Imperial College of Science, Technology and Medicine
Department of Computing

# Synthesis of Event-Based Controllers for Software Engineering

Nicolás Roque D'Ippolito

January 2013

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College and
the Diploma of Imperial College London

The work presented in this thesis has been done by myself in collaboration with my supervisors Dr. Sebastian Uchitel and Dr. Nir Piterman. All the work made by others mentioned in this thesis has been appropriately referenced.

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work

# Abstract

Behavioural modelling has been widely used to aid in the design of concurrent systems. Behaviour models have shown to be useful to uncover design errors in early stages of the development process. However, building correct behaviour models is costly and requires significant experience. Controller synthesis offers a way to build models that are correct by construction. Existing software engineering techniques for synthesising controllers have various limitations. Such limitations can be seen as restrictions in the expressiveness of the controller goals and environment model, or in the relation between the controllable and monitored actions. *The main aim of this thesis is the development of novel techniques overcoming known limitations of previous approaches and methodological guidelines for synthesising useful controllers.*

This thesis establishes the framework for controller synthesis techniques that support event-based models, expressive goal specifications, distinguish controllable from monitored actions and guarantee achievement of the desired goals. Together with these techniques, methodological guidelines are proposed to help in building more accurate descriptions of the environment and more effective controllers.

In addition, this thesis presents a tool that implements the proposed techniques. Evaluation of the techniques has been conducted using the tool to model known case studies from the literature, showing that by allowing more expressive controller goals and environment models, and explicitly distinguishing controllable and monitored actions such case studies can be more accurately modelled and solutions guaranteeing satisfaction of the goals can be achieved.

## Acknowledgements

I would like to express my deepest gratitude to Sebastián Uchitel. His guidance and support have been of invaluable importance not only for my PhD, but also in giving me the confidence to believe that all this was possible. His way of seeing things is truly fascinating and makes working with him an incredibly challenging and exciting experience. Seb is an example of supervisor, colleague and friend.

I am also deeply grateful to Nir Piterman, who has been most important in my development as a researcher. He has open my eyes to formal methods in such a natural way that, in many cases, it felt as if they were truly "simple matters". His brightness and joyful personality makes him a perfect partner with whom working long hours is extremely productive and fun.

I would also like to thank Victor Braberman, who among many other things, taught me that to understand great minds one have to listen and take (possibly long) time to think, so then, with some luck, one can understand.

Special thanks to Jeff Kramer and Jeff Magee with whom technical discussions are extremely challenging and their experience always triggers interesting research questions. I am grateful to the friends and colleagues at Imperial who have helped enormously in making coffees, lunches and even squash matches very enjoyable. Dalal, Ioana, Francesco, Rob and so many others. Many thanks to Esteban, Negro, Daniela, Diego, Ferto, Guido and Chucky for making "oficina 12" the amazing place it is. Also thanks to the DC crowd Mocskos, DFS, Flavia, Agustín and so many others that have made my visits to Buenos Aires incredibly fun.

I am deeply indebted to Karen and Will for their loving friendship that have made me feel some of the warmth London has to offer. Special thanks go to Nico, Romano y Lauta for all the shared fun, discussions and trips. To my great friends "Los químicos" (Pablo, Josy, Guto, Paula, Laura, Paola, Eze, Sergio and, recently, Ana and Ema) for so many dinners, mates and interesting, sometimes too heated, discussions. To all of you guys, thanks for making me feel so loved.

I want to specially thank Mamá, Papá, Flavia, Valeria, Franco, Taís, Mateo, Miranda and (my godson) Tomás for all their support and love.

This has been a beautiful journey. I have shared it with many special people, however, without any doubts, Ceci has been the most important of all. Without her this PhD would not have been possible. Muchisimas gracias amor de mi alma.

Financial support for this work has been provided by grant ERC PBMFIMBSE.

A Cecilia con todo mi amor.

# Contents

**Abstract** **5**

**Acknowledgements** **7**

**1 Introduction** **23**

1.1 Motivation . . . . . 23

1.2 Summary of Contributions . . . . . 28

1.3 Thesis Outline . . . . . 30

**2 Background Theory** **31**

2.1 The World and the Machine . . . . . 31

2.2 Transition Systems . . . . . 34

2.2.1 Labelled Transition Systems . . . . . 34

2.2.2 Modal Transition Systems . . . . . 36

2.3 Fluent Linear Temporal Logic . . . . . 38

2.4 Finite State Process . . . . . 40

2.5 Two-Player Games . . . . . 42

**3 LTS Control Synthesis** **46**

3.1 Motivating Example . . . . . 47

3.2 Event-Based control problem . . . . . 53

3.3 Safe Generalised Reactivity (1) . . . . . 56

3.3.1 Responsiveness in SGR(1) . . . . . 57

3.4 Assumptions and Anomalous Controllers . . . . . 59

3.5 Solving SGR(1) Control . . . . . 64

3.5.1 SGR(1) LTS control to GR(1) games . . . . . 65

3.5.2 Translating strategies to LTS Controllers . . . . . 66

3.6 Algorithm . . . . . 71

3.7 LTS Control Problem Determinacy . . . . . 79

3.8 Evaluation . . . . . 79

3.8.1 Autonomous Vehicles . . . . . 80

3.8.2 Purchase & Delivery . . . . . 84

3.8.3 Bookstore . . . . . 89

3.8.4 Production Cell . . . . . 93

3.9 Limitations . . . . . 94

3.9.1 Partially Defined Environment Models . . . . . 95

3.9.2 Fallible Environment Models . . . . . 96

3.9.3 Non-Deterministic Environment Models . . . . . 98

**4 Synthesis for Fallible Environments 99**

4.1 Motivating Example . . . . . 99

4.2 Failures Fairness . . . . . 104

4.3 Recurrent Success Control Problem . . . . . 113

4.4 Anomalous Controllers . . . . . 116

4.5 Unsupported Traces . . . . . 120

4.6 Evaluation . . . . . 124

4.6.1 Production Cell . . . . . 125

4.6.2 Pay & Ship . . . . . 126

4.6.3 Autonomous Vehicles . . . . . 128

4.6.4 Bookstore . . . . . 129

4.7 Limitations . . . . . 129

4.7.1 Production Cell . . . . . 130

**5 MTS Control Synthesis** **138**

5.1 MTS Control Problem . . . . . 139

5.1.1 Motivating Example . . . . . 141

5.1.2 Solving the MTS Control Problem . . . . . 142

5.2 Linear Reduction . . . . . 148

5.3 One Controller To Rule Them All . . . . . 154

5.4 Evaluation . . . . . 159

5.4.1 Production Cell . . . . . 159

5.5 Limitations . . . . . 161

**6 Tool Support** **163**

6.1 Implementation . . . . . 163

6.2 Modelling Control Problems . . . . . 167

6.2.1 Modelling the Environment . . . . . 167

6.2.2 Modelling Controller Goals . . . . . 168

6.3 Evaluation . . . . . 171

**7 Discussion and Related Work** **176**

7.1 Synthesis Techniques . . . . . 176

7.2 Environment Assumptions . . . . . 178

7.3 Modelling Language . . . . . 182

7.4 Fallible Environment Models . . . . . 183

7.5 Partially Specified Environment Models . . . . . 183

**8 Conclusions** **186**

8.1 Contributions . . . . . 186

8.2 Limitations . . . . . 189

8.3 Future Work . . . . . 191

8.4 Closing Remarks . . . . . 191

**A Proofs** **193**

**Bibliography** **208**

# List of Tables

3.1 Ranks for states in Strategy $\sigma_R$ . . . . . 75

6.1 Overview of the case studies results . . . . . 172

6.2 Progression in the size of the Production Cell . . . . . 173

# List of Figures

2.1 World and Machine Phenomena . . . . . 32

2.2 Semantics for the satisfaction operator . . . . . 39

2.3 FSP Example . . . . . 41

3.1 LTS Model for the Drill . . . . . 49

3.2 LTS Model for Raw Products Processing . . . . . 49

3.3 LTS Model for the Robot Arm . . . . . 50

3.4 Production process for products of type $A$ . . . . . 51

3.5 Event-Based Control Example. . . . . 55

3.6 Running Example . . . . . 57

3.7 Transition relation and Strategy for the game $G_R$ . . . . . 67

3.8 Pseudo-code of algorithm for solving SGR(1) games . . . . . 77

3.9 Furniture-sales service interface and User . . . . . 86

3.10 Controller for serving furniture requests . . . . . 89

3.11 Bookstore case study models . . . . . 91

3.12 Controller for reduced Production Cell . . . . . 94

4.1 Car Booking Service . . . . . 101

4.2 Ceramic Cooking Process . . . . . 104

4.3 Failing Ceramic Cooking Process ($E_2$) . . . . . 105

4.4 Ceramics cook-twice controller . . . . . 109

4.5 t-strong fairness is not enough . . . . . 111

4.6 Environment $E$ . . . . . 117

4.7 MDP for the Ceramic Cooking Problem . . . . . 123

4.8 FSP Example - Robot Arm . . . . . 131

4.9 FSP Example - Tools . . . . . 132

4.10 FSP Example - Supply & Pickup Restriction . . . . . 132

4.11 Environment Model Decoupled through Yielding Control . . . 136

5.1 Event-Based Control Example. . . . . 140

5.2 Server Example . . . . . 141

5.3 LTS Controller for the Server Example . . . . . 142

5.4 Translation from $E$ to $E^I$ . . . . . 144

5.5 Example for the case *All* . . . . . 157

5.6 Example for the case *Some* . . . . . 158

5.7 Second Oven for the extended Production Cell . . . . . 160

6.1 MTSA - Draw View . . . . . 168

6.2 MTSA - FSP Editor . . . . . 169

6.3 Controller Goals - FSP Example . . . . . 170

6.4 Production Cell - Time complexity. . . . . 174

A.1 Topology of a try-response cycle . . . . . 197

# Chapter 1

# Introduction

## 1.1 Motivation

Michael Jackson's Machine-World model [Jac95b] establishes a framework on which to approach the challenges of requirements engineering. In this model, requirements $R$ are prescriptive statements of the world expressed in terms of phenomena on the interface between the machine we are to build and the world in which the real problems to be solved live. Such problems are to be captured with prescriptive statements expressed in terms of phenomena in the world (but not necessarily part of the world-machine interface) called goals $G$ and descriptive statements of what we assume to be true in the world (environment assumptions $D$).

Within this setting, a key task in requirements engineering is to understand and document the goals and the characteristics of the environment in which

these are to be achieved, in order to formulate a set of requirements for the machine to be built such that if the environment assumptions and goals are valid, the requirements in such environment entail the goals, more formally $R, D \models G$.

Thus, a key problem of requirements engineering can be formulated as a synthesis problem. Given a set of descriptive assumptions on the environment behaviour and a set of system goals, construct an operational model of the machine such that when composed with the environment model, the goals are achieved. Such problem is known as the controller synthesis [PR89, RW89] problem and has been studied extensively resulting in techniques that have been used in various software engineering domains.

Synthesis of event-based operational models of intended system behaviour from scenario-based specifications (e.g. [UBC09a, DLvL06, BSL04]) allows integrating a fragmented, example-based specification into a model that 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 [LKMU08, KPR04].

In the domain of self-adaptive systems [HGS04] there has also been an increasing interest in behaviour model synthesis as such systems must be capable of adapting their strategies at run-time. Hence, they rely heavily on automated synthesis of behaviour models that will guarantee the satisfaction of requirements under the constraints enforced by the environment and the

capabilities offered by the self-adaptive system [KM07, GBMP97, DGM09].

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 [AITG04, IT07], and in particular in service oriented architectures [BIPT09].

Existing Software Engineering techniques for automatic synthesis of event-based controllers have various limitations. Such limitations can be seen as restrictions in the expressiveness of the system goals and environment assumptions, the relation between the controllable and monitored actions, or scalability problems.

Most techniques restrict controller goals and environment assumptions to safety properties. Hence, synthesis can be posed as a backward error propagation [RN95] variant where a behaviour model is pruned by disabling controllable actions that can lead to undesirable states. However, in many domains, and particularly in the realm of reactive systems [MP92], liveness requirements can be of importance and having synthesis techniques capable of dealing with them is desirable. Very few approaches to behaviour model synthesis that support liveness have been proposed [GT00, BCP+01, SHMK07, HSMK09b]. The problem with these approaches is that the distinction between controlled and monitored actions [PM95], and between descriptive and prescriptive behaviour [Jac95b] 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 ob-

tained when the self-adaptive system interacts with its environment due to non-valid assumptions that were not made explicit.

Making environment assumptions explicit is crucial, and even more so with liveness system goals. Jackson [Jac95b], and others (e.g., [vLL00, Lam01, PM95]) have argued the importance of distinguishing between descriptive and prescriptive assertions and, more specifically, between requirements (prescriptive statements to be achieved by the machine), system goals (prescriptive statements to be achieved by the machine and its environment) and environment assumptions (descriptive statements guaranteed or assumed to be guaranteed by the environment).

Domain assumptions play a key role in the validation process. Many system failures are due to invalid assumptions, many times related to an over-idealisation of the environment's behaviour. In other words, statements regarding environment behaviour that are not realistic are used to demonstrate the correctness of the requirements with respect to the goals. However, since the assumptions are invalid, the goals are not achieved. Explicit assumption modelling not only better supports validation but also exposes what is required for the system goals to be achieved, helping to set more realistic expectations.

Assumptions, and their relation with the synthesis problem has been studied recently [CGG07, CHJ08]. When dealing with liveness, assumptions play an even more prominent role: Typically, reasoning about liveness in behaviour models is performed under specific assumptions which correspond to liveness properties themselves. For instance, it is common to reason under some gen-

eral 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 [BCP+01, SHMK07, HSMK09b] that leave such assumptions implicit and do not allow for user tailored liveness assumptions entail some important risks and limitations for users. On the other hand, techniques that support explicit liveness assumptions such as [PPS06] do not provide the required methodological support to verify that the environment assumptions are indeed guaranteed by the environment.

One of the limitations of existing synthesis techniques is that they are designed to work in the context of idealised environment models. Situations in which the outcome of controlled actions is not guaranteed are dealt with by assuming that such actions never fail (e.g. [DBPU10]), by not considering liveness goals (e.g. [IT07, SHMK07]), or by building controllers that aim to be live but are not guaranteed to be so [GT00]).

Moreover, known controller synthesis techniques require complete descriptions of the environment. Typically, the environment is described in a formal language with its semantics defined as some variation of a two-valued state machine such as Labelled Transition Systems (LTS) [Kel76] or Kripke structures [Kri59]. Thus, the model of the environment is assumed to be complete up to some level of abstraction (i.e. with respect an alphabet of actions or propositions).

Traditional behaviour modelling frameworks based on LTS and Kripke structures are not well suited for describing partial knowledge about the en-

vironment. This is limiting in the context of iterative development processes [BT04], restricts the use of iterative techniques for requirements elaboration (e.g., [UKM04]), and prevents the use of controller synthesis techniques when a fully described environment is unavailable, undesirable or uneconomical.

In this thesis we set out to resolve these limitations. More specific details are provided in the next section.

## 1.2 Summary of Contributions

The main contribution of this thesis is the development of a number of novel techniques and methodological guidelines for synthesising event-based controllers. The techniques presented in this thesis work for an expressive subset of liveness properties, distinguish between controllable and monitored actions, and differentiate system goals from environment assumptions. Indeed, it is the assumptions that must be modelled carefully in order to avoid synthesising anomalous behaviour models. We propose the notion of assumption compatibility as a guideline and show that it guarantees non-anomalous controllers.

In addition, one aspect of an idealised environment model is removed 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.

A realistic stronger fairness condition on the behaviour of failures is identified, and a technique to construct controllers satisfying liveness specifications under these fairness conditions is defined.

Finally, we present a controller synthesis technique that handles models where the full description of the environment behaviour is not available, i.e. partial information models. Such environment models represent a set of complete behaviour models, called implementations, that capture full descriptions of the environment where the unknowns have been left as part of the behaviour or removed from the implementation. Given a partially defined environment model the problem is to answer if all, none or some of the implementations it describes, admit a controller that guarantees a given goal. A technique that solves this problem effectively is presented. It is also shown that the time complexity of the problem for partially specified environment models depends on the complexity of solving the control problem for a single implementation.

More specific contributions of this thesis are given in Section 8 after the technical details of the techniques are properly presented and developed.

The work presented in this thesis is based on and extends several papers that have been published in the last three years [DBPU10, DBPU11, DBPU12, DBPU13, D'I12]. This thesis should be regarded as the definitive account of the work. In addition, and thanks to what I have learned working on this thesis I have also published [FDB+10, DFG+10].

## 1.3 Thesis Outline

This thesis is organised as follows. We start by providing required formal background in Chapter 2. In Chapter 3, we present and evaluate the event-based controller synthesis techniques and methodological guidelines that establish the theoretical framework in which we build upon. Chapter 4 defines the problem of synthesising controllers for environments where failures are explicitly modelled. In Chapter 5, the problem of synthesis in the context of partially defined environment models is presented. Chapter 6 reports on the tool we have developed to evaluate the applicability and effectiveness of our techniques. Discussion and related work is presented in Chapter 7. Finally, future work and conclusions are reported in Chapter 8.

# Chapter 2

# Background Theory

## 2.1 The World and the Machine

We begin by providing an overview of the relevant requirements engineering notions. In particular, we present the requirements engineering view of Zave and Jackson [Jac95a, Jac95b, ZJ97] and of Letier and Van Lamsweerde [vLL00, Lam01]. Both views agree that distinguishing between the problem *world* and the *machine* solution is central to understanding whether the machine correctly solves the problem in question. Indeed, the effect of the machine on the world and the assumptions we make about this world are central to the requirements engineering process. The problem *world* defines a part of the real world that we want to improve by constructing a machine solution. Typically, it embodies some components that interact following known rules and processes. For instance, a drill tool, a robot arm and rules

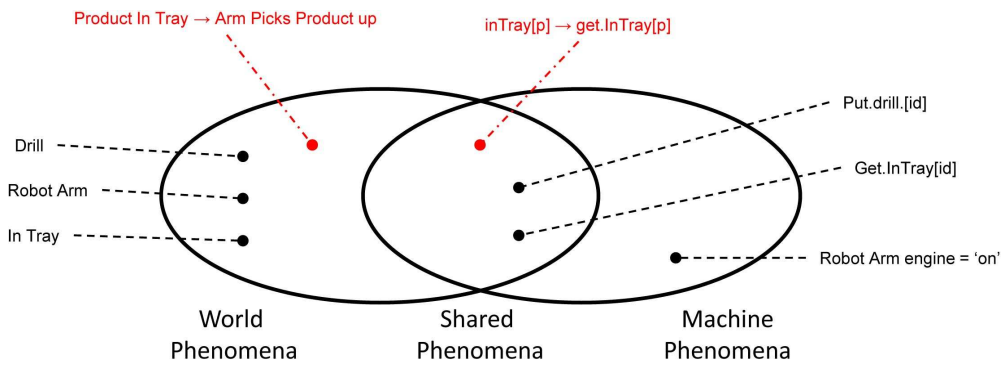



Figure 2.1: World and Machine Phenomena

for processing products that enter a production cell (see Figure 2.1). On the other hand, the *machine* solution is expected to solve the problem. For instance, the example in Figure 2.1 shows that the production cell should start processing products if they are available on the In Tray. Indeed, statement $inTray[p] \rightarrow get.InTray[p]$ shows that the robot arm is expected to pick up products from the in tray if they are ready to be processed. Finally, the shared phenomena is a portion of the problem world and the machine solution that is shared among them. Hence, it defines the interface through which the machine interacts with the world, represented as the intersection of the two sets in Figure 2.1. The *machine* is referred to in the context of synthesis as the *controller*, we shall use either term depending on the context. We may refer to the *problem world* as the *environment model*.

Statements describing phenomena of both the problem world and the machine solution may differ in scope and mood [Jac95a, PM95]. Statements may have indicative or optative mood. In [vL09], statements describing the system are characterised as *descriptive* and *prescriptive*. *Descriptive* statements represent properties about the system that hold independently of how the

system behaves. *Descriptive* statements are in indicative mood. *Prescriptive* statements state desirable properties which may hold or not. Indeed, *prescriptive* statements must be enforced by system components. Naturally, *prescriptive* statements may be changed, strengthened/weakened or even removed while *descriptive* cannot.

As mentioned above, statements may vary in scope. Both prescriptive and descriptive statements may refer to phenomena of the machine that is not shared with the world. Other statements may refer to phenomena shared by the machine and the world. More precisely, a *Domain property* is a descriptive statement about the problem world. It must hold regardless on how the system behaves. In this work we call *Environment Model*, the set of *domain properties* for a particular problem. An *Environmental Assumption* is a statement that may not hold and must be satisfied by the environment. A *Software Requirement*, or *Requirement* for short, is a prescriptive statement to be enforced by the machine regardless of how the problem world behaves and must be formulated in terms of the phenomena shared between the machine and the problem world. A *System Goal*, or *Goal* for short, is a prescriptive statement to be enforced by the machine, but not necessarily solely by it. Some collaboration with the environment might be needed.

Following [vLL00, Lam01] we say that an action is monitored/controllable if such action is observable/controllable by the machine. We may refer to monitored actions as uncontrollable actions, since they are controlled by the environment.

In this thesis we specify descriptive and prescriptive statements of the world

and machine using Fluent Linear Temporal Logic and Labelled Transitions Systems which we recall below.

## 2.2 Transition Systems

### 2.2.1 Labelled Transition Systems

We describe and fix notation for labelled transition systems (LTSs) [Kel76], which are widely used for modelling and analysing the behaviour of concurrent and distributed systems. LTS is a state transition system where transitions are labelled with actions. The set of actions of an LTS is called its communicating alphabet and constitutes the interactions that the modelled system can have with its environment.

**Definition 2.1.** (Labelled Transition Systems [Kel76]) *Let States be a universal set of states, Act be the universal set action labels. A* Labelled Transition System *(LTS) is a tuple* $E = (S_E, A_E, \Delta_E, s_{E_0})$*, where* $S_E \subseteq States$ *is a finite set of states,* $A_E \subseteq Act$ *is a finite alphabet,* $\Delta_E \subseteq (S_E \times A_E \times S_E)$ *is a transition relation, and* $s_0 \in S_E$ *is the initial state.*

Given $(s, \ell, s') \in \Delta_E$ we say that $\ell$ is enabled from $s$ in $E$. We say an LTS $E$ is *deterministic* if $(s, \ell, s')$ and $(s, \ell, s'')$ are in $\Delta_E$ implies $s' = s''$. For a state $s$ we denote $\Delta_E(s) = \{\ell \mid (s, \ell, s') \in \Delta_E\}$. Given an LTS $E$, we may refer to its alphabet as $\alpha E$.

**Definition 2.2.** (Parallel Composition) *Let* $M = (S_M,\ A_M,\ \Delta_M,\ s_{M_0})$ *and* $E = (S_E,\ A_E, \Delta_E,\ s_{E_0})$ *be LTSs.* Parallel composition $\|$ *is a symmetric operator such that* $E\|M$ *is the LTS* $E\|M = (S_E \times S_M,\ A_E \cup A_M,\ \Delta,\ (s_{E_0}, s_{M_0}))$*, where* $\Delta$ *is the smallest relation that satisfies the rules below, where* $\ell \in A_E \cup A_M$*:*

$$\frac{(s,\ell,s')\in\Delta_E}{((s,t),\ell,(s',t))\in\Delta}\ \ell \in A_E \backslash A_M \qquad \frac{(t,\ell,t')\in\Delta_M}{((s,t),\ell,(s,t'))\in\Delta}\ \ell \in A_M \backslash A_E$$

$$\frac{(s,\ell,s')\in\Delta_E,\ (t,\ell,t')\in\Delta_M}{((s,t),\ell,(s',t'))\in\Delta}\ \ell \in A_E \cap A_M$$

**Definition 2.3.** (Legal LTS) *Given* $E = (S_E, A_E, \Delta_E, s_{E_0})$*,* $M = (S_M,$ $A_M, \Delta_M, s_{M_0})$ *LTSs, and* $A_{E_u} \in A_E$*. We say that* $M$ *is a* Legal LTS *for* $E$ *with respect to* $A_{E_u}$*, if for all* $(s_E, s_M) \in E\|M$ *the following holds:* $\Delta_{E\|M}((s_E, s_M)) \cap A_{E_u} = \Delta_E(s_E) \cap A_{E_u}$

Intuitively, an LTS $M$ is a *Legal LTS* for and LTS $E$ with respect to $A_{E_u}$, if for all states in the composition $(s_E, s_M) \in S_{E\|M}$ hold that, an action $\ell \in A_{E_u}$ is disabled in $(s_E, s_M)$ if and only if it is also disabled in $s_E \in E$. In other words, $M$ does not restrict $E$ with respect to $A_{E_u}$.

**Definition 2.4.** (Traces) *Consider an LTS* $E = (S, A, \Delta, s_0)$*. A sequence* $\pi = \ell_0, \ell_1, \ldots$ *is a trace in* $E$ *if there exists a sequence* $s_0, \ell_0, s_1, \ell_1, \ldots$*, where for every* $i$ *we have* $(s_i, \ell_i, s_{i+1}) \in \Delta$*.*

**Definition 2.5.** (Reachable States) *Consider an LTS* $E = (S_E, A_E, \Delta_E, s_0)$*. A state* $s \in S_E$ *is reachable (from the initial state) in* $E$ *if there exists a sequence* $s_0, \ell_0, s_1, \ell_1, \ldots$*, where for every* $i$ *we have* $(s_i, \ell_i, s_{i+1}) \in \Delta$ *and* $s = s_{i+1}$*. We refer to the set of all reachable states in* $E$ *as* $Reach(E)$*.*

Throughout this thesis we restrict attention to LTSs $E$ such that for all states $s \in S_E$, $s$ is reachable.

