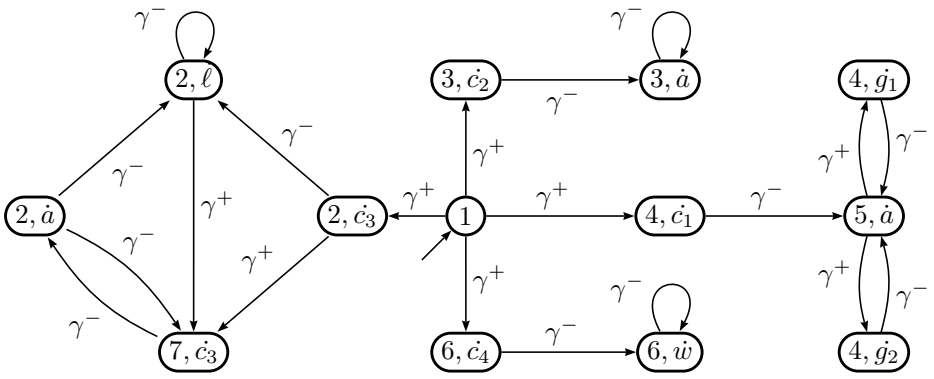
We now show how to extract an LTS controller from a winning strategy for the GR(1) game that was obtained from the SGR(1) LTS control problem as shown in Section 5.1.

Intuitively, the transformation is as follows: given an SGR(1) LTS control problem $\mathcal{E} = \langle E, H, A_c \rangle$, the game $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ obtained from $\mathcal{E}$ and a winning strategy for G, we build $M = (S_M, L, \Delta_M, s_{M_0})$ a solution to $\mathcal{E}$ by encoding in states of S_M a state of S_g and a state of the memory given by the winning strategy.

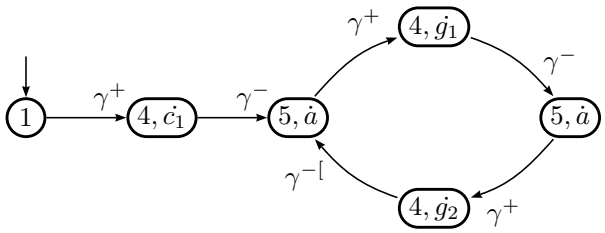
More precisely, let $E = (S_e, L, \Delta_e, s_{e_0})$, $\mathit{fl} = \{\mathit{fl}_1, \ldots, \mathit{fl}_k\}$ the set of fluents appearing in φ, $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ be the GR(1) game constructed from E as explained above, and let $\sigma : \Omega \times S_g \to 2^{S_g}$ and $u : \Omega \times S_g \to \Omega$ be a winning strategy in G. We construct the machine $M = (S_M, L, \Delta_M, s_{M_0})$ as follows.

To build $S_M \subseteq \Omega \times S_g$, consider two states $s_g = (s_e, \alpha_1, \ldots, \alpha_k)$ and $s'_g = (s'_e, \alpha'_1, \ldots, \alpha'_k)$. We say that action ℓ is *possible* from s_g to s'_g if $(s_g, s'_g) \in \Gamma^- \cup \Gamma^+$, there is some action ℓ such that $(s_e, \ell, s'_e) \in \Delta_e$ and for every fluent fl_i either (1) $\ell \notin I_{\mathit{fl}_i} \cup T_{\mathit{fl}_i}$ and $\alpha'_i = \alpha_i$, (2) $\ell \in I_{\mathit{fl}_i}$ and $\alpha'_i = \mathit{true}$, or (3) $\ell \in T_{\mathit{fl}_i}$ and $\alpha'_i = \mathit{false}$.

To build $\Delta_M \subset S_M \times L \times S_M$, consider a transition $(s_g, s'_g) \in \Gamma^-$. By definition of Γ^- there is an action $\ell \notin A_c$ such that ℓ is possible from s_g to s'_g. If $s'_g \in \sigma(\omega, s_g)$ then for every action ℓ such that ℓ is possible from s_g to s'_g we add $((\omega, s_g), \ell, (u(\omega, s_g), s'_g))$ to Δ_M. Similarly, consider a transition $(s_g, s'_g) \in \Gamma^+$. By definition of Γ^+ there is an action



(a) Transition relations for the game G_R.



(b) σ_R: Wining strategy for G_R.

Fig. 7.

$\ell \in A_c$ such that ℓ is possible from s_g to s'_g. If $s'_g \in \sigma(\omega, s_g)$ then for every action ℓ such that ℓ is possible from s_g to s'_g we add $((\omega, s_g), \ell, (u(\omega, s_g), s'_g))$ to Δ_M.

The initial state of M is defined as $s_{M_0} = (\omega_0, s_{g_0})$ where ω_0 is the initial value for the memory domain Ω. This completes the definition of M.

Consider the game G_R and a strategy (σ, u) that tries to fulfill the goals at the same time the environment fulfills its assumptions. That is, a strategy that satisfies $\square \diamondsuit \dot{a}$, $\square \diamondsuit \dot{g}_1$ and $\square \diamondsuit \dot{g}_2$. The only possible solution to such requirements is to have in (σ, u), a cycle visiting $(5, \dot{a})$, $(4, \dot{g}_1)$ and $(4, \dot{g}_2)$ in some order. A strategy satisfying this is shown in Figure 7(b). Note that some memory is needed to distinguish whether state $(4, \dot{g}_1)$ or $(4, \dot{g}_2)$ has to be visited after visiting $(5, \dot{a})$. Finally, in Figure 6(c) we show the LTS controller obtained by applying the conversion shown above to the strategy in Figure 7(b).

In Theorem 5.2 we show that if (σ, u) is winning strategy for a GR(1) game G constructed from a to a SGR(1) LTS control problem $\mathcal{E}$, then the LTS M constructed as explained above is a solution to $\mathcal{E}$. Note that to prove this proposition environment (E) determinism is needed.

THEOREM 5.1. (Completeness) *Let $\mathcal{E}$ be an SGR(1) LTS control problem, and G be a GR(1) game constructed by applying the conversion shown in Section 5.1 to $\mathcal{E}$. If C is a solution for the SGR(1) problem $\mathcal{E}$ then there exists a strategy (σ, u) such that: (σ, u) is winning for G and the LTS controller obtained by applying the translation shown in Section 5.2 to (σ, u) is equivalent to C.*

PROOF. The proof is organised as follows. First, we construct a winning strategy (σ, u). Second, we prove (σ, u) to be winning for G. The strategy is constructed by applying a similar reasoning as if we were applying the inverse of the transfor-

mation shown in Section 5.2 to M. That is, the states of M are used as the memory function for the strategy and given a pair of states $s_g = (s_e, \alpha_1, \ldots, \alpha_k)$ and $s'_g = (s'_e, \alpha'_1, \ldots, \alpha'_k)$ in G. The transition $(s_m, s'_m) \in \sigma_m$ iff there is, a controllable action $\ell \in L$, a transition $((s_e, s_m), \ell, (s'_e, s'_m)) \in \Delta_{E||M}$ and the truth values for the fluents are the expected by the definition of Γ^- and Γ^+. Furthermore, given that $E||M$ has no deadlocks and $E||M \models \Box\, \rho$, it follows that σ cannot reach a state such that $\Gamma^- \cup \Gamma^+ = \emptyset$, in other words, σ cannot reach a deadlock state. Also, by construction of Γ^- and Γ^+ the truth value of the fluents in the states is updated as expected. Finally, from the definition of the update of the values $\alpha_1, \ldots, \alpha_k$ in the states of G it is simple to see that a play is winning for controller iff it satisfies the FLTL formula $\Box\, \rho \wedge (\bigwedge_{i=1}^{n} \Box \Diamond\, \phi_i \Rightarrow \bigwedge_{j=1}^{m} \Box \Diamond\, \gamma_j)$. □

THEOREM 5.2. (Soundness) *Let $\mathcal{E}$ be a SGR(1) LTS control problem and G be a GR(1) game constructed by applying the conversion shown in Section 5.1 to $\mathcal{E}$, σ be a transition relation and u be an update function. If (σ, u) is winning strategy for G, and C is the LTS obtained by applying the conversion shown in Section 5.2, then it holds that C is a solution for $\mathcal{E}$.*

PROOF. Consider a joint computation $p = (s_{e_0}, s_{M_0}), \ell_0, (s_{p_1}, s_{c_1}), \ell_1, \ldots$ of $E||M$. Recall that states in $E||M$ are of the form (s_{e_0}, s_{M_0}) where $s_m = (m, s_p, \alpha_1, \ldots, \alpha_k)$. Now, by construction and the fact that E is deterministic, we know that for every fluent fl_i and for every $j \geq 0$ we have $s_{c_j} = (m_j, s_{p_j}, \alpha_1^j, \ldots, \alpha_k^j)$ and α_i is the truth value of fl_i^j at time j. Then, for some $j \geq 0$ and $s_m \in S_M$ we show that $s_{c_j} \in \phi_i$ iff ϕ_i is true at time j and similarly for γ_j. In addition, from the fact that the computation $s_{M_0}, \ell_0, s_{m_1}, \ell_1, \ldots$ is a computation of M, the sequence $s_{M_0}, s_{m_1}, \ldots$ is the product of a play in G that is consistent with (σ, u) and that (σ, u) is a winning strategy, it follows that this play in G correspond to a trace in M such that if the assumptions holds then the system goals will do so. Finally, given that (σ, u) is winning, it only produces infinite plays showing that $E||M$ states not satisfying ρ are not reachable and there are no deadlocks. □

5.3. Implementation

In this section we present the algorithm implemented extending the MTSA tool set [D'Ippolito et al. 2008]. The algorithm is based on ideas of [Juvekar and Piterman 2006]. For more details the reader is referred to Section 7.

