Synthesis of Live Behaviour Models for Fallible Domains *

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

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

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

[‡]University of Leicester,
Leicester, United Kingdom
{np183@le.ac.uk

ABSTRACT

We revisit synthesis of live controllers for event-based operational models. We remove one aspect of an idealised problem domain by allowing to integrate failures of controller actions in the environment model. Classical treatment of failures through strong fairness leads to a very high computational complexity and may be insufficient for many interesting cases. We identify a realistic stronger fairness condition on the behaviour of failures. We show how to construct controllers satisfying liveness specifications under these fairness conditions. The resulting controllers exhibit the only possible behaviour in face of the given topology of failures: they keep retrying and never give up. We then identify some well-structure conditions on the environment. These conditions ensure that the resulting controller will be eager to satisfy its goals. Furthermore, for environments that satisfy these conditions and have an underlying probabilistic behaviour, the measure of traces that satisfy our fairness condition is 1, giving a characterisation of the kind of domains in which the approach is applicable.

Categories and Subject Descriptors

D.2 [**Software Engineering**]

General Terms

Design, Algorithms

Keywords

controller synthesis, behavioural modelling

1. INTRODUCTION

We are interested in the automated construction of operational event-based models from specification of intended system behaviour. This topic has been the subject of extensive study in the software engineering community. Consider for example the research on synthesis from scenario-based (e.g. [27, 3]) and declarative (e.g. [17]) specifications. The aim is to provide an operational model that will support requirements elicitation and analysis. Analysis techniques may include model checking, simulation, animation and inspection (aided by automated slicing and abstraction).

Behaviour model synthesis is also used to automatically construct plans that are then enacted by a software component. For instance, synthesis of glue code and component adaptors has been studied. The main aim of these studies has been to achieve safe composition at the architecture level, and in particular in service oriented architectures (e.g. [14]). Also, there has been an increasing interest in applications to self-adaptive systems [10, 26]. All these systems rely heavily on controller synthesis techniques [24]. Such techniques guarantee the satisfaction of safety and even liveness [22] requirements. The proposed solutions work within the constraints enforced by the problem domain, the capabilities offered by the self-adaptive system, and under fairness and progress assumptions on the controller's environment. Recently, we suggested an approach for synthesis [5] in the context of discrete event systems. Our work emphasises the importance of explicit distinction, in controller synthesis, between controlled and monitored actions [19] and between descriptive and prescriptive behaviour [15]. We also provided appropriate methodological guidelines.

One of the limitations of existing synthesis techniques is that they are designed to work in the context of idealised problem domains. Situations in which the outcome of controller actions are not guaranteed are dealt with by assuming that controller actions never fail (e.g. [5]), by not-considering liveness goals (e.g. [14, 26]), or by building controllers that aim to be live but are not guaranteed to be so [10]).

In this paper we propose a technique for synthesising *live* behaviour models in the context of problem domains in which controlled actions can fail. The technique adapts and extends our previous work on synthesis of controllers for discrete event systems [5]. A key insight is the identification of a realistic fairness condition, *strong independent fairness*, which allows for a polynomial treatment of failures. In contrast, the complexity of the general problem is exponential.

Specifically, we consider models in the form of Labelled Transition Systems (LTS). We distinguish controllable from uncontrollable actions. In addition, some controllable actions have associated actions that constitute a success or a failure. The synthesis problem calls for the construction of a

*This work was partially supported by grants ERC PBM-FIMBSE, CONICET PIP955, UBACYT X021, and PICT PAE 2272.

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

model that when composed with its environment satisfies a given specification in FLTL [9]. The FLTL formulas we consider have 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 controller's environment, and $\Box \diamondsuit G_j$ models a liveness goal for the system. The expressions A_i and G_j are non-temporal fluent expressions [9]. The system safety goal, I, is expressed as a Fluent Linear Temporal Logic formula. We assume strong independent fairness of the successes and failures with respect to certain assumptions on the environment. Intuitively, the strong independent fairness condition states that every failure and every assumption must occur fairly (infinitely often if enabled infinitely often) but also independently of all other failures and assumptions and of the state of the environment and the controller. In other words, failures and assumptions cannot be coordinated. They must be "controlled" by different agents which must be oblivious to each other.

Technical contributions of this paper include *(i)* a discussion of the fairness conditions required for problem domains with failures. In particular the observation that strong fairness [8] of successful controlled actions may be insufficient to guarantee that reasonable controllers are synthesised; *(ii)* novel fairness conditions, i.e. t-strong fairness and strong independent fairness that are stronger than strong fairness and are good fits for realistic controller synthesis settings, *(iii)* the definition of a polynomial time LTS control problem, named *RSGR(1)* that supports safety and GR(1)-like liveness properties; *(iv)* the restrictions that an environment model requires in order to guarantee correctness of the synthesis procedure and to avoid controllers that fulfill their specification by trivialising it (i.e. anomalous controllers); and *(v)* a proof that if the environment can be thought of as a grounding of a probabilistic environment with non-zero probabilities on transitions, then the traces that are not strong independent fair have probabilistic measure zero, thus providing a characterisation of the domains in which our approach can be applied.

The paper is organised as follows. In Section 2 we motivate and present an overview of the approach. Section 3 includes the necessary background. In Section 4 we present the technique for synthesising LTS controllers in the presence of failures: in subsection 4.1 we discuss the notion of fairness required for the proposed controller synthesis technique; in subsection 4.2 we formalise the control synthesis problem that handles domains with failures and discuss how it can be solved efficiently; in subsection 4.3 we discuss the problem of anomalous controllers and how to avoid them; and in subsection 4.4 we present a probabilistic argument to show that behaviour that is not strong independent fair is irrelevant in the context of our synthesis approach. Finally, we report on case studies, discussion, related work and conclusions. Proofs can be found in [1].

2. MOTIVATION

In this section we discuss motivation for our approach. Technical details are provided in the next sections.

Consider the following simplified scenario: A travel agency wants to sell vacation packages on-line by orchestrating existing web-services for flight purchase, car hire and hotel booking. We want *an automated or semiautomated technique for building the agency's orchestration*, based on the known usage protocols for individual services and on the

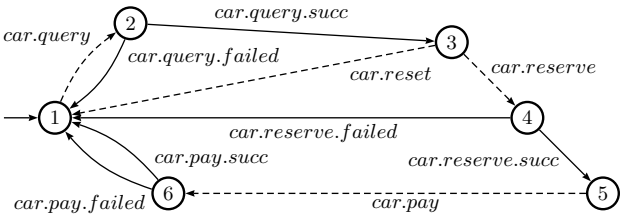

Figure 1: Car Booking Service.

agency's own requirements for the provision of packages.