### 2.2.2 Modal Transition Systems

Modal transition system (MTS) [LT88] are abstract notions of LTSs. They extend LTSs by distinguishing between two sets of transitions. Intuitively an MTS describes a set of possible LTSs by describing an upper bound and a lower bound on the set of transitions from every state. Thus, an MTS defines required transitions, which must exist, and possible transitions, which may exist. By elimination, other transitions cannot exist. Formally, we have the following.

**Definition 2.6.** (Modal Transition Systems [LT88]) *A* Modal Transition System (MTS) *is* $E = (S, A, \Delta^r, \Delta^p, s_0)$*, where* $S \subseteq States$*,* $A \subseteq Act$*, and* $s_0 \in S$ *are as in LTSs and* $\Delta^r \subseteq \Delta^p \subseteq (S \times A \times S)$ *are the required and possible transition relations, respectively.*

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

We define a refinement relation between MTSs. An LTS can be viewed as an MTS where $\Delta^p = \Delta^r$. Thus, the definition generalises to when an LTS refines an MTS. LTSs that refine an MTS $E$ are complete descriptions of the system behaviour and thus are called *implementations* of $E$.

**Definition 2.7.** (Refinement) *Let* $E = (S, A, \Delta_E^r, \Delta_E^p, s_0^E)$ *and* $N = (T, A, \Delta_N^r, \Delta_N^p, s_0^N)$ *be two MTSs. Relation* $H \subseteq S \times T$ *is a* refinement *between* $E$ *and* $N$ *if the following holds for every* $\ell \in A$ *and every* $(s, t) \in H$*.*

- *If* $(s, \ell, s') \in \Delta_E^r$ *then there is* $t'$ *such that* $(t, \ell, t') \in \Delta_N^r$ *and* $(s', t') \in H$*.*
- *If* $(t, \ell, t') \in \Delta_N^p$ *then there is* $s'$ *such that* $(s, \ell, s') \in \Delta_E^p$ *and* $(s', t') \in H$*.*

*We say that* $N$ *refines* $E$ *if there is a refinement relation* $H$ *between* $E$ *and* $N$ *such that* $(s_0^E, s_0^N) \in H$*, denoted* $E \preceq N$*.*

Intuitively, $N$ refines $E$ as every required transition of $E$ exists in $N$ and every possible transition in $N$ is possible also in $E$.

**Definition 2.8.** (Implementation and Implementation Relation) *An LTS* $N$ *is an* implementation *of an MTS* $E$ *if and only if* $N$ *is a refinement of* $E$ *(*$E \preceq N$*). We shall refer to the refinement relation between an MTS and an LTS as an implementation relation. We denote the set of implementations of* $E$ *as* $\mathtt{I}[M]$*. We assume strong refinement unless a different refinement notion is explicitly stated.*

An implementation is *deadlock free* if all states have outgoing transitions. We say that an MTS is *deterministic* if there is no state that has two outgoing possible transitions on the same action, and refer to the set of all deterministic implementations of an MTS $E$ as $\mathtt{I}^{det}[E]$.

## 2.3 Fluent Linear Temporal Logic

Linear temporal logics (LTL) are widely used to describe behaviour requirements [GM03, vLL00, LvL02, KPR04]. 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 [GM03]. Fluent Linear Temporal Logic (FLTL) [GM03] is a linear-time temporal logic for reasoning about fluents. A *fluent* $Fl$ is defined by a pair of sets and a Boolean value: $Fl = \langle I_{Fl}, T_{Fl}, Init_{Fl} \rangle$, where $I_{Fl} \subseteq Act$ is the set of initiating actions, $T_{Fl} \subseteq Act$ is the set of terminating actions and $I_{Fl} \cap T_{Fl} = \emptyset$. A fluent may be initially *true* or *false* as indicated by $Init_{Fl}$. Every action $\ell \in Act$ induces a fluent, namely $\dot{\ell} = \langle \ell, Act \setminus \{\ell\}, false \rangle$. Finally, the alphabet of a fluent is the union of its terminating and initiating actions.

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

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

where $Fl \in \mathcal{F}$. As usual we introduce $\wedge$, $\Diamond$ (eventually), and $\Box$ (always) as syntactic sugar. Let $\Pi$ be the set of infinite traces over $Act$. The trace $\pi = \ell_0, \ell_1, \ldots$ satisfies a fluent $Fl$ at position $i$, denoted $\pi, i \models Fl$, if and only if one of the following conditions holds:

- $Init_{Fl} \wedge (\forall j \in \mathbb{N} \cdot 0 \leq j \leq i \rightarrow \ell_j \notin T_{Fl})$

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

Figure 2.2: Semantics for the satisfaction operator

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

Given an infinite trace $\pi$, the satisfaction of a formula $\varphi$ at position $i$, denoted $\pi, i \models \varphi$, is defined as shown in Figure 2.2. We say that $\varphi$ holds in $\pi$, denoted $\pi \models \varphi$, if $\pi, 0 \models \varphi$. A formula $\varphi \in$ FLTL holds in an LTS $E$ (denoted $E \models \varphi$) if it holds on every infinite trace produced by $E$.

In Chapter 5 we modify LTSs and MTSs by adding new actions and adding states and transitions that use the new actions. It is convenient to change FLTL formulas to ignore these changes. Consider an FLTL formula $\varphi$ and a set of actions $\Gamma$ such that for all fluents $Fl = \langle I_{Fl}, T_{Fl}, Init_{Fl} \rangle$ in $\varphi$ we have $\Gamma \cap (I_{Fl} \cup T_{Fl}) = \emptyset$. We define the *alphabetised next* version of $\varphi$, denoted $\mathcal{X}_\Gamma(\varphi)$, as follows.

- For a fluent $Fl \in \mathcal{F}$ we define $\mathcal{X}_\Gamma(Fl) = Fl$.
- For $\varphi \vee \psi$ we define $\mathcal{X}_\Gamma(\varphi \vee \psi) = \mathcal{X}_\Gamma(\varphi) \vee \mathcal{X}_\Gamma(\psi)$.
- For $\neg\varphi$ we define $\mathcal{X}_\Gamma(\neg\varphi) = \neg\mathcal{X}_\Gamma(\varphi)$.
- For $\varphi \mathbf{U} \psi$ we define $\mathcal{X}_\Gamma(\varphi \mathbf{U} \psi) = \mathcal{X}_\Gamma(\varphi) \mathbf{U} \mathcal{X}_\Gamma(\psi)$.
- For $\mathbf{X}\varphi$ we define $\mathcal{X}_\Gamma(\mathbf{X}\varphi) = \mathbf{X}((\bigvee_{f \in \Gamma} f) \mathbf{U} \mathcal{X}_\Gamma(\varphi))$

Thus, this transformation replaces every next operator occurring in the formula by an until operator that skips uninteresting actions that are in $\Gamma$.

Note that the transformations in Section 5 force an action not in $\Gamma$ to appear after every action from $\Gamma$. Thus, the difference between **U** under even and odd number of negations is not important.

## 2.4 Finite State Process

Up to now, we have described LTSs (MTSs) by defining their components, i.e. states, actions, transition relation (required and possible), and initial state. This representation is suitable for LTSs (MTSs) with few states. However, such representation become impractical when working with larger LTSs (MTSs). For this reason, we use a simple process algebra notation called Finite State Processes (FSP) to textually specify LTSs [MKG97, MK06].

FSP is a specification language with well-defined semantics in terms of LTSs (MTSs), which provides a concise way of describing LTSs. Each FSP expression E can be mapped onto a finite LTS (MTS), we use lts(E) to denote the LTS (MTS) that corresponds to it. We now discuss briefly the syntax of FSP.

As an example, in Figure 2.3, we show the FSP code for a ceramic cooking process.

In FSP, process names start with uppercase letters and actions start with lowercase ones. The code for the Ceramic Cooking defines two FSP processes, one modelling a process that only stays idle, called `IDLE`, and another one called `DOMAIN`. In addition, `DOMAIN` defines auxiliary processes `COOKING`,

```
IDLE = idle -> IDLE.
DOMAIN = (idle->DOMAIN | cook->COOKING),
COOKING = (cooking->COOKING | cook->OH
            | finishedCooking->COOKED),
COOKED = (moveToBelt->DOMAIN | cook->COOKING),
OH = (overHeated->OH).
||IDLE_DOMAIN = (IDLE||DOMAIN).
```

Figure 2.3: FSP Example

`COOKED`, and `OH`. Auxiliary processes are local to the FSP process in which they have been defined. `DOMAIN` is defined using the action prefix operator `->` and recursion. For example, the process is defined to start by doing either `idle` staying in the same process or `cook` which leads to the local process `COOKING`.

FSP supports several composition operators such as LTS and MTS parallel compositions, or MTSs merge [FDB+10]. The parallel composition operator, which is denoted `||` is defined to preserve the semantics of LTS parallel composition presented in Definition 2.2. Thus, given two FSP processes `P` and `Q`, we have: `lts(P||Q) = lts(P)|| lts(Q)`.

In FSP processes that are defined by composing two non-auxiliary processes are called composite and their names are prefixed with `||`. Thus the parallel composition of the FSP processes for `IDLE` and `DOMAIN` components is `||IDLE_DOMAIN = (IDLE||DOMAIN)`.

Finally, FSP has a number of keywords that are used just before the definition of a process and that force MTSA to perform a complex operation on the process. For instance, the keyword, `minimal`, makes MTSA construct the minimal LTS/MTS with respect to strong semantic equivalence and the key-

word, `deterministic`, makes MTSA construct the minimal LTS with respect to trace

FSP also allows to define FLTL properties. A fluent that pinpoints the states where ceramic is being cooked can be expressed in FSP with the following code: `fluent Cooking = <cook, finishedCooking> initially 0`. `Cooking` is initially false, becomes true with `cook` and it is false again when `finishedCooking` occurs.

Summarising, FSP provides the support required to specify LTSs and FLTL formulas. Such support is required to express environment models and controller goals as it is shown in next chapters.

## 2.5 Two-Player Games

We consider two-player games played between two players, namely 1 and 2, where the goal of 1 is to satisfy the specification regardless of the actions of 2. Intuitively, of the actions possible in a state, 1 can choose to disable the action she controls. However, she cannot disable all possible actions as this would lead to a deadlock.

In this thesis we use such games in the context of controller synthesis. Hence, the games we consider are such that Player 1 (the controller) chooses which actions, from the set of controlled actions, to enable and Player 2 (the environment) chooses which actions to follow. Formally, we have the following.

**Definition 2.9.** (Two-player Game) *A* Two-player Game *(Game) is* $G =$

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

Notice that if in a controllable state $\gamma$ is empty the choice of controller may lead to a deadlock. This is prohibited later by defining this as a losing choice for the controller. From an uncontrollable state the controller may decide to disable all controllable actions. The choices of the controller are formalised in the form of a strategy. This is the policy that the controller applies. In general, the strategy may depend on the history. This is reflected in the strategy depending on a memory value in the domain $\Omega$ and updating this value according to the evolvement of the play.

Recall that this game is different from the one defined in [PPS06]. Piterman et al. define a game in which the environment chooses its next valuation and only then, the controller gets to choose what to do next.

**Definition 2.10.** (Strategy with memory) *A* strategy with memory $\Omega$ *for the controller is a pair of functions* $(\sigma, u)$, *where* $\Omega$ *is some memory domain with designated start value* $\omega_0$, $\sigma : \Omega \times S \to 2^S$ *such that* $\sigma(\omega, s) \subseteq \Gamma^+(s)$ *and* $u : \Omega \times S \to \Omega$.

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

**Definition 2.11.** (Consistency and Winning Strategy) *A finite or infinite play $p = s_0, s_1, \ldots$ is* consistent *with $(\sigma, u)$ if for every $n$ we have $s_{n+1} \in \sigma(\omega_n, s_n)$, where $\omega_{i+1} = u(\omega_i, s_{i+1})$ for all $i \geq 0$. A strategy $(\sigma, u)$ for controller from state $s$ is* winning *if every maximal play starting in $s$ and consistent with $(\sigma, u)$ is infinite and in $\varphi$. We say that controller* wins *the game $G$ if it has a winning strategy from the initial state.*

As the controller is defined as losing on all finite plays it follows that it cannot disable all controllable actions from a controllable state. We refer to checking whether controller wins a game $G$ as *solving* the game $G$. The *controller synthesis* problem is to produce a winning strategy for controller. If such winning strategy for controller exists we say that the control problem is *realisable* [RW89, MPS95]. It is well known that if controller wins a game $G$ and $\varphi$ is $\omega$-regular it can win using a finite memory strategy [PR89]. In general, winning conditions are defined using either nondeterministic Büchi automata (whose alphabet is the set of states of the game or additional labels that are added to the states) or LTL formulas (whose atomic propositions are the states of the game or labels that are added to the states). We only refer to these mechanisms in order to state the well known complexity results regarding solving games with such winning condition.

**Definition 2.12.** (Generalised Reactivity(1) [PPS06]) *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 $\varphi$ is $gr((\phi_1, \ldots, \phi_n), (\gamma_1, \ldots, \gamma_m))$.*

We refer to games (as defined in Definition 2.9) with Piterman's GR(1) winning conditions (Recalled in Definition 2.12) as GR(1) games.

**Theorem 2.1.** *Given a game $G = (S, \Gamma^-, \Gamma^+, s_{g_0}, \varphi)$ the complexity of deciding whether $G$ is winning and computing a winning strategy for controller is as follows.*

- *The complexity is 2EXPTIME-complete for $\varphi$ expressed in LTL [PR89].*
- *The complexity is EXPTIME-complete for $\varphi$ expressed as a nondeterministic Büchi automaton [PR89].*
- *The complexity is $O(nm|S|(|\Gamma^- \cup \Gamma^+|))$ for $\varphi$ a generalised reactivity(1) formula [KPP05, JP06].*

A game is called determined when the set of states from which the controller has a winning strategy are disjoint from the states from which the environment has a winning strategy. Martin shown that every game with a Borel type winning set is determined [Mar75, Kec95]. In [GTW02], Grädel et. al. proven that regular games are determined. From this result follows that the type of games we defined above are determined. More formally we have.

**Theorem 2.2.** *(*Determinacy*) Every regular game is determined.*

**Corollary 2.1.** *Every GR(1) game is determined.*

# Chapter 3

# LTS Control Synthesis

In this section we present a high level description of an event-based control problem following the world-machine model [Jac95b]. 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.

We start by providing a motivating example.

## 3.1 Motivating Example

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

Consider the following variation of the Production Cell case study [LL95]: A factory manufactures several kinds of products each of which requires a production process which involves different tools applied in a specified order. The factory production system is expected to adapt its production process depending on a number of factors such as the available tools (which is subject to change for instance when a tool breaks or a new instance of an existing tool type is introduced), the specification of how to process each product type (which can change because the production requirements for a product type changes), and other constraints (for example, an energy consumption requirement that constrains the concurrent use of certain tools).

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

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

To simplify the presentation, assume that the factory must produce two types of products, namely $A$ and $B$, with three different tools: an oven, a

drill and a press. Products of type $A$ require using the oven, then the drill and finally the press, while products of type $B$ are processed in the following order: drill, press, oven. In addition, there is a constraint on concurrent use of tools: the drill and the press cannot be used simultaneously. Finally, a liveness condition on the production of products of type $A$ and $B$ is also required, that is, the production of one kind of product cannot postpone indefinitely the production of products of the other kind.

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

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

We denote $Products$ the set $Products = \{(id, ty) \mid 0 \leq id \leq Max$ and $ty \in \{A.B\}\}$ and $Tools$ the set of available tools.

Figure 3.1 we show a behaviour model, describing the drill tool: Any product ($p \in Products$), can be *put* into the drill tool by the robot arm ($put.drill[p]$) and, subsequently, that product is processed ($drill.process[p]$) by the drill and can then be taken by the robot arm ($get.drill[p]$).

In Figure 3.2 we show a model that describes how raw products can be processed. A product ($p \in Products$) is *idle* until it appears in the in tray ($[p].inTray$), then it is picked up by the robot arm ($[p].getInTray$), subsequently, it can be freely placed and picked up from any tool (resp. $put.[t : Tools][p]$ and $get.[t : Tools][p]$) until the product processing is finished

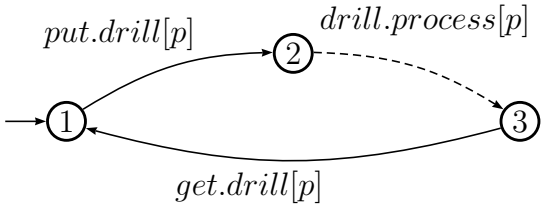



Figure 3.1: LTS Model for the Drill

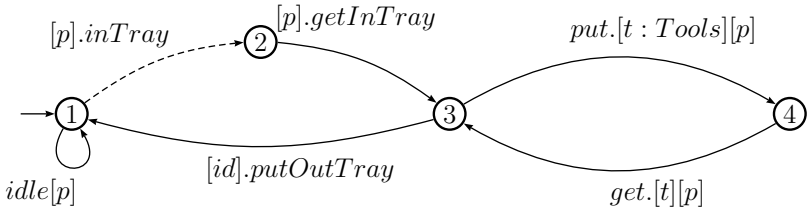



Figure 3.2: LTS Model for Raw Products Processing

and the product is placed in the out tray ($[p].putOutTray$). For simplicity, we model that an instance of a product can be reprocessed, hence, once put on the out tray, the product model is at the initial state again.

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 [Jac95b], 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 3.3) 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.

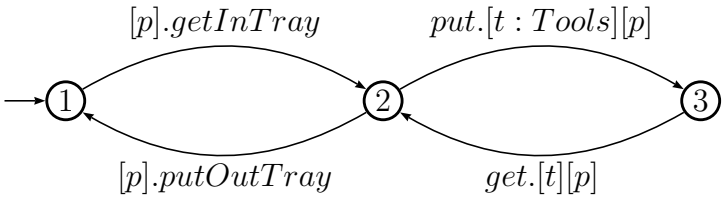



Figure 3.3: LTS Model for the Robot Arm

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. The LTS for this composition is too big to be shown, it can be constructed using the MTSA tool [DFCU08] and data available at [D'I].

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

The set of controllable 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.

Throughout this thesis we use dashed lines to denote monitored actions in figures.

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

order for each type of product. In Figure 3.4 we show how to model the processing requirements for products of type $A$ (denoted $id_a$), a temporal logic representation of such requirement is also possible (and can be constructed automatically from Figure 3.4) but more cumbersome. We omit it here but assume that the FLTL formula $\Box\, \varrho$ is obtained by transforming the LTS in Figure 3.4 to FLTL. Hence, $\Box\, \varrho$ is an FLTL formula that captures the requirements for products of type $A$ and $B$.

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

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

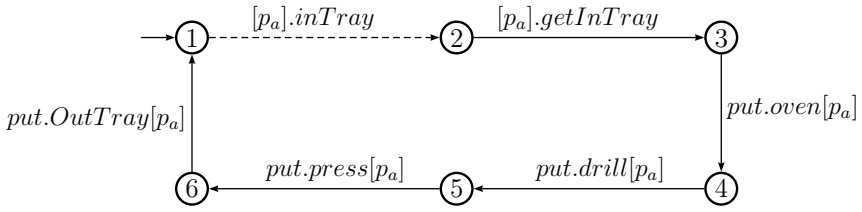



Figure 3.4: Production process for products of type $A$

product $(id, ty)$ has just been added to the out tray.

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

Consequently, we must assume that the environment will produce an infinite number of raw products of type $A$ and $B$:
$As = \bigwedge_{p \in Products}(\Box \Diamond AddedToInTray(p))$ where $AddedToInTray(p)$ holds if the product $p$ 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 ($\varrho$), that the drill and press are not used simultaneously ($\psi$) and that if the environment provides infinitely many raw products of both types ($As$) both types of products will be produced ($G$).

It is interesting to note that a controller for the robot arm that satisfies the specification above when composed with the model of the environment cannot be produced by simply pruning the environment model (as behaviour model synthesis techniques for safety properties do). This is because, in order to fulfill the liveness part of the specification, a controller must "remember"

if it has been postponing one type of product for too long. Say products of type $A$ have been postponed for too long, the controller must stop processing the other component type, $B$, giving way to the production of $A$ products. How much the controller waits before switching type could vary from one 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 3.5 we describe the procedure for synthesising behaviour models that satisfy the specification described above, and hence capable of, among other things, identifying the model's need for memorising specific aspects of the system behaviour in order to satisfy liveness properties.

## 3.2 Event-Based control problem

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

A solution for the *Event-Based control problem* is to find a machine whose

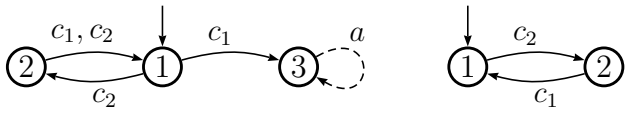
concurrent behaviour with an environment that satisfies the assumptions satisfies the goals.

We adopt labelled transition systems (LTS) and parallel composition in the style of CSP [Hoa78] as the formal basis for modelling the environment and for representing the synthesised controller, and FLTL, with its corresponding satisfiability notion, as a declarative specification language to describe both environment assumptions and system goals.

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

**Definition 3.1.** (LTS Control) *Given an environment model in the form of an LTS $E$, a set of controllable actions $A_C \in Act$, and a set $H$ of pairs $(As_i, G_i)$ where $As_i$ and $G_i$ are FLTL formulas specifying assumptions and goals respectively, the solution for the LTS control problem $\mathcal{E} = \langle E, H, A_C \rangle$ is to find an LTS $M$ such that $M$ is a legal LTS for $E$ with respect to the set of monitored actions $A_U = \overline{A_C}$, $E \| M$ is deadlock free, and for every pair $(As_i, G_i) \in H$ and for every trace $\pi$ in $E \| M$ the following holds: if $\pi \models As_i$ then $\pi \models G_i$.*

As a simple example, consider the control problem $\mathcal{E} = \langle E, H, A_C \rangle$, where $E$ is the LTS model in Figure3.5(a), $A_C = \{c_1, c_2\}$, $H = \{(\emptyset, \Box\, \neg \dot{a}), (\emptyset, \Box \diamond \dot{c_1})\}$ and the fluents are defined as follows: $\dot{a} = \langle \{a\}, \{c_1, c_2\}, \mathit{false} \rangle$, $\dot{c_1} = \langle \{c_1\}, \{a, c_2\}, \mathit{false} \rangle$, $\dot{c_2} = \langle \{c_2\}, \{c_1, a\}, \mathit{false} \rangle$. The model $M$ in Figure 3.5(b) is a solution to $\mathcal{E}$. It does so by restricting $c_1$ from the initial state which guarantees that in $E \| M$ the action $a$ cannot happen, hence, $\Box\, \neg \dot{a}$ is satisfied in $E \| M$. In addition, $M$ only enables from state 2 the transition on $c_1$, from which follows that traces in $E \| M$ are simple alternations of $c_2$ and then $c_1$, hence, $E \| M$ satisfies $\Box \diamond \dot{c_1}$.




(a) Environment $E$ (b) Controller $M$

Figure 3.5: Event-Based Control Example.

Just like in traditional (i.e. state-based) controller synthesis, the problem with using FLTL as the specification language for assumptions and goals is that the synthesis problem is 2EXPTIME complete (see Theorem 3.7). 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 [AMPS98] and GR(1) under the assumption of full observability. The latter can be seen as an extension of [AMPS98] to a more expressive liveness fragment of LTL.

## 3.3 Safe Generalised Reactivity (1)

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

**Definition 3.2.** (SGR(1) LTS Control) *An LTS control problem $\mathcal{E} = \langle E, H, A_C \rangle$ is SGR(1) if $E$ is deterministic, and $H = \{(\emptyset, I), (As, G)\}$, where $I = \Box\, \rho$, $As = \bigwedge_{i=1}^{n} \Box \diamondsuit \phi_i$, $G = \bigwedge_{j=1}^{m} \Box \diamondsuit \gamma_j$, and $\phi_i$, $\rho$ and $\gamma_j$ 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 3.6(a), $A_C = \{c_1, c_2, c_3, c_4, g_1, g_2\}$, $H = \{(\emptyset, I), (As, G)\}$, $I = \Box\, \neg \dot{w}$, $As = \Box \diamondsuit \dot{a}$ and $G = \Box \diamondsuit \dot{g}_1 \wedge \Box \diamondsuit \dot{g}_2$. Recall that for all $\ell$ in the alphabet of $E$, the fluent $\dot{\ell}$ is defined as the fluent that becomes true when $\ell$ occurs and becomes false when any other action occurs.