Intuitively, the algorithm aims to avoid, through restricting controllable actions, cycles of states satisfying all the assumptions but not all of the goals. The existence of such cycles would allow for traces in which the controller looses the GR(1) game. In order to avoid such cycles the algorithm searches, for every state, a strategy that guarantees satisfaction of all goals. To do so, it chooses an order in which it will attempt to satisfy the goals. The algorithm applies a fixed point iteration for computing the best way each state has to satisfy the next goal. In order to measure the "quality" of different successor states with respect to satisfying the next goal, a ranking system is used. The rank for a particular successor will measure the "distance" to the next goal in terms of the number of times that all assumptions will be satisfied before reaching the goal. If this number tends to infinity then this means that from the current state a trace is possible in which the environment assumptions hold infinitely often but the system goals do not. Hence, such state should be avoided by the strategy for the controller.

Consider a game $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi)$, where $\varphi = gr((\phi_1, \ldots, \phi_n), (\gamma_1, \ldots, \gamma_m))$. A *ranking function* for a goal γ_j is a function $R_j : S_g \to (\mathbb{N} \times \{1, \ldots, n\}) \cup \{\infty\})$. Intuitively, $R_j(s_g) = (k, l)$ means that in order to reach from s_g a state in which γ_j holds, all paths will make assumption ϕ_l hold at most k times, ϕ_1 through ϕ_{l-1} will hold at least $k + 1$

times and assumptions ϕ_{l+1} through ϕ_n will hold at least k times. $R(s) = \infty$ means that s is a *loosing* state, i.e. from s there is no strategy for the controller that can avoid a trace which satisfies infinitely often all assumptions but does not satisfy infinitely often all goals. A *ranking system* for G and φ is a sequence R_1, ..., R_m of ranking functions, each associated with a specific goal γ_j.

As mentioned above, the computation of the ranking system is a fixed point iteration, where the rank of a state for a goal γ_j is computed based on the rank of its successors. For instance, if s_g is controllable and γ_j is satisfied in s then the best choice for the controller would be to move to a state in which satisfaction of $\gamma_{j\oplus 1}$ is likely. Hence, its best choice is the successor state with the lowest ranking for goal $\gamma_{j\oplus 1}$ where $j \oplus 1$ is $(j \bmod m) + 1$. On the other hand, if s_g is controllable but does not satisfy γ_j, then the best choice is the successor with the best ranking for the same goal, i.e. γ_j. If s_g is a non-controllable state, the difference is that the ranking must consider the worst possible scenario: It is the environment, rather than the controller, that picks the successor state and it picks the state that is least likely to achieve the next goal. Hence the rank of s_g will depend on the highest rank of its successors.

The above intuition is encoded in the following function $sr : S_g \to (\mathbb{N} \times \{1, \ldots, n\}) \cup \{\infty\})$. The function also encodes the fact that deadlocking states (states with no successors) are ranked ∞. In addition, note that we order ranks using the lexicographical order. Given a state s_g and a goal γ_j, $sr(s_g, j)$ is defined as follows:

— If $\Gamma^+(s_g) \cup \Gamma^-(s_g) = \emptyset$, then $sr(s_g, j) = \infty$, otherwise
— If s_g is controllable and $s_g \in \gamma_j$ then $sr(s_g, j) = min_{s'_g \in \Gamma^+(s_g)}\ R_{j\oplus 1}(s'_g)$.
— If s_g is controllable and $s_g \notin \gamma_j$ then $sr(s_g, j) = min_{s'_g \in \Gamma^+(s_g)} R_j(s'_g)$.
— If s_g is uncontrollable and $s_g \in \gamma_j$ then $sr(s_g, j) = max_{s'_g \in \Gamma^-(s_g)} R_{j\oplus 1}(s'_g)$.
— If s_g is uncontrollable and $s_g \notin \gamma_j$ then $sr(s_g, j) = max_{s'_g \in \Gamma^-(s_g)} R_j(s'_g)$.

Function $sr(s_g, j)$ computes the rank of the successor state that should be used to compute $R_j(s_g)$. It does so assuming that ranks of all successor states have been previously computed. In order to compute the true ranks of all states, we must do a fixed point iteration. The fixed point is when the rank of every state is *stable* with respect to every goal.

We say that s_g *is stable in* R_j if all the following hold.

— If $s_g \in \gamma_j$ and $sr(s_g, j \oplus 1) = \infty$ then $R_j(s_g) = \infty$.
— If $s_g \in \gamma_j$ and $sr(s_g, j \oplus 1) \neq \infty$ then $R_j(s_g) = (0, 1)$.
— If $s_g \notin \gamma_j$, $R_j(s_g) = (k, l)$ and $s_g \in \phi_l$ then $R_j(s_g) > sr(s_g, j)$.
— If $s_g \notin \gamma_j$, $R_j(s_g) = (k, l)$ and $s_g \notin \phi_l$ then $R_j(s_g) \geq sr(s_g, j)$

The intuition for this definition is as follows: If goal γ_j is satisfied in state s_g but its successors cannot achieve the goal ($sr(s_g, j \oplus 1) = \infty$) then s is losing and its rank for γ_j should be ∞. However, if the successors of s_g are winning then, as γ_j holds in s_g, no assumptions need to be visited before satisfying γ_j. Hence, best possible rank is $(0, 1)$.

If goal γ_j is not satisfied in state s_g but ϕ_l is, then the number of times ϕ_l will be satisfied before achieving γ_j must be greater than the number of times that its successors will satisfy ϕ_l before satisfying γ_j ($R_j(s_g) > sr(s_g, j)$) On the other hand, if neither γ_j nor ϕ_l are satisfied in state s_g, then the number of times ϕ_l will be satisfied before achieving γ_j must not be lower than the number of times that its successors will satisfy ϕ_l before satisfying γ_j.

The algorithm for solving the GR(1) game consists of three steps. First, it initialises the ranking system so that $R_j(s_g) = (0, 1)$ for all states and goals. Second it iterates until the ranking system is stable. If it is not stable, then the rank for some state and goal needs to be incremented, and stability is checked again. It is known that every

	1	$(4, c_1)$	$(5, a)$	$(4, g_1)$	$(4, g_2)$
$\dot{g_1}$	$(1,1)$	(1.1)	$(1,1)$	$(0,1)$	$(1,1)$
$\dot{g_2}$	$(1,1)$	(1.1)	$(1,1)$	$(1,1)$	$(0,1)$

Table I. Ranks for states in Strategy σ_R.

rank greater than $(max_j\, max_i\, |\phi_i - \gamma_j|, n)$ is effectively equivalent to ∞, where $|\phi_i - \gamma_j|$ is the number of states in G that satisfy ϕ_i and do not satisfy γ_j, which guarantees termination of the algorithm. Finally, if a stable ranking in which the initial state has a non-infinite ranking for the first goal ($R_1(s_{g_0})$) then a winning strategy is constructed. This last step is supported by the following theorem, which has been proven correct but the proof is out the scope of this paper.

THEOREM 5.3. (Algorithm Soundness) *If R_1, ..., R_m is a stable ranking system, then for every state s_g such that $R_1(s_g) \neq \infty$ there exists a winning strategy from s_g.*

PROOF. Let $M = \{1, \ldots, m\}$ be the memory range of the strategy. The memory is updated by $u(v,t) = v$ if $t \notin G_v$ and $u(v,t) = v \oplus 1$ otherwise. The function $\sigma(v,t) = \{t' \in L^+ \mid t' \prec_j t\}$. By definition of stability, $\sigma(v,t)$ includes all uncontrollable successors of t. It is simple to see that following this strategy the reachable states are always contained in T and the reachable states have a finite rank.

Consider a computation induced by this strategy $p = t_0, s_1, \ldots$. Let $j_0, j_1, \ldots$ be the sequence of memory values used by u and let $r_0, r_1, \ldots$ be the sequence of ranks, where $r_i = R_{j_i}(t_i)$. The only way in which $j_{i+1} \neq j_i$ is if G_{j_i} is visited by t_i. If for infinitely many locations $j_{i+1} \neq j_i$ then the computation visits all G_j infinitely often. Otherwise, from some location i_0 we have for all $i > i_0$ $G_i = G_{i_0}$. Consider the sequence of ranks $r_{i_0}, r_{i_0+1}, \ldots$. By assumption, for all o we have $r_{i_{o+1}} \leq r_{i_o}$ and furthermore if $r_{i_o} = (k,l)$ and $t_{i_o} \in A_l$ then $r_{i_{o+1}} < r_{i_o}$. By well-foundedness of $\mathbb{N} \times \{1, \ldots, n\}$ we conclude that from some point onwards r_i is constant and for some l we have A_l is visited finitely often. It follows that all computations are in φ and the strategy is winning as required. □

In Figure I we show the ranks values for states in σ_R, the strategy shown in Figure 7(b). The columns represent states in the strategy and the rows, which goal is being considered. The ranks are mostly $(1,1)$ since from most of states for both goals, $\dot{a}$ holds before $\dot{g_i}$ hold for $i \in \{1,2\}$. As expected, the rank for $(4, g_1)$ and $(4, g_2)$, according to $\dot{g_1}$ and $\dot{g_2}$ respectively is $(0,1)$.