An example of what the protocol for a car rental web-service is the one depicted in Figure 2. The service requires a query with information on dates, car type, and other preferences ($car.query$). The service can either respond with a list of items satisfying the specified criteria ($car.query.succ$) or with not-found ($car.query.failed$). Subsequently, if a list is retrieved, a particular item can be reserved ($car.reserve$) or the process can be aborted ($car.reset$). Reservation can fail ($car.reserve.failed$) or succeed ($car.reserve.succ$). In the latter case, payment is enabled ($car.payment$) and can succeed ($car.payment.succ$) or fail ($car.payment.\ failed$).

The web-service protocols for hotel and flight bookings will typically be similar to that of car rentals: a sequence of actions is required to progress towards a purchase and a number of problems may arise, which lead to the failure of these actions (no flights, communication errors, insufficient funds, etc.). Without loss of generality, the protocol for hotel and flight bookings is analogous to that for cars with actions such as $flight.query$ and $hotel.reserve.succ$.

The problem for the travel agency orchestration is to coordinate the individual services in order to provide a cohesive comprehensive vacation package web-service. For instance, it must attempt to avoid booking hotel and car for a customer when no flights are available for the desired dates. Such a requirement can be formalised, for instance, in temporal logic as $I_1 = \Box(\forall srv \in Services \cdot TryToBuy(srv) \Rightarrow AllReserved)$. Another requirement, if the agency is charged for reservations, might be not to reserve before all queries have returned viable items: $I_2 = \Box(\forall srv \in Services \cdot TryToReserve(srv) \Rightarrow AllFound)$.

The agency should coordinate the services to achieve its own requirements. It should do that while following the protocols of the services and deal with the various failures that may occur. For instance, if a failure occurs when reserving a flight, then the Flight service must be re-queried; but if the result of the new query returns $notFound$ then reservations for car and hotel must be cancelled (and the user may consider a different holiday).

Finally, the travel agency orchestration must be live. That is, it must actually succeed in providing package holidays. Of course this depends on actually having requests pending to be processed. Such a requirement should be formalised distinguishing the assumptions on the environment ($A_1 = \Box \diamondsuit PendingPackageRequests$) and prescriptions on the orchestration ($G_1 = \Box \diamondsuit package.deliver$). We require that if the environment satisfies A_1 then the orchestration will satisfy G_1.

We distinguish between the travel agency's controlled and monitored actions (double and single lines in figures, respectively). Actions such as $car.query$, $car.reset$, $flight.reserve$, and $hotel.payment$ are controlled by the orchestrator for the travel agency while the rest are monitored. With such distinction we apply controller synthesis. We attempt to pro-

duce a controller such that, when interacting with the Car, Flight and Hotel services, will achieve safety (e.g. I_1, I_2) and liveness (e.g. G_1) under relevant assumptions (e.g. A_1).

Unfortunately, our previous approach [5] cannot produce controllers that guarantee such goals. This is quite reasonable as for achieving such a goal, the controller must rely on a number of domain assumptions. For instance, it cannot be the case that queries, reservations and payments always fail. Under such assumptions it would seem feasible to construct a controller for the travel agency: the controller would have to retry actions in the case of failures knowing that after some number of reattempts it will succeed[1]. The assumption mentioned (if the controller tries often enough, it will eventually succeed) is a typical fairness condition sometimes referred to as strong fairness [8]. Strong fairness is not supported by polynomial time algorithms such as [5]. It requires exponential algorithms such as [7]. It could be argued that many exponential worst case algorithms are well-behaved in practice. Unfortunately, this is not the case here. The best case complexity of all known algorithms that deal with strong fairness is exponential [21]. More precisely, the size of the controller is always $N \times k!$ where N is the size of the environment model and k is the number of strong fairness conditions. The best time complexity of all known algorithms is $N^k \times k!$. Again, the $k!$ factor is never reduced. In other words, unlike symbolic model checking in which many practical settings are not worst case, here space and time blow up in every reasonable sized example.

Interestingly, strong fairness assumptions on success of queries, reservations and payments are insufficient to achieve the goals. The (strong fair) behaviour in which failures "take turns" would prevent achieving the goal. Consider the scenario in which the controller first queries a car successfully and then fails querying for a hotel. The controller must reset the car service and re-query for cars and hotels. But now the hotel query succeeds and the car query fails forcing the controller to reset the hotel service, and so on. Thus, a synthesis algorithm relying on strong fairness would declare that no controller realising this goal exists.

In conclusion, it would seem possible to build a reasonable orchestration of the services towards achieving the goals of the travel agency. However, non-trivial assumptions on the environment behaviour are required to guarantee such goals are achieved by a reasonable controller. This lays out two research questions. Firstly, *how can an orchestration for the travel agency be constructed automatically* and secondly, *what are the required assumptions*, which will enable to guarantee the goals.

In Section 4 we present a technique for automatically synthesising controllers for settings such as the travel agency. We handle cases where the environment can exhibit failures to controller actions, and show what are the environment assumptions required for such controllers to succeed.

3. BACKGROUND

In this section we present background for controller synthesis in the context of event-based operational models. We assume the problem domain for which a controller is to be built is described as a labelled transition system.

[1]Note that even in this simple example retrying is not trivial. For example, if a payment fails it is necessary to re-query and re-reserve before re-attempting to pay.

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* $\Delta(s) = \{s' \mid (s, a, s') \in \Delta\}$*. We say an LTS is deterministic if* (s, ℓ, s') *and* (s, ℓ, s'') *are in* Δ *implies* $s' = s''$*. An execution of* P *is a word* $s_0, a_0, s_1, \ldots$ *where* $(s_i, a_i, s_{i+1}) \in \Delta$*. A word* π *is trace of* P *if there is an execution* ε *of* P *such that* $\varepsilon|_L = \pi$*. We define* $tr(P)$ *to define the set of traces of* P*.*

We describe specifications (e.g. the prescriptions for controller) using Fluent Linear Temporal Logic (FLTL) [9]. Linear temporal logics (LTL) are widely used to describe behaviour requirements [9]. 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 $\dot{\ell}$ 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) and $\mathbf{U}$ (strong until) as follows: $\varphi ::= fl \mid \neg\varphi \mid \varphi \vee \psi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \psi$, where $fl \in \Phi$. We introduce $\wedge$, $\Diamond$ (eventually), and $\Box$ (always) as usual.

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:

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

The problem of controller synthesis can be expressed as follows: Given an LTS model E of the environment, a set of controllable actions L_c, assumptions As_i and goals G_i expressed in FLTL, build an LTS M such that when composed in parallel with E (i.e. $E \| M$), the controller does not block all non-controlled actions in the environment and for every trace of $E \| M$ if the trace satisfies the assumption A_i, then the trace also satisfies the goal G_i.

We use a standard definition of parallel composition [13]. The parallel composition is the LTS that models the asynchronous execution of composed models. It interleaves non-shared actions and forces synchronisation on shared actions. The notion of a controller that does not block the actions of the environment that it does not control is built on that of *legal environment* for Interface Automata [4]. Intuitively, it says that M is a legal environment for E if in every state (m, e) of $M \| E$ where m and e are states of M and E respectively, if an action a not controlled by M is enabled in e then it is also enabled in (m, e).

