Synthesis of Live Behaviour Models *

Nicolás D'Ippolito[†*] Victor Braberman[*] Nir Piterman[†] Sebastián Uchitel[†*]

[†] Imperial College London
London, United Kingdom
{npiterma, su2}@imperial.ac.uk

[*] Universidad de Buenos Aires,
Buenos Aires, Argentina
{ndippolito, vbraber}@dc.uba.ar

ABSTRACT

We present a novel technique for synthesising behaviour models that works for an expressive subset of liveness properties and conforms to the foundational requirements engineering World/Machine model, dealing explicitly with assumptions on environment behaviour and distinguishing controlled and monitored actions. This is the first technique that conforms to what is considered best practice in requirements specifications: distinguishing prescriptive and descriptive assertions. Most previous attempts at using synthesis of behavioural models were restricted to handling only safety properties. Those that did support liveness were inadequate for synthesis of operational event based models as they did not include the bespoke distinction between system goals and environment assumptions.

Categories and Subject Descriptors

D.2 [**Software Engineering**]

General Terms

Design, Algorithms

Keywords

controller synthesis, behavioural modelling

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. [32, 7, 5]) 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.

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 [2], and in particular in service oriented architectures [4]. More recently, there has been an increasing interest in self-adaptive systems [11] which must be capable of designing at runtime adaptation strategies. Consequently, such systems 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.

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 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 [25], 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 [13, 31], which have been applied in self-adaptive systems. The problem with these approaches is that the distinction between controlled and monitored actions [26], and between descriptive and prescriptive behaviour [17] 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 assumptions that were not made explicit.

Making assumptions explicit is crucial, and even more so with liveness system goals. Jackson [17], and others (e.g., [21, 26]) have 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. When dealing with liveness, assumptions play an even more prominent role: Typically, reasoning about

*This work was partially supported grants PICT-PAE 2272 37229, ERC PBM-FIMBSE, UBACyT X021, CONICET and PIP112-200801-00955. NP is supported by grant UK EPSRC EP/E028985/1

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.

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 [3, 31] that leave such assumptions implicit and do not allow for user tailored liveness assumptions entails some important risks and limitations for users.

In this paper we propose a technique for synthesising behaviour models that works for an expressive subset of liveness properties, that distinguishes between controlled and monitored actions, and differentiates between system goals and environment assumptions. The technique adapts and extends recent advances [27] in synthesis of controllers for discrete event systems [29].

More specifically, we adapt the controller synthesis technique GR(1) [27] to work in the context of event-based specifications using LTS semantics, parallel composition and to support safety properties as part of the specification. The synthesis procedure, given a descriptive specification of the environment in the form of an LTS and a set of controllable actions, constructs a behaviour model that when composed with the environment satisfies a given FLTL [12] formula of the form $\square\, I \wedge (\bigwedge_{i=1}^{n} \square \diamondsuit A_i \rightarrow \bigwedge_{j=1}^{m} \square \diamondsuit G_j)$ where $\square\, I$ is a safety system goal, $\square \diamondsuit A_i$ represents a liveness assumption on the behaviour of the environment, $\square \diamondsuit G_j$ models a liveness goal for the system and A_i and G_j are non-temporal fluent expressions [12], while I is a system safety goal expressed as a Fluent Linear Temporal Logic formula [12].

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; *(iv)* the restrictions that an event-based setting requires in order to guarantee correctness of the synthesis procedure and to avoid anomalous controllers. Interestingly, and perhaps not surprisingly, the restrictions to achieve the latter correspond to following the methodological and theoretical guidelines dictated by the goal-oriented requirements engineering approach [21].

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. We finish with a short description of a case study, discussion, related work and conclusions. Due to lack of space proofs are omitted.

2. OVERVIEW

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

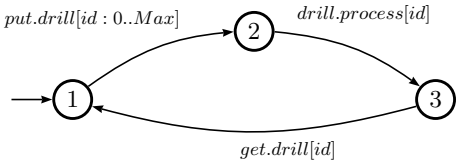

Figure 1: Drill.