The construction of a winning strategy (σ, u) from a stable ranking where $R_1(s_{g_0}) \neq \infty$ is straightforward: The strategy will attempt to reach goals in turns, that is it will first reach γ_j before it attempts to reach $\gamma_{j\oplus 1}$. To reach a goal γ_j from a state s_g it will pick a successor of s_g such that it has a smaller ranking for γ_j ($\sigma(j, s_g) = s'_g$ such that $R_j(s_g) > R_j(s'_g)$). When it reaches γ_j, it will simply pick a successor state with non-infinite rank for the next goal ($\sigma(j, s_g) = s'_g$ such that $R_{j\oplus 1}(s'_g) \neq \infty$). The memory update function u simply changes the goal to be satisfied if the current goal is satisfied at the current state: $u(j, s_g) = j$ if γ_j is not satisfied in s_g and $u(j, s_g) = j \oplus 1$ otherwise. Note that each ranking function depicts a plan for reaching its own goal. Thus, using these plans, goals can be pursued in any order.

What remains is to show that the algorithm is complete:

THEOREM 5.4. (Algorithm Completeness) *If there is a winning strategy wins from $s_g \in G$, there exists a stable ranking system $R_1, \ldots, R_m$ such that for every $s_g \in S$ and $j \in \{1, \ldots, m\}$ we have $R_j(s_g)$ is either ∞ or (k, l) with $k \leq max_l|\phi_l - (\gamma_j)|$.*

PROOF. An analogous proof is provided in [Kesten et al. 2005]. □

We show the complete algorithm in Figure 8 computes a stable ranking such that for every state $s_g \in T$ if s_g is winning for controller (i.e. $R_1(t) < \infty$). At high level, the algorithm has two major parts, the initialisation and the stabilisation. The initialisation sets the initial rank for every state in the game and initialises the queue of states $pending$ to be processed. A state is added to $pending$ if does not satisfy any guarantee and do satisfy assume ϕ_1. All the states in every ranking function are initialised with $(0, 1)$ (i.e. the minimum possible rank) except for states such that $\Gamma^- \cup \Gamma^+ = \emptyset$ which are initialised with ∞. Notice that states with ∞ rank are those which either does not satisfy ρ or are deadlock states in E. The stabilisation part is a fixed point that iterates on $pending$ until is empty. We now describe the stabilisation procedure. The function `is_stable(state,g)` returns true if the g-th ranking function is stable for `state`. The function `unstablePred(state,g)` returns a set of pairs of predecessors of `state` and a rankings g for which the ranking is unstable. The function `best(state,g)` returns the value of $best(\texttt{state}, \texttt{g})$, as defined above. Finally, $\texttt{inc}(k, l), \texttt{state}, \texttt{g})$ returns $(0, 1)$ if `state` is in $\gamma_\texttt{g}$, it returns (k, l) if `state` is not in $\texttt{assumption}_l$, and it returns the minimal value greater than (k, l) otherwise. Notice that $\texttt{inc}(\infty, \texttt{state}, \texttt{g})$ is ∞ and if $n = max_l(|\phi_l - (\gamma_\texttt{g})|)$ and `state` is in $\phi_m - \gamma_\texttt{g}$ then $inc((n, m), \texttt{state}, \texttt{g})$ is ∞. This algorithm computes the minimal existing stable ranking. Based on the ideas in [Etessami et al. 2005] and [Juvekar and Piterman 2006], this algorithm can be implemented to work in time $O(m \cdot n \cdot |S|^2)$.

The following theorem follows from the algorithm described above and Theorems 5.4 and 5.3.

THEOREM 5.5.
Given a SGR(1) control problem $\mathcal{E} = \langle E, H, A_c \rangle$, where $H = \{(\emptyset, I), (As, G)\}$, $I = \square\, \rho$, $As = \bigwedge_{i=1}^{n} \square \diamondsuit \phi_i$ and $G = \bigwedge_{j=1}^{m} \square \diamondsuit \gamma_j$. $\mathcal{E}$ is solvable in $O(m \cdot n \cdot |S|^2)$ time.

6. CASE STUDIES

In this section we show the results of applying our technique to different case studies. The case studies were performed using an implementation of the control synthesis algorithm described above integrated into the Modal Transition System Analyser [D'Ippolito et al. 2008]. The modified tool and case studies can be downloaded from http://sourceforge.net/projects/mtsa/

Autonomous Vehicles.
We present a variation of the case study originally presented in [Heaven et al. 2009] in the context of self-adaptive systems. In [D'Ippolito et al. 2011], a similar version, including failures as part of the environment model, was considered and applied in the context of controller synthesis for fallible domains.

Consider a situation in which a two-bedroom house has collapsed leaving only one small passage between the two rooms (referred to as $north$ and $south$ rooms). The entrance door of the house is in the south room and there is a group of people trapped in the north room. The task of bringing aid packages to the occupants trapped inside is too dangerous for humans, hence, a robotic system is required. A robot that has a wide range of movements and has an arm capable of loading and unloading packages. The robot has a number of sensors which can be used, among other things to check if a loading operation, which is of a significant amount of complexity and uncertainty, is

```
SolveGame(game=(states,transitions),safe,
          guarantees,assumptions,fault)
{
//Initialisation
  for (state : states) {
    for (g : guarantees) {
      rank_g(state)=(0,1);
    } // for (g)
  } // for (s)
  Queue pending;
  for (state : states) {
    if (∃ g : guarantees . state ∉ g &&
        state ∈ assume_1) {
      pending.push(pair(state,g));
    } // if
    if (Γ^−(state) = ∅ && Γ^+(state) = ∅) {
      for (g : guarantees) {
        rank_g(state)=∞;
      } // for (g)
      pending.push(unstablePred(state,g));
    } // if
  } // for (s)
//Stabilisation
  while (!pending.empty()) {
    (state,g) = pending.pop();
    if (rank_g(state)==∞)
      cont;
    if (is_stable(rank_g(state)))
      cont;
    rank_g(state)=inc(best(state,g),state,g);
    pending.push(unstablePred(state,g));
  } // while ()
} // SolveGame
```

Fig. 8. Pseudo-code of algorithm for solving SGR[1] games.

successful or not. The situation is complicated by the presence of a door between the two rooms. The door cannot be opened by the robot. However, although the structure is unstable, it is known that once the door is open, it can be held open by the trapped occupants.

A descriptive model of the environment was constructed by composing a model of the robot (with actions such as $moveNorth$), its robot arm (with actions such as $getPackage$ or $putPackage$) and sensors (e.g. $getPackageOk$, $getPackageFailed$), a model of the door (e.g. $openDoor$), and a topological model of the house which restricts movements according to the position of the robot and the status of the door. For instance, it describes that the robot only can cross the door if it is near it and in which positions it ends up after crossing it. Whenever the robots moves it senses the destination position from the environment (i.e. $southNear$, $southFar$, $northNear$ $northFar$).

The aim is to automatically synthesise a behaviour model that will control the robot and will achieve the task of retrieving aid packages from the outside to the room where the occupants are trapped. Hence, the set of controllable actions is the set of actions that correspond to the actions that can be performed by the robot and its arm (e.g.

$moveNorth$, $getPackage$, $putPackage$) excluding actions not controlled by the robot (e.g. $openDoor$, $getPackageOk$, $northFar$).

The formalisation of the prescriptive goals for the controller is divided into two parts: safety and liveness.

The safety part prescribes the expected places for loading and unloading the robot. That is, (i) the robot can only be loaded while is stopped and near the entrance of the house. Consequently, the robot cannot move until it is successfully loaded with an aid package. (ii) Packages must not unloaded in rooms other than the north room.

The liveness part of the goal states that the robot must be at the far end of north room and have just unloaded infinitely often: $G = \Box \diamondsuit (\dot{northFar} \wedge \dot{putPackage})$. The control problem, as defined up to this point is not realisable, as the robot has no guarantees that the door will be open for it to move freely to and from the north and south rooms. Some assumption on the behaviour of the door must be included. We introduce the assumption that the door is infinitely often open ($As = \Box \diamondsuit \dot{doorOpen}$). This is still insufficient, since the robot has no control over the success or failure of attempting to load an aid package using the arm. Thus, packages may never be loaded successfully. This shows that there is a missing assumption stating that if the robot attempts to load a package it will eventually succeed: $\Box(\dot{getPackage} \rightarrow \diamondsuit \dot{getPackageOk})$. When assumptions regarding the door being open and package loading being successful are included in the SGR(1) LTS control problem, it becomes realisable. Hence, a solution exists and is constructed automatically by the tool.