The LTS $M_1$, $M_2$ and $M_3$ of Figures 3.6(c) to 3.6(d) are some of the possible solutions to $\mathcal{R}$: $E \| M_1$ has no traces satisfying the assumptions $As$, hence it is not obliged to satisfy $G$; all traces in $E \| M_2$ satisfy $As$ and also $G$; and

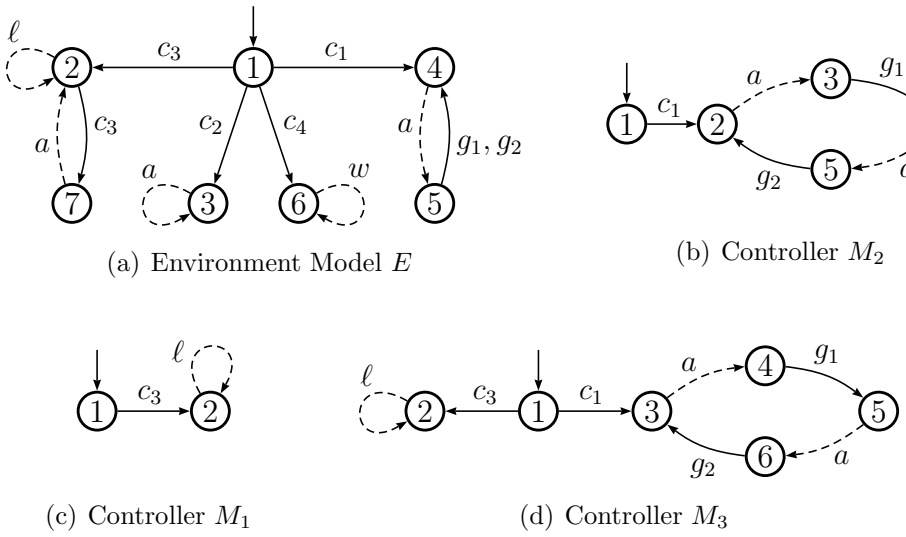



Figure 3.6: Running Example

traces in $E \| M_3$ 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 $M_2$ nor $M_3$ can be obtained only by pruning $E$. Both models introduce new states which allow the controller to "remember" which is the next goal that must be achieved ($g1$ or $g2$). The automated construction of these "memory" states will be described in detail in section 3.6.

### 3.3.1 Responsiveness in SGR(1)

The SGR(1) control problem restricts the form of the environment assumptions and system goals. Thus, a valid concern is the impact of this restriction on expressiveness in practice. A closer look at the family of liveness formulas reveals it is not arbitrary: they are designed to capture a Büchi acceptance

condition. More concretely, any liveness property specifiable by a deterministic Büchi automaton can be handled by the proposed approach. The trick is, basically, to compose the Büchi automaton structure with the original plant LTS and then use assumptions and goals to express that their acceptance conditions will/should (respectively) be visited infinitely often. Typical responsiveness assumptions and goals (e.g. $\Box(\phi \rightarrow \Diamond \psi)$) could be treated in this way [PPS06]. In the context of LTS and FLTL this kind of assumptions can be handled without explicitly generating the deterministic Büchi automaton. In many cases, this can be done by encoding the responsiveness with fluents and assumptions in GR(1) form.

Recall the Production Cell case study in Section 3.1. Consider the case where 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 $\bigwedge_{p \in Products} \Box(WaitForProcessing(p) \rightarrow \Diamond [p].put.\dot{O}utTray)$ where $WaitForProcessing(p) = \langle [p].inTray, [p].getInTray, false \rangle$. This is an example of a responsiveness goal that does not fit the syntactic requirements of SGR(1) but could be dealt with by means of the SGR(1) encoding. To encode the assumption as a GR(1)-like formula we do as follows, define a fluent with the initiating action of the antecedent as the initiating action and the initiating action of the consequent as the terminating action. Consequently, we first define the fluent $InOut = \langle \{[t].[id].inTray\}, \{[t].[id].put.OutTray\}, false \rangle$ and then we add the formula $\Box \Diamond \neg InOut$ as an environmental assumption in the GR(1) specification.

## 3.4 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 behaviour? We now answer this question.

Consider the LTS controller $M_1$ discussed in the previous section. $M_1$ solves the SGR(1) control problem $\mathcal{R}$ by simply ensuring that for all trace $\pi \in tr(E\|M_1)$ $\pi \not\models As$. Such a solution, from an engineering perspective is unsatisfactory: $M_1$ should "play fair" by trying to achieve $G$ when $As$ holds rather than trying to avoid $As$. In this sense, $M_2$ and $M_3$ 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 $\sigma$, no other controller that achieves $G$ could have allowed $As$ after $\sigma$.

**Definition 3.3.** (Best Effort Controller) *Given an SGR(1) LTS control problem $\mathcal{E}$ with assumptions $As$ and an LTS $M$ such that $M$ is a solution for $\mathcal{E}$, we say that $M$ is a* best effort controller *for $\mathcal{E}$ if for all finite traces $\sigma \in tr(E\|M)$ if there is no $\sigma'$ where $\sigma.\sigma' \in tr(E\|M)$ and $\sigma.\sigma' \models As$ then there is no other solution $M'$ to $\mathcal{E}$ such that $\sigma \in tr(E\|M')$ and there exists $\sigma'$ such that $\sigma.\sigma' \in tr(E\|M')$ and $\sigma.\sigma' \models As$*

Controller $M_1$ is not a best effort controller as $\epsilon$, the empty trace in $E\|M_1$ cannot be extended in $E\|M_1$ to satisfy $As$, yet it can be extended by $\sigma' = c_1, 2, a, 3, g_1, 4, a, 5, g_2, \cdots$ in $E\|M_2$ such that $\epsilon.\sigma'$ satisfies $\Box \Diamond As$. On the other hand, given that there are no traces in $E\|M_2$ violating $As$, $M_2$ is a best effort controller for $\mathcal{R}$. $M_3$ is also a best effort controller as the only finite trace violating $As$ in $M_3$ is $\sigma = c_3, \cdots$ and there are no extension of $\sigma$ satisfying $As$ and $G$.

Note that controller $M_3$ also could be argued to be anomalous from an engineering perspective: Although $M_3$ does play fair when choosing action $c_1$ to state 3, it can also choose action $c_3$ to state 2, leading $E\|M_3$ 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 3.4.** (Assumption Preserving Controller) *Given an SGR(1) LTS control problem $\mathcal{E}$ with assumptions $As$ and an LTS $M$ such that $M$ is a solution for $\mathcal{E}$, we say that $M$ is an* assumption preserving controller *for $\mathcal{E}$ if for all finite traces $\sigma \in tr(E\|M)$ if there is no $\sigma'$ where $\sigma.\sigma' \in tr(E\|M)$ and $\sigma.\sigma' \models As$ then there does not exist $\sigma''$ such that $\sigma.\sigma'' \in tr(E)$ and $\sigma.\sigma'' \models (I \wedge As)$*

**Theorem 3.1.** *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.*

*Proof.* Assume that $M$ is not best effort. It follows that there exists $\sigma \in$

$tr(E\|M)$ that can be extended to satisfy assumptions $As$, i.e. there exists $\sigma'$ such that $\sigma.\sigma' \in tr(E\|M)$ and $\sigma.\sigma' \models As$. Furthermore, there exists $M'$ solution to $\mathcal{E}$ such that $\sigma \in tr(E\|M')$ and $\sigma.\sigma' \in tr(E\|M')$. By Definition 2.2 (LTS parallel composition) $\sigma.\sigma' \in tr(E)$, hence $M$ is not assumption preserving controller. □

By Theorem 3.1 and the fact that $M_1$ is not best effort it follows that $M_1$ is not an assumption preserving controller. Although $M_3$ is best effort, it is not an assumption preserving controller since the trace $\sigma = c_3, c_3, a, c_3, \ldots$ in $E$ is a valid extension to $\sigma = c_3, \ldots$ in $E\|M_3$ which satisfies $As$ while violating $G$. On the other hand, given that every infinite trace in $M_2$ satisfies both $As$ and $G$, $M_2$ is an assumption preserving controller.

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

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 [LvL02].

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

**Definition 3.5.** (Assumption compatibility) *Given an SGR(1) LTS control problem* $\mathcal{E} = \langle E, H, A_C \rangle$ *and* $H = \{(\emptyset, I), (As, G)\}$*, we say that the As is compatible with E if for every state s in E there is no solution for the SGR(1) LTS control problem* $\mathcal{E}_s = \langle E_s, H', A_C \rangle$*, where* $H' = \{(\emptyset, I), (As, false)\}$ *and* $E_s$ *is the result of changing the initial state of E to s.*

Hence, when the assumptions of an SGR(1) LTS control problem are compatible with the environment, it is guaranteed that anomalous controllers

(such as those that are not best effort and assumption preserving) will not be produced.

**Theorem 3.2.** *Given an SGR(1) LTS control problem $\mathcal{E}$ with assumptions As and environment E, if As is compatible with E then all solutions to $\mathcal{E}$ are best effort and assumption preserving.*

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

Note that the running example $\mathcal{R}$ violates Definition 3.5 and hence, has anomalous controllers such as $M_1$, which is not *Best Effort* nor *Assumption Preserving*, or $M_3$ which is *Best Effort* but not *Assumption Preserving*.

Our notion of anomalous controllers is tightly coupled to the problem of properties which are trivially satisfied in system model [BB94]. A typical pattern of vacuity [BBDER97] is one in which the left hand side of an implication is never fulfilled by the system model. A controller that that achieves its goals by falsifying assumptions can be thought of as the cast of the vacuity problem in controller synthesis.

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

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

## 3.5 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) [PPS06].

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 3.5.1). Secondly, a solution $(\sigma, u)$ to the GR(1) game is used to build a solution $M$ (i.e. an LTS controller) for $\mathcal{E}$ (Section 3.5.2). We also show that our approach is sound and complete. That is, a solution to the SGR(1) LTS control problem $\mathcal{E}$ exists if and only if a solution to the GR(1) game $G$ exists. Furthermore, the LTS controller $M$ built from $(\sigma, u)$ is a solution to $\mathcal{E}$.

The reader not interested details of the mapping of SGR(1) into GR(1) can skip directly to Section 3.6 where we comment on the algorithm of the synthesis technique, or to Section 3.8 where we show a controller for a reduced version of the Production Cell case study and present the case studies we have used to evaluate our technique.

### 3.5.1 SGR(1) LTS control to GR(1) games

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

More precisely, consider an SGR(1) LTS control problem $\mathcal{E} = \langle E, H, A_C \rangle$, where, $H = \{(\emptyset, I), (As, G)\}$, $E = (S_e, A, \Delta_e, s_{e_0})$, $As = \bigwedge_{i=1}^{n} \Box \Diamond \phi_i$, $I = \Box \rho$ and $G = \bigwedge_{j=1}^{m} \Box \Diamond \gamma_j$. Let $\mathit{fl} = \{\dot{1}, \ldots, \dot{k}\}$ be the set of fluents used in $As$ and $G$ and $\dot{i} = \langle I_i, T_i, \mathit{Init}_i \rangle$. The game $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ is constructed as follows.

We build $S_g$ from $E$ such that states encode a state in $E$ and truth values for all fluents in $\varphi$: 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 $\mathit{fl}_i$, we say that $s_g$ satisfies $\mathit{fl}_i$ if $\alpha_i$ is *true* and $s_g$ does not satisfy $\mathit{fl}_i$ otherwise. We generalise satisfaction to Boolean combination of fluents in the natural way.

We build transition relations $\Gamma^-$ and $\Gamma^+$ using the following rules. Consider a state $s_g = (s_e, \alpha_1, \ldots, \alpha_k)$. If $s_g$ does not satisfy $\rho$ (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 $\Gamma^\beta$, where $\beta$ is $+$ if $\ell \in A_C$, $\beta$ is $-$ if $a \notin A_C$ and (1) $\alpha'_i$ is $\alpha_i$ if $\ell \notin I_{\mathit{fl}_i} \cup T_{\mathit{fl}_i}$, (2) $\alpha'_i$ is *true* if $\ell \in I_{\mathit{fl}_i}$ and (3) $\alpha'_i$ 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 $\varphi_g$, defined to be a set of infinite traces, from

$AS$ and $G$ as follows: We abuse notation and denote by $\phi_i$ the set of states $s_g$ such that $s_g$ satisfies the assumptions $\phi_i$ and by $\gamma_i$ the set of states $s_g$ such that $s_g$ satisfies the goal $\gamma_i$. 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.

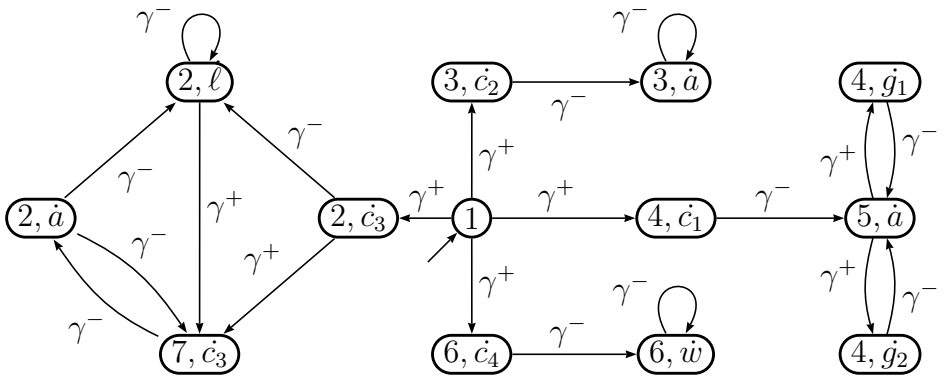
It can be shown that if there is a solution to an SGR(1) LTS control problem then there is a winning strategy for a controller in the constructed GR(1) game (refer to Theorem 3.3).

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

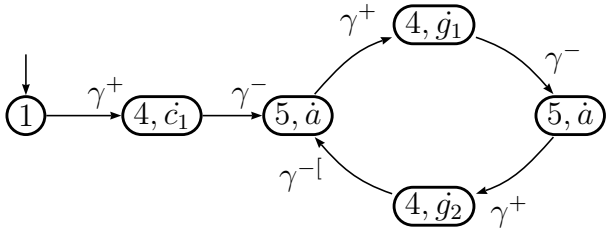
Consider the control problem $\mathcal{R}$ defined above and its corresponding environment model shown in Figure 3.6(a). Let $G_R$ be the game obtained by applying to $\mathcal{R}$ the procedure described above. Figure 3.7(a) shows the transition relations $\Gamma^-$ and $\Gamma^+$ for $\mathcal{R}$. Transitions in $\Gamma^-$ and $\Gamma^+$ are marked as $\gamma^-$ and $\gamma^+$ respectively. States are labelled with a state in the original LTS model (i.e. model $E$ in Figure 3.6(a)) and the set of fluents holding in the state of the LTS model.

### 3.5.2 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




(a) Transition relation for $G_R$




(b) $\sigma_R$: Wining strategy for $G_R$

Figure 3.7: Transition relation and Strategy for the game $G_R$

as shown in Section 3.5.1.

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

More precisely, let $E = (S_e, A, \Delta_e, s_{e_0})$, $\mathit{fl} = \{\mathit{fl}_1, \ldots, \mathit{fl}_k\}$ the set of fluents appearing in $\varphi$, $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ be the GR(1) game constructed

from $E$ as explained above, and let $\sigma : \Omega \times S_g \to 2^{S_g}$ and $u : \Omega \times S_g \to \Omega$ be a winning strategy in $G$. We construct the machine $M = (S_M, A, \Delta_M, s_{M_0})$ 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 $\ell$ is *possible* from $s_g$ to $s'_g$ if $(s_g, s'_g) \in \Gamma^- \cup \Gamma^+$, there is some action $\ell$ such that $(s_e, \ell, s'_e) \in \Delta_e$ and for every fluent $fl_i$ either (1) $\ell \notin I_{fl_i} \cup T_{fl_i}$ and $\alpha'_i = \alpha_i$, (2) $\ell \in I_{fl_i}$ and $\alpha'_i = \mathit{true}$, or (3) $\ell \in T_{fl_i}$ and $\alpha'_i = \mathit{false}$.

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

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

Consider the game $G_R$ and a strategy that satisfies $\Box \Diamond \dot{a}$, $\Box \Diamond \dot{g}_1$ and $\Box \Diamond \dot{g}_2$. The only possible solution to such requirements is to have in $(\sigma, u)$, a cycle visiting $(5, \dot{a})$, $(4, \dot{g}_1)$ and $(4, \dot{g}_2)$ in some order. A strategy satisfying this is shown in Figure 3.7(b). Note that some memory is needed to distinguish whether state $(4, \dot{g}_1)$ or $(4, \dot{g}_2)$ has to be visited after visiting $(5, \dot{a})$. Finally, in Figure 3.6(b) we show the LTS controller obtained by applying

the conversion shown above to the strategy in Figure 3.7(b).

In Theorem 3.4 we show that if $(\sigma, u)$ is winning strategy for a GR(1) game $G$ constructed from a SGR(1) LTS control problem $\mathcal{E}$, then the LTS $M$ constructed as explained above is a solution to $\mathcal{E}$. Note that to prove this proposition environment ($E$) determinism is needed. This is because LTSs can only see the performed actions and not the actual state. Hence, having a non-deterministic environment model $E$ would make impossible for the machine $M$ to guarantee $M \| E \models \varphi$.

**Theorem 3.3.** (Completeness) *Let $\mathcal{E}$ be an SGR(1) LTS control problem, and $G$ be a GR(1) game constructed by applying the conversion shown in Section 3.5.1 to $\mathcal{E}$. If $M$ is a solution for the SGR(1) problem $\mathcal{E}$ then there exists a strategy $(\sigma, u)$ such that: $(\sigma, u)$ is winning for $G$ and the LTS controller obtained by applying the translation shown in Section 3.5.2 to $(\sigma, u)$ is equivalent to $M$.*

*Proof.* The proof is organised as follows. First, we construct a winning strategy $(\sigma, u)$. Second, we prove $(\sigma, u)$ to be winning for $G$. The strategy is constructed by applying a similar reasoning as if we were applying the inverse of the transformation shown in Section 3.5.2 to $M$. That is, the states of $M$ are used as the memory function for the strategy and given a pair of states $s_g = (s_e, \alpha_1, \ldots, \alpha_k)$ and $s'_g = (s'_e, \alpha'_1, \ldots, \alpha'_k)$ in $G$. The transition $(s_m, s'_m) \in \Delta_M$ iff there is, a controllable action $\ell \in A$, a transition $((s_e, s_m), \ell, (s'_e, s'_m)) \in \Delta_{E||M}$ and the truth values of the fluents are the expected by the definition of $\Gamma^-$ and $\Gamma^+$. Furthermore, given that $E||M$ has no

deadlocks and $E||M \models \square\,\rho$, it follows that $\sigma$ cannot reach a state such that $\Gamma^- \cup \Gamma^+ = \emptyset$, in other words, $\Delta$ cannot reach a deadlock state. Also, by construction of $\Gamma^-$ and $\Gamma^+$ the truth value of the fluents in the states is updated as expected. Finally, from the definition of the update of the values $\alpha_1, \ldots, \alpha_k$ in the states of $G$ it is simple to see that a play is winning for controller iff it satisfies the FLTL formula $\square\,\rho \wedge (\bigwedge_{i=1}^{n} \square \diamondsuit\, \phi_i \Rightarrow \bigwedge_{j=1}^{m} \square \diamondsuit\, \gamma_j)$. □

**Theorem 3.4.** (Soundness) *Let $\mathcal{E}$ be a SGR(1) LTS control problem and $G$ be a GR(1) game constructed by applying the conversion shown in Section 3.5.1 to $\mathcal{E}$, $\sigma$ be a transition relation and $u$ be an update function. If $(\sigma, u)$ is winning strategy for $G$, and $M$ is the LTS obtained by applying the conversion shown in Section 3.5.2, then it holds that $M$ is a solution for $\mathcal{E}$.*

*Proof.* Consider a joint computation $p = (s_{e_0}, s_{M_0}), \ell_0, (s_{e_1}, s_{c_1}), \ell_1, \ldots$ of $E||M$. Recall that states in $E||M$ are of the form $(s_{e_0}, s_{M_0})$ where $s_M = (m, s_e, \alpha_1, \ldots, \alpha_k)$. Now, by construction and the fact that $E$ is deterministic, we know that for every fluent $\mathit{fl}_i$ and for every $j \geq 0$ we have $s_{M_j} = (m_j, s_{p_j}, \alpha_1^j, \ldots, \alpha_k^j)$ and $\alpha_i$ is the truth value of $\mathit{fl}_i^j$ at time $j$. Then, for some $j \geq 0$ and $s_M \in S_M$ we show that $s_{M_j} \in \phi_i$ iff $\phi_i$ is true at time $j$ and similarly for $\gamma_j$. In addition, from the fact that the computation $s_{M_0}, \ell_0, s_{M_1}, \ell_1, \ldots$ is a computation of $M$, the sequence $s_{M_0}, s_{M_1}, \ldots$ is the product of a play in $G$ that is consistent with $(\sigma, u)$ and that $(\sigma, u)$ is a winning strategy, it follows that this play in $G$ correspond to a trace in $M$ such that if the assumptions holds then the system goals will do so. Finally, given that $(\sigma, u)$ is winning, it only produces infinite plays showing that $E||M$ states not satisfying $\rho$ are not reachable and there are no deadlocks. □

## 3.6 Algorithm

In this section we present the algorithm implemented extending the MTSA tool set [DFCU08]. The algorithm is based on ideas of [JP06]. Implementation details are provided in Section 6.1.

Intuitively, the algorithm aims to avoid, through restricting controllable actions, cycles of states satisfying all the assumptions but not all of the goals. The existence of such cycles would allow for traces in which the controller looses the GR(1) game. In order to avoid such cycles the algorithm searches, for every state, a strategy that guarantees satisfaction of all goals. To do so, it chooses an order in which it will attempt to satisfy the goals. The algorithm applies a fixed point iteration for computing the best way each state has to satisfy the next goal. In order to measure the "quality" of different successor states with respect to satisfying the next goal, a ranking system is used [Jur00]. The rank for a particular successor will measure the "distance" to the next goal in terms of the number of times that all assumptions will be satisfied before reaching the goal. If this number tends to infinity then this means that from the current state a trace is possible in which the environment assumptions hold infinitely often but the system goals do not. Hence, such state should be avoided by the strategy for the controller.

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

most $k$ times, $\phi_1$ through $\phi_{l-1}$ will hold at least $k+1$ times and assumptions $\phi_{l+1}$ through $\phi_n$ will hold at least $k$ times. $R(s) = \infty$ means that $s$ is a *loosing* state, i.e. from $s$ there is no strategy for the controller that can avoid a trace which satisfies infinitely often all assumptions but does not satisfy infinitely often all goals. A *ranking system* for $G$ and $\varphi$ is a sequence $R_1, \ldots,$ $R_m$ of ranking functions, each associated with a specific goal $\gamma_j$.

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

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

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

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

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

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

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

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

The algorithm for solving the GR(1) game consists of three steps. First, it initialises the ranking system so that $R_j(s_g) = (0, 1)$ for all states and goals. Second it iterates until the ranking system is stable. If it is not stable, then the rank for some state and goal needs to be incremented, and stability is checked again. It is known that every rank greater than $(max_j\, max_i\, |\phi_i - \gamma_j|, n)$ is effectively equivalent to $\infty$, where $|\phi_i - \gamma_j|$ is the number of states in $G$ that satisfy $\phi_i$ and do not satisfy $\gamma_j$, which guarantees termination of the algorithm. Finally, if a stable ranking in which the initial state has a non-infinite ranking for the first goal ($R_1(s_{g_0})$) then a winning strategy is constructed. This last step is supported by the following theorem.

**Theorem 3.5.** (Algorithm Soundness) *If $R_1$, ..., $R_m$ is a stable ranking system, then for every state $s_g$ such that $R_1(s_g) \neq \infty$ there exists a winning strategy from $s_g$.*

*Proof.* Let $M = \{1, \ldots, m\}$ be the memory range of the strategy. The memory is updated by $u(v, s) = v$ if $s \notin G_v$ and $u(v, s) = v \oplus 1$ otherwise. The function $\sigma(v, s) = \{s' \in \Gamma^+ \mid s' \prec_j s\}$. By definition of stability, $\sigma(v, s)$ includes all uncontrollable successors of $s$. It is simple to see that following

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

Table 3.1: Ranks for states in Strategy $\sigma_R$

this strategy the reachable states are always contained in $S$ and the reachable states have a finite rank.

Consider a computation induced by this strategy $p = s_0, s_1, \ldots$. Let $j_0, j_1, \ldots$ be the sequence of memory values used by $u$ and let $r_0, r_1, \ldots$ be the sequence of ranks, where $r_i = R_{j_i}(t_i)$. The only way in which $j_{i+1} \neq j_i$ is if $G_{j_i}$ is visited by $t_i$. If for infinitely many locations $j_{i+1} \neq j_i$ then the computation visits all $G_j$ infinitely often. Otherwise, from some location $i$ we have for all $i > i_0$ $G_i = G_{i_0}$. Consider the sequence of ranks $r_{i_0}, r_{i_0+1}, \ldots$. By assumption, for all $o$ we have $r_{i_{o+1}} \leq r_{i_o}$ and furthermore if $r_{i_o} = (k, l)$ and $t_{i_o} \in A_l$ then $r_{i_{o+1}} < r_{i_o}$. By well-foundedness of $\mathbb{N} \times \{1, \ldots, n\}$ we conclude that from some point onwards $r_i$ is constant and for some $l$ we have $A_l$ is visited finitely often. It follows that all computations are in $\varphi$ and the strategy is winning as required. □

In Table 3.1 we show the rank values for states in $\sigma_R$, the strategy shown in Figure 3.7(b). The columns represent states in the strategy and the rows, which goal is being considered. The ranks are mostly $(1, 1)$ since from most of states for both goals, $\dot{a}$ holds before $\dot{g_i}$ hold for $i \in \{1, 2\}$. As expected, the rank for $(4, g_1)$ and $(4, g_2)$, according to $\dot{g_1}$ and $\dot{g_2}$ respectively is $(0, 1)$.

The construction of a winning strategy $(\sigma, u)$ from a stable ranking where $R_1(s_{g_0}) \neq \infty$ is straightforward: The strategy will attempt to reach goals in

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

What remains is to show that the algorithm is complete.