DEFINITION 3.2. (LTS Control) *Given a specification for a problem domain in the form of an environment LTS* E*, a set of controllable actions* L_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* $\mathcal{L} = \langle E, H, L_c \rangle$ *is to find an LTS* M *with controlled actions* L_c *and uncontrolled* $\overline{L_c}$ *such that* M *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 the synthesis problem is 2EXPTIME complete [23]. 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 GR(1) [22]. We have presented an adaptation of GR(1) in the context of LTS in [5]. It is defined as follows:

DEFINITION 3.3. (SGR(1) LTS Control) *An LTS control problem $\mathcal{L} = \langle E, H, L_c \rangle$ 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 ρ, ϕ_i and γ_j are Boolean combinations of fluents.*

4. SYNTHESIS FOR FALLIBLE DOMAINS

We now present a technique for synthesising controllers even when their environment exhibits failures. In Subsection 4.1 we discuss the notion of fairness required for the proposed controller synthesis technique. In Subsection 4.2 we formalise the control synthesis problem and discuss how it can be solved efficiently. We then discuss in Subsection 4.3 the problem of anomalous controllers and how to avoid them, and finally in Subsection 4.4 we present a probabilistic argument to show that non-fair environment behaviour is irrelevant in many realistic settings.

Note that the examples in this section are simplistic and with obvious solutions to allow conveying the main issues involved in controller synthesis in domains with failures.

4.1 Fair Environments

We consider controller synthesis in the context of environments that exhibit failures. We call this setting *synthesis with failures*. We present examples showing that fairness of failures and successes is both necessary and subtle. A malicious environment typically cannot be controlled to achieve the goals. However, we propose realistic fairness assumptions that allow for controllers that behave as expected, i.e., do not give up and keep retrying.

Consider E, the simple environment model in Figure 2, where a ceramics cooking process is described. The aim of the controller is to produce cooked ceramics by taking raw pieces from the in-tray, placing them in the oven and moving them once cooked to a conveyor belt. A natural solution for such a problem is to attempt to build a controller (using SGR(1)) with a liveness goal $G = \Box \Diamond \dot{moveToBelt}$ and an assumption $A = \Box \Diamond \neg \dot{cooking}$. Note that the assumption A is required to ensure that the controller's environment progresses when cooking ceramics; without the assumption no controller can guarantee production of ceramics. Indeed, a controller for this trivial problem is the one that chooses to *cook* rather than be *idle*, and is constructed automatically by solving the $SGR(1)$ problem $sg_1 = \langle E, H, L_c \rangle$, where $H = \{(A, G)\}$ and $L_c = \{idle, cook, moveToBelt\}$. The solution to sg_1 is a controller M that (composed with E) produces infinitely many cooked pieces if the oven finishes cooking infinitely often (i.e. $E \| M \models \Box \Diamond A$ implies $E \| M \models \Box \Diamond G$).

A slight twist to the ceramics cooking problem is the scenario in which some pieces may break during cooking. The reasons for why the pieces may break (e.g. impurities in the ceramics, heat stability in oven, etc) are abstracted in the model (Figure 3). Such abstraction of the causes for failure is a common approach to behaviour modelling of problem domains. The assumption is not that the controller's environment chooses whether the piece breaks, rather that the choice is made by a number of factors that are beyond the scope of the model.

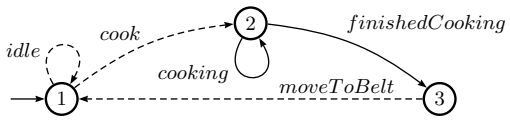

Figure 2: Ceramic Cooking Process.

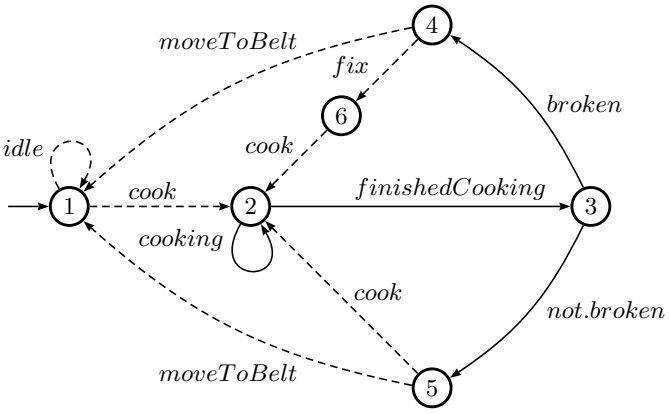

Figure 3: Failing Ceramic Cooking Process (E_2).

We distinguish failures from other actions as follows. For each control problem we define a set of try-response triples. Such a triple captures the relation between controlled actions and their success or fail reactions. Note that we require 1) the "try" action to be controlled, 2) all actions in a try-response triple to be unique with respect to other triples in the set, 3) re-tries cannot occur before a response, 4) responses can only occur as a result of a try, 5) maximum of one response occurs for every try, and 6) the decision of whether to fail or succeed cannot be enforced by other actions, hence failure is enabled if and only if success is enabled.

DEFINITION 4.1. (Try-Response) *Given an LTS $M = (S_M, L_M, \Delta_M, s_{M_0})$, where $L_C \subseteq L_M$, we say that a set $T = \{(try_i, suc_i, fail_i)\}$ is a* try-response *set for M if the following hold for all i:*

1. $try_i \in L_C$, $suc_i, fail_i \in L \backslash L_C$ and $suc_i \neq fail_i$,
2. For all $j \neq i$, $\{try_i, suc_i, fail_i\} \cap \{try_j, suc_j, fail_j\} = \emptyset$,
3. $\neg(\dot{fail_i} \vee \dot{suc_i}) W \dot{try_i}$,
4. $\Box(\dot{try_i} \implies \bigcirc(\neg \dot{try_i} \mathrm{W} (\dot{fail_i} \vee \dot{suc_i})))$,
5. $\Box((\dot{fail_i} \vee \dot{suc_i}) \implies \bigcirc(\neg(\dot{fail_i} \vee \dot{suc_i}) \mathrm{W}\ \dot{try_i}))$, *and*
6. *For all $s \in S_M$, $fail_i$ is enabled from s iff suc_i is enabled from s.*

We return to the ceramics cooking problem and add a failure to it. Consider the model of Figure 3 with the try-response set $T = \{(cook, not.broken, broken)\}$. The controller for this problem is required to accomplish two things. First, to produce cooked pieces and place them on the conveyor belt. Second, to ensure that only unbroken pieces are placed on the belt while broken pieces are fixed and re-cooked.

A naive attempt to build a controller for the modified problem simply adds a safety goal $S = \Box\ \dot{moveToBelt}$

$\Rightarrow \neg Broken$ to sg_1, where $Broken$ is a fluent defined as $\langle$ *broken*, *cook*$\rangle$. In other words, attempting to solve $sg_2 = \langle E_2, H, L_c\rangle$, where $H = \{(\emptyset, S), (A, G)\}$.