It is interesting to note that without the safety restriction disallowing the robot to move unless successfully loaded, the assumptions would not be compatible. Specifically, in the case in which the robot is near the door, not moving and unloaded, the robot cannot leave its position if it is not successfully loaded. Thus, after a failed load operation the robot is forced to retry. Consequently, no controller would be able to prevent the environment to fulfil its promise. Nevertheless, if after a loading fail the robot could not only to retry but also to move then the environment would not be able to fulfil its assumptions on its own and would depend on the controller's decision to retry or not. This illustrates how compatibility would be actually violated and although in this particular case our algorithm yields a best effort controller it can not be guaranteed in general. Moreover, this shows the usefulness of best effort controllers. Specifically, without the restriction an assumption preserving controller would not be possible, while best effort controllers exist.

Intuitively, the assumptions should represent the requirements of a reasonable environment. These assumptions will enable the robot to make progress. Compare, for example, the assumptions $\Box \diamondsuit \dot{getPackageOk}$ and $\Box \diamondsuit (\dot{getPackageOk} \vee \neg \dot{getPackage})$. The first, superficially matches our intuition that the environment should make it possible for the machine to pick up a package. It is, however, too strong as the machine may not try to pick packages at all. The second, depicts a reasonable environment: a machine that keeps trying to pick up a package should be allowed to do so. Hence, it is not surprising that the second assumptions satisfies the assumption compatibility condition 4.6. Furthermore, $As = \Box \diamondsuit (\dot{getPackageOk} \vee \neg \dot{getPackage}) \wedge \Box \diamondsuit \dot{doorOpen}$ allows synthesising a solution to the LTS control problem $\mathcal{E} = \langle E, \{(\emptyset, I), (As, G)\}, \{\alpha ARM \cup \alpha ROBOT\}\rangle$ that tries as much as possible to load packages. In other words, for this case study we successfully synthesised a behaviour model that controls the robot arm which ensures that if the assumptions are satisfied by the environment the machine satisfies its goals.

The synthesised behaviour model is too big to be shown here. Nevertheless, the tool and FSP source code for the case study is available at [D'Ippolito et al. 2008].

Comparing our approach to the original case study, note that in [Sykes et al. 2007] *(i)* no assumptions are explicitly given, and as a result *(ii)* no guarantees can be given as to whether the synthesised controller will satisfy the goal of delivering aid packages, and *(iii)* although under certain conditions the plan synthesised by [Sykes et al. 2007] will work, it is not clear what those circumstances are. We overcome these issues by modelling the robot, its arm and sensors, and the house restrictions separately from the controller goals. This allowed us to discover implicit environmental assumptions. Specifically, in order to guarantee that the robot will successfully deliver aid packages infinitely often, the door must be open infinitely many times. Furthermore, the arm has to successfully pick up aid packages infinitely many times. In other words, by applying Jackson's [Jackson 1995a] approach, we discovered the environmental assumptions required to guarantee satisfaction of the goals, shown how the assumptions can be explicitly encoded in our SGR(1) control problem and checked the assumptions to be compatible with the environment.

Purchase & Delivery.
In [Pistore et al. 2004] a case study involving the synthesis of a plan for composing and monitoring of distributed web services is presented.

More specifically, purchase requests must be processed by a web-service by buying on a furniture-sales service and booking a shipping service. Consequently, this web service must handle the interaction between user requests and both services by controlling messages and forwarding between the parts.

Additionally, both the furniture-sales and shipping services may fail processing a request. Naturally, failures may prevent the composed web-service to succeed purchasing and delivering furniture. Consequently, the goal proposed states that if there is a failure while trying to pay and ship furniture, then the planning goal changes to one in which all, the user, the furniture-sales and the shipping service reach a failure state. This aims to avoid inconsistent states in which some services succeed and some fail. For instance, if the user refuses the offer, the composed service is expected to reach a state in which both the purchase and delivery requests have been cancelled.

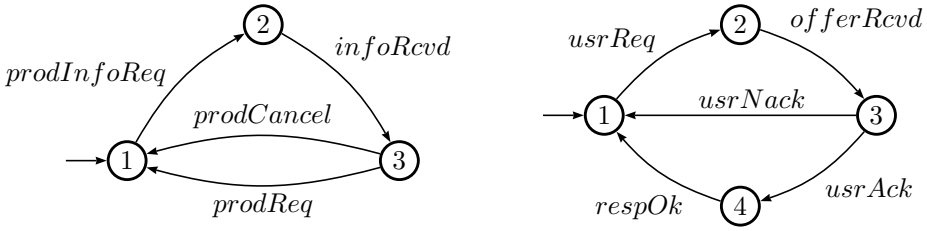
We restrict the analysis and synthesis to a failure-free version of the problem. Hence, both furniture-sales and shipping services always respond positively on a request by the model to be synthesised. Even though the failure-free assumption may seem to restrictive it allows us to handle some of the, so-considered, failures in [Pistore et al. 2004]. For instance, the user refusing a furniture-delivery pair is considered a failure, which violates the intuition that failures are not controlled by users, instead they are supposed to happen unexpectedly, e.g. a server crashes.

Even though in [Pistore et al. 2004] a behaviour model for this domain in which failures are possible is synthesised, there are no guarantees that the goal of satisfying purchase requests is achieved. In fact, achieving the goals stated in [Pistore et al. 2004] requires assuming some progress on the environment and fairness conditions on the success of operations on the furniture-sales and shipping services, as we show below.

Now we describe how this case study is handled by our approach. By providing descriptions about the services and user behaviour, and prescribing the desired goals for the controlled environment, considering first the safety prescriptions and then the liveness ones.

The interface of the furniture-sales service, described by the LTS in Figure 9(a), allows requesting for information on a particular product ($prodInfoReq$) then once the information has been received ($infoRcvd$) it is possible place a request for the product ($prodReq$). The protocol to interact with the shipping service is analogous.

A model describing how the user interacts with the composed web-service is shown in Figure 9(b). The user can place a request for some product ($usrReq$) then he may get



(a) Furniture-sales service interface. (b) User.

Fig. 9.

either an offer for product and shipping combination ($offerRcvd$) or a denial of service ($reqDenial$) resetting the protocol. If an offer is received, the user can either confirm the order ($usrAck$) or decline it ($usrNack$). If the order is confirmed, the user waits for its product to be shipped as arranged ($respOk$).

There are several safety prescriptions for the controller-to-be. (i) The service should only check for some product or shipping information if the user placed a request first. This restricts the composed web service from spontaneously generating queries without a triggering user request. (ii) It is only possible to offer the user with a combination of product and shipping service if both services have confirmed availability. (iii) Both product and shipping requirements should be placed only if the user acknowledged the purchase. (iv) The service will only cancel product or shipping requests if the user does not acknowledge the offer he received. (v) The service only flags that the request has been cancelled when both product ordering and shipping services have cancelled the request. (vi) The service finishes successfully only after the product ordering and shipping services have successfully handled their requests. These requirements can be easily expressed in FLTL. For the full specifications the reader is referred to [D'Ippolito et al. 2008].

The liveness prescription for the behaviour model to be synthesised is simply to buy infinitely many product-shipping pairs, which can be encoded as $\square \diamondsuit \dot{respOk}$.

Without any collaboration of the environment it is not possible for a controller to satisfy its liveness goals. For instance, if the user never acknowledges for purchase and delivery of furniture, then it will not be possible to fulfil controller goals. Consequently, we will be able to generate a controller only if it is possible to assume that the environment will acknowledge requests infinitely many times, in FLTL this is $\varphi = \square \diamondsuit \dot{usrAck}$. However, the environment cannot acknowledge product and shipping combinations if the controller does not provide such combinations. In other words, the environment cannot fulfil this assumption on its own, which shows that φ is not capturing our intuition correctly. Furthermore, φ does not satisfy the compatibility condition 4.6, which as shown before lead to undesired controllers.

Our assumptions must state that the environment has to acknowledge combinations only if he received an offer $\square(\dot{offerRcvd} \rightarrow \diamondsuit \dot{usrAck})$. As shown in Section 4.1 this assumptions can be expressed with the FLTL formula $\varphi' = \square \diamondsuit \neg \dot{OfferAckd}$, where $\dot{OfferAckd} = \langle \dot{offerRcvd}, \dot{usrAck}, \bot \rangle$. This means that if the environment receives infinitely many offers it has to acknowledge them infinitely often, which is compatible with the environment and captures our intuition more closely.

We modelled the case study as an SGR(1) problem and applied the MTSA tool set to generate a controller, which is shown in Figure 10. As one may expect the controller only synchronises the message passing between the user, furniture and delivery services. The environment model is a compatible with respect to the assumptions that

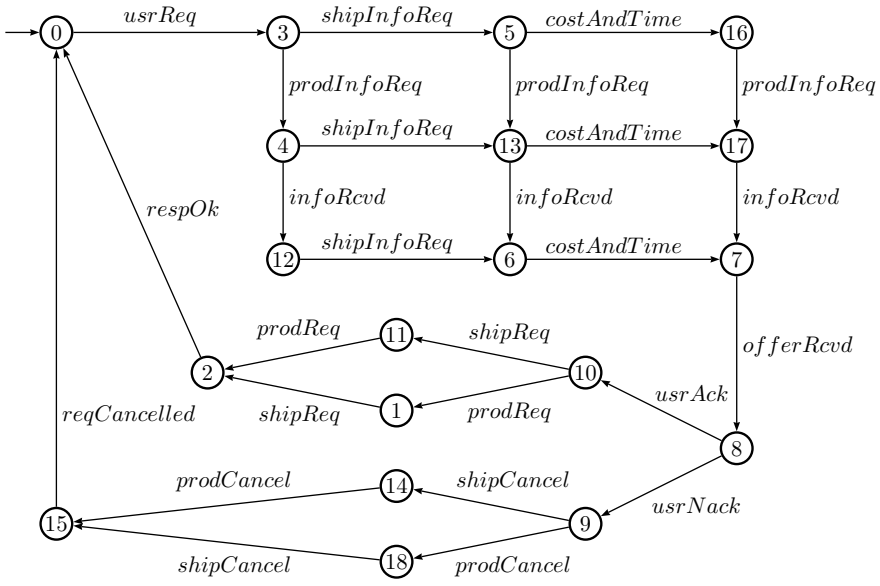


Fig. 10. Controller for serving furniture requests.