Consider the following variation of the Production Cell case study [23]: 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 when a tool breaks or a new instance of an existing tool type is introduced for example), 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 requirements 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 the behaviour model for 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

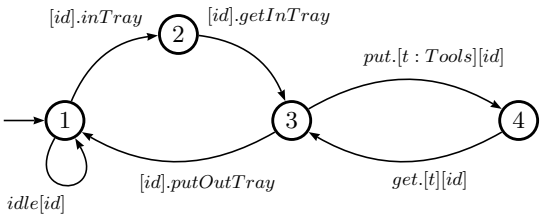

Figure 2: Products.

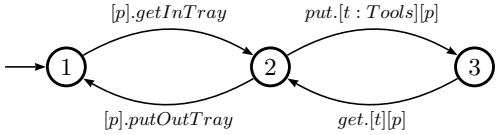

Figure 3: Robot arm.

model is at the initial state again. Note that *id* represents a particular product and $Tools$ the available tool set.

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 [17], 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 [10].

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 it is 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 system specification consists 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 products. In Figure 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 ϱ 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 $I = \varrho \wedge \psi$.

The liveness part of the system specification must capture the requirement of not indefinitely postponing the production of any product type. Such requirement can be formalised in temporal logic as follows:
$G = \bigwedge_{type \in \{A,B\}} \Box \Diamond (\bigvee_{id=0}^{MAX} AddedToOutTray(type, id))$
where $AddedToOutTray(t, i)$ 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 $\Box I \wedge \Box \Diamond G$ 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:
$As = \bigwedge_{type \in \{A,B\}, 0 \leq id \leq MAX} (\Box \Diamond AddedToInTray(type, id))$
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 controller 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 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 of synthesising controllers that satisfy the specification described above, and hence capable of, among other things, identifying the controller's need for memorising specific aspects of the system behaviour in order to satisfy liveness properties.

3. BACKGROUND

We use Labelled Transition Systems as defined below.

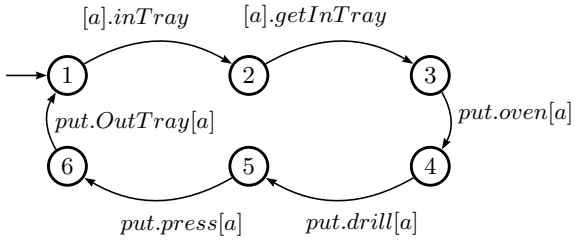

Figure 4: Specification for products of type A.

DEFINITION 3.1. (Labelled Transition Systems) *A* Labelled Transition System *(LTS) is* $P = (S, L, \Delta, s_0)$*, where* S *is a finite set of states,* $L \subseteq Act$ *is its* communicating alphabet, $\Delta \subseteq (S \times L \times S)$ *is a transition relation, and* $s_0 \in S$ *is the initial state. We denote* $\Delta(s) = \{s' \mid (s, a, s') \in \Delta\}$ *and traces*(P) *the set of traces* $t = s, \ell, s', \ell', \cdots$ *of* P*. We say an LTS is deterministic if* (s, ℓ, s') *and* (s, ℓ, s'') *are in* Δ *implies* $s' = s''$*.*

The following definition is based on that of *Interface Automata* and *Legal Environment* presented in [8].

DEFINITION 3.2. (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* $L_M = L_{M_c} \cup L_{M_u}$*,* $L_{M_c} \cap L_{M_u} = \emptyset$*,* $L_P = L_{P_c} \cup L_{P_u}$ *and* $L_{P_c} \cap L_{P_u} = \emptyset$*. We say that* M *is a legal environment for* P *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* $P' = \langle S_P, \{s_{P_0}\}, L_{P_u}, L_{P_c}, \emptyset, \Delta_P\rangle$*.*

We describe specifications (i.e. goals) using Fluent Linear Temporal Logic (FLTL) [12]. Linear temporal logics (LTL) are widely used to describe behaviour requirements [12, 22, 19]. 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 [12]. FLTL is a linear-time temporal logic for reasoning about fluents. A *fluent* fl is defined by a set of initiating actions I_{fl}, a set of terminating actions T_{fl}, and an initial value $Initially_{fl}$. That is, $fl = \langle I_{fl}, T_{fl}\rangle_{initially_{fl}}$, where $I_{fl}, T_{fl} \subseteq Act$ and $I_{fl} \cap T_{fl} = \emptyset$. When we omit $Initially_{fl}$, we assume the fluent is initially *false*. We use fl_ℓ as short for the fluent defined as $fl = \langle \ell, Act \setminus \{\ell\}\rangle$.