Unfortunately, sg_2 does not have a solution. Furthermore, in general, there is no controller that will work if the environment is malicious. A controller cannot succeed if its environment breaks all ceramics. In other words, for a controller to produce cooked unbroken ceramics we must assume that if enough pieces are cooked, one will eventually not break. That is, that the response to trying *cook* is not always the failure *broken*. This could be a reasonable assumption for the problem domain. Another attempt at automatically building a controller could be to strengthen the assumption A in sg_2 to be $A' = \Box \Diamond \neg cooking \wedge \Box \Diamond not.broken$. This leads to $sg_3 = \langle E_2, H, L_c\rangle$, where $H = \{(\emptyset, S), (A', G)\}$.

The problem with sg_3 is that it admits as a solution a controller M which only does *idle*. This is because by never performing *cook*, the assumption A' and more specifically $\Box \Diamond not.broken$ does not hold. Hence, the controller has no obligation to achieve G. Formally, $E_2 \| M \models \Box \Diamond A$ implies $E_2 \| M \models \Box \Diamond G$ holds if $E_2 \| M \not\models \Box \Diamond A$. Clearly, A' is not a reasonable assumption for the controller's environment. The environment depends on the controller to achieve A'. Or in van Lamsweerde's terms [16], the assumption is not realisable by the controller's environment. As we show in [5], unrealisable assumptions, in addition to not following best practices in Requirements Engineering, can lead to controllers that satisfy their goals vacuously. Just like the controller that always idles in our example satisfies its specification vacuously (see also Subsection 4.3).

In order to introduce an assumption that is realisable by the controller's environment, we must state that if pieces are cooked infinitely often, *not.broken* is taken infinitely often (i.e. $\Box \Diamond cook \Rightarrow \Box \Diamond not.broken$). However, this condition amounts to requiring strong fairness [8] of *not.broken* actions which cannot be encoded in SGR(1). Although more general controller synthesis techniques can deal with strong fairness [7], these take the algorithmic complexity of synthesis from polynomial (the SGR(1) case), to exponential. Moreover, sometimes strong fairness is not sufficiently strong for synthesising controllers in simple, yet common, problem domains. This is shown in the next example.

Consider another variation of the ceramic cooking problem in which pieces must be cooked twice before being moved to the conveyor belt. A controller for such a problem will need to "remember" how many successful consecutive *cook*'s have occurred. Requiring strong fairness on try-response triples of $T = \{(cook, not.broken, broken)\}$ is insufficient to allow the construction of a controller that achieves its goals. There is no controller that can deal with the case in which pieces break at least once every two consecutive attempts to cook them. For instance, consider M a potential controller for the problem (Figure 4). It is possible to construct a strongly fair trace by always succeeding in the first cook (taking the *not.broken* transition from state 3) but always failing after the second cook (taking the *broken* transition in state 7). If an infinite number of *cook* are tried then $\Box \Diamond cook \Rightarrow \Box \Diamond not.broken$ holds, yet the controller never succeeds in placing a twice cooked unbroken piece on the conveyor belt.

The above example shows that a stronger notion of strong fairness is required. Informally, it should state that every individual attempt to cook should be treated fairly. That is,

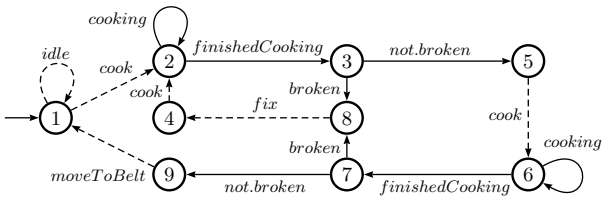

Figure 4: Ceramics cook-twice controller.

attempting first cook of a piece (transition from 1 to 2 in Figure 3) infinitely often should yield an infinite number of non broken once-cooked pieces (transition 3 to 5) and attempting a second cook of a piece (transition 5 to 6) infinitely often should yield an infinite number of non broken twice-cooked pieces (transition 7 to 9).

This stronger notion of fairness is in fact tightly coupled with the structure of the environment and controller behaviour models. What is needed is that for every global state (a state of $E_2 \| M$), if a cook on that state is attempted infinitely often then the cooking process will not fail infinitely often. An alternative intuition is that the decision whether to fail the cooking process should be fair and be taken independently of state of the environment model or the controller. In the two-cooks-a-piece example, the decision to fail the second cook of a piece process should be fair and independent of the first *cook* on the same piece.

The following definition captures this stronger notion of fairness. It requires that for every transition labelled with a *try*, if it is taken infinitely often then infinitely often *success* occurs before another *try*.

Definition 4.2. (t-strong fairness) *Given an LTS M and a try-response T for M. A trace $\pi \in tr(M)$ is* t-strong fair *with respect to M and T if for all $(try_i, suc_i, fail_i) \in T$ and for all transitions $t = (s, try_i, s')$ the following holds: $\pi' \models \Box \Diamond try'_i \Rightarrow \Box \Diamond(\neg try_i \text{ U } suc_i)$, where $\pi' = \varepsilon'|_{L_M \cup \{try'_i\}}$, $\varepsilon' = \varepsilon|_{[s.try_i.s'/s.try_i.try'_i.s']}$, and ε is an execution of M such that $\varepsilon|_{L_M} = \pi$.*

Note that $w|_A$ is the projection of word w over the alphabet A, and $w_{[v/v']}$ is the result of replacing in word w all occurrences of word v with v'.

One issue remains regarding the fairness conditions that are relevant to enable automated synthesis with failures.

Consider the synthetic example in Figure 5. In this example *try* is the only controlled action, $(try, succ, fail)$ the only try-response triple, ℓ is an arbitrary event, and G and A represent goals and assumptions respectively. The trace $try, success, \ell, try, fail, A, try, success, \ell, \ldots$ is an example of a trace that satisfies strong fairness and t-strong fairness, the assumptions hold infinitely often and yet the goal is never achieved. Note that no controller could prevent this trace as *try* is the only controlled action. The trace shows some peculiar behaviour: the environment never chooses to take the assumptions on state 3, and it can do so because it relies on the fact that it fails sometimes and through failing achieves its assumptions.

Although contrived, the example shows that the assumptions and failures can be systematically combined to make a controller unsuccessful: the environment can avoid assumptions when actions succeed (state 3 in Figure 5) and achieve assumptions when actions fail (state 5). However, a natural expectation is that the assumptions on the environ-

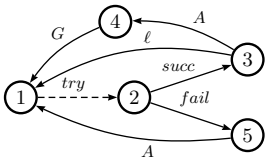

Figure 5: t-strong fairness is not enough.

ment should be independent of failures; particularly because the choice of failure or success is understood as nondeterministic given that it abstracts the actual cause for failure and success.