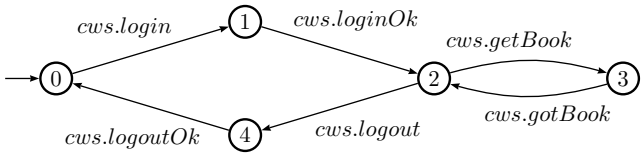
customers confirm infinitely many products and delivery options. Thus, the resulting controller is guaranteed to be non-anomalous and the environment assumptions under which it achieves its goals are explicit.

It is important to note that in the original case study there are no explicit environment assumptions. Consequently, it may be uncertain if and when the controller behaves properly. In other words, there are no guarantees that goals are to be achieved. This is due to the fact that the problem is not properly modelled. Following the World-Machine model i.e., properly modelling the relevant descriptive and prescriptive statements, helped us discover the required environmental assumptions on the user behaviour which allow for guaranteeing the satisfaction of the prescriptive goals. For instance, modelling the user ack/nack responses as part of the description of the domain and providing the safety prescription, lead to the assumption on user's acknowledgements, central to the generating of the controller.

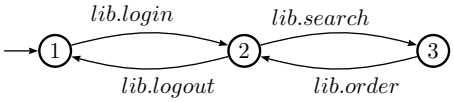
Bookstore.
The web-service composition scenario in [Inverardi and Tivoli 2007], is in a sense similar to that of Pay & Ship, in that two services must be coordinated to provide a more complex service. The main difference is that Inverardi et al. provide no explicit liveness goals for the controller nor liveness assumptions on the environment behaviour.

Following the world-machine approach, we describe the environment models as the parallel composition of models for the composed web-service (CWS) and the books search and order services.

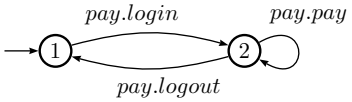
More specifically, the composed web-service (CWS), for which the interaction protocol is presented in Figure 11(a), must coordinate a service for search and order books, and a payment service. The interaction protocol for the search and order service is shown in Figure 11(b). Once the user has logged in he can either choose to search and



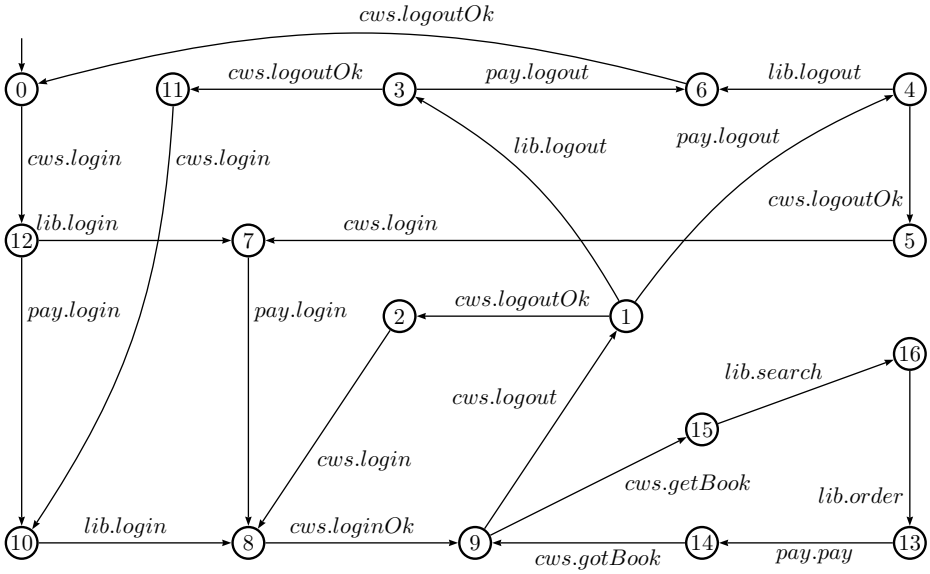
(a) CWS Interaction protocol.



(b) Service for book search and order



(c) Payment service



(d) Search-Order-Pay web-service.

Fig. 11. Bookstore case study models.

order books or to logout terminating the interaction. Similarly, the payment service (Figure 11(c)) requires a log in to enable for payments to occur. Moreover, while the user is logged in, he can place as many payments as required.

As in [Pistore et al. 2004], the main goal of the safety prescriptions is to prescribe the ordering between the actions of the services involved. For instance, the composite web-service can only be considered logged in, if both the (sub-)services have been successfully, logged in.

In the following we show the safety properties prescribing how the composed web-service must orchestrate the (sub-)services.

(i) The user logs on (off) triggering $cws.login$ ($cws.logout$) action, then the service should log into both payment and library services triggering $pay.login$ ($pay.logout$) and

$lib.login$ ($lib.logout$) respectively, then (ii) only after both search and payment services have logged in the CWS must inform the user that the login (logout) operation was successful by triggering the $cws.loginOk$ ($cws.logoutOk$) action. Naturally, (iii) the CWS should not attempt a login action on neither of the (sub-)services if there no $cws.login$ action first. The procedure of searching, ordering and finally paying for books must also be coordinated by the CWS. (iii) CWS can only search for books $lib.search$ if a request has been placed $cws.getBook$, and either of them can only happen if the user has been successfully logged in ($cws.loginOk$). (iv) Only after a book has been ordered ($lib.order$), CWS triggers the payment ($pay.pay$). Finally, (v) CWS must only confirm that a book has been successfully bought $cws.gotBook$ it is successfully paid for it. There is only one liveness goal for the controller. It must sell (i.e. successfully order and pay) books infinitely often, i.e. $\square \diamondsuit cws.gotBook$. However, this cannot be guaranteed if the user (i.e. environment) does not try to get books infinitely many times, i.e. $\square \diamondsuit cws.getBook$. Note that the environment can only try to get books if some other actions have occurred before, i.e. logins. Hence, it may seem that such assumptions would be non-compatible, it turns out that they are. Intuitively, the actions required for the environment to get books are dependant on the environment solely, i.e. $cws.login$.

Therefore, we defined the SGR(1) problem with system goals $\square \diamondsuit cws.gotBook$ and environmental assumptions $\square \diamondsuit cws.getBook$. The controller we obtained, shown in Figure 11(d), guarantees that for every trace in which the environment assumptions are satisfied the controller goals will also be satisfied. Note that since the environment assumptions are compatible the controller guarantees that is going to do it best to fulfil the system goals.

It is important to note that, since the technique used in [Inverardi and Tivoli 2007] does not allow for liveness goals or environmental assumptions, the achievement of the main goal of the service, i.e., to sell books, cannot be guaranteed. On the contrary, as we specify the case study as a SGR(1) control problem, we can explicitly provide with the liveness conditions required to successfully sell books. Such conditions are required but not sufficient, since without any collaboration of the environment the goals cannot be achieved. As said above, the environment has to try to get books for the controller to successfully sell them.

Production Cell.

This case study has been explained as a running example in Section 2. We presented the descriptive and prescriptive statements for the problem and presented some observations on the generated controller.

The controller was obtained using the MTSA tool set [D'Ippolito et al. 2008]. Since the size of the models is too big to be depicted in this paper, we show the controller for a smaller version, which has only one tool (an oven) and can only process one instance of each product type at a time. The synthesised controller is shown in Figure 12. As noted in Section 2 the controller must "remember" if it has been postponing one type of product for too long. Consequently, the algorithm add memory to the original states encoding the last product type processed in order to guarantee the system goals. The controller waits for products of type A to be processed first (see states 4 and 6) regardless of whether there are products of type B ready to be processed (see states 5 and 6). It then does the same for products of type B (see states 10, 2 and 1).