**Theorem 3.6.** (Algorithm Completeness) *If there is a winning strategy from $s_g \in G$, there exists a stable ranking system $R_1$, ..., $R_m$ such that for every $s_g \in S$ and $j \in \{1, \ldots, m\}$ we have $R_j(s_g)$ is either $\infty$ or $(k, l)$ with $k \leq max_l|\phi_l - (\gamma_j)|$.*

*Proof.* An analogous proof is provided in [KPP05]. □

The algorithm in Figure 3.6 computes a stable ranking such that for every state $s_g \in T$ if $s_g$ is winning for controller (i.e. $R_1(t) < \infty$). At high level, the algorithm has two major parts, the initialisation and the stabilisation. The initialisation sets the initial rank for every state in the game and initialises the queue of states *pending* to be processed. A state is added to *pending* if it satisfies no guarantee and satisfies assumptions. All the states in every ranking function are initialised with $(0, 1)$ (i.e. the minimum possible rank)

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

Figure 3.8: Pseudo-code of algorithm for solving SGR(1) games

except for states such that $\Gamma^- \cup \Gamma^+ = \emptyset$ which are initialised with $\infty$. Notice that states with $\infty$ rank are those which either do not satisfy $\rho$ or are deadlock states in $E$. The stabilisation part is a fixed point that iterates on *pending* until is it empty. We now describe the stabilisation procedure. The function `is_stable(state,g)` returns true if the `g`-th ranking function is stable for `state`. The function `unstablePred(state,g)` returns a set of pairs of predecessors of `state` and a rankings `g` for which the ranking is unstable. The function `best(state,g)` returns the value of $best(\texttt{state}, \texttt{g})$, as defined above. Finally, `inc((`$k,l$`),state,g)` returns $(0, 1)$ if `state` is in $\gamma_{\texttt{g}}$, it returns $(k, l)$ if `state` is not in $\texttt{assumption}_l$, and it returns the minimal value greater than $(k, l)$ otherwise. Notice that `inc(`$\infty$`,state,g)` is $\infty$ and if $n = max_l(|\phi_l - (\gamma_{\texttt{g}})|)$ and `state` is in $\phi_m - \gamma_{\texttt{g}}$ then $inc((n, m), \texttt{state}, \texttt{g})$ is $\infty$. This algorithm computes the minimal existing stable ranking. Based on the ideas in [EWS05] and [JP06], this algorithm can be implemented to work in time $O(m \cdot n \cdot |S|^2)$.

The following theorem follows from the algorithm described above and Theorems 3.6 and 3.5.

**Theorem 3.7.** *Given a SGR(1) control problem* $\mathcal{E} = \langle E, H, A_C \rangle$*, where* $H = \{(\emptyset, I), (As, G)\}$*,* $I = \Box\, \rho$*,* $As = \bigwedge_{i=1}^{n} \Box \Diamond\, \phi_i$ *and* $G = \bigwedge_{j=1}^{m} \Box \Diamond\, \gamma_j$*.* $\mathcal{E}$ *is solvable in* $O(m \cdot n \cdot |S|^2)$ *time.*

## 3.7 LTS Control Problem Determinacy

The LTS control problem leads naturally to the concept of a game, where one player (the controller) chooses which actions to enable and the other player (environment) chooses which actions to follow.

The following lemma shows that determinacy results 2.2 in the field of two player games also hold for the above mentioned LTS control approach.

**Lemma 3.1.** *(*LTS Determinacy*) Given* $\mathcal{E} = \langle E, \varphi, A_C \rangle$ *an LTS control problem where* $E = (S,\ A,\ \Delta,\ s_0)$*. If* $\mathcal{E}$ *is unrealisable then there exists an LTS* $E'$ *such that from every state in* $E'$ *all actions in* $A_C$ *are enabled, and for every possible controller* $M$*, we have* $E' \| E \| M \models \neg\varphi$*.*

*Proof.* Follows from Gurevich and Harrington [GH82] results that showed that finite-memory strategies suffice to win Muller games, and the proof for solving LTS control problems with two-player games.

## 3.8 Evaluation

In this section we show the results of applying our technique to different case studies. The case studies were performed using the tool we present in Chapter 6. The full version of these case studies can be found in [D'I]

### 3.8.1 Autonomous Vehicles

We present a variation of the case study originally presented in [HSMK09b] in the context of self-adaptive systems.

Consider a situation in which a two-bedroom house has collapsed leaving only one small passage between the two rooms (referred to as *north* and *south* rooms). The entrance door of the house is in the south room and there is a group of people trapped in the north room. The task of bringing aid packages to the occupants trapped inside is too dangerous for humans, hence, a robotic system is required. A robot that has a wide range of movements and has an arm capable of loading and unloading packages. The robot has a number of sensors which can be used, among other things to check if a loading operation, which is of a significant amount of complexity and uncertainty, is successful or not. The situation is complicated by the presence of a door between the two rooms. The door cannot be opened by the robot. However, although the structure is unstable, it is known that once the door is open, it can be held open by the trapped occupants.

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

crossing it. Whenever the robots moves it senses the destination position from the environment (i.e. $southNear$, $southFar$, $northNear$ $northFar$).

The aim is to automatically synthesise a behaviour model that will control the robot and will achieve the task of retrieving aid packages from the outside to the room where the occupants are trapped. Hence, the set of controllable actions is the set of actions that correspond to the actions that can be performed by the robot and its arm (e.g. $moveNorth$, $getPackage$, $putPackage$) excluding actions not controlled by the robot (e.g. $openDoor$, $getPackageOk$, $getPackageFail$, $northFar$).

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

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

The liveness part of the goal states that the robot must be at the far end of north room and have just unloaded infinitely often: $G = \square \diamondsuit(\dot{northFar} \wedge \dot{putPackage})$. The control problem, as defined up to this point is not realisable, as the robot has no guarantees that the door will be open for it to move freely to and from the north and south rooms. Some assumption on the behaviour of the door must be included. We introduce the assumption that the door is infinitely often open ($As = \square \diamondsuit \dot{doorOpen}$). This is still

insufficient, since the robot has no control over the success or failure of attempting to load an aid package using the arm. Thus, packages may never be loaded successfully. This shows that there is a missing assumption stating that if the robot attempts to load a package it will eventually succeed: $\square(\dot{getPackage} \rightarrow \diamondsuit \dot{getPackageOk})$. When assumptions regarding the door being open and package loading being successful are included in the SGR(1) LTS control problem, it becomes realisable. Hence, a solution exists and is constructed automatically by the tool.

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

Intuitively, the assumptions should represent the requirements of a reasonable environment. These assumptions will enable the robot to make

progress. Compare, for example, the assumptions $\square \diamondsuit \dot{getPackageOk}$ and $\square \diamondsuit (\dot{getPackageOk} \vee \neg \dot{getPackage})$. The first, superficially matches our intuition that the environment should make it possible for the machine to pick up a package. It is, however, too strong as the machine may not try to pick packages at all. The second, depicts a reasonable environment: a machine that keeps trying to pick up a package should be allowed to do so. Hence, it is not surprising that the second assumption satisfies the assumption compatibility condition 3.5. Furthermore, $As = \square \diamondsuit (\dot{getPackageOk} \vee \neg \dot{getPackage}) \wedge \square \diamondsuit \dot{doorOpen}$ allows synthesising a solution to the LTS control problem $\mathcal{E} = \langle E, \{(\emptyset, I), (As, G)\}, \{\alpha ARM \cup \alpha ROBOT\}\rangle$ that tries as much as possible to load packages. In other words, for this case study we successfully synthesised a behaviour model that controls the robot arm which ensures that if the assumptions are satisfied by the environment the machine satisfies its goals.

The synthesised behaviour model is too big to be shown here. Nevertheless, the tool is available at [DFCU08] and, the and FSP source code for the case study can be found at in [D'I].

Comparing our approach to the original case study, note that in [SHMK07] *(i)* no assumptions are explicitly given, and as a result *(ii)* no guarantees can be given as to whether the synthesised controller will satisfy the goal of delivering aid packages, and *(iii)* although under certain conditions the plan synthesised by [SHMK07] will work, it is not clear what those circumstances are. We overcome these issues by modelling the robot, its arm and sensors, and the house restrictions separately from the controller goals. This allowed

us to discover implicit environmental assumptions. Specifically, in order to guarantee that the robot will successfully deliver aid packages infinitely often, the door must be open infinitely many times. Furthermore, the arm has to successfully pick up aid packages infinitely many times. In other words, by applying Jackson's [Jac95a] approach, we discovered the environmental assumptions required to guarantee satisfaction of the goals, shown how the assumptions can be explicitly encoded in our SGR(1) control problem and checked the assumptions to be compatible with the environment.

### 3.8.2 Purchase & Delivery

In [PBB+04] a case study involving the synthesis of a plan for composing and monitoring of distributed web services is presented.

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

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

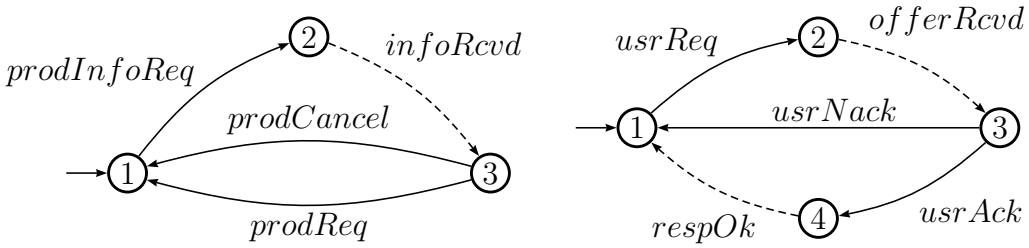
refuses the offer, the composed service is expected to reach a state in which both the purchase and delivery requests have been cancelled.

We restrict the analysis and synthesis to a failure-free version of the problem. Hence, both furniture-sales and shipping services always respond positively on a request by the model to be synthesised. Even though the failure-free assumption may seem to restrictive it allows us to handle some of the, so-considered, failures in [PBB+04]. For instance, the user refusing a furniture-delivery pair is considered a failure, which violates the intuition that failures are not controlled by users, instead they are supposed to happen unexpectedly, e.g. a server crashes.

Even though in [PBB+04] a behaviour model for this environment in which failures are possible is synthesised, there are no guarantees that the goal of satisfying purchase requests is achieved. In fact, achieving the goals stated in [PBB+04] requires assuming some progress on the environment and fairness conditions on the success of operations on the furniture-sales and shipping services, as we show below.

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

The interface of the furniture-sales service, described by the LTS in Figure 3.9(a), allows requesting for information on a particular product ($prod\ Info\ Req$) then once the information has been received ($infoRcvd$) it is pos-




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

Figure 3.9: Furniture-sales service interface and User

sible place a request for the product (*prodReq*). The protocol to interact with the shipping service is analogous.

A model describing how the user interacts with the composed web-service is shown in Figure 3.9(b). The user can place a request for some product (*usrReq*) then he may get either an offer for product and shipping combination (*offerRcvd*). If an offer is received, the user can either confirm the order (*usrAck*) or decline it (*usrNack*). If the order is confirmed, the user waits for its product to be shipped as arranged (*respOk*).

There are several safety prescriptions for the controller-to-be. (*i*) The service should only check for some product or shipping information if the user placed a request first. This restricts the composed web service from spontaneously generating queries without a triggering user request. (*ii*) It is only possible to offer the user with a combination of product and shipping service if both services have confirmed availability. (*iii*) Both product and shipping requirements should be placed only if the user acknowledged the purchase. (*iv*) The service will only cancel product or shipping requests if the user does not acknowledge the offer he received. (*v*) The service only flags that the request has been cancelled when both product ordering and shipping services

have cancelled the request. ($vi$) The service finishes successfully only after the product ordering and shipping services have successfully handled their requests. These requirements can be easily expressed in FLTL. For the full specifications the reader is referred to [DFCU08].

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

Without any collaboration of the environment it is not possible for a controller to satisfy its liveness goals. For instance, if the user never acknowledges for purchase and delivery of furniture, then it will not be possible to fulfil controller goals. Consequently, we will be able to generate a controller only if it is possible to assume that the environment will acknowledge requests infinitely many times, in FLTL this is $\varphi = \Box \diamondsuit \dot{usrAck}$. However, the environment cannot acknowledge product and shipping combinations if the controller does not provide such combinations. In other words, the environment cannot fulfil this assumption on its own, which shows that $\varphi$ is not capturing our intuition correctly. Furthermore, $\varphi$ does not satisfy the compatibility condition 3.5, which as shown before lead to undesired controllers.

Our assumptions must state that the environment has to acknowledge combinations only if he received an offer $\Box(\dot{offerRcvd} \rightarrow \diamondsuit \dot{usrAck})$. As shown in Section 3.3 this assumptions can be expressed with the FLTL formula $\varphi' = \Box \diamondsuit \neg \dot{OfferAckd}$, where $\dot{OfferAckd} = \langle offerRcvd, usrAck, \bot \rangle$. This means that if the environment receives infinitely many offers it has to acknowledge them infinitely often, which is compatible with the environment

and captures our intuition more closely.

We modelled the case study as an SGR(1) problem and applied the MTSA tool set to generate a controller, which is shown in Figure 3.10. As one may expect the controller only synchronises the message passing between the user, furniture and delivery services. The environment model is a compatible with respect to the assumptions that customers confirm infinitely many products and delivery options. Thus, the resulting controller is guaranteed to be non-anomalous and the environment assumptions under which it achieves its goals are explicit.

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

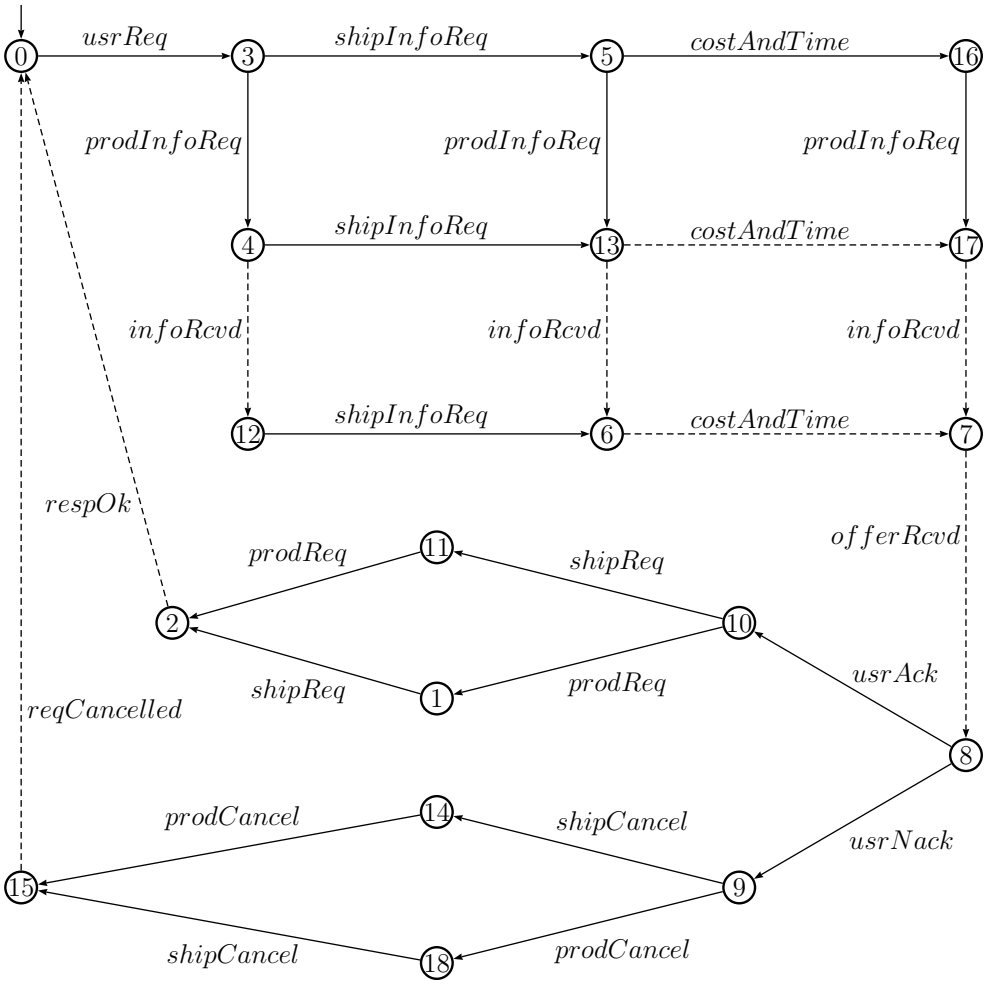



Figure 3.10: Controller for serving furniture requests

### 3.8.3 Bookstore

The web-service composition scenario in [IT07], is in a sense similar to that of Pay & Ship, in that two services must be coordinated to provide a more complex service. The main difference is that Inverardi et al. provide no explicit liveness goals for the controller nor liveness assumptions on the environment behaviour.

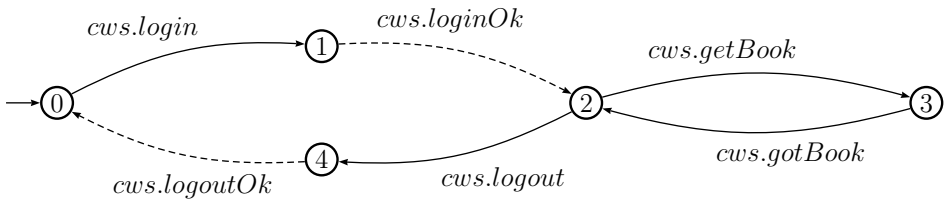
Following the world-machine approach, we describe the environment models as the parallel composition of models for the composed web-service (CWS) and the books search and order services.

More specifically, the composed web-service (CWS), for which the interaction protocol is presented in Figure 3.11(a), must coordinate a service for search and order books, and a payment service. The interaction protocol for the search and order service is shown in Figure 3.11(b). Once the user has logged in she can either choose to search and order books or to logout terminating the interaction. Similarly, the payment service (Figure 3.11(c)) requires a log in to enable for payments to occur. Moreover, while the user is logged in, she can place as many payments as required.

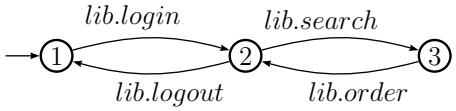
As in [PBB+04], the main goal of the safety prescriptions is to prescribe the ordering between the actions of the services involved. For instance, the composite web-service can only be considered logged in, if both the (sub-)services have been successfully, logged in.

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

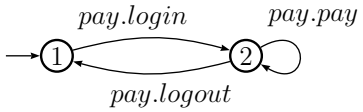
($i$) The user logs on (off) triggering $cws.login$ ($cws.logout$) action, then the service should log into both payment and library services triggering $pay.login$ ($pay.logout$) and $lib.login$ ($lib.logout$) respectively, then ($ii$) only after both search and payment services have logged in the CWS must inform the user that the login (logout) operation was successful by triggering the $cws.loginOk$ ($cws.logoutOk$) action. Naturally, ($iii$) the CWS should not attempt a login



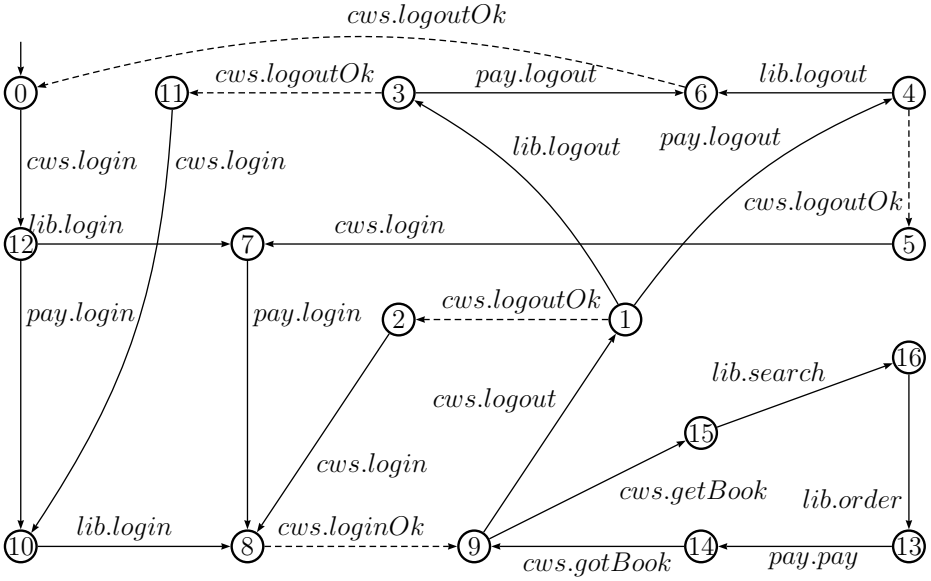

(a) CWS Interaction protocol




(b) Service for book search and order




(c) Payment service




(d) Search-Order-Pay web-service

Figure 3.11: Bookstore case study models

action on neither of the (sub-)services if there no *cws.login* action first. The procedure of searching, ordering and finally paying for books must also be coordinated by the CWS. (*iii*) CWS can only search for books *lib.search* if a request has been placed *cws.getBook*, and either of them can only happen

if the user has been successfully logged in ($cws.loginOk$). ($iv$) Only after a book has been ordered ($lib.order$), CWS triggers the payment ($pay.pay$). Finally, ($v$) CWS must only confirm that a book has been successfully bought $cws.gotBook$ it is successfully paid for it. There is only one liveness goal for the controller. It must buy (i.e. successfully order and pay) books infinitely often, i.e. $\square \diamondsuit cws.\dot{g}otBook$. However, this cannot be guaranteed if the user (i.e. environment) does not try to get books infinitely many times, i.e. $\square \diamondsuit cws.\dot{g}etBook$. Note that the environment can only try to get books if some other actions have occurred before, i.e. logins. Hence, it may seem that such assumptions would be non-compatible, it turns out that they are. Intuitively, the actions required for the environment to get books are dependant on the environment solely, i.e. $cws.login$.

We defined the SGR(1) problem with system goals $\square \diamondsuit cws.\dot{g}otBook$ and environmental assumptions $\square \diamondsuit cws.\dot{g}etBook$. The controller we obtained, shown in Figure 3.11(d), guarantees that for every trace in which the environment assumptions are satisfied the controller goals will also be satisfied. Note that since the environment assumptions are compatible the controller guarantees that is going to do it best to fulfil the system goals.

It is important to note that, since the technique used in [IT07] does not allow for liveness goals or environmental assumptions, the achievement of the main goal of the service, i.e., to sell books, cannot be guaranteed. On the contrary, as we specify the case study as a SGR(1) control problem, we can explicitly provide with the liveness conditions required to successfully sell books. Such conditions are required but not sufficient, since without any

collaboration of the environment the goals cannot be achieved. As said above, the environment has to try to get books for the controller to successfully sell them.

### 3.8.4 Production Cell

This case study has been explained as a running example in Section 3.1. We presented the descriptive and prescriptive statements for the problem and presented some observations on the generated controller.

Since the size of the models is too big to be depicted[1], we show the controller for a smaller version, which has only one tool (an oven) and can only process one instance of each product type at a time. The synthesised controller is shown in Figure 3.12. As noted in Section 3.1 the controller must "remember" if it has been postponing one type of product for too long. Consequently, the algorithm adds memory to the original states encoding the last product type processed in order to guarantee the system goals. The controller waits for products of type $A$ to be processed first (see states 4 and 6) regardless of whether there are products of type $B$ ready to be processed (see states 5 and 6). It then does the same for products of type $B$ (see states 10, 2 and 1).

[1]For a full description of this case study we refer the reader to [D'I]

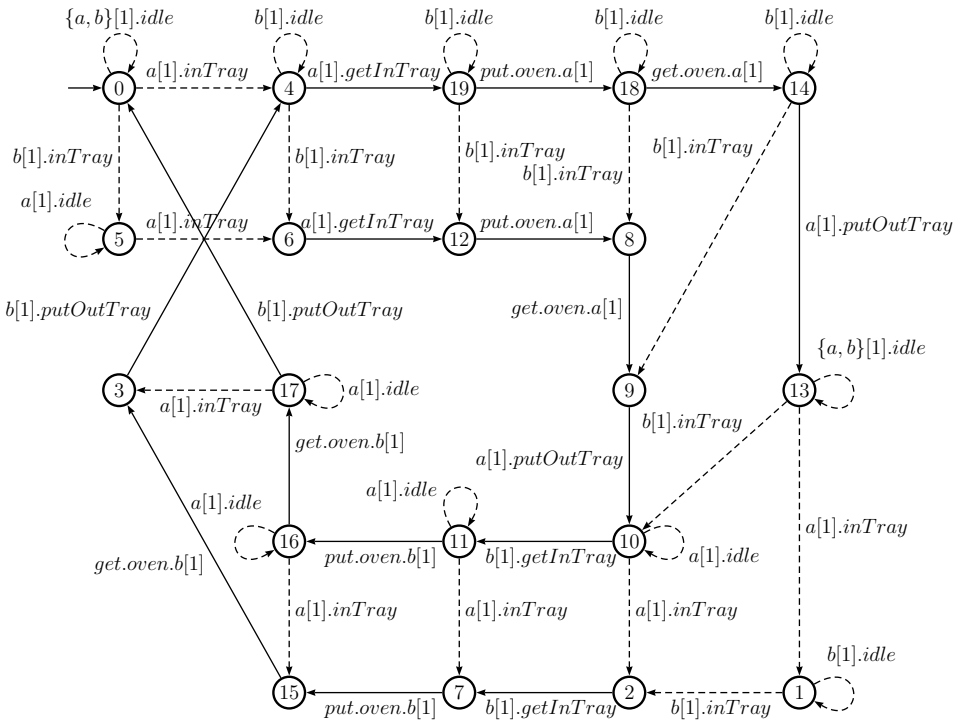



Figure 3.12: Controller for reduced Production Cell

## 3.9 Limitations

Although we have successfully applied SGR(1) to a number of case studies from different domains, showing that it is useful in many real life problems, there are some situations in which it might not be expressive enough. In the next sections we analyse some limitations of SGR(1) and motivate techniques, presented further on in this thesis, that overcome such limitations.