If required to construct a controller for Figure 5 what should the controller do? Naturally, the controller should keep taking *try* hoping that eventually assumptions are not coordinated with failures. A synthesis algorithm that only assumes strong fairness or even t-strong fairness would say that this is impossible and fail to produce a controller. Our goal is then to come up with a setting in which such a controller would be automatically generated by the synthesis algorithm. In order to do so we formalise the notion that assumptions and failures must be independent.

We formalise that assumptions and failures must be independent of each other in the following way. We restrict traces of interest to those that satisfy that assumptions must be attainable infinitely often without seeing failures. Or more precisely, if the controller tries often enough, then not only will it succeed but also it will succeed and all assumptions are fulfilled. That is, if assumptions and failures are truly independent, trying often enough guarantees that at some point after a try, no failures will occur until all assumptions are satisfied.

DEFINITION 4.3. (Strong Independent Fairness) *Given an LTS M, a try-response T for M and A a set of FLTL formulas. A trace $\pi \in tr(M)$ is* Strong Independent Fair *with respect to A if for all $(try_i, suc_i, fail_i) \in T$ and for all transition $t = (s, try_i, s')$ the following holds: $\pi' \models \Box \Diamond try'_i \Rightarrow \Box \Diamond((\neg try_i \ \mathrm{U} \ suc_i) \wedge (\bigwedge_{i=1}^{n}(\neg(\bigvee_{j=1}^{n} fail_j)\mathrm{W}A_i)))$, where $\pi' = \varepsilon'|_{L_M \cup \{try'_i\}}$, $\varepsilon' = \varepsilon|_{[s.try_i.s'/s.try_i.try'_i.s']}$, and ε is an execution of M such that $\varepsilon|_{L_M} = \pi$.*

In the next subsection we formalise the control problem with the fairness discussed above. We show that this problem can be solved efficiently by encoding it into the SGR(1) control problem. The encoding relies on strong independent fairness. Finally, as further motivation, we reason about domains that are considered as probabilistic (with non-zero probabilities on all transitions). We show that in such domains, if the environment is well structured, then the probabilistic measure of traces that do not satisfy this fairness conditions (and consequently the traces for which controllers have no obligations) is zero.

4.2 Recurrent Success Control Problem

We now formalise the *recurrent success control problem*. For traces that are strong independent fair, it guarantees general safety and liveness properties, which are GR(1)-like. We extend the SGR(1) control problem we defined in [5] by introducing failures and expectations on the fairness of the environment.

DEFINITION 4.4. (Recurrent Success) *Given an SGR(1) LTS control problem $\mathcal{L} = \langle E, H, L_C \rangle$ and a try-response T for $\mathcal{L}$, the solution for the* Recurrent Success *control problem $\mathcal{R} = \langle \mathcal{L}, T \rangle$ is to find an LTS M such that M with controlled actions L_c and uncontrolled actions $\overline{L_c}$ is a legal environment for E, $E||M$ is deadlock free, and for every pair $(As_i, G_i) \in H$, for every $(try_i, suc_i, fail_i)$ and for strong independent fair trace π in $M||E$ the following holds: if $\pi \models As_i$ then $\pi \models G_i$.*

Notice the requirement of independence between decisions on when to fail and when to achieve assumptions. This is key to the tractable treatment of RSGR(1) problems: RSGR(1) can be reduced to a SGR(1) problem leading to more efficient algorithms than those needed to solve strong fairness in general.

THEOREM 4.1. *Given $\mathcal{R} = \langle \mathcal{L}, T \rangle$ an RSGR(1) control problem, it holds that there exists an SGR(1) control problem $\mathcal{S}$ such that $\mathcal{R}$ is realisable iff $\mathcal{S}$ is realisable. Furthermore, the controller extracted from $\mathcal{S}$ can be used to control $\mathcal{R}$.*

The reduction can be explained in two steps: RSGR(1) can be solved by constructing a controller for an alternative control problem named Finitely Many Failures (FMF). Solutions for FMF control problems construct controllers that guarantee $\Box \Diamond G$ on a trace if on the same trace $\Box \Diamond As_i$ holds and also a finite number of failures occur (i.e. $\Diamond \Box \neg \bigvee_j fail_j$). An FMF problem can be coded as an SGR(1) problem where the goal is $\Box \Diamond (G \vee \bigvee_j fail_j)$.

The key to the coding of RSGR(1) into FMF is the strong independent fairness requirement, and in particular what it adds on top of t-strong fairness: if a *try*-transition is taken infinitely often, then not only will it succeed infinitely often but also that infinitely often no failures will be observed (for that *try* or any other action that can potentially fail) until all assumptions have occurred.

We sketch the proof that every solution to FMF is a solution to RSGR(1). Suppose, by way of contradiction, that M is an FMF-controller that is not an RSGR(1)-controller. Then there must be a strong independent fair trace π in $E||C$ that satisfies the assumptions infinitely but not the goals. In π there must be an infinite number of failures (otherwise it would be a counter-example to M being an FMF controller) and hence there must be at least one *try*-transition taken infinitely often. As π is strong independent fair, the *try*-transition must be successful and infinitely often no failures occur before assumptions occur. Hence, there is cycle covered by π in which no failures occur, all assumptions do occur and goals are not achieved. This cycle can be used to construct a trace in $E||M$ which has finitely many faults and in which goals are not achieved even though assumptions hold. This contradicts that M was assumed to be an FMF-controller.

We give an alternative intuition of why RSGR(1) can be reduced to FMF. In FMF the controller knows that at some point there will be no more failures but does not know at which point this will happen. It follows that its strategy is to reattempt knowing that eventually all its attempts will be successful. The same strategy works for RSGR(1). Indeed, because of strong independent fairness, it may be the case that failures are infrequent enough and non-systematically occurring. In such cases eventually all the successes needed to achieve the goals will occur "consecutively" (i.e. with no failures occurring before reaching the goal).

The MTSA tool set[6] implements RSGR(1).

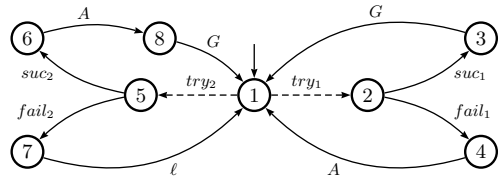

Figure 6: Environment E.

4.3 Anomalous Controllers

Anomalous controllers, as we defined in [5], are an important issue to consider in the context of automatic controller synthesis. Intuitively, an anomalous controller tries to discharge its obligation of achieving goals by preventing the environment from fulfilling its own obligations.

We revisit the issue of anomalous controllers for RSGR(1). In RSGR(1) the controller may discharge its obligation to achieve goals by either preventing assumptions from occurring or by forcing non strong independent fair traces.

The following definition of best effort controller extends that of [5] for domains with failures. It states that a controller for an RSGR(1) problem is best effort if it prevents infinitely many occurrences of assumptions and strong independent fair traces "as least as possible". That is, every other controller prevents these cases as much as the best effort one or more. Informally, the definition states that for every point at which it is no longer possible to satisfy the assumptions infinitely often or it is not a strong fair independent trace, the same would occur for every other controller.