Given the set of fluents Φ, an FLTL formula is defined inductively using the standard boolean connectives and temporal operators $\mathbf{X}$ (next), $\mathbf{U}$ (strong until) as follows: $\varphi ::= fl \mid \neg\varphi \mid \varphi \vee \psi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\psi$, where $fl \in \Phi$. As usual we introduce $\wedge$, $\Diamond$ (eventually), and $\Box$ (always) as syntactic sugar.

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

- $Initially_{fl} \wedge (\forall j \cdot 0 \leq j \leq i \Rightarrow a_j \notin T_{fl})$
- $\exists j \cdot (j \leq i \wedge a_j \in I_f) \wedge (\forall k \in \mathbb{N} \cdot j < k \leq i \Rightarrow a_k \notin T_{fl})$

From the FLTL definition it follows that many results for LTL can be easily extended to FLTL.

DEFINITION 3.3. (Two-player Game) *A* Two-player Game *(Game) is* $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi)$*, where* S_g *is a finite set of states,* $\Gamma^-, \Gamma^+ \subseteq S_g \times S_g$ *are transition relations,* $s_{g_0} \in S_g$ *is the initial state, and* $\varphi \subseteq S_g^\omega$ *is a winning condition. We denote* $\Gamma^-(s_g) = \{s'_g \mid (s_g, s'_g) \in \Gamma^-\}$ *and similarly for* Γ^+*. A state* s_g *is* uncontrollable *if* $\Gamma^-(s_g) \neq \emptyset$ *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* $s_{g_{n+1}} \in \gamma \cup \Gamma^-(s_{g_{n+1}})$ *and adds* $s_{g_{n+1}}$ *to* p*.*

A strategy with memory Ω *for the controller is a pair of functions* (σ, u) *where,* $\sigma : \Omega \times S_g \rightarrow 2^{S_g}$ *such that* $\sigma(\varpi, s_g) \subseteq \Gamma^+(s_g)$ *and* $u : \Omega \times S_g \rightarrow \Omega$ *such that* Ω *is some memory domain with a designated start value* ϖ_0*. Intuitively,* σ *tells controller which states to enable as possible successors and* u *tells controller how to update her memory. If* Ω *is finite, we say that the strategy uses finite memory. A finite or infinite play* $p = s_{g_0}, s_{g_1}, \ldots$ *is* consistent *with* (σ, u) *if for every* n *we have* $s_{g_{n+1}} \in \sigma(\varpi_n, s_{g_n}) \cup \Gamma^-(s_{g_n})$*, where* $\varpi_{i+1} = u(\varpi_i, s_{g_{i+1}})$ *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.*

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. It is well known that if controller wins a game G and φ is ω-regular she can win using a finite memory strategy [28]. We now define the class of winning conditions φ that is of our interest.

DEFINITION 3.4. Generalised Reactivity(1) [27] *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* $gr((\phi_1, \ldots, \phi_n), (\gamma_1, \ldots, \gamma_m))$*.*

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 [17]. 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 controller that restricts the events it controls. When deployed in a suitable environment such a controller will ensure the satisfaction of a given set of system goals. Satisfaction of these goals depends on the satisfaction of prescriptive 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 [15] as the formal basis for modelling the environment and the controller to be synthesised, 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 $\mathcal{L} = \langle E, H, A_c \rangle$ 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 [28]. 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 [1] and GR(1). The latter can be seen as an extension of [1] to a more expressive liveness fragment of LTL.

We now define the SGR(1) control, 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, and 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.