### 3.9.1 Partially Defined Environment Models

In practice, requirements engineering is not a waterfall process. Engineers do not build a complete description for environment model before they construct or synthesise a controller. Typically the environment model is elaborated incrementally. Furthermore, multiple variations of partial models are explored to asses risk, cost and feasibility [vL09]. In particular, a key question that drives requirements engineering forward and consequently drives elaboration of a partial description of the environment is if it is feasible to extend the environment model such that a controller for the extension can be synthesised.

In this context, SGR(1) and, in general, existing controller synthesis techniques are are not well suited as they require complete descriptions of the environment model. Indeed, the environment is modelled with two-valued state machines such as LTS. In order to support partially specified environment models, more flexible formalisms are required.

An appropriate formalism to support modelling when behaviour information is lacking is one in which currently unknown aspects of behaviour can be explicitly modelled [UBC09b]. A number of such formalisms exist such as Modal Transition Systems (MTS) [LT88] and Disjunctive MTS [LX90]. Partial behaviour models can distinguish between required, possible, and proscribed behaviour.

In Chapter 5, we define controller synthesis in the context of partially specified environment models. More specifically, we study the problem of checking the existence of an LTS controller (i.e. controller realisability) capable

of guaranteeing a given goal when deployed in a completely defined LTS domain model that conforms to the partially defined domain model given as an MTS.

### 3.9.2 Fallible Environment Models

SGR(1) is design to work with idealised environment models. In other words, environment models in which controllable actions are always successful. For example, consider the production cell case study presented in Section 3.8.4. The robot arm is modelled as a perfect device that never fails, e.g. it always succeeds when picking up a product. However, the real arm may have some difficulties in picking up products from the tools and trays, e.g. it may miss the target location for loading or unloading due to traction problems. Modelling such failures would lead to an unrealisable control problem as synthesis algorithm assumes that the arm may fail in picking up products indefinitely. This is due to a limitation of SGR(1) that treats failures as environment (i.e. monitored) actions and consequently treats them as actions that can systematically be used to beat the controller.

However, failures can be the result of errors that are independent from the environment behaviour (e.g. have a stochastic behaviour). In these cases, requiring the controller to achieve its goals when the environment controls failures is too strong and may lead to SGR(1) claiming the absence of a controller when under a weaker assumption such controller exists. For example, the environment cannot control the arm failing to pick up products if it is

due to imprecisions in locating the target to be picked up (as if that was not the case, retrying would only result in failure). In such case, the best controller we can get is one that that after a failure retries to pick up. In general, assuming that failures are independent from the environment would lead to controllers that after a failure keep retrying and never give up. Note that, such controllers are interesting when failures are not persistent in time, as if that was not the case, retrying would only result in failure. For instance, if failures of the robot arm are the result of a broken component of the arm that controls the pressure of the "hand", trying to pick up would fail even if the controller retries. Hence, even when it is possible to assume failures being independent of the environment, extra conditions must be studied.

In Chapter 4 we present a synthesis technique that works for environment models where failures as result of controllable actions are explicitly modelled. 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.

### 3.9.3 Non-Deterministic Environment Models

SGR(1) is restricted to deterministic environment models. Such restriction can be limiting in some cases. Consider a variation of the production cell case study (Section 3.8.4) where after a product is placed in the drill, it may block due to an internal error. Such behaviour cannot be captured by a deterministic model.

In general, as LTS controllers guarantee the satisfaction of their goals through parallel composition, having nondeterministic environment models means that the controller would not be able to know the exact state of the environment model. This leads to imperfect information, as the controller would only be able to deduce which *set* of states the environment model is in. Such a setting is much more computationally complex than synthesis with full information.

Only in recent years a few approaches towards imperfect information have started to emerge [RCDH07]. However, most of them are far from actual applications. In a setting of a nondeterministic environment model but giving the controller full information of actions and states, our technique works with no changes.

Hence, we envisage as future work relaxing the requirement on determinism for the environment model. In fact, as showed above, this is closely related to incomplete information of the controller of events controlled by the environment, an area that we also intend to further investigate.

# Chapter 4

# Synthesis for Fallible Environments

In this chapter we present a technique for synthesising controllers that achieve their goals even when their environment exhibits failures.

## 4.1 Motivating Example

In this section we discuss motivation for the 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 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 4.1. 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 it succeeds 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

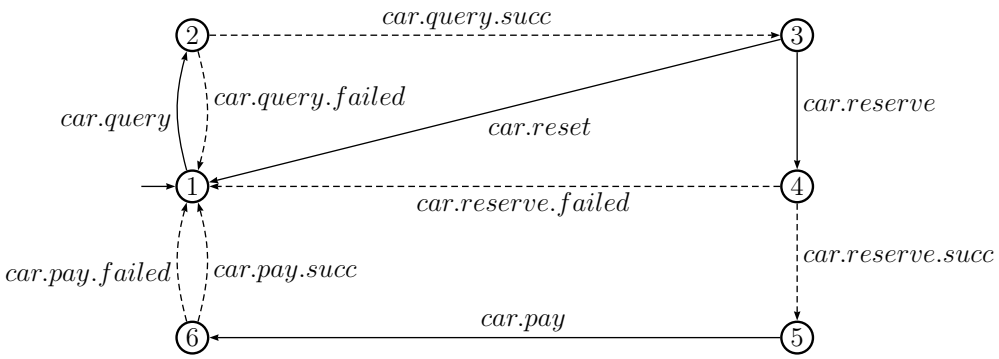



Figure 4.1: Car Booking Service

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 ($A_1 = \Box \Diamond PendingPackageRequests$) and prescriptions on the orchestration ($G_1 = \Box \Diamond package.deliver$). 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 produce 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 [DBPU10] 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 environment 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 [Fra86]. Strong fairness is not supported by polynomial time algorithms such as the one presented in Section 3.3. It requires exponential algorithms such as [EJ88]. 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 [PP06]. More precisely, the size of the controller is always $N \times k!$ where $N$ is the size of the environment model

[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.

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.

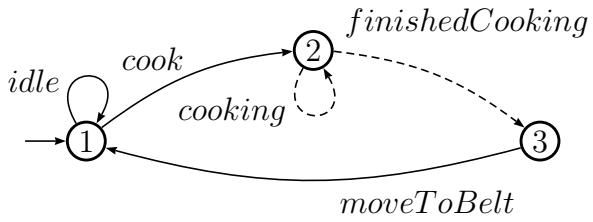



Figure 4.2: Ceramic Cooking Process

## 4.2 Failures Fairness

We consider controller synthesis in the context of environments that exhibit failures. We call this setting *synthesis for fallible domains*. 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 4.2, 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 $G = \Box \diamondsuit \dot{moveToBelt}$ and an assumption $A = \Box \diamondsuit \neg \dot{cooking}$. 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 $\mathcal{E} = \langle E, H, A_C \rangle$,

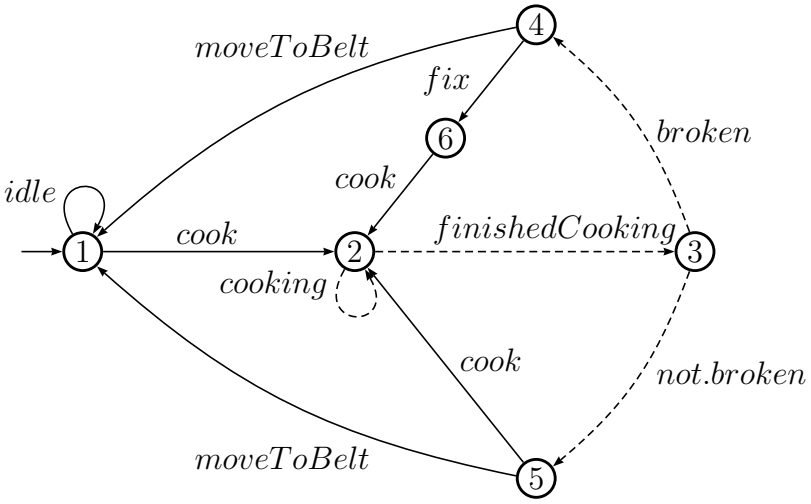



Figure 4.3: Failing Ceramic Cooking Process ($E_2$)

where $H = \{(A, G)\}$ and $A_C = \{idle, cook, moveToBelt\}$. The solution to $\mathcal{E}$ 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 4.3). 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.

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.

Recall that, $\dot{\ell}$ is the short for the fluent that indicates that $\ell$ has just occurred (namely $\dot{\ell} =< \ell, A \setminus \{\ell\} >$, where $A$ is the complete set of actions).

Intuitively, we consider a failure to be an unexpected environment response to an controlled action. For example, the *car.payment* action in the travel agency example is expected to lead to a *car.payment.succ* action, however, payment may not succeed, resulting in a *car.payment.failed* action in the environment. In other words, we assume that system controlled actions can have two distinguished responses that can be monitored by the controller (i.e. controlled by the environment), one that denotes the successful, expected, or sunny day response to the system action and the other the undesired response from the environment.

**Definition 4.1.** (Try-Response) *Given an LTS* $E = (S, A, \Delta, s_0)$*, and a set of actions* $A_C \subseteq A$*, we say that a set* $T = \{(try_i, suc_i, fail_i)\}$ *is a* try-response *set for* $E$ *if the following hold for all* $i$*:*

1. $try_i \in A_C$, $suc_i, fail_i \in A \setminus A_C$ 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. $\Box(\dot{try}_i \Rightarrow \mathbf{X}(\neg \dot{try}_i \mathrm{W}(\dot{fail}_i \vee \dot{suc}_i)))$,
5. $\Box((\dot{fail}_i \vee \dot{suc}_i) \Rightarrow \mathbf{X}(\neg(\dot{fail}_i \vee \dot{suc}_i) \mathrm{W}\ \dot{try}_i))$, *and*
6. *For all* $s \in S$, $fail_i$ *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 4.3 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 $I = \Box \dot{moveToBelt} \Rightarrow \neg Broken$ to $\mathcal{E}$, where $Broken$ is a fluent defined as $\langle broken, cook \rangle$. In other words, attempting to solve $\mathcal{E}_2 = \langle E_2, H, A_C \rangle$, where $H = \{(\emptyset, I), (A, G)\}$.

Unfortunately, $\mathcal{E}_2$ 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 environment. Another attempt at automatically building a controller could be to strengthen the assumption $A$ in $\mathcal{E}_2$ to be $A' = \Box \diamondsuit \neg \dot{cooking} \wedge \Box \diamondsuit \dot{not.broken}$. This leads to $\mathcal{E}'_2 = \langle E_2, H', A_C \rangle$,

where $H' = \{(\emptyset, S), (A', G)\}$.

The problem with $\mathcal{E}_2'$ 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 $\Box \diamondsuit \dot{not.broken}$ does not hold. Hence, the controller has no obligation to achieve $G$. Formally, $E_2 \| M \models \Box \diamondsuit A$ implies $E_2 \| M \models \Box \diamondsuit G$ holds if $E_2 \| M \not\models \Box \diamondsuit A$. Clearly, $A'$ is not a reasonable assumption for the environment behaviour. The environment depends on the controller to achieve $A'$. In van Lamsweerde's terms [vLL00], the assumption is not realisable by the environment. As we showed in Section 4.4, 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 4.4).

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. $\Box \diamondsuit \dot{cook} \Rightarrow \Box \diamondsuit \dot{not.broken}$). However, this condition amounts to requiring strong fairness [Fra86] of *not.broken* actions which cannot be encoded in SGR(1). Although more general controller synthesis techniques can deal with strong fairness [EJ88], 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

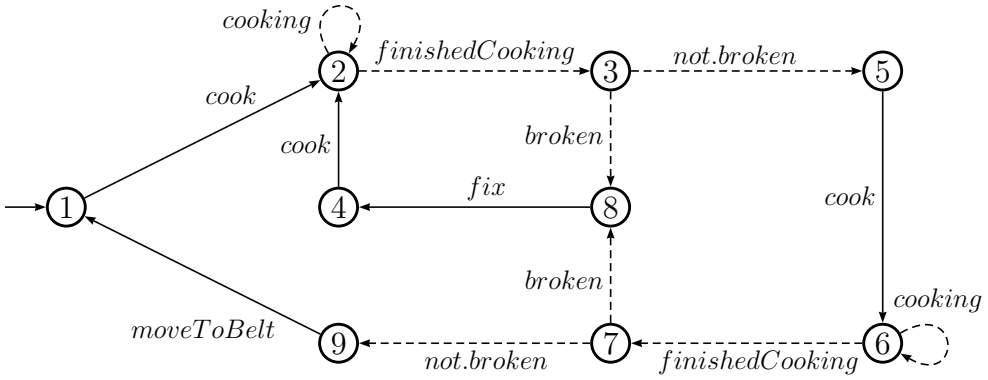



Figure 4.4: Ceramics cook-twice controller

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 4.4). 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 $\Box \diamondsuit \dot{cook} \Rightarrow \Box \diamondsuit \dot{not.broken}$ 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, attempting first cook of a piece (transition from 1

to 2 in Figure 4.3) 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* $E = (S, A, \Delta, s_0)$ *and a try-response* $T = \{(try_i, suc_i, fail_i)\}$ *for* $E$. *A trace* $\pi \in tr(E)$ *is* t-strong fair *with respect to* $E$ *and* $T$ *if for all* $(try_i, suc_i, fail_i) \in T$ *and for all transitions* $t = (s, try_i, s') \in \Delta$ *the following holds:* $\pi' \models \Box \Diamond try'_i \Rightarrow \Box \Diamond (\neg try_i \text{ U } suc_i)$, *where* $\pi' = \varepsilon'|_{A \cup \{try'_i\}}$, $\varepsilon' = \varepsilon|_{[s.try_i.s'/s.try_i.try'_i.s']}$, *and* $\varepsilon$ *is an execution of* $E$ *such that* $\varepsilon|_A = \pi$.

Note that $w|_A$ is the projection of word $w$ over the alphabet $A$, and $w_{[v/v']}$ is

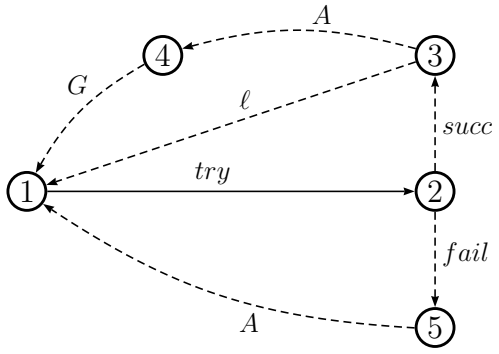



Figure 4.5: t-strong fairness is not enough

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 4.5. In this example $try$ is the only controlled action, $(try, succ, fail)$ the only try-response triple, $\ell$ 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 4.5)

and achieve assumptions when actions fail (state 5). However, a natural expectation is that the assumptions on the environment should be independent of failures; particularly because the choice of failure or success is understood as non-deterministic given that it abstracts the actual cause for failure and success.

If required to construct a controller for Figure 4.5 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* $E = (S, A, \Delta, s_0)$*, a try-response* $T = \{(try_i, suc_i, fail_i)\}$ *for* $E$*, and* $As$ *a set of FLTL formulas. A trace* $\pi \in tr(E)$ *is* Strong Independent Fair *with respect to* $As$

*if for all* $(try_i, suc_i, fail_i) \in T$ *and for all transition* $t = (s, try_i, s') \in \Delta$ *the following holds:* $\pi' \models \Box \diamondsuit try'_i \Rightarrow \Box \diamondsuit((\neg try_i \text{ U } suc_i) \wedge (\bigwedge_{i=1}^{n}(\neg \mathcal{F} \text{W} As_i)))$, *where* $As_i \in A$, $\mathcal{F} = (\bigvee_{j=1}^{n} fail_j)$, $\pi' = \varepsilon'|_{A \cup \{try'_i\}}, \varepsilon' = \varepsilon|_{[s.try_i.s'/s.try_i.try'_i.s']}$, *and* $\varepsilon$ *is an execution of* $E$ *such that* $\varepsilon|_A = \pi$.

In the next section we formalise the control problem with the fairness condition 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.

## 4.3 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 Section 3.3 by introducing failures and expectations on the fairness of the environment.

**Definition 4.4.** (Recurrent Success) *Given an SGR(1) LTS control problem* $\mathcal{E} = \langle E, H, A_C \rangle$ *and a try-response* $T$ *for* $\mathcal{R}$, *the solution for the* Recurrent

Success *control problem* $\mathcal{R} = \langle \mathcal{E}, T \rangle$ *is to find an LTS* $M$ *such that* $M$ *is a legal LTS for* $E$ *with respect to the set of monitored actions* $\overline{A_C}$, $E \| M$ *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* $\pi$ *in* $M \| E$ *the following holds: if* $\pi \models As_i$ *then* $\pi \models G_i$.

Note 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* $\mathcal{R} = \langle \mathcal{E}, T \rangle$ *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. Moreover, the controller for* $\mathcal{S}$ *can be used to control* $\mathcal{R}$.

*Proof.* Refer to Appendix A.1.

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 $\Box \diamondsuit G$ on a trace if on the same trace $\Box \diamondsuit As_i$ holds and also a finite number of failures occur (i.e. $\diamondsuit \Box \neg F$). An FMF problem can be coded as an SGR(1) problem where the goal is $\Box \diamondsuit (G \vee F)$.

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) (Full proof is given in Appendix A). 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 $\pi$ in $E\|M$ that satisfies the assumptions infinitely but not the goals. In $\pi$ 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 $\pi$ 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 $\pi$ 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).

We proved that it is possible to reduce an RSGR(1) problem to a SGR(1) problem for which it is assumed that failures occur only finitely many times. We now show how the assumption of finitely many failures can be handled as part of a SGR(1) control problem.

Let $\mathcal{E} = \langle E, H, A_C \rangle$ an SGR(1) control problem, where $H = \{(\emptyset, I), (A, G)\}$ and $A_C$ some set of controllable actions. Without loss of generality, $A$ and $G$ are singletons. To solve $\mathcal{E}$ under the assumption of finitely many failures means generate a controller $M$ such that $E \| M \models \Diamond \Box \neg F \Rightarrow (\Box \Diamond A \Rightarrow \Box \Diamond G)$. Distributing the negation of $F$ and replacing implication for disjunctions it follows that $E \| M \models \Box \Diamond F \vee \neg \Box \Diamond A \vee \Box \Diamond G)$. Finally, from distribution of $\vee$ over $\Box \Diamond$ it follows that $E \| M \models (\Box \Diamond A \Rightarrow \Box \Diamond (G \vee F))$.

## 4.4 Anomalous Controllers

Anomalous controllers, as we defined in Section 3.4, 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

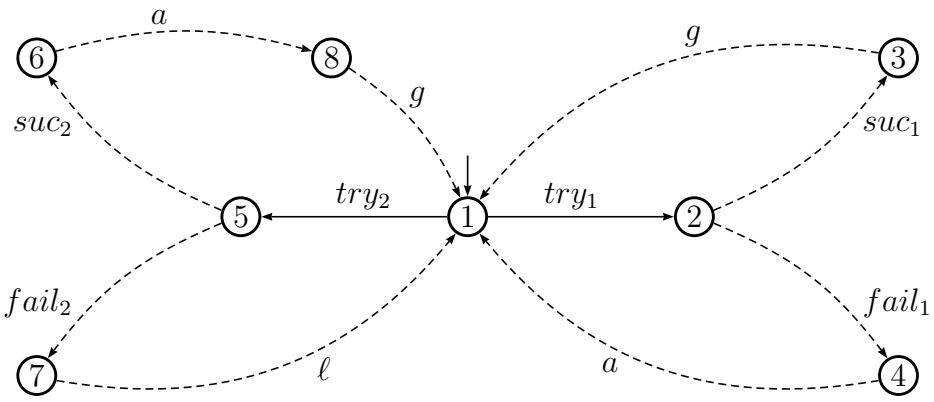



Figure 4.6: Environment $E$

assumptions from occurring or by forcing non strong independent fair traces.

The following definition of best effort controller extends that of Section 3.4 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 $As = \bigwedge_{i=1}^{n} \Box\Diamond A_i$. 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 $\sigma \in tr(E\|M)$ such that for all $\sigma'$ where $\sigma.\sigma' \in tr(E\|M)$, 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 $\sigma \in tr(E\|M')$, every $\sigma''$ such that $\sigma.\sigma'' \in tr(E\|M')$ either $\sigma.\sigma'' \models (\neg \bigwedge_{i=1}^{n} \Box\Diamond A_i)$ or $\sigma.\sigma''$ is not strong independent fair.*

Consider the RSGR(1) LTS control problem $\mathcal{R} = \langle \mathcal{E}, T \rangle$ where $\mathcal{E} = \langle E, H, A_C \rangle$, where $E$ is the LTS in Figure 4.6, $A_C = \{try_1, try_2\}$, $H = \{(As, G)\}$, $As = \Box \diamondsuit \dot{a}$ and $G = \Box \diamondsuit \dot{g}$. 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 fulfill 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 Section 3.4 for SGR(1) but not the one above for RSGR(1).

In Section 3.4 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 Section 3.4 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 $\mathcal{R} = \langle \mathcal{E}, T \rangle$, where $\mathcal{E} = \langle E, H, A_C \rangle$ 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 $\langle E_s, H', A_C \cup F\rangle$, 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 4.6 is not assumptions compatible with $A$. A controller $M$ which never takes $try_2$ nor $fail_1$ forces $E \| M$ to not satisfy $\square \diamondsuit A$, which means that the controller has no obligation of satisfying $false$. Hence, there is a solution for the problem $\langle E, H, A_C \cup F\rangle$, where $H' = \{(\emptyset, \text{true}), (A, \text{false})\}$.

Similarly to Section 3.4, the assumptions-compatibility condition is related to the definition of best effort controller. Intuitively, if the environment 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 $\mathcal{R} = \langle \mathcal{E}, T\rangle$ 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 Appendix A.2.

The theorem above is applicable for $E$ in Figure 4.6 since $E$ is not assumptions compatible with $A$; in effect, there are non best effort controllers for $\mathcal{E}$.

However, the orchestration problem discussed in Section 4.1 is assumptions compatible, the theorem applies and all solutions to the RSGR(1) orchestra-

tion 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 fulfills $A_1$.

## 4.5 Unsupported Traces

In the previous section 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 4.5. Assumption $A$ is compatible with the environment. Thus, solutions to the control problem are guaranteed to be best effort. Consequently, a controller that repeatedly at-

tempts *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 they are.

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 [Bel57] (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 environment model $E$, compatible assumptions $As$, 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* $\pi$ *matches a trace of* $E$ *that satisfies assumptions infinitely often but is not strong independent fair in* $M \| E\}$ *is zero.*

*Proof.* Refer to Appendix A.

For instance, the MDP $E_p$ in Figure 4.7 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 4.3. 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 A.3, controllers to the RSGR(1) problem with environment $E_2$, assumption $A$, goal $moveToBelt$, try-response triple (*cook*, *broken*, *not.broken*) and safety $moveToBelt \Rightarrow CookedTwice \wedge \neg Broken$ 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 4.1. Its environment is compatible with the assumptions on pending package request. The question to ask now is if the environment 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

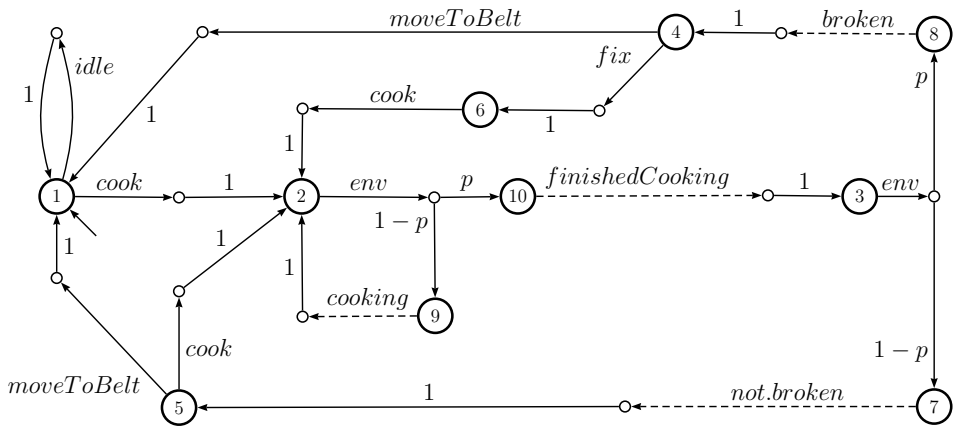



Figure 4.7: MDP for the Ceramic Cooking Problem

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 environment means that the denotation of, for instance, failures must be 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.

The proof of the above theorem is based on the fact that if the environment is assumption compatible then for every state there is a strategy (path) for

the environment model $E$ to fulfil its assumptions infinitely often without failing. Now, given a fair scheduler, there are traces for that scheduler such that the $E$ fulfil its assumptions infinitely often without failing. It follows that from every state of $E\|M$ a cycle with assumptions and no failures is reachable by taking non-failure uncontrollable actions. Additionally, since the probabilities on transitions can only be zero on failures, it follows that the probability of reaching the cycle is non-zero. We can show that in such an environment, as the environment is "well behaved", from every state of the environment, the probability of a cycle that is strongly successful with respect to the assumptions is not zero.

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

## 4.6 Evaluation

In this section we report on further refinements of the case studies presented in Section 3.8. We evaluate the applicability of our approach for synthesis when controlled actions may fail and the benefits it provides with respect to existing synthesis techniques, including our own. Applicability is evaluated based on the following criteria *i*) is RSGR(1) applicable to the case studies as described in the literature, *ii*) is RSGR(1) applicable to richer versions, which introduce environment 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 environment model.