Definition 4.5. (Best-Effort Controller) *Let $\mathcal{R}_\mathcal{S}$ be an RSGR(1) LTS control problem with assumptions A_s. We say that a solution M for $\mathcal{R}_\mathcal{S}$ is a* best effort controller *for $\mathcal{R}_\mathcal{S}$ if for all finite traces $\sigma \in traces(E\|M)$ such that for all σ' where $\sigma.\sigma' \in traces(E\|M)$, we have $\sigma.\sigma' \models (\neg \bigwedge_{i=1}^{n} \Box \Diamond A_i)$ or $\sigma.\sigma'$ is not strong independent fair, then for all other solutions M' to $\mathcal{R}_\mathcal{S}$ such that $\sigma \in traces(E\|M')$, every σ'' such that $\sigma.\sigma'' \in traces(E\|M')$ either $\sigma.\sigma'' \models (\neg \bigwedge_{i=1}^{n} \Box \Diamond A_i)$ or $\sigma.\sigma''$ is strong independent fair.*

Consider the RSGR(1) LTS control problem $\mathcal{R} = \langle \mathcal{L}, T \rangle$ where $\mathcal{L} = \langle E, H, L_c \rangle$, where E is the LTS in Figure 6, $L_c = \{try_1, try_2, G\}$, $H = \{(As, Gs)\}$, $As = \Box \Diamond \dot{A}$ and $Gs = \Box \Diamond \dot{G}$. The controller enabling only try_1, is a valid controller for $\mathcal{R}$ but it forces the environment to fulfil its assumptions by failing, while the controller enabling only try_2 is also a valid controller for $\mathcal{R}$ but it doesn't force the environment to a place in which the only possibility to fulfil its assumptions is by failing. Such a controller is more desirable and is what we expect from a *Best Effort* Controller in the context of Recurrent Success control problems.

Note that $\mathcal{R}$ satisfies the best effort condition defined in [5] for SGR(1) but not the one above for RSGR(1).

In [5] a sufficient condition for ensuring best effort is defined. It essentially dictates that it must be possible for the environment to fulfill its assumptions regardless of how a controller behaves. In the context of domains with failures and RSGR(1), this condition is not sufficient. For RSGR(1), we must require that the environment be able to achieve its assumptions independently of how the controller behaves and how decisions on failures occur. The assumptions-compatibility definition that follows is identical to that of [5] except that the set of controlled actions is extended with failure actions. The definition states that the assumptions are compatible if there is no controller that can prevent assumptions from happening even when controlling failures.

Definition 4.6. (Assumptions Compatibility) *Given an RSGR(1) LTS control problem $\mathcal{R} = \langle \mathcal{L}, T \rangle$, where $\mathcal{L} = \langle E, H, L_c \rangle$ and $H = \{(\emptyset, I), (As, G)\}$, we say that the As is compatible with E according to T, if for every state s in E there is no solution for the SGR(1) LTS control problem $\langle E_s, H', L_c \cup F \rangle$, where $H' = \{(\emptyset, I), (As, false)\}$, and E_s is the result of changing the initial state of E to s and F is the set of all $fail_i$ in T.*

It is straightforward to see that the environment E in Figure 6 is not assumptions compatible with A. A controller M which never takes try_2 nor $fail_1$ forces $E\|M$ to not satisfy $\Box \Diamond A$, which means that the controller has no obligation of satisfying $false$. Hence, there is a solution for the problem $\langle E, H, L_c \cup F \rangle$, where $H' = \{(\emptyset, \text{true}), (A, \text{false})\}$.

Similarly to [5], the assumptions-compatibility condition is related to the definition of best effort controller. Intuitively, if the domain is such that the environment can produce all its assumptions without requiring the use of failures, then every controller is best effort.

Theorem 4.2. *Given an RSGR(1) LTS control problem $\mathcal{R} = \langle \mathcal{L}, T \rangle$ with environment model E and assumptions As, if As is assumptions compatible with E according to T then all solutions to $\mathcal{R}$ are best effort controllers.*

Proof. *Refer to [1].*

The theorem above is applicable for E in Figure 6 since E is not assumptions compatible with A; in effect, there are non best effort controllers for E. However, the orchestration problem discussed in Section 2 is assumptions compatible, the theorem applies and all solutions to the RSGR(1) orchestration problem are best effort. The environment for the orchestration problem is assumptions compatible because the assumption A_1 requires there be package requests pending to be processed infinitely often. A controller (controlling failures too, as in Definition 4.6) cannot impede the environment from achieving the assumptions because failures will simply delay package processing and while a package is being processed that package is pending. On the other hand, once the controller has processed the package, it is blocked until a new package arrives. Hence, the environment is free to deliver a new package request, which becomes pending and which fulfils A_1.

4.4 Unsupported Traces

In the previous subsection we discussed assumptions compatibility. Under this condition a controller cannot discharge obligations by either forcing assumptions not to occur or by forcing strong independent fairness not to hold. However, even in the case of satisfying the assumptions-compatibility condition, the environment may still choose not to satisfy strong independent fairness. Clearly, for such traces the controller is not obligated to satisfy the goals. Consequently, applicability of our technique severely depends on how many or how relevant are the traces in which goals are not necessarily achieved?

More concretely, consider the example in Figure 5. Assumption A is compatible with the environment. Thus, solutions to the control problem are guaranteed to be best effort. Consequently, a controller that repeatedly attempts *try* is a good controller: it does not try to achieve its goals vacuously

and succeeds in achieving its goals for all strong independent fair traces. However, the trace $try, success, \ell, try, fail, A, try, success, \ell, \ldots$ is not strong independent fair. Hence the controller is not obliged to, and in fact does not, satisfy its goals. How good is this controller? How relevant is it that the controller does not achieve its goals for this trace? Are there other traces for which the controller's obligations are discharged and how relevant are they?

We consider this question in contexts where the environment can be thought of as a probabilistic model in which all transitions (or at least non-failing ones) have non-zero probability. We show that if we restrict our attention to assumptions-compatible environments, then the measure of the set of paths in which the environment progresses but the controller has no obligations is zero. That is, when working with assumptions-compatible models, the traces for which the controller does not achieve the goals are negligible.