DEFINITION 4.2. (SGR(1) LTS Control) *An LTS control problem $\mathcal{L} = \langle E, H, A_c \rangle$ is SGR(1) if E is deterministic, and $H = \{(\emptyset, I), (As, G)\}$, where $I = \Box\, \rho$, $As = \bigwedge_{i=1}^{n} \Box \diamondsuit \phi_i$, $G = \bigwedge_{j=1}^{m} \Box \diamondsuit \gamma_j$, 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 5(a), $A_c = \{c_1, c_2, c_3, c_4, g_1, g_2\}$, $H = \{(\emptyset, I),\ (As, G)\}$, $I = \Box \neg fl_w$, $As = \Box \diamondsuit fl_a$ and $G = \Box \diamondsuit fl_{g_1} \wedge \Box \diamondsuit fl_{g_2}$. Recall that fl_ℓ is a fluent that becomes true when ℓ occurs and becomes fals when any other action occurs.

The LTS C_1, C_2 and C_3 of Figures 5(b) to 5(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 acheived ($g1$ 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 Buchi acceptance condition. More concretely, any liveness property specifiable by a deterministic Buchi automaton can be handled by the proposed approach. The trick is, basically, to compose the Buchi 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. $\Box(\phi \rightarrow \diamondsuit \psi)$) could be treated in this way [27]. 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 ($\bigwedge_{id=0}^{MAX} \bigwedge_{type \in \{A,B\}} \Box(WaitingForProcessing(type,\ id) \rightarrow \diamondsuit\, put.OutTray(type,\ id))$ where $WaitingForProcessing(type, id)$ are fluents initiated by events $[type].[id].inTray$ and terminated by events $[type].[id].getInTray$.

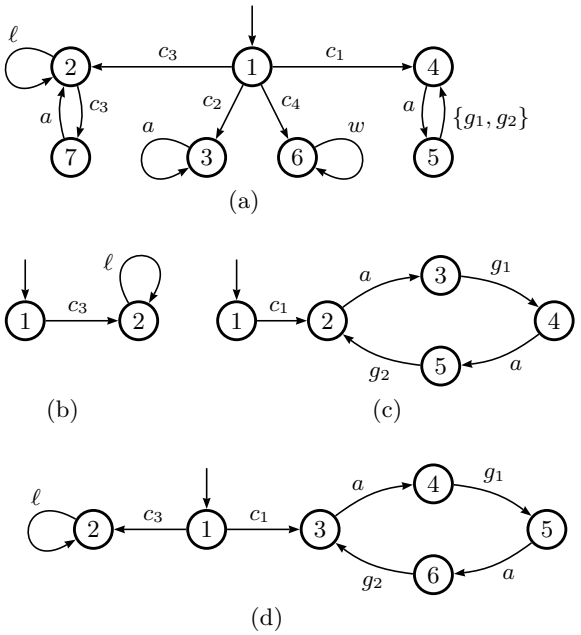

Figure 5: (a) Environment Model E (b) Controller C_1 (c) Controller C_2 (d) Controller C_3

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 $As = \bigwedge_{i=1}^{n} \Box \diamondsuit \phi_i$ 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 $\mathcal{L}$ with assumptions As and an LTS M such that M is a solution for $\mathcal{L}$, we say that M is a* best effort controller *for $\mathcal{L}$ 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 $\mathcal{L}$ 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 effor 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 $\square \diamondsuit As$. 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 $\mathcal{L}$ with assumptions As and an LTS M such that M is a solution for $\mathcal{L}$, we say that M is an* assumption preserving controller *for $\mathcal{L}$ 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 $\sigma.\sigma' \models (I \wedge As)$*

PROPERTY 4.1. *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.*

Property 4.1 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 *assumptions preserving* controller because the trace $\sigma = c_3, c_3, a, c_3, \cdots$ in E is a valid extension to $\sigma = c_3, \cdots$ 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 *assumptions preserving* controller.

Note that the *Best Effort* criterion compares two controllers while *Assumption Preserving* compares the behavior of the controlled environment against the environment.

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 [22].

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 any choice it may make and the behaviour of any controller that it might be composed with. This is ensured by checking that for every state [1] in E there is no strategy for the controller to falsify As.

DEFINITION 4.5. (Environment Assumption Compatibility) *Given an SGR(1) LTS control problem $\mathcal{L} = \langle E, H, A_c \rangle$ 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 $\langle E_s, H', A_c \rangle$ and $H' = \{(\emptyset, I), (As, false)\}$, where 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.