7. RELATED WORK

In this paper we extend the work previously presented [D'Ippolito et al. 2010] providing the full theoretical background, proofs, a description of a rank-based implementa-

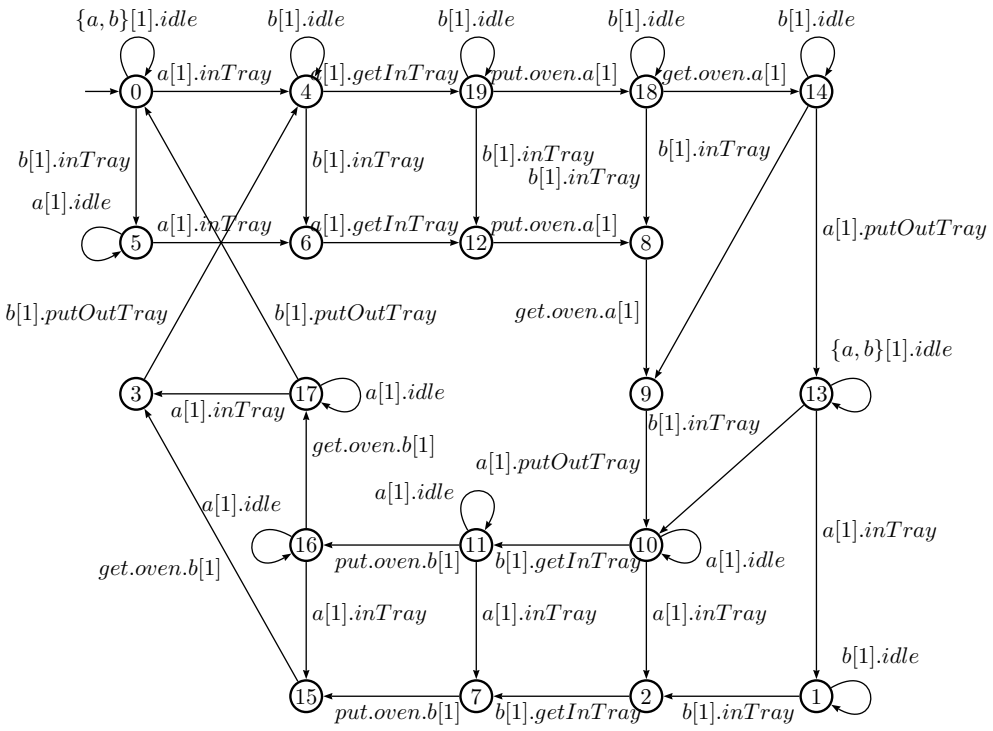


Fig. 12. Controller for reduced Production Cell.

tion and further evaluation. In [D'Ippolito et al. 2011] we have reported on a different controller synthesis technique for fallible domains. Its focus is on how to build controllers that can achieve their goals even when the expected outcome of the operations they control can sometimes fail. In contrast, this paper highlights the methodological challenges of applying controller synthesis event-based systems, hence focusing on the distinction between prescription and description, the notion of anomalous controllers, and the relation with assumption realisability.

Our work builds on that of the controller synthesis community and particularly on the generalised reactivity synthesis algorithm GR(1) proposed in [Piterman et al. 2006]. This line of work originates in updating Büchi, Landweber, and Rabin's work [Buchi and Landweber 1969; Rabin 1970] to modern terms [Pnueli and Rosner 1989]. While [Piterman et al. 2006] handles only a subset of the possible specifications, other recent work tries to bypass the hurdles involved in solving the general problem (e.g., [Kupferman and Vardi 2005] and its extension in [Schewe and Finkbeiner 2007]). The community has largely focused on controllers for embedded systems and digital circuits (cf. [Bloem et al. 2007; Sohail et al. 2008]), hence adopting a shared memory model: The controller is aware of changes in the environment by querying the state space shared with the environment. For instance, GR(1) uses kripke structures, state machines with propositional valuations on states, where the environment and the controller update and read respectively controlled and monitored propositions. However, in many settings such as requirements engineering, architectural design and self-adaptive systems, a message passing communication model in the context of a distributed system is typically considered. Hence, controller synthesis techniques require

adaptation to the notion of event-based communicating machines [Hoare 1978]. This adaptation, specifically to Labelled Transition Systems (LTS) [Keller 1976] semantics and CSP-like parallel composition [Hoare 1978], is a contribution of this paper. The change from state-based to event-based semantics introduces the need for determinism of the environment to guarantee that the controller has sufficient information about the state of the environment to guarantee it satisfies its goals (see Definition 4.2 and Theorem 5.2). The change also introduces the need for a sound methodological approach to the definition of assumptions in order to avoid anomalous controllers.

Even though several behaviour model synthesis techniques have been studied (e.g. [Damas et al. 2006; Young and Devanbu 2006; Bontemps et al. 2004]) these are restricted to user-defined safety requirements applying variations of the backward error propagation technique [Russell and Norvig 1995]. The exceptions that we are aware of relate to the self-adaptive systems and the planning communities.

In the self-adaptive systems community many architectural approaches for adaptation have been proposed. At the heart of many adaptation techniques, there is a component capable of designing at run-time a strategy for adapting to the changes in the environment, system and requirements (e.g. [Dalpiaz et al. 2009; Huang et al. 2004; Gat et al. 1997; Kramer and Magee 2007]). These architectures do not prescribe the mechanism for constructing adaptation strategies. The technique we propose here could be used in the context of all of these architectures. In fact, we believe, that the methodological guidance that our approach offers could help integrating the controller synthesis techniques into these architectures in a sound way. Furthermore, by providing support for explicitly modelling the environmental assumptions, our approach allows to clearly understand what system goals are guaranteed to be satisfied.

A few approaches to automated construction of adaptation strategies exist. Sykes et al. in [Sykes et al. 2007; Heaven et al. 2009] build on the "Planning as Model Checking" framework [Giunchiglia and Traverso 2000] to construct plans that aim to guarantee reaching a particular goal state. Thus, this technique can handle certain liveness requirements. However, the execution of the plan is restarted every time the environment behaves unexpectedly. Hence, there is an implicit assumption that the environment behaves "well enough" for the system to eventually reach the goal state. Validating that the environment will behave "well enough" is not possible as the notion is not defined and it is not clear what guarantees are provided by the plans. In the proposed approach, assumptions are explicit and hence guarantees are clear. In addition, being explicit, it is possible to validate if the assumptions needed in order to achieve the goals hold in the environment in which the controller is to be deployed. For instance, in the *Autonomous Vehicles* case study from previous section, without any assumption on how the door behaves, there would not be a plan for the robot that guarantees bringing aid packages form south to north room infinitely often. The same occurs in the *Purchase & Delivery* case study, where without the assumption that if the user receives combination of furniture and delivery options often enough, he will acknowledge such combinations often enough.

More generally, the planning as model checking framework (e.g., [Giunchiglia and Traverso 2000]), supports CTL goals, requires a model in the form of a Kripke structure and does not consider the problem of composing the environment with a machine that is responsible for guaranteeing the system's goals. Consequently, it does not distinguish between controlled and monitored actions and the plans that are generated would not be realisable by the software system. Moreover, since planning as model checking synthesises from CTL formulas it is not possible to distinguish which are the assumptions on the environment behaviour required to guarantee the satisfaction of the controller goals.

Assumptions on environment behaviour are a key part of the synthesis problem. In general, without proper assumptions, controllers cannot be produced. In [Chatterjee et al. 2008] a technique for correcting unrealisable specifications is proposed. Given an unrealisable specification φ, they compute an environment assumption ψ such that $\psi \Rightarrow \varphi$ is realisable and ψ is *environment realisable*. The notion of *environment realisability* is somehow related to our *assumption compatibility* condition. However, *environment realisability* asserts that there exists an environment that satisfies ψ, which in our case is not applicable since the environment is input to our technique. Similar to our motivation, Chatterjee et al. notice that unrealisable specifications for environments lead to abnormalities in the behaviour of their algorithm but do not relate this to the kind of controllers produced. In other words, they do not provide a notion similar to our anomalous controllers. Consequently, if the specification is actually realisable there is no warning that the assumptions are not realisable by the environment and therefore produced controllers could be anomalous. Additionally, we can check for *assumption compatibility* in polynomial time without requiring probabilistic games.

The notion of assumption compatibility was discovered independently in various contexts and used differently. In [Chechik et al. 2007], it is used in the context of vacuity detection (cf. [Beer et al. 2001] and a large body of other work [Kupferman and Vardi 2003; Armoni et al. 2003; Gurfinkel and Chechik 2004; Chechik et al. 2007]) and for debugging environment models. They say that an environment model E *guarantees* a property φ iff for every possible controller C, $E||C \models \varphi$. In other words, a property φ is called an *environment guarantee* iff the environment satisfies φ regardless of what the controller might do.

In the context of [Chechik et al. 2007] this is considered bad. Here, we consider it as a good thing. The fact that Chechik et al. consider CTL and not LTL is a minor difference. However, they implement only a pre-condition for checking environment guarantees. Furthermore, it has since been discovered that implementing the complete check for the test suggested by Chechik et al. is in fact equivalent to realisability [Godefroid and Piterman 2009] and, as mentioned previously, unless restricted to a manageable fragment of the logic does not work well in practice. Our approach is thorough (i.e. complete) but for applicability reasons we restrict the specifications to GR(1) formulas.

In [Cobleigh et al. 2003] a technique for automatic generation of assumptions in the context of assume-guarantee reasoning is proposed. Given a property and a parallel composition of two models, the technique produces the assumptions one of the model has in order to satisfy the property. If the second model satisfies these assumptions then it is possible to conclude that the composition does too. It could be argued that this technique is related to ours as the generated assumptions could be considered the controller. However, there are two main differences. First, the techniques used by Cobleigh et al. are inappropriate for usage in a context that distinguishes between monitored and controlled actions (essentially using model checking instead of game solving). Second, the pruning of unsafe states, can ensure safety but cannot handle liveness requirements. More recently, in [Emmi et al. 2008] Emmi et al. have presented a technique for assume-guarantee verification for interface automata. The technique distinguishes between controllable and monitored actions. However, liveness requirements are not supported.