Consider an environment E for an RSGR(1) problem $\mathcal{R}$ that can be seen as an abstraction of a Markov Decision Process [2] (MDP) E_p. It is possible to show that if E is assumptions compatible with respect to the assumptions of $\mathcal{R}$ then the measure of paths that are not strong independent in E_p is zero. More formally:

THEOREM 4.3. *Given an RSGR(1) problem $\mathcal{R}$ with an assumptions-compatible environment E, and an MDP E_p such that the underlying LTS $E_p \downarrow$ is simulation equivalent to E then, for every controller M, for every fair scheduler s of E_p consistent with M, the following holds: the measure of the set $B = \{\pi | \pi$ is a trace of E_p under scheduler s and π matches a trace of E that satisfies assumptions infinitely often but is not strong independent fair in $M||E\}$ is zero.*

PROOF. *Refer to [1].*

For instance, the MDP E_p in Figure 7 is a model of the Ceramic Cooking problem. It is straightforward to see that the grounding of E_p (i.e. $E_p \downarrow$) is simulation equivalent to the LTS E_2 of Figure 3. In addition, E_2 is assumptions compatible with $A = \neg cooking$ as the only way of not achieving the assumption is by performing *cooking*, which is controlled by the environment. So, by Theorem 4.3, controllers to the RSGR(1) problem with environment E_2, assumption A, goal $moveToBelt$, try-response triple ($cook$, $broken$, $not.broken$) and safety $moveToBelt \Rightarrow Cooked\ Twice \wedge \neg Broken$ are best effort and achieve the goals with probability one. Traces that are not strong independent fair (e.g. if a piece is broken at least once in every two cooks) are negligible.

Consider now the orchestration problem of Section 2. Its environment is compatible with the assumptions on pending package request. The question to ask now is if the problem domain can be thought of as an MDP for the theorem above to be applicable. This amounts to validating if the environment's choices can be thought of as probabilistic choices over some memoryless probabilistic distribution. All choices of the environment are related to failures: Queries on availability of cars, hotels and plains can fail; reservations on these can fail; and so can payments. Modelling each query failure/success as an independent probabilistic choice entails the following. Either resources are transiently unavailable (e.g. cars of a certain model eventually become available) or users will vary their criteria reasonably (e.g. making it less restrictive) in order to succeed in queries. Hence, Theorem 4.3 is not free. Requiring an MDP model of the domain means that the denotation of, for instance, failures must be

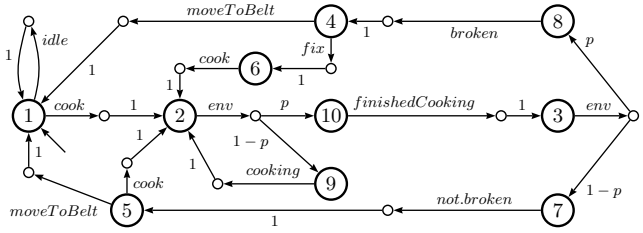

Figure 7: MDP for Ceramic Cooking Problem.

compatible with probabilistic choice. In many setting such a denotation is possible and realistic, as with the orchestration problem, but this is not necessarily the case. If, for instance, payment failure denotation includes failures due to incorrect program logic in one of the services, then assuming probabilistic behaviour of these failures may not be valid. For example, the logic may be such that it consistently fails once every n payments of the same client, where n is the number of services that a package includes.

Summarising, Theorem 4.3 shows that the restriction to strong independent fair paths is not severe if the problem domain can be modelled probabilistically.

5. CASE STUDIES

In this section we report on a number of case studies. We evaluate the applicability of our approach and the benefits it provides with respect to existing synthesis techniques. The case studies are taken from existing literature on behaviour model synthesis. Applicability is evaluated based on the following criteria *i*) is RSGR(1) applicable to the case study as described in the literature, *ii*) is RSGR(1) applicable to richer versions, which introduce domain relevant failures. Benefits with respect to existing techniques is evaluated based on *i*) the ability to generate controllers automatically, *ii*) the guarantees provided by the resulting controller, and *iii*) the degree of idealisation of the problem domain. For the controller synthesis tool and all case studies, including the orchestration problem of Section 2, see [6].

Production Cell. In [5] we presented a control problem based on the Production Cell [18]: a robotic arm coordinates the application of various tools to construct a product fulfilling some safety and liveness requirements. The liveness requirement is that infinitely many products are constructed. The assumption is that if the controller is waiting for pieces to construct a new product, it eventually receives them.

Both the original problem formulation and that of [5] take an idealised view of the problem. They assume that all controllable arm actions succeed. For instance, it is possible to order an arm to lift a piece from the conveyor belt, and it is assumed that this always succeeds. We refined the problem in order to account for failed arm movement actions. We define a set of try-success triples of the form ($put.tool_i$, $tool_i.succ$, $tool_i.fail$) modelling the action of placing a piece to be processed by tool i and the possible success or failure of the action. The resulting model is a compatible environment (see Definition 4.6) for the following assumption: if the controller is waiting for pieces the environment provides then. Hence, the RSGR(1) problem is guaranteed to produce a best effort controller (see Theorem 4.2).

It could be argued that we model failures to our advantage in order to obtain a compatible environment. However, we find it very natural that failures and assumptions (in this case) are independent. Notice that failures can occur

only when the controller is busy working on existing pieces. Hence, it would be impossible to "not satisfy" assumptions when failures occur.

Another possible criticism could be that strong independent fair traces are not sufficient in this domain. However, for instance, suppose that failures are abstracting imprecision of arm movements. In such a case, arms miss the target location for loading or unloading due to traction problems. It is reasonable to assume that the imprecision measured in millimetres is a memory-less random variable. Hence, failure would be related to the imprecision being above a certain threshold. Consequently, the probability of a failure is independent of the global state of the environment, that of the controller, and of the history of previous failures.

Consider a denotation of failures such as the one above. By Proposition 4.3 the traces for which the controller provides no guarantees have probabilistic measure zero. Obviously, if the failure denotes also the possibility of the arm breaking or getting permanently stuck, then the measure of such traces is not zero. In fact, under such scenarios no controller could achieve its production goals (unless another action repair or get-unstuck is added).

Pay & Ship. Pistore et al. synthesise a plan for composing distributed web services and monitoring them [20]. More specifically, a web-service coordinates purchase requests by buying on a furniture-sales service and booking a shipping service. The case study includes these failures: Both the furniture-sales and shipping services may respond positively or negatively to a request by the controller-to-be.

The controller synthesised in [20] gives no guarantees that the goal of satisfying purchase requests is achieved. In fact, achieving the goals stated in [20] requires assuming progress on the environment and fairness conditions on the success of operations on the furniture-sales and shipping services.

We modelled this case study as an RSGR(1) problem. In our setting, it is possible to check that the model is a compatible environment with respect to the following assumptions. First, that the purchase requests occur infinitely often. Second, that customers confirm infinitely many products and delivery options. Thus, the resulting controller is guaranteed to be best-effort. Furthermore, the environment assumptions under which it achieves its goals are explicit. Finally, the probabilistic argument of Section 4.4 is applicable: If failures are assumed, for instance, to be a result of lack of periodically renewed resources (no stock of selected furniture or no delivery trucks available at the moment of request). If, on the other hand, failures denote the application of a commercial policy related to the characteristics of the purchase, then a probabilistic argument may not apply. Clearly, users of our technique have to analyze its adequacy for their specific problem. They have to understand the implications of assuming strong fair independece on traces and the implications of deploying a service which does not provide guarantees in these cases.