PROPERTY 4.2. *Given an SGR(1) LTS control problem $\mathcal{L}$ with assumptions As and environment E, if As is compatible with E then all solutions to $\mathcal{L}$ are best effort and assumption preserving.*

Note that the running example $\mathcal{R}$ violates Definition 3.2 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:
$\square\ (AddedToOutTray(t, i) \rightarrow \diamondsuit\ AddedToInTray(t, i))$.
Such a formula can, as mentioned previously and discussed in [27], be encoded into our framework.

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.

[1] The adds no computational complexity to the control problem

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 $\mathcal{L}$ (Section 5.2). We also show that our approach is sound and complete. That is, a solution to the SGR(1) LTS control problem $\mathcal{L}$ 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 $\mathcal{L}$.

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 $\mathcal{L} = \langle E, As, G, A_c\rangle$ 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 $\mathcal{L} = \langle E, As, G, A_c\rangle$, where, $E = (S_e, L, \Delta_e, s_{e_0})$, $As = \bigwedge_{i=1}^{n} \Box \Diamond \phi_i$ and G is separated in $G = \Box \rho$ and $\bigwedge_{j=1}^{m} \Box \Diamond \gamma_j$. Let $\mathit{fl} = \{\mathit{fl}_1, \ldots, \mathit{fl}_k\}$ be the set of fluents used in As and G and $\mathit{fl}_i = \langle I_{\mathit{fl}_i}, T_{\mathit{fl}_i}\rangle_{initially_i}$. 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 Proposition 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 $\Box \rho \wedge (\bigwedge_{i=1}^{n} \Box \Diamond \phi_i \implies \bigwedge_{j=1}^{m} \Box \Diamond \gamma_j)$

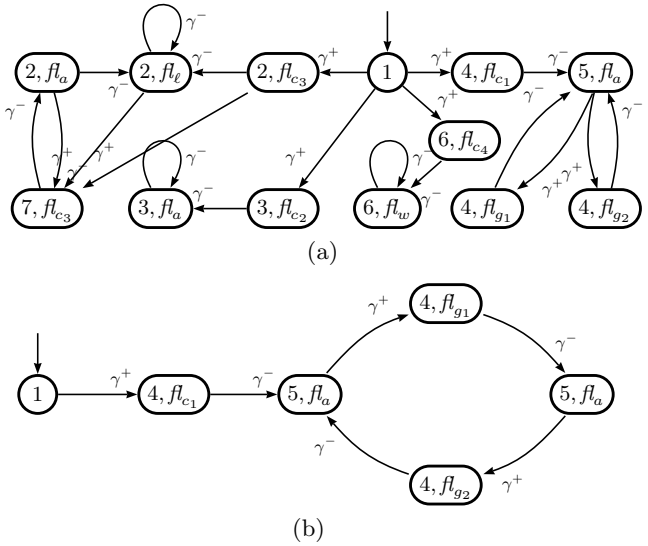

Figure 6: (a) Transition relations for the game G_R (b) Wining strategy for G_R

Figure 6(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 5(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 $\mathcal{L} = \langle E, As, G, A_c\rangle$, the game $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ obtained from $\mathcal{L}$ and a winning strategy for G, we build $M = (S_m, L, \Delta_m, s_{m_0})$ a solution to $\mathcal{L}$ 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, consider an SGR(1) LTS control problem $\mathcal{L} = \langle E, As, G, A_c\rangle$. 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 and $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ be the GR(1) game constructed from E as 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 $s'_g \in \sigma(m, s_g)$ then for every action ℓ such that ℓ is possible from s_g to s'_g we add $((m, s_g), \ell, (u(m, s_g), s'_g))$ to Δ_g. Similarly, consider a transition $(s_g, s'_g) \in \Gamma^+$. By definition of Γ^+ there is an action $\ell \in A_c$ such that ℓ is possible from s_g to s'_g. If $s'_g \in \sigma(\varpi, s_g)$ then for every action ℓ such that ℓ is possible from s_g to s'_g we add $((\varpi, s_g), \ell, (u(\varpi, s_g), s'_g))$ to Δ_g.