For all case studies, including the orchestration problem of Section 4.1, see [D'I].

### 4.6.1 Production Cell

In Section 3.8 we have presented a control problem based on the Production Cell [LL95]: 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 raw products to construct a new product, it eventually receives them.

Both the original problem formulation [LL95] and that of Section 3.8 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 product 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 product 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 products the environment provides them. 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 products. 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).

### 4.6.2 Pay & Ship

In Section 3.8 we have presented a simplified version of the case study originally presented by Pistore et al. [PBB+04]. The goal of the case study is to

apply their technique in order to synthesise a plan for composing distributed web services and monitoring them.

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 [PBB+04] gives no guarantees that the goal of satisfying purchase requests is achieved. In fact, achieving the goals stated in [PBB+04] 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 4.5 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 analyse its adequacy for their specific problem. They have to under-

stand the implications of assuming strong fair independence on traces and the implications of deploying a service which does not provide guarantees in these cases.

### 4.6.3 Autonomous Vehicles

We revisit the robotics case study from Section 3.8. Recall that 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.

In Section 3.8 we presented an idealised version of the case study, originally introduced in [HSMK09a]. Note that due to the limitations of the SGR(1) 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.

The synthesis algorithm presented in [HSMK09a] 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 [HSMK09a], implicitly 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 (un-modelled) 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.

### 4.6.4 Bookstore

We consider the web-service composition scenario in Section 3.8.3, which again 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.

## 4.7 Limitations

In the previous section we have shown that the technique presented in this chapter can successfully model several examples from the software engineer-

ing literature in a precise and effective way. Although, the technique, supporting SGR(1) goals and failures, have shown to be sufficient in many contexts, it may fall short in some others.

In the next section we model a variation of the production cell case study presented in Section 4.6.1. We show how to model controller goals and environment in order to synthesise a controller. Then, we show that the case study, as simple as it may look, has some limitations that can be avoided but by being very careful in the way we model it. Nevertheless, we discuss other modelling approaches that require more expressive synthesis techniques.

### 4.7.1 Production Cell

Consider a scenario in which we have a robot arm, a drill, a painting tool, an in tray, an out tray, a recycle bin and some additional sensors. The arm can be moved freely and there is no fixed connection between any tool or tray. We aim to use the arm aided by sensors to move objects from the in tray, through a combination of tools according to a high-level production process, to the out tray.

The control mechanisms of the robot arm are provided through a general purpose API with support for moving the arm to a specific coordinate, and for identifying and grabbing objects. In addition, due to traction issues, the arm may fail when trying to grab an object, a situation that can be sensed through an operation that reads whether the gripper is fully closed or not.

We model the environment in terms of high level actions (e.g. picking up from

```
ARM = (pickupfrom[l:GetLocations][c:olours]
        ->GET_RESULT[l][c]),
GET_RESULT[l:GetLocations][c:Colours]=
            (pickupfrom_success[l][c]->PICKED_UP[c]
            | pickupfrom_fail[l][c]->ARM),
PICKED_UP[c:Colours] = (putdownat[l:PutLocations][c]
                        ->putdownat_success[l][c]-> ARM).
```

Figure 4.8: FSP Example - Robot Arm

the drill can only succeed if an object was previously placed there, the paint tool paints objects red); *ii)* specification of a high-level production process (e.g. produce alternating coloured objects $(red.yellow)^{\star}$, only painted objects that have been drilled); *iii)* specification of environment assumptions (e.g. yellow objects will be supplied indefinitely, the probability of successfully grabbing an object is greater than 0); and finally *iv)* synthesis of a controller in the form of a behaviour model that encodes the arm's strategy for achieving the production process (e.g. what to do if it needs to output a red object but is receiving only yellow objects).

The FSP process `ARM`, in Figure 4.8, models the fact that picking objects up from a location can fail. As expected, the arm must successfully pick up an object to be able to put it somewhere else. Figure 4.9 shows the definition for two processes. First, `PAINT` that models the behaviour of the painting tool which paints red any object it is given. Second, `TOOLS` is a parametric model that captures the behaviour of tools that receive objects, and only after their processing is done they allow for objects to be picked up.

`SUPPLY`, in Figure 4.10, models that objects are only supplied when the in

```
PAINT = (putdownat_success['paint][Colours]->COLORING),
COLORING =(ready['paint]['red]
            ->pickupfrom_success['paint]['red]->PAINT)
                    +ready['paint]['yellow],
                    pickupfrom_success['paint]['yellow].
TOOL(T='any)=(putdownat_success[T][c:Colours]
            ->ready[T][c]->pickupfrom_success[T][c]->TOOL).
||TOOLS = (forall[t:Tools] TOOL(t) || PAINT).
```

Figure 4.9: FSP Example - Tools

tray is empty. Note that objects remain in the in tray until they are successfully picked up. In addition, Figure 4.10 depicts the `FORCE_PICKUP` process. `FORCE_PICKUP` models that once a yellow object has been provided, the arm must pickup such object after a fixed number of moves.

Finally, the environment model is constructed as the parallel composition of the LTSs modelling the robot arm, the tools, and the supplier. Note that objects can be either red or yellow, and depending on what is required either colour must be required in the out tray.

There are three safety goal for the controller. First, objects must be produced alternating their colour, i.e. objects must be produce in a $(red, yellow)^\star$

```
SUPPLY = (supply[c:Colours]
        ->pickupfrom_success['in][c]->SUPPLY).
||SUPPLIER = SUPPLY.
FORCE_PICKUP = (supply['yellow]->COUNT[0]
                | A\{supply['yellow]}->FORCE_PICKUP),
COUNT[id:Count] = (A\{pickupfrom['in]['yellow]}->COUNT[id+1]
                | pickupfrom['in]['yellow] -> FORCE_PICKUP),
COUNT[MAX+1] = (pickupfrom['in]['yellow]->FORCE_PICKUP).
```

Figure 4.10: FSP Example - Supply & Pickup Restriction

sequence. Second, objects must be drilled before they are painted. Third, the arm can attempt to pick up only if a object is at the specified location (e.g. the drill).

As solving a control problems under *strong independent fairness* assumption can be reduced to SGR(1), the events assumed to be strongly independent fair are specified simply as a fluent that indicates when any failure has occurred.

The liveness part of the goal is to guarantee that infinitely many red objects are produced. The controller cannot guarantee such goal unless the environment provides with objects. In this case, we only require that yellow objects are provided by the environment infinitely often. Hence, the liveness part of the controller goal guarantees that infinitely many times red objects are placed in the out tray, only if there infinitely many yellow objects are delivered in the in tray.

It is important to note that if the `FORCE_PICKUP` process is removed from the specification `ENV` is not compatible with the environment assumptions. This is because the controller may never pick up an object from the in tray preventing the environment to provide yellow objects. Note that an alternative way of achieving a compatible control problem is to allow the environment to enqueue objects in the in tray.

Although we cannot show the controller due to space restrictions (it has over 5000 states, compared to the over 10000 states of the environment model) we describe some of the behaviour its exhibits: $i$) If it is the turn to output a red objects but the environment provides a yellow one, the controller has to

drill it, then paint it red and finally put it down on the out tray. *ii*) When it is the turn to output yellow ones but the environment provides a red one, then the controller can assume that at some point a yellow object will be supplied and, hence, discard the current red object and wait for the next yellow to appear. *iii*) When it is the turn for red objects to be produced, if the controller is processing (i.e. drilling, then painting) a yellow to get a red but a red is supplied, the controller may choose to discard the yellow being processed and just output the red waiting in the in tray.

As mentioned above, `FORCE_PICKUP` is required to avoid controllers that prevent the environment of supplying objects by not trying to pick up objects from the in tray. Moreover, by specifying `FORCE_PICKUP` we are explicitly restricting attention to controllers that actively try to pick up objects after a fixed number of movements. Up to this point, we have successfully synthesised a controller for the robot arm.

Although we have synthesised controllers that are desired, restricting to a fixed number the moves before the arm tries to pick up an object is a rather strong requirement and it may significantly increase the state space to be explored.

An alternative to modelling `FORCE_PICKUP` as a bounded liveness [LKMU05, MNP07] restriction is to model the real a liveness goal for the controller. Specifically, a liveness goal prescribing the controller to pickup objects from the in tray infinitely often. Such goal would produce a controller. However, the environment model would not be compatible with the environment assumptions. Indeed, anomalous controllers exists. For instance, a contro-

ller that never picks up objects from the in tray is an anomalous controller for this problem, as it violates assumptions in order to satisfy the specification. Interestingly enough, such controller not only disables the environment to satisfy its assumptions but also it must actively avoid pursuing its own goals, i.e. picking up objects from the in tray. Although, the algorithm we have implemented would not produce such controller, modelling an environment that is not compatible with its liveness assumptions is not a reasonable solution.

A different approach to model this case study is to decouple the environment's attempt to supply an object from the actual successful supply to the in tray. This is done by modelling an arm that can attempt to supply objects independently of whether the in tray is empty or not (see Figure 4.11). Such environment can freely attempt to supply objects to the in tray at any point. Hence, environment assumptions modelled in terms of the supply attempts would be compatible with the environment model. However, the environment can attempt to supply indefinitely, hence, leaving the controller with no chance to move the arm at all. To avoid such undesired behaviour two restrictions are added. First, a safety requirement adding to the environment capabilities of yielding the control to the controller (See process `SCHED` in Figure 4.11. Second, a liveness assumption prescribing that the environment must yield the control infinitely often.

There are a number of issues when decoupling the environment from the controller and introducing yield actions. Consider a scenario in which the controller has tried to pick up an object from the in tray after the environ-

```
SUPPLY = (attemptSupply[c:Colours]->SUPPLY).

SUPPLYCONSTRAINT = EMPTY,
EMPTY = (attemptSupply[c:Colours]->supply[c]->FULL[c]),
FULL[c:Colours] = (attemptSupply[cNew:Colours]->FULL[c]
                  | pickupfrom_success['in][c]->EMPTY).
||SUPPLIER = (SUPPLY||SUPPLYCONSTRAINT).

SCHED = ({Alphabet\{yield}}->SCHED | yield->YIELD),
YIELD = ({ControllableActions}->SCHED).
```

Figure 4.11: Environment Model Decoupled through Yielding Control

ment has supplied the object successfully. As the environment can hold the notification of the pickup being successful, the controller cannot process objects. Consequently, there is no controller that guarantees the red and yellow objects be processed alternately. Hence, assumptions on the responsiveness of all tools and trays must be added. Unfortunately, giving such freedom to the environment results in an unrealisable control problem. Indeed, the environment fulfils its assumptions by simply trying to supply a yellow after a red has been successfully supplied, hence, it never actually supplies yellow objects. Consequently, as only red objects are supplied no controller can achieve the goals.

This case study can be solved simply by adding an assumption requiring the environment to provide with yellow objects infinitely often if the in tray is empty infinitely often (i.e. $\Box \Diamond \mathit{intray.empty} \rightarrow \Box \Diamond \mathit{supply.yellow}$). Such assumptions guarantee that yellow objects will be provided infinitely often. Moreover, as the assumptions also hold while the in tray is full, the controller must pickup and produce objects in order to guarantee its goals. In other

words, these assumptions are compatible with the environment. Unfortunately, GR(1)-like formulas are not expressive enough to encode assumptions such as the one above (which essentially constitutes strong fairness). Strong fairness assumptions are out of the scope of the techniques we have presented in this work.

This case study shows the need for synthesis techniques supporting more expressive goals. However, small variations in its configuration make it tractable with effective synthesis techniques such as SGR(1). For instance, having separate in trays, one for red and one for yellow objects would allow to apply SGR(1).

Summarising, we have found that in some cases our technique, even supporting fallible environments and GR(1) goals, might not be expressive enough. Nonetheless, we have shown that a number of modelling approaches (e.g. bounded liveness) can be applied, allowing for efficient techniques such as SGR(1) to be applicable.

# Chapter 5

# MTS Control Synthesis

In this chapter we define controller synthesis in the context of partially specified environment models. More specifically, we study the problem of checking the existence of an LTS controller (i.e., controller realisability) capable of guaranteeing a given goal when deployed in a completely defined LTS domain model that conforms to a partially defined problem domain given as an MTS.

The semantics of MTS is given in terms of a set of LTS implementations in which each LTS provides the required behaviour described in the MTS and does not provide any of the MTS proscribed behaviour. We define the *MTS control problem* as follows: given an MTS we ask if *all*, *none* or *some* of the LTS implementations it describes admit an LTS controller that guarantees a given goal given as a FLTL formula. The realisability question we address in the context of MTS has a three valued answer.

From a model elaboration perspective, a *none* response indicates that there is no hope of building a system that satisfies the goals independently of the aspects of the domain that have been modelled as uncertain. This entails that either goals must be weakened or stronger assumptions about the domain must be made. An *all* response indicates that the partial domain knowledge modelled is sufficient to guarantee that the goals can be achieved, consequently further elaboration may not be necessary. Finally, a *some* response indicates that further elaboration is required.

In the next sections we present formal definitions, evaluation and limitations of the MTS control problem.

## 5.1 MTS Control Problem

The problem of control synthesis for MTS is to check whether all, none or some of the LTS implementations of a given MTS can be controlled by an LTS controller. More specifically, given an MTS, an FLTL goal and a set of controllable actions, the answer to the MTS control problem is *all* if all implementations of the MTS can be controlled, *none* if no implementation can be controlled and *some* otherwise. This is defined formally below.

**Definition 5.1.** (Semantics of MTS Control) *Given a deterministic MTS* $E = (S, A, \Delta^r, \Delta^p, s_0)$*, an FLTL formula* $\varphi$ *and a set* $A_C \subseteq A$ *of controllable actions, to solve the* MTS control problem $\mathcal{E} = \langle E, \varphi, A_C \rangle$ *is to answer:*

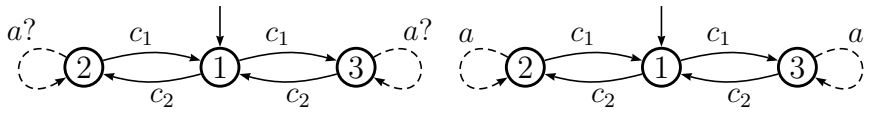
- ***All****, if for all LTS* $I \in \mathcal{I}^{det}[E]$*, the control problem* $\langle I, \varphi, A_C \rangle$ *is realis-*

*able,*

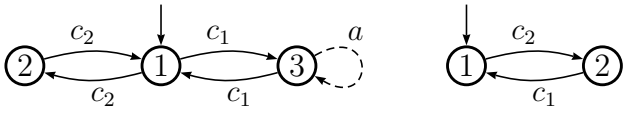
- ***None**, if for all LTS $I \in \mathcal{I}^{det}[E]$, the control problem $\langle I, \varphi, A_C \rangle$ is unrealisable,*
- ***Some**, otherwise.*

We say that the MTS control problem is *unrealisable* if its answer is none and *realisable* otherwise. Consequently, we may refer as solving the *realisability* question to solving the MTS control problem.

As a simple example, consider the MTS control problem $\mathcal{E} = \langle E, \varphi, A_C \rangle$, where $E$ is the MTS in Figure 5.1(a), $A_C = \{c_1, c_2\}$, $\varphi = \Box \neg \dot{a}$ and $\dot{a} = \langle \{a\}, \{c_1, c_2\}$. Consider $I_1$ and $I_2$ two implementations of $E$. There is no controller for $I_1$ such that $a$ can be restricted to occur. $I_2$, on the other hand, can be controlled by the LTS controller in Figure 5.1(d). Hence, the realisability question for this example is **Some**.



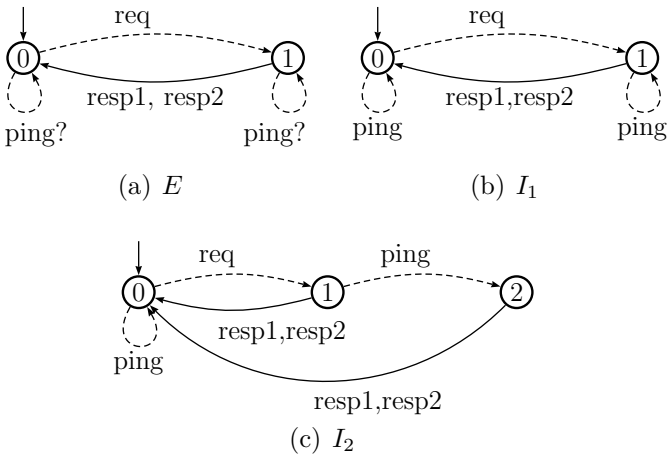

(a) MTS Environment $E$ (b) Iplementation $I_1$




(c) Iplementation $I_2$ (d) Controller $M_2$

Figure 5.1: Event-Based Control Example.

Note that, as in the case of LTS control problem, we restrict attention to deterministic environment models. This follows from the fact that out solution




(a) $E$ (b) $I_1$

(c) $I_2$

Figure 5.2: Server Example

for MTS realisability is by a reduction to LTS realisability.

### 5.1.1 Motivating Example

As an example, consider $E$, the environment model in Figure 5.2(a) describing interactions between a server and clients. Note that although it is certain that client requests be responded by the server, definitions regarding when clients may ping the server have not been made yet. Suppose that we want to build a controller for this server such that the server guarantees that after receiving a request it will eventually yield a response and if there are enough requests responses of both kinds will be issued. We formally describe this requirement as the FLTL formula: $\varphi = \Box \diamondsuit \mathit{Response} \wedge (\Box \diamondsuit \dot{req} \Rightarrow (\Box \diamondsuit \dot{resp}_1 \wedge \Box \diamondsuit \dot{resp}_2))$, where $\mathit{Response} = \langle \{req\}, \{resp_1, resp_2\}, false \rangle$. As expected, the server can only control the response. Hence, we have the MTS control problem $\mathcal{E} = \langle E, \varphi, \{resp_1, resp_2\} \rangle$. Consider the implementation $I_1$, shown in Figure 5.2(b). The uncontrollable self loop over ping

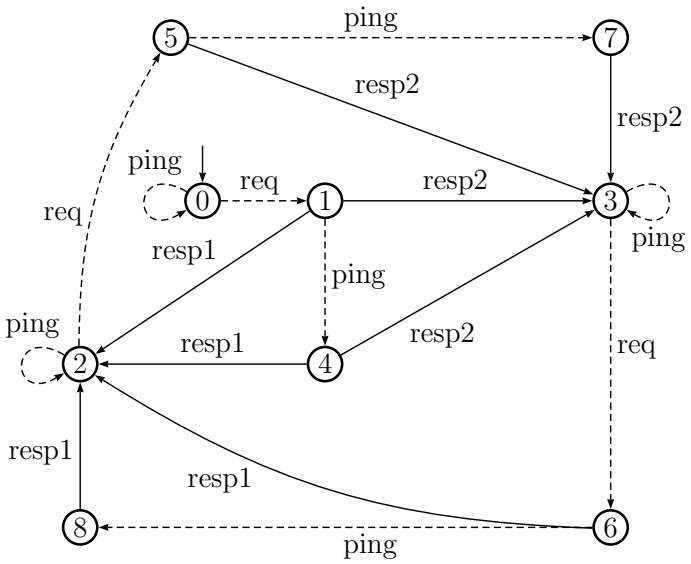



Figure 5.3: LTS Controller for the Server Example

in state 1 allows the environment to ping indefinitely impeding the controller from eventually producing a response (i.e. no controller can avoid the trace $req, ping, ping, \ldots$). The implementation $I_2$, shown in Figure 5.2(c), allows only a bounded number of pings after a request, hence, the server cannot be flooded and a controller for the property exists (see Figure 5.3). Consequently, the answer for the MTS control problem $\mathcal{E}$ is some.

### 5.1.2 Solving the MTS Control Problem

A naive approach to the MTS control problem may require to evaluate an infinite number of LTS control problems. Naturally, such approach is not possible, hence, it is mandatory to find alternative ways to handle MTS control problems.

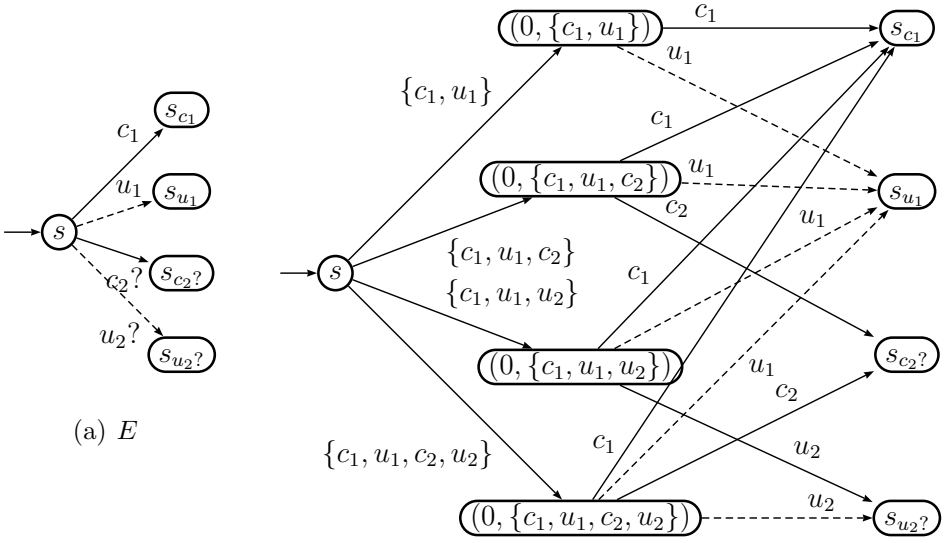
We reduce the MTS control problem to two LTS control problems. The first LTS control problem encodes the problem of whether there is a controller for each implementation described by the MTS. It does so by modelling an environment that can pick the "hardest" implementation to control. In fact, in the LTS control problem, the environment will pick at each point the subset of possible transitions of the MTS that are available. If there is a controller for this environment, there is a controller for all implementations.

The second LTS control problem encodes the problem of whether there is no controller for every implementation of the MTS. Similarly, this is done by modelling an LTS control problem in which the controller can pick the implementation (in fact, it is now the controller that picks the subset of possible transitions of the MTS that are available at each point). If there is no controller in this setting, then for every implementation there is no controller.

The two LTS problems are defined in terms of the same LTS. The only difference is who controls the selection of the subset of possible actions, i.e. implementation choice. We now define the LTS $E^I$ in which additional transition labels are added to model explicitly when either the controller or the environment choose which subset of possible transitions of the MTS are available.

**Definition 5.2.** *Given an MTS* $E = (S, A, \Delta^r, \Delta^p, s_0)$. *We define* $E^I = (S_{E^I}, A_{E^I}, \Delta_{E^I}, s_0)$ *as follows:*

- $S_{E^I} = S \cup \{(s, i) \mid s \in S \wedge i \subseteq A \text{ and } \Delta^r(s) \subseteq i\}$




(a) $E$

(b) $E^I$

Figure 5.4: Translation from $E$ to $E^I$

- $A_{E^I} = A \cup \overline{A}$, *where* $\overline{A} = \{\ell_i \mid i \subseteq A\}$
- $\Delta_{E^I} = \{(s, \ell_i, (s, i)) \mid i \subseteq A\} \ \cup \ \{((s, i), \ell, s') \mid (s, \ell, s') \in \Delta^p \wedge \ell \in i\}$

States in $E^I$ are of two kinds. Those that are of the form $s$ with $s \in S$ encode states in which a choice of which subset of possible transitions are implemented has to be made. Choosing a subset $i \subseteq A$, leads to a state $(s, i)$. States of latter form $(s, i)$ have outgoing transitions labelled with actions in $i$. A transition from $(s, i)$ on an action $\ell \in i$ leads to the same state $s'$ in $E^I$ as taking $\ell$ from $s$ in $E$. For example, the model in Figure 5.4(b) is obtained by applying Definition 5.2 to model in Figure 5.4(a).

The LTS $E^I$ provides the basis for tractably answering the MTS control

question. The following algorithm shows how to compute the solution for the MTS control problem.

**Algorithm 1.** (MTS Control Solving) *Given an MTS control problem* $\mathcal{E} = \langle E, \varphi, A_C \rangle$. *If* $E^I$ *is the LTS model obtained by applying Definition 5.2 to* $E$, *then the answer for* $\mathcal{E}$ *is computed as follows.*

- ***All****, if there exists a solution for* $\mathcal{E}_A^I = \langle E^I, \mathcal{X}_{\overline{A}}(\varphi), A_C \rangle$
- ***None****, if there is no solution for* $\mathcal{E}_N^I = \langle E^I, \mathcal{X}_{\overline{A}}(\varphi), A_C \cup \overline{A} \rangle$
- ***Some****, otherwise.*

Algorithm 1 shows how to compute the answer for a given MTS control problem $\langle E, \varphi, A_C \rangle$.

Intuitively, if we give control over the new actions $\ell_i$ to the environment, it can choose the hardest implementation to control. Thus, this solves the question of whether all implementations are controllable.

Lemma 5.1 proves that the case *all* in Algorithm 1 is sound and complete.