We use Labelled Transition Systems as our modelling framework. We distinguish between controlled and uncontrolled actions in the definition of the control problem. It could be argued that LTSs do not provide proper support for expressing the required domain models. There are a number of formalisms to describe behaviour distinguishing controlled from monitored actions, such as Input/Output Automata [Lynch and Tuttle 1989] or Interface Automata [de Alfaro and Henzinger 2001] (IA). Since I/O automata require the input actions to be enabled from every state, they do not fit our

domain model. There is no technical limitation that prevents us from using Interface Automata. Adapting our approach to IA is relatively simple. We treat IA output actions as controllable actions and input actions as monitored actions. The conversion of the control problem to a game remains as it is now with the slight difference of considering output actions as controller actions. Computing the strategy and translating it to an interface automaton can be done straightforwardly. Finally, the generated controller will be a valid controller since it satisfies the goals and the requirement of being a legal environment for the environment model.

Regarding the implementation, in [Piterman et al. 2006] an efficient algorithm for solving games with GR(1) winning conditions. However, the notion of game differs from the one used in this paper. Although, both games are turn-based, the order is slightly different, e.g. in [Piterman et al. 2006], the environment chooses its next valuation and only then, the controller gets to choose what to do next. In our case (see Definition 3.4), the controller first restricts transitions over the set of controllable actions and then the environment gets to choose the next state of the game. To use the implementation in [Piterman et al. 2006] a preprocessing to convert one game to the other would be necessary. In addition, the algorithm proposed in [Piterman et al. 2006] manipulates sets of states using a symbolic representation in the form of BDDs [Bryant 1986]. In other words, it uses a data structure that manipulates sets of states. The algorithm requires efficient implementations of set union, intersection, and the application of a transition on a set, i.e. given a set of states, to compute the set of its predecessors, and negation. It is most suitable for cases where the states of the "game graph" are obtained by setting values to variables. There is no natural way to represent sets except by lists of states resulting in a linear overhead for every set operation. Thus, the symbolic algorithm that computes $O(m \cdot n \cdot |S|^2)$ symbolic operations would result in an algorithm that in practice uses $O(m \cdot n \cdot |S|^3)$ operations, where $|S|$ is the number of states, m is the number of environmental assumptions and n is the number of controller goals. Hence, the symbolic algorithm is not the best suited in our context.

In [Juvekar and Piterman 2006], an enumerative solution to games with GR(1) winning condition, based on the same ideas, is presented. However, states of their games are partitioned into two (one controlled by each player). Hence, states are expected to be either uncontrollable or controllable. Furthermore, in [Juvekar and Piterman 2006], strategies are defined as a partial function, which takes sequences of states and yields a state. In other words, given a play conforming with the game their controller has to choose a particular successor while in our case, the controller may choose a set of possible successors. This notion of strategy favours the construction of *best effort* controllers. It is easy to prove that for some cases were assumptions and environment model are not compatible, our approach produces best effort controllers while the algorithm proposed in [Juvekar and Piterman 2006] does not guarantee so. Hence, we implement a variation of [Juvekar and Piterman 2006] which produces controllers complying with our notion of strategy and handles every state independently and its run time is $O(m \cdot n \cdot |S| \cdot |E|)$, where $|E|$ is the number of transitions.

Finally, our work is heavily influenced by the work on requirements engineering by Jackson [Jackson 1995b], van Lamsweerde [van Lamsweerde and Letier 2000; Lamsweerde 2001] and Parnas [Parnas and Madey 1995] who argued the importance of distinguishing between descriptive and prescriptive assertions, between software requirements, system goals and environment assumptions, and the key role that the latter play in the validation process.

8. CONCLUSIONS AND FUTURE WORK

Synthesis of controllers (i.e. behaviour models) that guarantee liveness goals in event-based systems poses not only algorithmic but also methodological challenges. In this

paper, we proposed a technique that works for an expressive subset of liveness properties, that distinguishes between controlled and monitored actions [Parnas and Madey 1995], differentiates between prescriptive and descriptive [Jackson 1995a] aspects of the specification of system goals, environment behaviour, and environment assumptions.

We presented the *event-based* control problem and defined the *LTS* and *SGR(1)* control problems, which are control problems set in a theoretical framework adequate for event-based models. The first acts as a general definition, the second grounds the specification language to LTSs and FLTL, and the third supports safety and GR(1)-like properties. For the latter, we provide a solution that works in polynomial time and is based on a rank computation [Jurdziński 2000]. We also provide proofs for correctness and completeness for the presented algorithm.

Another important contribution is that we identify a characterisation of anomalous controllers that even correct and complete algorithms like ours might yield if no further conditions are required for the assertions acting as liveness assumptions on the environment. Furthermore, we identify an effective condition for assumptions that rules out those anomalies and in line with methodological guidelines in requirements engineering regarding realisability [Letier and van Lamsweerde 2002].

We have reported on the application of our approach to a number of case studies from different domains, such as, robotics, web-services composition and industrial product line control.

The rank-based algorithm and the whole SGR(1) control synthesis technique has been implemented as part of the MTSA tool set [D'Ippolito et al. 2008].

There are a number of avenues for future work. We aim at relaxing the requirement on determinism for the environment model that is currently in place for assuring the soundness of our approach. In fact, this is closely related to non-observability of events controlled by the environment, an area that we also intend to further investigate.

Finally, we are investigating ways of providing feedback for the case in which a controller that satisfies the specified goals does not exist. For instance, by providing a counter-strategy for the environment or perhaps, applying debugging techniques such as [Konighofer et al. 2009].

ACKNOWLEDGMENTS

We thank anonymous FSE and TOSEM referees for their comments first on a conference version of this paper [D'Ippolito et al. 2010] and then on an earlier version of the journal paper. We gratefully acknowledge the support of UBACYT 20020100100813, CONICET PIP 0955, ANPCyT PICT PAE 2272 and 2278.

REFERENCES

ARMONI, R., FIX, L., FLAISHER, A., GRUMBERG, O., PITERMAN, N., TIEMEYER, A., AND VARDI, M. 2003. Enhanced vacuity detection in linear temporal logic. In *Financial Cryptography*, R. Wright, Ed. Lecture Notes in Computer Science Series, vol. 2742. Springer Berlin / Heidelberg, 368–380.

ASARIN, E., MALER, O., PNUELI, A., AND SIFAKIS, J. 1998. Controller synthesis for timed automata. In *Proceedings of the IFAC Symposium on System Structure and Control*.

AUTILI, M., INVERARDI, P., TIVOLI, M., AND GARLAN, D. 2004. Synthesis of "correct" adaptors for protocol enhancement in component-based systems. In *Specification and Verification of Component-Based Systems*. ACM, 79.

BEATTY, D. L. AND BRYANT, R. E. 1994. Formally verifying a microprocessor using a simulation methodology. In *Proceedings of the 31st annual Design Automation Conference*. DAC '94. ACM, New York, NY, USA, 596–602.

BEER, I., BEN-DAVID, S., EISNER, C., AND RODEH, Y. 1997. Efficient detection of vacuity in actl formulas. In *Proceedings of the 9th International Conference on Computer Aided Verification*. CAV '97. Springer-Verlag, London, UK, 279–290.

BEER, I., BEN-DAVID, S., EISNER, C., AND RODEH, Y. 2001. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design 18*, 141–163.

BERTOLI, P., CIMATTI, A., PISTORE, M., ROVERI, M., AND TRAVERSO, P. 2001. MBP: a model based planner. In *Proceedings of the IJCAI01 Workshop on Planning under Uncertainty and Incomplete Information*.

BERTOLINO, A., INVERARDI, P., PELLICCIONE, P., AND TIVOLI, M. 2009. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ESEC/FSE '09. ACM, New York, NY, USA, 141–150.

BLOEM, R., GALLER, S., JOBSTMANN, B., PITERMAN, N., PNUELI, A., AND WEIGLHOFER, M. 2007. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In *Proceedings of the conference on Design, automation and test in Europe*. DATE '07. EDA Consortium, San Jose, CA, USA, 1188–1193.

BONTEMPS, Y., SCHOBBENS, P.-Y., AND LÖDING, C. 2004. Synthesis of open reactive systems from scenario-based specifications. *Fundamenta Informaticae - Application of Concurrency to System Design (ACSD'03) 62*, 139–169.

BRYANT, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers 35*, 677–691.

BUCHI, J. AND LANDWEBER, L. 1969. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 295–311.

CHATTERJEE, K., HENZINGER, T. A., AND JOBSTMANN, B. 2008. Environment assumptions for synthesis. In *Proceedings of the 19th international conference on Concurrency Theory*. CONCUR '08. Springer-Verlag, Berlin, Heidelberg, 147–161.

CHECHIK, M., GHEORGHIU, M., AND GURFINKEL, A. 2007. Finding environment guarantees. In *Proceedings of the 10th international conference on Fundamental approaches to software engineering*. FASE'07. Springer-Verlag, Berlin, Heidelberg, 352–367.

COBLEIGH, J. M., GIANNAKOPOULOU, D., AND PĂSĂREANU, C. S. 2003. Learning assumptions for compositional verification. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*. TACAS'03. Springer-Verlag, Berlin, Heidelberg, 331–346.