The initial state of M is defined as $s_{m_0} = (\varpi_0, s_{g_0})$ where ϖ_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 $\square \diamondsuit fl_a$, $\square \diamondsuit fl_{g_1}$ and $\square \diamondsuit fl_{g_1}$. The only possible solution to such requirements is to have in (σ, u), a cycle visiting $(5, fl_a)$, $(4, fl_{g_1})$ and $(4, fl_{g_2})$ in some order. A strategy satisfying this is shown in Figure 6(b). Note that some memory is needed to distinguish whether state $(4, fl_{g_1})$ or $(4, fl_{g_2})$ has to be visited after visiting $(5, fl_a)$. Finally, in Figure 5(c) we show the LTS controller obtained by applying the conversion shown above to the strategy in Figure 6(b).

In Proposition 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 $\mathcal{L}$, then the LTS M constructed as explained above is a solution to $\mathcal{L}$. Note that to prove this proposition environment (E) determinism is needed.

PROPOSITION 5.1. (Completeness) *Let $\mathcal{L}$ 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 $\mathcal{L}$. If M is a solution for the SGR(1) problem $\mathcal{L}$ 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 M.*

PROPOSITION 5.2. (Soundness) *Let $\mathcal{L}$ 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 $\mathcal{L}$, σ be a transition relation and u be an update function. If (σ, u) is winning strategy for G, and M is the LTS obtained by applying the conversion shown in Section 5.2, then it holds that M is a solution for $\mathcal{L}$.*

5.3 Implementation

The original algorithm for solving GR(1) games [27] manipulates sets of states using a symbolic representation in the form of BDDs. We implemented a rank-based algorithm, that is better suited for explicit state space representation allowing for integration within the MTSA tool set [10].

To test our implementation we applied it to the case study presented in Section 2. The size of the environment model is over 2000 states, the size of the synthesised controller is above 1000 states and it takes 3s to compute the strategy. 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 (a drill) and can only process one instance of each product type at a time. The synthesised controller is shown in Figure 7. Note the states introduced by the algorithm to remember 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 2 and 7) regardless of whether there are products of type B ready to be processed (see state 9). It then does the same for products of type B.

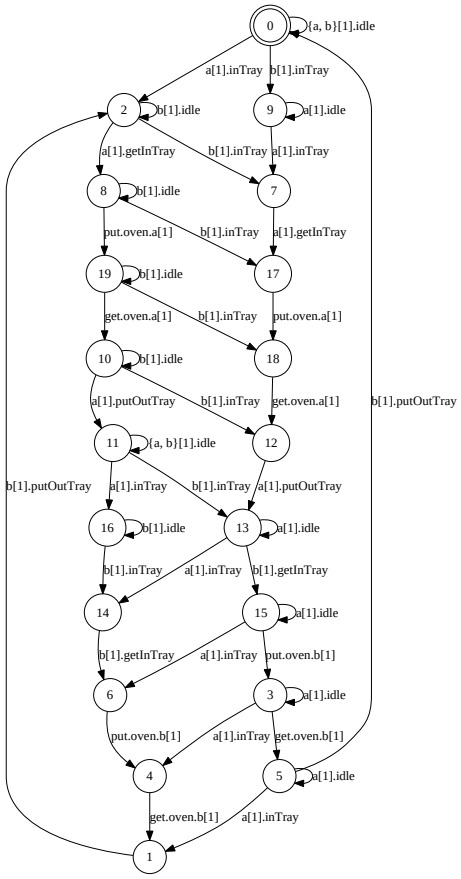

Figure 7: Controller for reduced Production Cell.

6. RELATED WORK

Our work builds on that of the controller synthesis community and particularly on the generalised reactivity synthesis algorithm GR(1) proposed in [27]. The community has largely focused on controllers for embedded systems and digital circuits, 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 being adapted to the notion of event-based communicating machines [15]. This adaptation, specifically to Labelled Transition Systems (LTS) [20] semantics and CSP-like parallel composition [15], is a contribution of this paper. The change from state-based to event-based 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 Property 5.2). The change also introduces the need for a sound methodological approach to the definition of assumptions in order to avoid anomalous controllers.

Although many behaviour model synthesis techniques have been studied (e.g. [7, 5]) these are restricted to user-defined safety requirements. The exceptions that we are aware of relate to the self-adaptive systems and the planning communities.