**Lemma 5.1.** (All) *Given an MTS control problem* $\mathcal{E} = \langle E, \varphi, A_C \rangle$ *where* $E = (S, A, \Delta^r, \Delta^p, s_{0_E})$. *If* $E^I$ *is the LTS obtained by applying Definition 5.2 to* $E$, *then the following holds. The answer for* $\mathcal{E}$ *is* all *iff the LTS control problem* $\mathcal{E}_A^I = \langle E^I, \mathcal{X}_{\overline{A}}(\varphi), A_C \rangle$ *is realisable.*

*Proof.* Refer to Appendix A.1

We give an intuition of the proof for Lemma 5.1. Assume there is no solution to $\mathcal{E}_A^I$. Then by Determinacy Lemma 3.1 there exists an LTS $N$ such that

for every possible controller $M$, $E^I \| M \| N \models \neg \mathcal{X}_{\overline{A}}(\varphi)$. From $N$ we construct an implementation $I \in \mathtt{I}^{det}[E]$. Intuitively, states of the form $s$ of $E^I$ are uncontrollable (i.e., have only uncontrollable transitions). Thus, $N$ instructs in every such state of $E^I$ which successor of the form $(s, i)$ to choose. The set of actions $i$ is the set of transitions to implement from a state representing $s$ in $I$. We then show that this implementation is not controllable. Otherwise, we could combine the controller for $I$ with $N$. By construction, traces in the parallel composition should both satisfy and not satisfy the respective formulas, which is impossible.

In the other direction, assume there is a controller $M$ for $\mathcal{E}_A^I$. Given an implementation $I \in \mathtt{I}^{det}[E]$ we show that there exists a controller for $\mathcal{I} = \langle I, \varphi, A_C \rangle$. The construction is directed by the refinement relation for $I$. If $t$ is a state of $I$ that refines state $s$ of $E$ and implements the set $i$ of transitions, then the controller controls $i$ in $I$ by enabling the same set of actions that $M$ enables to control $(s, i)$ in $E^I$. As before, every trace in the parallel composition between the controller and $I$ corresponds to a trace in $E^I \| M$. As traces in $E^I \| M$ must satisfy the winning condition, so is the case for traces in the parallel composition between the controller and $I$.

Consider the case in which the answer for the MTS control problem is *none*. The answer to $\mathcal{E}$ is *none*, if there is no solution to the LTS control problem $\mathcal{E}_N^I$. Intuitively, if we give control over the new actions $\ell_i$ to the controller, it can choose the easiest implementation to control. Thus, this solves the question of whether no implementation is controllable.

Lemma 5.2 proves that the case *none* in Algorithm 1 is sound and complete.

**Lemma 5.2.** ***(None)*** *Given an MTS control problem $\mathcal{E} = \langle E, \varphi, A_C \rangle$ where $E = (S, A, \Delta^r, \Delta^p, s_0)$. If $E^I$ is the LTS obtained by applying Definition 5.2 to $E$, then the following holds.*

*The LTS control problem $\mathcal{E}^I_N = \langle E^I, \mathcal{X}_{\overline{A}}(\varphi), A_C \cup \overline{A} \rangle$ is realisable iff there exists $I \in \mathcal{I}^{det}[E]$ such that the LTS control problem $\mathcal{I} = \langle I, \varphi, A_C \rangle$ is realisable.*

*Proof.* Refer to Appendix A.2.

We give an intuition of the proof for Lemma 5.2. Assume there is a controller $M$ for $\mathcal{E}^I_N$. We construct an implementation $I$ and a controller $N$ for $\mathcal{I} = \langle I, \varphi, A_C \rangle$. Intuitively, states of the form $s$ of $E^I$ are controllable (i.e., have only controllable transitions). Thus, $M$ instructs in every such state of $E^I$ which successor of the form $(s, i)$ to choose. The set of actions $i$ is the set of transitions to implement from a new state of $I$ matching state $s$ of $E^I$. Then, $N$ enables the same actions as $M$ enables from $(s, i)$ in $E^I$. We then show that $N$ indeed controls $\mathcal{I}$. This is done by showing a correspondence between traces of $I \| N$ and traces of $E^I \| M$. Thus, both must satisfy the respective winning conditions. In the other direction, given an implementation $I \in \mathtt{I}^{det}[E]$ and a controller $N$ for $\mathcal{I}$, we construct a controller $M$ for $\mathcal{E}^I_N$. The construction is directed by the refinement relation for $I$. If $t$ is a state of $I$ that refines state $s$ of $E$ and implements the set $i$ of transitions, then $M$ enables the successor $(s, i)$ of $s$ in $E^I$. Then, it enables from $(s, i)$ the same set of transitions enabled by $N$ to control $t$ in $I$. A correspondence between

between traces of $I\|N$ and traces of $E^I\|M$ is established. Both traces satisfy the respective winning conditions.

## 5.2 Linear Reduction

Algorithm 1 shows that the MTS control problem can be reduced to two LTS control problems. Hence, our solution to the MTS control problem is, in general, doubly exponential in the size of $E^I$ (cf. [PR89, DBPU13]). Unfortunately, the state space of $E^I$ is exponential in the branching degree of $E$, which in turn is bounded by the size of the alphabet of the MTS. More precisely, for a state $s \in E^I$ the number of successors of $s$ is bounded by the number of possible combinations of labels of maybe transitions from $s$ in $E$. In this section we show that to compute the answer for $\mathcal{E}^I_A$and $\mathcal{E}^I_N$it is enough to consider only a small part of the states of $E^I$. Effectively, it is enough to consider at most linearly (in the number of outgoing transitions) many successors for every state. This leads to the MTS control problem being 2EXPTIME-complete.

First, we analyse $E^I$ in the context of $\mathcal{E}^I_A$. We define a fragment $\mathcal{E}^{I^+}_A$of $\mathcal{E}^I_A$. Let $E^{I^+} = (S_{E^I}, A_{E^I}, \Delta^+, s_{0_{E^I}})$, where only the following transitions from $\Delta_{E^I}$ are included in $\Delta^+$.

1. Consider a state $s \in E$ that has at least one required uncontrollable successor. In $\Delta^+$ we add to $s$ only the transition $(s, \ell_i, (s, i))$, where $i = \Delta^r_E(s) \cup (\Delta^p_E(s) \cap A_\mu)$. That is, in addition to required transitions

from $s$ we include all uncontrollable possible successors of $s$.

2. Consider a state $s \in E$ that has no required uncontrollable successors but has a required controllable successor. In $\Delta^+$ we add to $s$ only the transitions $(s, \ell_i, (s, i))$, where $i$ is either $\Delta^r_E(s)$ or $i$ is $\Delta^r_E(s) \cup (\Delta^p_E(s) \cap A_\mu)$. That is, we include a transition to all required transitions from $s$ as well as augmenting all required transitions by all uncontrollable possible transitions.
3. Consider a state $s \in E$ that has no required successors. In $\Delta^+$ we add to $s$ a transition to $(s, \ell_i, (s, i))$, where $i = \Delta^p_E(s) \cap A_\mu$, and for every $\ell \in \Delta^p_E(s) \cap A_C$ we add to $s$ the transition $(s, \ell_{\{\ell\}}, (s, \{\ell\}))$. That is, we include a transition to all possible uncontrollable transitions from $s$ and for every possible controllable transition a separate transition.
4. For a state $(s, i)$ we add to $\Delta^+$ all the transitions in $\Delta_{E^I}$.

Let $\mathcal{E}^{I^+}_A = \langle E^{I^+}, \mathcal{X}_{\overline{A}}(\varphi), A_C \rangle$.

**Lemma 5.3.** *The problem $\mathcal{E}^I_A$ is controllable iff $\mathcal{E}^{I^+}_A$ is controllable.*

*Proof.* The following correspondence between controller for $\mathcal{E}^I_A$ and controller for $\mathcal{E}^{I^+}_A$ establishes the lemma.

1. Consider $s \in E$ with at least one required uncontrollable successor. Recall that in $E^{I^+}$ state $s$ has a unique successor $(s, i)$ with $i = \Delta^r_E(s) \cup (\Delta^p_E(s) \cap A_\mu)$. Consider a potential controller for $\mathcal{E}^I_A$. This controller must be able to control the unique successor as in $E^{I^+}$. Consider a potential controller for $E^{I^+}$ and some other successor $(s, i')$ of $s$ in $E^I$. By construction, $(s, i')$ has all the uncontrollable required

successors of $s$ as well as potentially some possible uncontrollable and some controllable transitions. However, as $(s, i')$ has an uncontrollable successor all controllable successors can be removed by the controller. It follows that the transitions enabled to control $(s, i)$ will suffice to control all other successors $(s, i')$.

2. Consider $s \in E$ with no required uncontrollable successors but with some required controllable successor.

   Recall that in $E^{I^+}$ state $s$ has two successors $(s, i_1)$ and $(s, i_2)$ where $i_1 = \Delta^r_E(s)$ and $i_2 = \Delta^r_E(s) \cup (\Delta^p(s) \cap A_\mu)$. Consider a potential controller for $\mathcal{E}^I_A$. This controller must be able to control both $(s, i_1)$ and $(s, i_2)$ in $E^I$. Consider a potential controller for $E^{I^+}$. Let $(s, i')$ be a successor of $s$ in $E^I$ such that $i'$ includes only controllable transitions. By construction, $(s, i')$ has all the controllable required successors of $s$ and potentially some possible controllable transitions. However, as $(s, i')$ is controllable it follows that the set of actions enabled to control $(s, i_1)$ can be used to control all such successors $(s, i')$. Let $(s, i')$ be a successor of $s$ in $E^I$ such that $i'$ includes some uncontrollable successor. By construction, $(s, i')$ has all the controllable required successors of $s$ as well as potentially some possible uncontrollable and some possible controllable transitions. However, as $(s, i')$ has an uncontrollable successor all controllable successors can be removed by the controller. Since the uncontrollable successors $(s, i')$ are a subset of the uncontrollable successors of $(s, i_2)$, it follows that the set of actions enabled to control $(s, i_2)$ can be used to control all other such successors $(s, i')$.

3. Consider $s \in E$ with no required successors.

Recall that in $E^{I^+}$ state $s$ has a successor $(s, i)$, where $i = \Delta_E^p(s) \cap A_\mu$ and for every $\ell \in \Delta_E^p(s) \cap A_C$ a successor $(s, \{\ell\})$. Consider a potential controller for $E^I$. This controller must be able to control $(s, i)$ and all $(s, \{\ell\})$ in $E^I$. Consider a potential controller for $\mathcal{E}_A^I$. Let $(s, i')$ be a successor of $s$ in $E^I$ such that $i' \cap A_\mu \neq \emptyset$. By construction, $(s, i')$ has some uncontrollable transitions of $s$ as well as potentially some controllable transitions. However, as $(s, i')$ has an uncontrollable successor all controllable successors can be removed by the controller. Since the uncontrollable successors of $(s, i')$ are a subset of the uncontrollable successors of $(s, i)$, it follows that the set of actions enabled to control $(s, i)$ can be used to control all such successors $(s, i')$. Let $(s, i')$ be a successor of $s$ in $E^I$ such that $i' \subseteq A_C$. By construction, $(s, i')$ has some possible controllable transitions of $s$ and no uncontrollable transitions of $s$. As $(s, i')$ is controllable it follows that the set of actions enabled to control some $(s, \{\ell\})$ where $\ell \in i'$ can be used to control this successor.

We now analyse $E^I$ in the context of the $\mathcal{E}_N^I$. We define a fragment $\mathcal{E}_N^{I^-}$ of $\mathcal{E}_N^I$. Let $E^{I^-} = (S_{E^I}, A_{E^I}, \Delta^-, s_{0,E^I})$, where only the following transitions from $\Delta_{E^I}$ are included in $\Delta^-$.

1. Consider a state $s \in E$ that has at least one required uncontrollable successor. In $\Delta^-$ we add to $s$ only the transition $(s, \ell_i, (s, i))$, where $i = \Delta_E^r(s)$. That is, include only the required transitions from $s$.
2. Consider a state $s \in E$ that has no required uncontrollable successors. In $\Delta^-$ we add to $s$ a transition to $(s, \ell_i, (s, i))$, where $i = \Delta_E^r(s) \cup (\Delta_E^p(s) \cap A_C)$, and for every $\ell \in \Delta_E^p(s) \cap A_\mu$ we add to $s$ the transition

to $(s, \ell_{\Delta^r_E(s)\cup\{\ell\}}, (s, \Delta^r_E(s) \cup \{\ell\}))$. That is, we include a transition to all controllable transitions from $s$ and for every possible uncontrollable transition a separate transition.

3. For a state $(s, i)$ we add to $\Delta^-$ all the transitions in $\Delta_{E^I}$.

Let $\mathcal{E}^{I^-}_N = \langle E^{I^-}, \mathcal{X}_{\overline{A}}(\varphi), A_C \cup \overline{A}\rangle$.

**Lemma 5.4.** *The problem $\mathcal{E}^I_N$ is controllable iff $\mathcal{E}^{I^-}_N$ is controllable.*

*Proof.* The following correspondence between a controller for $\mathcal{E}^I_N$ and controller for $\mathcal{E}^{I^-}_N$ establishes the lemma.

1. Consider $s \in E$ with at least one required uncontrollable successor.
   Recall that in $E^{I^-}$ state $s$ has a unique successor $(s, i)$ with $i = \Delta^r_E(s)$. Consider a potential controller for $\mathcal{E}^I_N$. Suppose that it chooses some other successor $(s, i')$ of $s$ in $E^I$. By construction, $(s, i')$ has all the uncontrollable required successors of $s$ as well as potentially some possible uncontrollable and some controllable transitions. However, as $(s, i')$ has an uncontrollable successor, all controllable successors can be removed by the controller. It follows that the set of actions enabled from $(s, i')$ can also be enabled to control $(s, i)$. Consider a potential controller for $\mathcal{E}^{I^-}_N$. This controller controls the successor $(s, i)$ and can do the same in $\mathcal{E}^I_N$.
2. Consider $s \in E$ with no required uncontrollable successors.
   Recall that in $E^{I^-}$ state $s$ has a successor $(s, i)$ with $i = \Delta^r_E(s) \cup (\Delta^p_E(s) \cap A_C)$ and for every $\ell \in \Delta^p_E(s) \cap A_\mu$ the successor $(s, \Delta^r_E(s) \cup \{\ell\})$. Consider a potential controller for $\mathcal{E}^I_N$. Suppose that it chooses some

other successor $(s, i')$ of $s$ in $E^I$. If $(s, i')$ has some uncontrollable successor then the set of actions enabled to control this successor can be used by a controller that controls $E^{I^-}$ from state $(s, \Delta^r_E(s) \cup \{\ell\})$ for $\ell \in i'$ by removing all controllable successors in $\Delta^r_E(s)$. If $(s, i')$ has only controllable successors then $i' \subseteq \Delta^r_E(s) \cup (\Delta^p_E(s) \cap A_C)$ and the set of actions enabled by this controller can be used to control $(s, \Delta^r_E(s) \cup (\Delta^p_E(s) \cap A_C)$ in $E^{I^-}$. Consider a potential controller for $\mathcal{E}^{I^-}_N$. This controller controls some successor $(s, i)$ and can do the same in $\mathcal{E}^I_N$.

Using $\mathcal{E}^{I^+}_A$ and $\mathcal{E}^{I^-}_N$ can establish the complexity of the MTS control problem.

**Theorem 5.1.** (MTS Control Complexity)

*Given an MTS control problem $\mathcal{E} = \langle E, \varphi, A_C \rangle$ it is 2EXPTIME-complete to decide whether the answer to $\mathcal{E}$ is all, none, or some.*

*Proof.* Hardness follows from MTS control being more general than LTS control and the latter's 2EXPTIME-hardness.

Every state in either $E^{I^+}$ or $E^{I^-}$ has at most linearly many successors in the number of original transitions in $E$. As LTS control is in 2EXPTIME membership in 2EXPTIME follows. □

## 5.3 One Controller To Rule Them All

In this section we study whether it is possible to automatically construct controllers for the implementations of MTSs where the answer for the respective MTS control problem is either *all* or *some*. In particular, we focus in understanding whether the controller computed while solving $\mathcal{E}$ (i.e. solving $\mathcal{E}_A^{I^+}$) can be used as "template" to produce solutions to the LTS control problems induced by the implementations of $E$.

Let $\mathcal{E} = \langle E, \varphi, A_C \rangle$ be an MTS control problem. Assume that the answer for $\mathcal{E}$ is *all*. We claim that in such case, there exists an LTS controller that can control all implementations of $E$. $\mathcal{E}$ is answered by solving the LTS control problem $\mathcal{E}_A^{I^+}$. We prove that from the solution to $\mathcal{E}_A^{I^+}$ we can build a controller that can control all implementations of $E$.

The next definition presents an LTS, constructed from a solution of $\mathcal{E}_A^{I^+}$, that can control all implementations of $E$.

**Definition 5.3.** *Let $\mathcal{E} = \langle E, \varphi, A_C \rangle$ be an MTS control problem, $E = (S,\ A,\ \Delta^r,\ \Delta^p,\ s_0)$, and $M = (S_M, A_M, \Delta_M, s_{M_0})$ be a solution to $\mathcal{E}_A^{I^+}$. We define $M' = (S', A', \Delta', s'_0)$ as follows:*

- $A' = A$
- $s'_0 = s_{M_0}$
- $\Delta' = \{(s, \ell, s'') \mid i \subseteq A \text{ and } (s, \ell_i, s') \in \Delta_M \text{ and } (s', \ell, s'') \in \Delta_M\}$
- $S' = s'_0 \cup \{s' \mid (s, \ell, s') \in \Delta'\}$

Consider $M'$, constructed from a solution $M$ to $\mathcal{E}_A^{I^+}$ as defined above. Recall

that $M$, have two kinds of states: $i$) states $s$ that only enables actions $\ell_i$ where $i \subseteq A$ represents the set of actions that are chosen to be implemented and; $ii$) states $(s, i)$ representing a particular implementation choice with enabled actions $\ell \in A \cap i$. Since actions $\ell_i$ are uncontrollable and $M$ is solution to $\mathcal{E}_A^{I^+}$, it follows that all implementation choices from states $s$ can be controlled by $M$. Consequently, each state in $M'$ enables all transitions over actions in $i$, for all $i$ such that $\ell_i$ is enabled from $s$ in $M$, i.e. all possible transitions that can be chosen to be implemented. From the fact that $M$ is solution to $\mathcal{E}_A^{I^+}$and that every state in $M'$ enables all possible transitions, it follows that the parallel composition of $M'$ with any implementation is deadlock free and satisfies $\varphi$.

The following theorem formally shows that if the answer to an MTS control problem is *all*, it is possible to control every implementation with a single LTS controller. To this end, it proves that given a solution to the MTS control problem, the LTS defined in Definition 5.3 can control all implementations of $E$.

**Theorem 5.2.** *Let $\mathcal{E} = \langle E, \varphi, A_C \rangle$ be an MTS control problem. If the answer for $\mathcal{E}$ is* all *then there exists an LTS $M'$ such that for every $I$ implementation of $E$, $I \| M' \models \varphi$.*

*Proof.* Let $M$ be a solution to $\mathcal{E}_A^{I^+}$.

We prove this lemma in two steps. First, we construct $M'$ (a candidate controller for all implementations) from $M$. Second, wee prove that for any implementation $I \in \mathtt{I}^{det}[E]$, the parallel composition $I \| M'$ satisfies $\varphi$ (i.e.

$I \| M' \models \varphi$).

Let $M'$ be the LTS obtained from applying Definition 5.3 to $M$.

Let $I \in \mathtt{I}^{det}[E]$ be an implementation of $E$ and, $R$ be a refinement relation between $E$ and $I$.

We now show that $I \| M'$ is deadlock free and $I \| M'$ satisfies $\varphi$ (i.e. $I \| M' \models \varphi$). Consider a state $(s_I, s_{M'}) \in I \| M'$, and $s_E$ such that $(s_E, s_I) \in R$.

By construction, all monitored actions enabled in $s_I$ are also enabled in $s_{M'}$. Hence, $M'$ can follow $I$. Consider the case in which there are no monitored actions enabled in $s_I$. By MTS refinement, there are no required monitored actions enabled in $s_E$ and there must be some controllable action enabled in $s_E$. By construction of $M'$, at least one of the controllable required transitions enabled from $s_E$ is enabled in $s_{M'}$, or, if there are no controllable required transitions in $s_E$, then all controllable maybe transitions enabled in $s_E$ are enabled also from $s_{M'}$.

By construction of $M'$ and the proof of completeness for the *All* case (Theorem 5.1), it follows that $tr(I \| M') \subseteq tr(\mathcal{E}_A^I \| M)|_A$ where $A$ is the alphabet of $E$. Consequently, $I \| M' \models \varphi$. □

Consider an MTS control problem $\mathcal{E} = \langle E, \varphi, A_C \rangle$, where $E$ is the MTS in Figure 5.5(a), $\varphi = \Box \neg \dot{e}$, and $A_C = \{c_1, c_2\}$. The solution $M$ to $\mathcal{E}_A^{I^+}$is shown in Figure 5.5(b). $M'$ (Figure 5.5(c) is the result of applying Definition 5.3 to $M$. It is easy to see that all implementations are controllable. For instance, $I_1$ (Figure 5.5(d)) and $I_2$ (Figure 5.5(d)) are such that $I_1 \| M' \models \varphi$ and

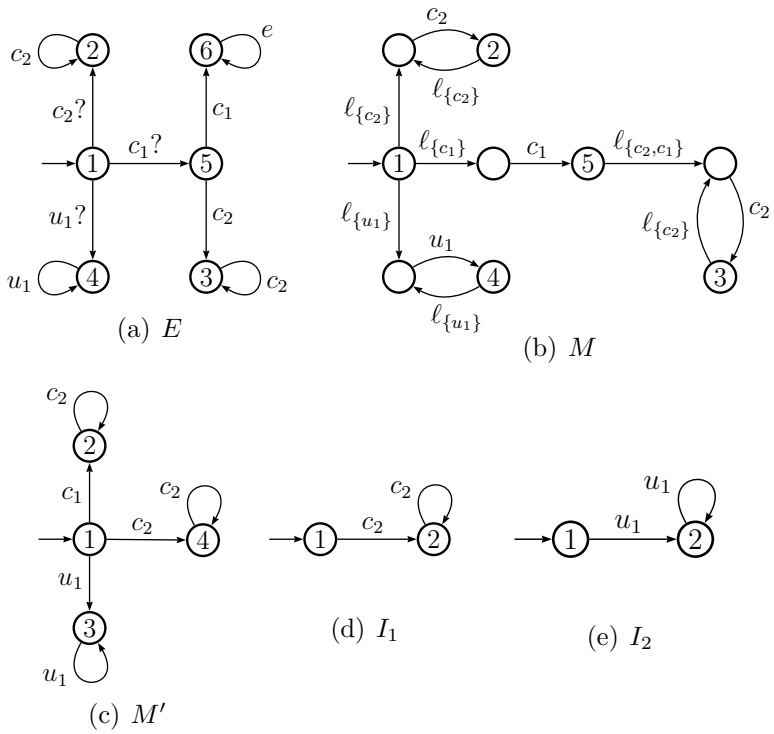



Figure 5.5: Example for the case *All*

$I_2 \| M' \models \varphi$. Note that as $\mathcal{E}_A^{I^+}$is realisable all implementations of $E$ are controllable. We simply choose $I_1$ and $I_2$ as examples.

Note that although $M'$ is a controller according to Definition 3.1, it violates the intuition that the controller only restricts controllable actions that are enabled in the environment model. In order to control any implementation of $E$, $M'$ must enable all monitored actions that could be implemented and every controllable action that does not lead to goal violations. For instance, from the initial state $M'$ enables $u_1$ which is not implemented in $I_1$. Intuitively, $M'$ restricts controllable actions that are not desired and $I_1$ "prunes" the controller by disallowing non-implemented transitions.

Unfortunately, when the answer for an MTS control problem is is *some*, there

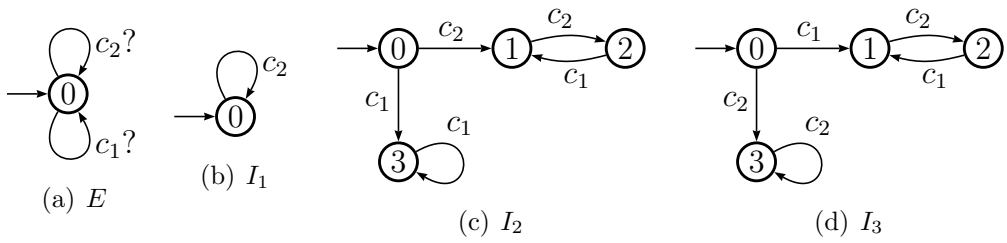



Figure 5.6: Example for the case *Some*

is no unique controller that can control all implementations.

Consider the MTS control problem $\mathcal{E} = \langle E, \varphi, A_C \rangle$, where $E$ is the MTS in Figure 5.6(a), $\varphi = \Box \Diamond \dot{c_1} \wedge \Box \Diamond \dot{c_2}$ and $A_C = \{c_1, c_2\}$. Consider the LTS control problem $\mathcal{I}_1 = \langle I_1, \varphi, A_C \rangle$ where $I_1$ is the implementation of $E$ shown in Figure 5.6(b). As $I_1$ does not implement $c_1$ it follows that there ir no controller for $\mathcal{I}_1$. Consider LTS control problem $\mathcal{I}_2 = \langle I_2, \varphi, A_C \rangle$ where $I_2$ is the implementation of $E$ shown in Figure 5.6(c). $\mathcal{I}_2$ is controllable, in fact, we can construct a controller for $\mathcal{I}_2$ by disabling $c_1$ from 0 in $I_2$. Hence, the answer for $\mathcal{E}$ is *some*.

Now we show that there is no single controller for all implementations of $E$. Consider $\mathcal{I}_2$ and $\mathcal{I}_3$, implementations of $E$ in Figure 5.6(c) and 5.6(d) respectively. Let $M$ be a solution for $\mathcal{I}_2$. $M$ must necessarily disable $c_1$ from the initial state. However, by disabling $c_1$ from the initial state, $M \| I_3$ cannot satisfy $\varphi$. Consequently, $M$ is not a solution to $\mathcal{I}_3$. Dually, let $M$ be a solution to $\mathcal{I}_3$. $M$ must disable $c_2$ from the initial state. However, $M \| I_2$ cannot satisfy $\varphi$. Hence, $M$ is not a solution to $\mathcal{I}_2$.