Autonomous Vehicles. We consider the robotics case study originally presented in [12]. It presents a disaster recovery scenario in which a robot must travel within a collapsed house taking supplies to people trapped in one of the rooms. In addition a number of obstacles may intermittently impede movement of the robot. The synthesis algorithm presented in [12] considers two types of failures as a result of movements of the robot: *i*) the robot does not get to expected position after moving, for instance due to roughness of the terrain, and *ii*) the package is dropped, as a result, for instance, of sharp movements of the robot. The goal of the controller is to get to the target location with supplies. However, there is no guarantee that the controller achieves this goal.

The environment model, as presented in [12], assumes the following. First, the robot is loaded with supplies infinitely often. Second, intermittent obstacles disappear infinitely often. We find that the environment is compatible with these assumptions. Thus, posing this case study as RSGR(1) produces a controller that is guaranteed to achieve its goals for strong independent fair traces. Furthermore, if failures due to movement attempts are considered to be independent (i.e. that the rubble may compromise an attempt at moving, but that the robot does not encounter an unsurmountable (unmodelled) obstacle such as a wall); and if the loss of supplies has a probability lower than one, then strong independent fair traces have a probabilistic measure of one.

Note that in [5] an adaptation of the case study is presented. Due to the limitations of the technique in dealing with failures, either failures must be restricted or removed altogether, or if fully specified, the technique reports that no controller can be built.

Bookstore. We consider the web-service composition scenario in [14], which structurally resembles Pay & Ship. Similarly to Pay & Ship, two services are to be coordinated to provide a more complex service. The difference is that no explicit liveness properties are stated. Furthermore, an idealised version of the services is provided in which no failures can occur. The introduction of failures to this problem results in a problem that is, in essence, the same as Pay & Ship and which our approach can deal with.

6. DISCUSSION AND RELATED WORK

Our work builds on that of the controller synthesis community and particularly on the generalised reactivity synthesis algorithm GR(1) [22]. In [5], we revisit GR(1) and adapt it to a message passing communication model, rather than for a shared memory model. The message passing model matches the paradigm in behaviour modelling and analysis in software engineering (e.g. requirements engineering and architectural design). Specifically, in [5] a controller synthesis algorithm, SGR(1), is studied for LTS and CSP-like parallel composition [13]. In particular, we provided a sound methodological approach to the definition of assumptions in order to avoid anomalous controllers. The technique presented herein extends both the controller synthesis algorithms and methodological definitions of [5] to account for domains with failures.

Although numerous behaviour model synthesis techniques have been studied (e.g. [3]) these are restricted to user-defined safety requirements. The exceptions that we are aware of relate to the self-adaptive systems and planning.

Sykes et al. build plans for reachability (a limited form of liveness) goals [26]. In their setting, 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. As "well enough" is not defined, it is not clear what guarantees are provided by the resulting plans. More generally, planning as model checking [10] supports CTL goals for kripke structures. Thus, the problem

of environment-controller composition, distinction between controllable and monitored actions and realizability are not considered. In addition, as with [26], when failures are supported, there are no guarantees as to when goals are actually achieved by plans.

In [14] the problem of constructing an adaptation strategy is studied. However, it is limited to enforcing safety properties and uses a backward error propagation technique [25] to construct controllers. The lack of explicit live conditions makes failures and fairness conditions irrelevant.

Finally, our work is heavily influenced by the work on requirements engineering by Jackson [15] van Lamsweerde [16] and Parnas [19]. They 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. CONCLUSIONS

We have presented a controller synthesis technique for event-based operational models. Our technique supports a restricted, yet expressive, form of liveness. It conforms to foundational requirements engineering best practices: (i) it makes explicit the assumptions on the environment behaviour and (ii) distinguishes between controlled and monitored actions. Our technique integrates failures, allowing less idealised environment models. In order to handle failures we introduce explicit fairness conditions that are required to guarantee controller goals. We provide methodological guidelines for providing well constructed assumptions, which are in line with standard realisability notions. We show that these guidelines guarantee controllers that are eager to satisfy goals and avoid discharging obligations by invalidating environment assumptions. Furthermore, for environments that satisfy these guidelines and have an underlying probabilistic behaviour, the measure of traces that satisfy our fairness condition is 1. This gives further evidence to the usefulness of these guidelines.

A key aspect of our technique is that, unlike general controller synthesis techniques, it remains within polynomial complexity. We restrict users to write specification in GR(1), which has been used in complex problem settings such as autonomous vehicles [11]. The fairness condition required in our technique is crucial to reducing the problem to SGR(1) and then GR(1). The tradeoff between algorithmic complexity and expressiveness is captured by strong independent fairness. Informally, strong independent fairness requires environments not to orchestrate failure occurrences and satisfaction of assumptions. As shown in discussions and case studies, there are relevant problem domains in which environments have such characteristics. In particular, there are problem domains where environment choices can be characterised by memoryless probabilistic choices, which make our technique even more appealing.

8. REFERENCES

[1] Tech. Report. http://www.doc.ic.ac.uk/ srdipi/tech.

[2] R. Bellman. A Markovian decision process. *Journal of Mathematics and Mechanics.*, 6:679–684, 1957.

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

[4] L. De Alfaro and T. Henzinger. Interface automata. *ESEC/FSE-9*, pages 109–120, 2001.

[5] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesis of Live Behaviour Models. In *FSE*. ACM, 2010.

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

[7] E. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *FOCS*. 1988.

[8] N. Francez. Fairness. Springer-Verlag, 1986.

[9] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. *ESEC/FSE-11*, pages 257–266, 2003.

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

[11] H. Kress-Gazit, D. Conner, H. Choset, A. Rizzi, and G. Pappas. Courteous Cars: Decentralized Multiagent Traffic Coordination. *IEEE Robotics & Automation*, 15(1):30–38, 2008.

[12] W. Heaven, D. Sykes, J. Magee, J. Kramer. A Case Study in Goal-Driven Architectural Adaptation. *SESAS*, 2009.

[13] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):677, 1978.

[14] P. Inverardi and M. Tivoli. A reuse-based approach to the correct and automatic composition of web-services. In *FSE ESSPE*, page 33. ACM, 2007.

[15] M. Jackson. The world and the machine. In *ICSE*, 1995.

[16] A. V. Lamsweerde. Goal-oriented requirements engineering: A guided tour. *RE*, page 249, 2001.

[17] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *ASE*, 2008.

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

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

[20] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. *Artificial Intelligence: Methodology, Systems, and Applications*, pages 106–115, 2004.

[21] N. Piterman and A. Pnueli. Faster solutions of Rabin and Streett games. In *LICS*, pages 275–284, 2006.

[22] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. *VMCAI*, pages 364–380, 2006.

[23] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*. ACM, 1989.

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

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

[26] D. Sykes, W. Heaven, J. Magee, and J. Kramer. Plan-directed architectural change for autonomous systems. In *SAVCBS*, pages 15–21. ACM, 2007.

[27] S. Uchitel, G. Brunet, and M. Chechik. Behaviour model synthesis from properties and scenarios. In *IEEE Trans. Software Eng.*, pages 384-406. IEEE, 2009.