In the self-adaptive 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. [6, 11]). These architectures do not prescribe the mechanism for constructing adaptation strategies. The technique we propose could be used in the context of any of these architectures. In fact, we believe, that the methodological guidance that our approach offers will help integrating the controller synthesis techniques into these architectures in a sound way.

Various approaches to automated construction of adaptation strategies exists. Sykes et al. in [31] build on the "Planning as Model Checking" framework [13] to construct plans that aim to guarantee reaching a particular goal state, hence a certain liveness requirement must be dealt with. 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, hence the plans do not provide the expected guarantees.

More generally, the planning as model checking framework (e.g., [13]), 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 goals. Consequently, it does not distinguish between controllable and monitorable actions, and the plans that are generated would not be realisable by the software system.

Finally, our work is heavily influenced by the work on requirements engineering by Jackson [17] van Lamsweerde [21] and Parnas [26] who have 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.

7. FURTHER EVALUATION

In this section we show the results of applying our technique to a case study originally presented in [14] in the context of self-adaptive systems. The case study was performed using an implementation of the control synthesis algorithm described above integrated into the Modal Transition System Analyser. The modified tool together with the running example and the case study are available at [10].

Consider a situation in which a two-bedroom house has collapsed leaving only one small passage between the two rooms (refereed 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 successful or not. The situation is complicated by the presence of a door between the two rooms. The door cannot be opened by the Koala robot. However, although the structure is unstable, it is assumed that the door can be opened and closed by the trapped occupants.

A 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 $getPackage$ and sensors (e.g. $getPackageOk$, $getPackageFailed$), a model of the door (e.g. $openDoor$) and the 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 be loaded near the door and it can't move until the loading is accomplished successfully.

The aim is to automatically synthesise a controller for the robot that will achieve the task of retrieving aid packages from the outside to the room where the trapped 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. $moveNorth$, $getPackage$) excluding actions events such as $openDoor$ or $getPackageOk$.

The formalisation of the goals is divided into two parts. The safety part I prescribes the legal places for loading and unloading the robot (e.g. the robot must not unloaded packages in rooms other than the north room) The liveness part of the goal states that, infinitely often, the robot must be at the far end of north room and have just unloaded: $G = \Box \Diamond Pos4 \wedge JustUnLoadeded$.

The control problem, as defined up to this point is not solvable 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. Introducing the assumption that the door is infinitely often open ($As = \Box \Diamond DoorOpen$) is still insufficient as our implementation reports that the SGR(1) LTS control problem cannot be solved. The problem is that the robot has no control over the success or failure of attempting to load an aid package using the arm. Thus, a missing assumption stating that if the robot attempts to load a package it will eventually succeed: $\Box(getPackage \Rightarrow \Diamond GetPackageOk)$. When assumptions regarding the door being open and package loading being successful are included in the SGR(1) LTS control problem, a solution exists and is constructed automatically by the tool. It is interesting to note that the last assumption is compatible with the environment thanks to the topology of the plant. Consider the case in which the robot is near to the door it's not moving and it's unloaded. In such case the topological model indicates that the robot can't leave its position if it is not successfully loaded. Thus, after a failed loading the robot is forced to retry. Thus, 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. Therefore, this illustrates how compatibility would be actually violated and although our algorithm yields a best effort controller that could not be taken for granted.

Comparing our approach to the original case study, note that in [31] *(i)* no assumptions are explicitly given, as a result *(i)* no guarantees can be given as to wether the synthesised controller will satisfy the goal of delivering aid packages, and *(iii)* although under certain conditions the synthesised plan will work, it is not clear what those circumstances are.

8. CONCLUSIONS AND FUTURE WORK

Synthesis for liveness goals of 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 be-

tween controlled and monitored actions [26], differentiates between prescriptive and descriptive [16] aspects of the specification of system goals, environment behaviour, and environment assumptions.

We presented the *event-based* 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. We provide a solution that works in polynomial time and is based on a rank computation [18], which is more suitable for explicit state space representation. Besides, we identify a class of anomalous controllers that even correct and complete algorithms like ours might yield if no further restrictions were required for the assertions acting as liveness assumptions on the environment. Furthermore, we identify an effective condition for assumptions that rules out those anomalies.

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. Finally, we aim to validate if our definitions of controller anomalies are complete. In other words, if they match our intuitions of what a good controller is.