DALPIAZ, F., GIORGINI, P., AND MYLOPOULOS, J. 2009. An architecture for requirements-driven self-reconfiguration. In *Proceedings of the 21st International Conference on Advanced Information Systems Engineering*. CAiSE '09. Springer-Verlag, Berlin, Heidelberg, 246–260.

DAMAS, C., LAMBEAU, B., AND VAN LAMSWEERDE, A. 2006. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. SIGSOFT '06/FSE-14. ACM, New York, NY, USA, 197–207.

DE ALFARO, L. AND HENZINGER, T. A. 2001. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. ESEC/FSE-9. ACM, New York, NY, USA, 109–120.

D'IPPOLITO, N., FISCHBEIN, D., CHECHIK, M., AND UCHITEL, S. 2008. Mtsa: The modal transition system analyser. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. ASE '08. IEEE Computer Society, Washington, DC, USA, 475–476.

D'IPPOLITO, N. R., BRABERMAN, V., PITERMAN, N., AND UCHITEL, S. 2010. Synthesis of live behaviour models. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. FSE '10. ACM, New York, NY, USA, 77–86.

D'IPPOLITO, N. R., BRABERMAN, V., PITERMAN, N., AND UCHITEL, S. 2011. Synthesis of live behavior models for fallible domains. In *to appear in the 33rd International Conference of Software Engineering*. ICSE '11. ACM, New York, NY, USA.

EMMI, M., GIANNAKOPOULOU, D., AND PĂSĂREANU, C. S. 2008. Assume-guarantee verification for interface automata. In *Proceedings of the 15th international symposium on Formal Methods*. FM '08. Springer-Verlag, Berlin, Heidelberg, 116–131.

ETESSAMI, K., WILKE, T., AND SCHULLER, R. A. 2005. Fair simulation relations, parity games, and state space reduction for büchi automata. *SIAM Journal on Computing 34*, 1159–1175.

GAT, E., BONNASSO, R. P., MURPHY, R., AND PRESS, A. 1997. On three-layer architectures. In *Artificial Intelligence and Mobile Robots*. AAAI Press, 195–210.

GIANNAKOPOULOU, D. AND MAGEE, J. 2003. Fluent model checking for event-based systems. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. ESEC/FSE-11. ACM, New York, NY, USA, 257–266.

GIUNCHIGLIA, F. AND TRAVERSO, P. 2000. Planning as model checking. In *Proceedings of the 5th European Conference on Planning: Recent Advances in AI Planning*. Springer-Verlag, London, UK, 1–20.

GODEFROID, P. AND PITERMAN, N. 2009. Ltl generalized model checking revisited. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI '09. Springer-Verlag, Berlin, Heidelberg, 89–104.

GURFINKEL, A. AND CHECHIK, M. 2004. How vacuous is vacuous? *Tools and Algorithms for the Construction and Analysis of Systems*, 451–466.

HEAVEN, W., SYKES, D., MAGEE, J., AND KRAMER, J. 2009. Software engineering for self-adaptive systems. Springer-Verlag, Berlin, Heidelberg, Chapter A Case Study in Goal-Driven Architectural Adaptation, 109–127.

HOARE, C. A. R. 1978. Communicating sequential processes. *Communications of the ACM 21*, 666–677.

HUANG, A.-C., GARLAN, D., AND SCHMERL, B. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. In *Proceedings of the First International Conference on Autonomic Computing*. IEEE Computer Society, Washington, DC, USA, 276–277.

INVERARDI, P. AND TIVOLI, M. 2007. A reuse-based approach to the correct and automatic composition of web-services. In *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*. ESSPE '07. ACM, New York, NY, USA, 29–33.

JACKSON, M. 1995a. *Software requirements & specifications: a lexicon of practice, principles and prejudices*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

JACKSON, M. 1995b. The world and the machine. In *Proceedings of the 17th international conference on Software engineering*. ICSE '95. ACM, New York, NY, USA, 283–292.

JURDZIŃSKI, M. 2000. Small progress measures for solving parity games. In *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings*, H. Reichel and S. Tison, Eds. Lecture Notes in Computer Science Series, vol. 1770. Springer-Verlag, Lille, France, 290–301.

JUVEKAR, S. AND PITERMAN, N. 2006. Minimizing generalized bchi automata. In *Proceedings 18th International Conference on Computer Aided Verification*, T. Ball and R. Jones, Eds. Lecture Notes in Computer Science Series, vol. 4144. Springer Berlin / Heidelberg, 45–58.

KAZHAMIAKIN, R., PISTORE, M., AND ROVERI, M. 2004. Formal verification of requirements using spin: A case study on web services. In *Proceedings of the Software Engineering and Formal Methods, Second International Conference*. SEFM '04. IEEE Computer Society, Washington, DC, USA, 406–415.

KELLER, R. M. 1976. Formal verification of parallel programs. *Communications of the ACM 19*, 371–384.

KESTEN, Y., PITERMAN, N., AND PNUELI, A. 2005. Bridging the gap between fair simulation and trace inclusion. *Information and Computation 200*, 35–61.

KONIGHOFER, R., HOFFEREK, G., AND BLOEM, R. 2009. Debugging formal specifications using simple counterstrategies. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*. IEEE, 152–159.

KRAMER, J. AND MAGEE, J. 2007. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*. FOSE '07. IEEE Computer Society, Washington, DC, USA, 259–268.

KUPFERMAN, O. AND VARDI, M. 2003. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer (STTT) 4,* 2, 224–233.

KUPFERMAN, O. AND VARDI, M. Y. 2005. Safraless decision procedures. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*. FOCS '05. IEEE Computer Society, Washington, DC, USA, 531–542.

LAMSWEERDE, A. V. 2001. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*. Vol. 0. IEEE Computer Society Washington, DC, USA, IEEE Computer Society, Los Alamitos, CA, USA, 0249.

LETIER, E., KRAMER, J., MAGEE, J., AND UCHITEL, S. 2008. Deriving event-based transition systems from goal-oriented requirements models. *Automated Software Engineering 15*, 175–206.

LETIER, E. AND VAN LAMSWEERDE, A. 2002. Agent-based tactics for goal-oriented requirements elaboration. In *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. ACM, New York, NY, USA, 83–93.

LEWERENTZ, C. AND LINDNER, T., Eds. 1995. *Formal Development of Reactive Systems - Case Study Production Cell*. Lecture Notes in Computer Science Series, vol. 891. Springer-Verlag, London, UK.

LYNCH, N. AND TUTTLE, M. 1989. An introduction to input/output automata. *CWI Quarterly 2*, 219–246.

MALER, O., PNUELI, A., AND SIFAKIS, J. 1995. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*. Vol. 95. Springer, 229–242.

MANNA, Z. AND PNUELI, A. 1992. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA.

PARNAS, D. L. AND MADEY, J. 1995. Functional documents for computer systems. *Science of Computer Programming 25,* 1, 41 – 61.

PISTORE, M., BARBON, F., BERTOLI, P., SHAPARAU, D., AND TRAVERSO, P. 2004. Planning and monitoring web service composition. In *Artificial Intelligence: Methodology, Systems, and Applications*, C. Bussler and D. Fensel, Eds. Lecture Notes in Computer Science Series, vol. 3192. Springer Berlin, 106–115. 10.1007/978-3-540-30106-6_11.

PITERMAN, N., PNUELI, A., AND SA'AR, Y. 2006. Synthesis of reactive (1) designs. *Lecture notes in computer science 3855*, 364–380.

PNUELI, A. AND ROSNER, R. 1989. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '89. ACM, New York, NY, USA, 179–190.

RABIN, M. 1970. Weakly definable relations and special automata. In *Proc. Symp. Math. Logic and Foundations of Set Theory*. 1–23.

RAMADGE, P. AND WONHAM, W. 1989. The control of discrete event systems. *Proceedings of the IEEE 77,* 1, 81–98.

RUSSELL, S. AND NORVIG, P. 1995. Artificial intelligence: a modern approach. *New Jersey*.

SCHEWE, S. AND FINKBEINER, B. 2007. Bounded synthesis. In *Proceedings of the 5th international conference on Automated technology for verification and analysis*. ATVA'07. Springer-Verlag, Berlin, Heidelberg, 474–488.

SOHAIL, S., SOMENZI, F., AND RAVI, K. 2008. A hybrid algorithm for ltl games. In *9th International Conference on Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science Series, vol. 4905. Springer, 309–323.

SYKES, D., HEAVEN, W., MAGEE, J., AND KRAMER, J. 2007. Plan-directed architectural change for autonomous systems. In *SAVCBS*, A. Poetzsch-Heffter, Ed. ACM, 15–21.

UCHITEL, S., BRUNET, G., AND CHECHIK, M. 2009. Synthesis of partial behavior models from properties and scenarios. *IEEE Transactions on Software Engineering 35*, 384–406.

VAN LAMSWEERDE, A. 2009. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley.

VAN LAMSWEERDE, A. AND LETIER, E. 2000. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering 26*, 978–1005.

YOUNG, M. AND DEVANBU, P. T., Eds. 2006. *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2005, Portland, Oregon, USA, November 5-11, 2006*. ACM.

ZAVE, P. AND JACKSON, M. 1997. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM) 6*, 1–30.