9. REFERENCES

[1] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *Proc. SSC*, pp. 469–474. 1998.

[2] M. Autili, P. Inverardi, M. Tivoli, and D. Garlan. Synthesis of" correct" adaptors for protocol enhancement in component-based systems. *Proc. SAVCBS*, pp. 79, 2004.

[3] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a model based planner. In *Proc. IJCAI Workshop on Planning under Uncertainty and Incomplete Information*. 2001.

[4] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proc. FSE*, pp. 141–150, 2009.

[5] Y. Bontemps, P. Schobbens, and C. Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundamenta Informaticae*, 62(2):139–169, 2004.

[6] F. Dalpiaz, P. Giorgini, and J. Mylopoulos. An Architecture for Requirements-Driven Self-reconfiguration. In *Proc. AISE*, pp. 260. Springer, 2009.

[7] C. Damas, B. Lambeau, and A. van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proc. FSE*, pp. 197–207, 2006

[8] L. De Alfaro and T. Henzinger. Interface automata. *ACM Software Engineering Notes*, 26(5):120, 2001.

[9] N. D'Ippolito. MTSA Tool., 2010. http://www.doc.ic.ac.uk/ srdipi/fse2010/.

[10] N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. MTSA: The modal transition system analyser. In *Proc. ASE*, pp. 475–476, 2008.

[11] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, pages 46–54, 2004.

[12] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. *ACM Software Engineering Notes*, 28(5):257–266, 2003.

[13] F. Giunchiglia and P. Traverso. Planning as model checking. In *Proc. ECP RAAP*, pages 1–20, 2000.

[14] W. Heaven, D. Sykes, J. Magee, and J. Kramer. A Case Study in Goal-Driven Architectural Adaptation. In *Proc. SEfSAS*, page 127. Springer, 2009.

[15] C. Hoare. Communicating sequential processes. *CACM*, 21(8):677, 1978.

[16] M. Jackson. *Software Specifications and Requirements: a lexicon of practice, principles and prejudices*. Addison-Wesley, 1995.

[17] M. Jackson. The world and the machine. In *Proc. ICSE*, pp. 283–292, 1995.

[18] M. Jurdziński. Small progress measures for solving parity games. In *Proc. STACS*, LNCS 1770, pp. 290–301, 2000. Springer-Verlag.

[19] R. Kazhamiakin, M. Pistore, and M. Roveri. "Formal Verification of Requirements using SPIN: A Case Study on Web Services". In *Proc. SEFM*, 2004.

[20] R. Keller. "Formal Verification of Parallel Programs". *CACM*, 19(7):371–384, 1976.

[21] A. V. Lamsweerde. Goal-oriented requirements engineering: A guided tour. *Requirements Engineering, IEEE International Conference on*, 0:0249, 2001.

[22] E. Letier and A. Van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proc. ICSE*, volume 24, pages 83–93, 2002.

[23] C. Lewerentz and T. Lindner, editors. *Formal Development of Reactive Systems - Case Study Production Cell*, LNCS 891. Springer, 1995.

[24] J. Magee and J. Kramer. *Concurrency: state models & Java programs*. Wiley New York, 2006.

[25] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer, 1992.

[26] D. L. Parnas and J. Madey. Functional documents for computer systems. *SCP*, 25(1):41 – 61, 1995.

[27] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. In *Proc. VMCAI*, LNCS 3855, pp. 364-381, 2006.

[28] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. POPL*, pp. 179–190, 1989.

[29] P. Ramadge and W. Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77(1):81–98, 1989.

[30] S. Russell and P. Norvig. Artificial intelligence: a modern approach. New Jersey, 1995.

[31] D. Sykes, W. Heaven, J. Magee, and J. Kramer. Plan-directed architectural change for autonomous systems. In *Proc. FSE*, pp. 15–21, 2007.

[32] S. Uchitel, G. Brunet, and M. Chechik. Behaviour model synthesis from properties and scenarios. In *Proc. ICSE*, pp. 34–43, 2007.

[33] A. van Lamsweerde and E. Letier. "Handling Obstacles in Goal-Oriented Requirements Engineering". *Trans. on SE*, 26(10):978–1005, 2000.