Summarising, we have shown that when the answer to an MTS control problem is all, there exists a controller that for all implementations of the MTS

environment model. If the answer is some, it is not possible to build a single controller for all implementations. Yet, it would be interesting to characterise the set of implementations that can be controlled. Moreover, such characterisation would lead to MTS control problems with all answer. Hence, allowing for single controller controlling all implementations. This ideas envisage promising future work.

## 5.4 Evaluation

In this section we present the results of applying our technique for checking realisability of MTS control problems to an extension of the production cell case study presented in Section 3.8.4.

### 5.4.1 Production Cell

Assume that in addition to the robot arm, oven, drill, press and, in and out trays, the factory now has a second oven to cook products. Due to resource limitations, it is not known in advance which (if any) oven will be available for production use. In addition, the exact model of the new oven is not available, hence, certain part of its behaviour is yet undefined.

For simplicity, we refer to the oven presented in Section 3.8.4 as Oven 1, and to the new oven as Oven 2. In Figure 5.7 we show an MTS modelling the behaviour of the ovens. Oven 1 is very reliable as it always finish cooking. On the other hand, Oven 2 may be less reliable as it admits an implementation in

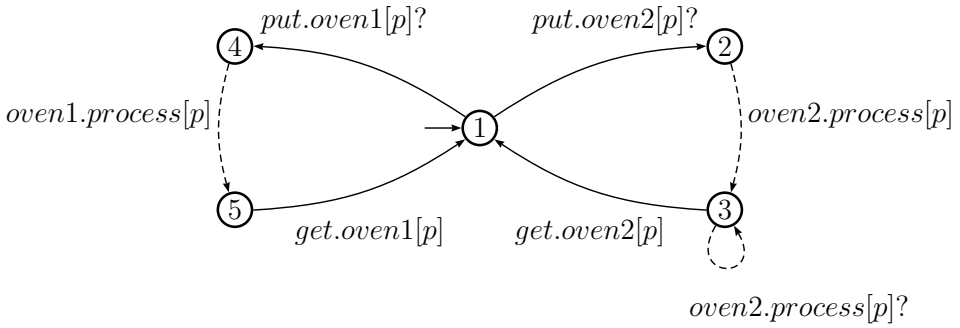



Figure 5.7: Second Oven for the extended Production Cell

which it may loop cooking forever. Note that the maybe loop in which Oven 2 cooks forever (i.e. state 3, Figure 5.7) denotes that it is not yet defined which model (i.e. implementation) of Oven 2 will be available. In addition, as it is uncertain which ovens (if any) will be available, transitions on the "put" action are maybe (see Figure 5.7).

Regarding the production process, any oven is equally useful to cook products. Consequently, products of type $A$ require using one of the ovens, then the drill and finally the press, while products of type $B$ are processed in the following order: drill, press, oven (either of them).

As described in Section 3.8.4, the drill and the press cannot be used simultaneously. In addition, the ovens must not be used simultaneously either.

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

The liveness goal of the factory is to produce products of type $A$ and $B$ without postponing indefinitely the production of either type.

The answer for the MTS control problem is *some* as any implementation that includes Oven 1 is controllable (see Section 3.8.4) and, there is no controller for an implementation that does not include Oven 1 and implements the Oven 2 that can remain processing forever.

Interestingly enough, there exists a unique controller that can control all implementations of this extension of the production cell. Indeed, the controller that includes both ovens and assumes that Oven 2 finish cooking after a finite number of process actions can control any controllable implementation.

Summarising, we have applied our technique to an extension of the production cell case study. We have successfully answered the realisability question and found out that this example allows for a template controller that can control any controllable implementation.

## 5.5 Limitations

The MTS control problem is limited to deterministic environment models. However, as MTS are supposed to be useful in the context of incremental elaboration, in many cases, it is reasonable to assume that nondeterminism will appear in initial models and will then tend to disappear as our knowledge on the real environment increases.

In addition, as the MTS control problem is reduced to LTS control, in a setting where the controller has full information of actions and state of the environment our technique can handle the setting of nondeterministic MTSs

and considering only deterministic implementations.

Finally, solving the synthesis problem for nondeterministic MTS, which corresponds to imperfect information games, is not straightforward. Nevertheless, we believe that it would reduce to synthesis for nondeterministic LTS in much the same way as with the deterministic variant.

# Chapter 6

# Tool Support

We have extended the MTSA [DFCU08] tool to support the techniques presented in this thesis. In this chapter we describe our tool, provide insights on how it is implemented and show how to use it to model the control problems presented in this thesis. Our extension of MTSA is available as an open source project in sourceforge (`http://sourceforge.net/projects/mtsa`).

## 6.1 Implementation

Regarding the implementation of SGR(1), in [PPS06] an efficient algorithm for solving games with GR(1) winning conditions is presented. However, the notion of game differs from the one used in this work. Although, both games are turn-based, the order is slightly different, e.g. in [PPS06], the environment chooses its next valuation and, only then, the controller gets

to choose what to do next. In our case (see Definition 2.9), the controller first restricts transitions over the set of controllable actions and then the environment gets to choose the next state of the game. To use the implementation in [PPS06] a preprocessing step to convert one game to the other would be necessary. In addition, the algorithm proposed in [PPS06] manipulates sets of states using a symbolic representation in the form of Binary Decision Diagrams (BDDs [Bry86]). In other words, it uses a data structure that manipulates sets of states. The algorithm requires efficient implementations of set union, intersection, negation, and the computation of the set of predecessors of a state. It is most suitable for cases where the states of the "game graph" are obtained by setting values to variables. This is not the case in our setting where every state is an entity in its own right. There is no natural way to represent sets except by lists of states resulting in a linear overhead for every set operation. Thus, the symbolic algorithm that computes $O(m \cdot n \cdot |S|^2)$ symbolic operations would result in an algorithm that in practice uses $O(m \cdot n \cdot |S|^3)$ operations, where $|S|$ is the number of states, $m$ is the number of environmental assumptions and $n$ is the number of controller goals. Hence, the symbolic algorithm is not the best suited in our context.

A different approach is to translate the LTS control problem to BDDs and apply the algorithm proposed in [PPS06]. How such a translation is to be made is far from being trivial and requires further research. It could be done by translating the composition of the complete environment model and the properties to BDDs and then applying the BDD-based algorithm. Such par-

allel composition does not preserve the structure of the components and, as is well known, using the structure of the models to be translated can greatly improve the efficiency of the BDDs. Hence, it is not clear whether the resulting BDDs would result in better performance when used as inputs to the algorithm. On the other hand, the translation could be approached by translating each LTS to a number of BDDs. However, such an approach would need to consider the structure of each component and the communication between them. This is not trivial as there is no well-fitting correspondence between variables and states. Consequently, BDDs as support for solving LTS control problems can be an interesting option and it requires further research. Even though this is not the focus of our work it would definitely constitute challenging future lines of investigation.

Reducing game solution to other symbolic solutions such as SAT solving or QBF [le2] solving is an interesting research problem in its own right. Currently it is not known how to use SAT solvers for efficiently solving synthesis problems [HKLN12]. This is ongoing research in the community and it is yet not clear whether effective solutions for control problems will arise from it. In any case, in order to use such approaches for synthesis in our context the problem of efficient symbolic encoding of LTSs through variables (as for BDDs) needs to be solved.

In [JP06], an enumerative [1] solution to games with the GR(1) winning condition, based on the same ideas, is presented. However, the states of their games are partitioned into two (one controlled by each player). Hence, states are

[1] Intuitively, enumerative algorithms handle states individually and separately. This is contrasted with symbolic algorithms, which handle sets of states together.

expected to be either uncontrollable or controllable. Furthermore, in [JP06], strategies are defined as a partial function, which takes sequences of states and yields a state. In other words, given a play conforming with the game their controller has to choose a particular successor while, in our case, the controller may choose a set of possible successors. This notion of strategy favours the construction of *best effort* controllers. It is easy to prove that for some cases where assumptions and environment model are not compatible, our approach produces best effort controllers while the algorithm proposed in [JP06] does not guarantee so. Hence, we implement a variation of [JP06] which produces controllers complying with our notion of strategy and handles every state independently and its run time is $O(m \cdot n \cdot |S| \cdot |E|)$, where $|E|$ is the number of transitions. A detailed description of our algorithm is provided in Section 3.6.

In this work we have focused on the development of novel synthesis techniques and building tools to evaluate them. Hence, there is a wide spectrum of optimisation techniques that have yet to be evaluated as possible improvements of our implementation, such as partial order reduction or bit caching [Hol97]. Such implementations could be interesting in the wider context of game solving and the study of their application to synthesis could be of interest to the games community.

## 6.2 Modelling Control Problems

In this section we show the details on how to model control problems using our tool.

### 6.2.1 Modelling the Environment

In MTSA environment models are described with an extension of the FSP language, presented in Section 2.4.

The extension of FSP provided by the MTSA tool includes traditional operators for describing behaviour models, such as action prefix (`->`), choice (`|`), sequential composition (`;`), and parallel composition (`||`). Extensions support modelling partial behaviour models [FDB+10]. Maybe transitions are denoted with a question mark `?` as suffix. MTSA supports several MTS operations, such as MTS refinement (Definition 2.7), merge [FDB+10], abstract and constraint. For more details see [DFCU08].

Our tool integrates functionality to construct, analyse and elaborate both MTS and LTS models and provides a graphical environment aimed to facilitate these tasks. A snapshot of the MTSA tool is shown in Figure 6.1.

In our tool environment models are described in FSP (see Section 2.4). In Figure 6.2 we show a snapshot of the FSP editor provided by our tool.

The code shown in Figure 6.2 corresponds to that of the ceramic cooking example presented in Section 4.2. It defines a process called `CERAMIC` that

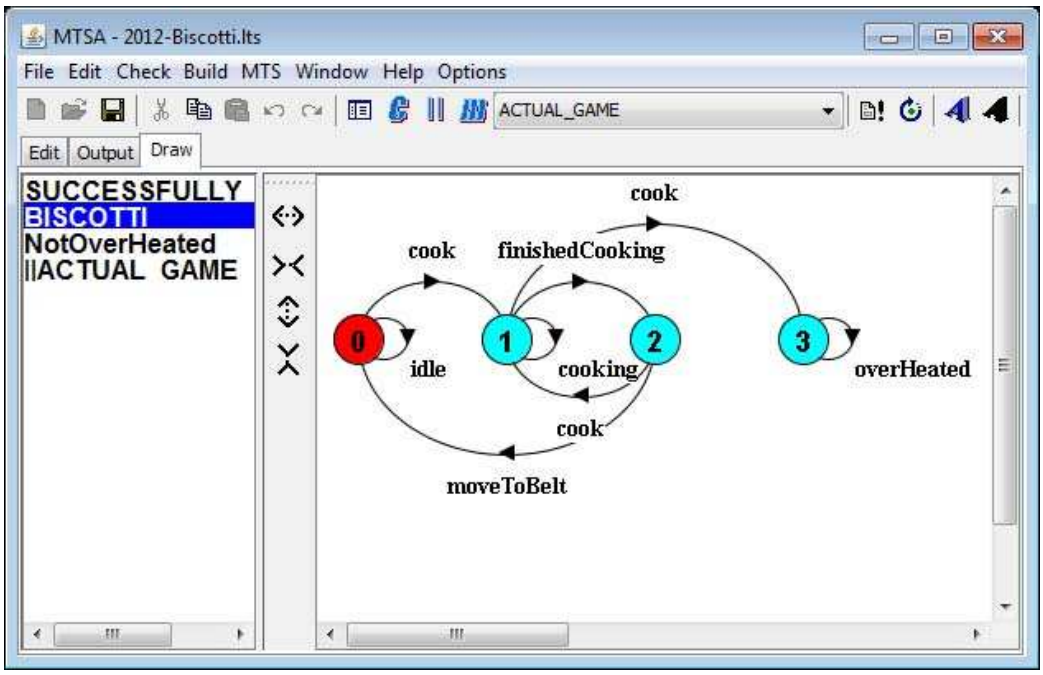



Figure 6.1: MTSA - Draw View

allows for either staying idle (looping to itself), or starting to cook, leading to the auxiliary process `COOKING`. Similarly, `COOK` may loop to itself while the oven is cooking, or it may lead to the auxiliary process `OVERHEATED` if there is a second attempt to cook while the oven is still cooking. Additionally, if the oven signals that it has finished cooking the model transitions to `FINISHED`. Finally, `FINISHED` transitions either to `COOKING` by starting to cook, or it resets to `CERAMIC` by moving the cooked ceramic to the conveyor belt.

### 6.2.2 Modelling Controller Goals

We have added a set of keywords to FSP to support *i)* Synthesis of LTS controllers, *ii)* Assumptions compatibility checking, and *iii)* Answering the MTS

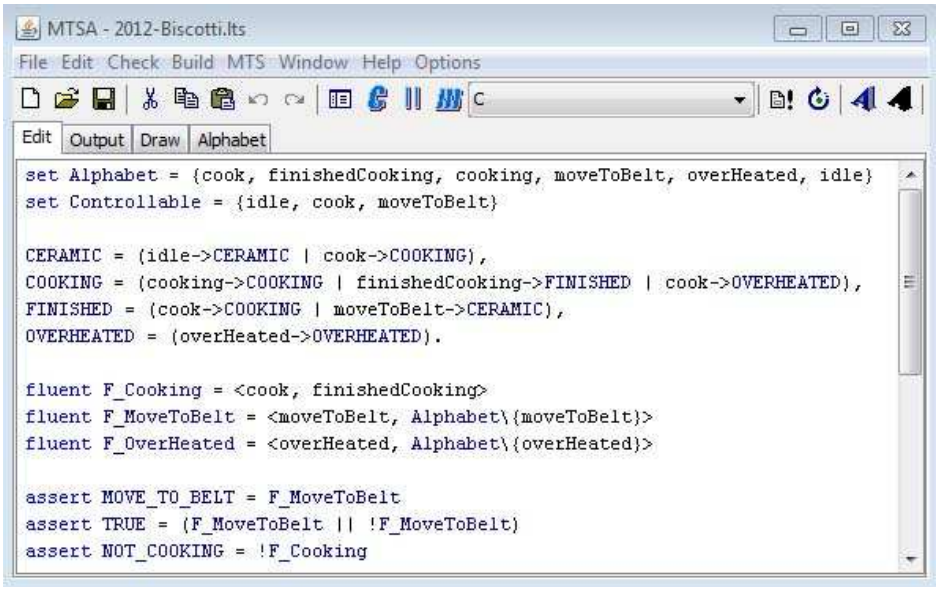



Figure 6.2: MTSA - FSP Editor

control problem (restricted to GR(1) goals). In Figure 6.3 we show the FSP code for assumptions compatibility checking and synthesising a controller for the ceramic cooking example presented in Section 4.7.

It is important to note that the tool has been developed to support GR(1)-like goals. Hence, more expressive specifications are not yet supported.

The `controller` operator returns (if exists) a controller that satisfies a specification for a given environment model. For example, in Figure 6.3 we show the FSP code modelling the controller goals for the ceramic cooking example where `controller` is applied to the environment model `CERAMIC` and the goal `G1`.

The `checkCompatibility` operator checks if the assumptions on the environment behaviour are realisable by the environment, if this is the case the model

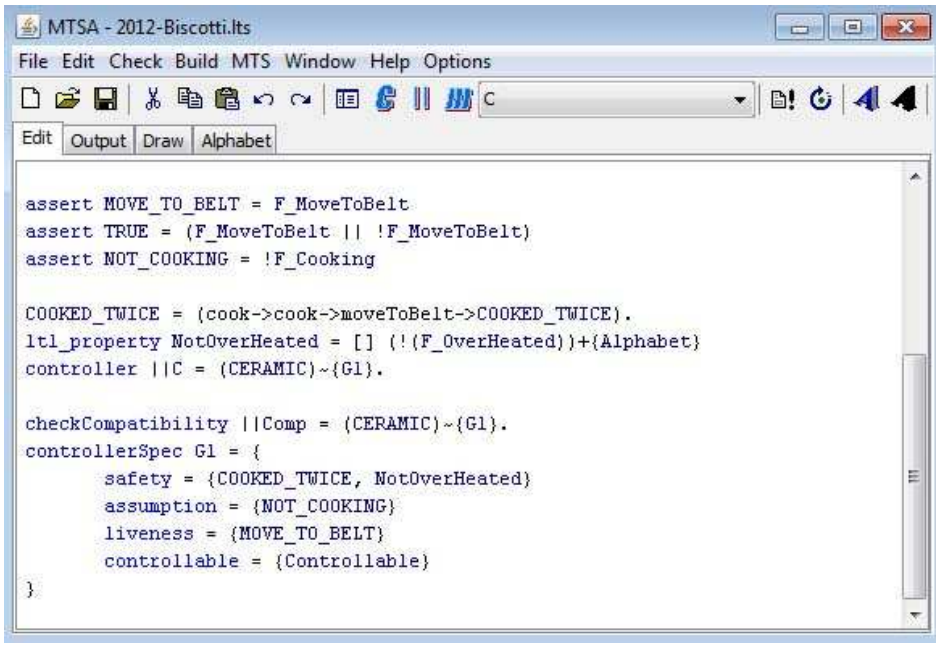



Figure 6.3: Controller Goals - FSP Example

returned by `controller` is guaranteed to be non-anomalous (See Section 3.4). If the environment model is an MTS, the `controller` operator answers the realisability question, i.e. returns all, some or none.

Controller goals are defined within the `controllerSpec` operator. The `safety` keyword allows for defining safety requirements. There are two ways of defining such restrictions. First, by providing an LTS model representing expected behaviour, e.g. `COOKED_TWICE` in Figure 6.3. Second, using the `ltl_property` operator which given a safety formula $\varphi$ builds an LTS model $E$ in which any trace violating $\varphi$ leads to the error state in $E$ (e.g. `TOOL_ORDER`).

The liveness part of the controller goal is defined using the `assumption` and `liveness` operators that specify the environment assumptions and controller liveness goals respectively. Both are defined with FLTL assertions repre-

sented with the keyword `assertion`. In the example, `NOT_COOKING` it is defined as the only environment assumption and, `MOVE_TO_BELT` as the only controller liveness expectation. Thus, the synthesised controller will guarantee that if infinitely many times the oven is not cooking, objects are placed in the belt infinitely often.

## 6.3 Evaluation

In this section we report on the results we have obtained by running the tool with the case studies we have presented in this thesis.

Although the tool has been developed as a prototype to evaluate the proposed techniques, its running times are within the expected parameters yielding controllers in less than 30 seconds even for games of 16000 states. Memory consumption can be seen as a characteristic of the implementation to be improved. Nevertheless, all the case studies we have run required less 600 Mb, which in modern terms is within reasonable limits.

We focus on the complexity of the case studies in terms of the size of the environment model, the translated game and the controller synthesised. We also report on the running time and memory consumption of the tool, which gives an intuition on the size of problems it can handle.

All experiments were performed on an Intel Core I5 1.8GHz, with 4GB RAM.

In Figure 6.1 we show the results we have obtained running MTSA on the above-mentioned case studies. The first column gives a descriptive name and

| | **Env. Model # States** | **Game # States** | **Controller # States** | **Time ms** | **Memory MB** |
|---|---|---|---|---|---|
| Production Cell (3.1) | 1393 | 6962 | 4778 | 12000 | 214 |
| Running example (3.3) | 7 | 11 | 3 | 65 | 10 |
| Autonomous Vehicles (3.8.1) | 43 | 57 | 38 | 75 | 15 |
| Purchase & Delivery (3.8.3) | 17 | 21 | 17 | 70 | 9 |
| Production Cell (3.8.4) | 18 | 51 | 41 | 240 | 11 |
| Travel Agency (4.1) | 3082 | 4507 | 2352 | 2200 | 130 |
| Ceramic Cooking (4.2) | 11 | 14 | 11 | 90 | 12 |
| Production Cell (4.6.1) | 2481 | 15994 | 12496 | 28800 | 470 |
| Pay & Ship (4.6.2) | 91 | 95 | 94 | 260 | 164 |
| Autonomous vehicles (4.6.3) | 45 | 99 | 49 | 140 | 159 |
| Bookstore (4.6.4) | 32 | 36 | 35 | 80 | 10 |
| Production Cell (4.7.1) | 8621 | 10159 | 5377 | 5200 | 270 |
| Server Example (5.1.1) | 6 | 15 | 16 | 175 | 79 |
| Production Cell (5.4.1) | 271 | 432 | 114 | 3600 | 26 |

Table 6.1: Overview of the case studies results

a reference to the section in the thesis where the example is presented. The next three columns show the number of states of the environment model, the game obtained from it, and the synthesised controller. The last two columns report on the synthesis time in seconds and memory consumption in megabytes.

The last two rows correspond to MTS control problems. Here we report on the one of the LTS control problems solved in order to answer the MTS realisability question.

In order to provide an impression of the scalability of the techniques and the tool we have run the Production Cell case study in Section 4.6.1 with different environment model sizes.

Recall that the case study describes a scenario in which we have a robot arm, a drill, a painting tool, an in tray, an out tray, a recycle bin and some additional sensors. The goal is to use the arm aided by sensors to move objects from the in tray, through a combination of tools according to a high-

| Moves | Env. Model # States | Game # States | Controller # States | Time ms | Memory MB |
|---|---|---|---|---|---|
| 5 | 7229 | 8539 | 4429 | 3600 | 260 |
| 10 | 9549 | 11239 | 6092 | 6700 | 314 |
| 15 | 11869 | 13939 | 7799 | 6800 | 406 |
| 20 | 14189 | 16639 | 9506 | 11117 | 451 |
| 25 | 16509 | 19339 | 11239 | 13168 | 545 |
| 30 | 18829 | 22039 | 12925 | 15323 | 611 |
| 35 | 21149 | 24739 | 14624 | 19173 | 611 |
| 40 | 23469 | 27439 | 16362 | 23419 | 764 |
| 50 | 28109 | 32839 | 19769 | 34547 | 829 |
| 60 | 32749 | 38239 | 23198 | 49965 | 1,060 |
| 80 | 42029 | 49039 | 30016 | 81968 | 1,360 |
| 100 | 51309 | 59839 | 36873 | 175586 | 1,630 |
| 150 | 74509 | 86839 | 53971 | 638682 | 2,129 |
| 200 | 97708 | n/a | n/a | n/a | n/a |

Table 6.2: Progression in the size of the Production Cell

level production process, to the out tray.

The controller for the arm satisfies three safety requirements: objects are produced alternating their colour, objects must be drilled before they are painted, and the arm can attempt to pick up only if a object is at the specified location.

The liveness part of the goal is to guarantee that infinitely many red objects are produced, and it is assumed that yellow objects will be provided infinitely often. In addition, after a fixed number of moves the controller is forced to try to pick up objects from the in tray; this is to avoid anomalous controllers. The number of moves after the arm is forced to pick up objects is a parameter of the environment model. In Table 6.2 we report the results of running the case study for different bounds for the pick up restriction.

The first column indicates the number of moves after which the arm is forced to try to pick up an object from the in tray. The next three columns show the number of states of the environment model, the game obtained from it,

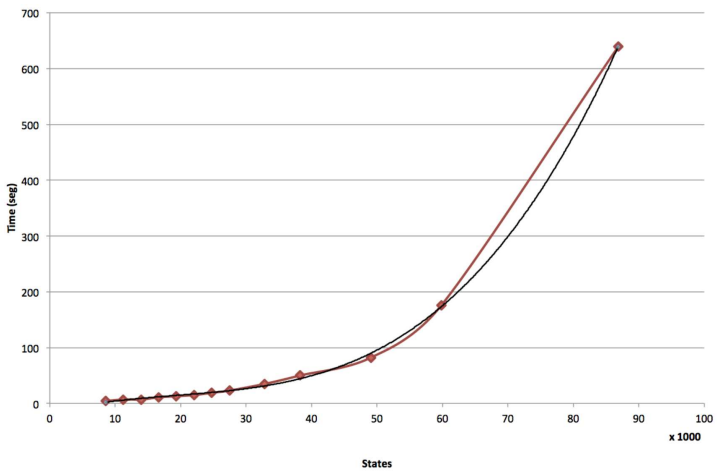



Figure 6.4: Production Cell - Time complexity.

and the synthesised controller. The last two columns report on the synthesis time in seconds and memory consumption in megabytes.

As expected the time required for solving the problem grows as a function of the size of the model and the size of the safety requirements. Specifically, as shown in Figure 6.4 the time to compute a controller, in this case, fits a quadratic curve. This is not surprising since the cubic (worst case) time complexity of our algorithm arises when solving an unrealisable control problem. Note that in this example the size of the model grows with the number of moves the arm can do before being forced to pick up, or the size of the safety requirements could grow, for instance, if the alternation pattern between red and yellow objects becomes more complex. Recall that the complexity of our algorithm is $O(a \times g \times |s| \times |E_G|)$ where $a$ is the number of assumptions, $g$ is the number of liveness goals, $s$ is the size of the safety restrictions and $E_G$ is the size of the game.

In this case, with one environment assumption, one liveness goal and a model

up to 75,000 states, our tool yielded a controller successfully. For models greater than 97,000 the tool runs out of memory. Nevertheless, these results are promising as the tool has not been optimised in any way. We have implemented the algorithms as described in the thesis with regular data structures and no focus on limiting use of resources. We strongly believe that further optimisations on the memory consumption of the tool will lead to support for synthesis problems of much greater complexity.

# Chapter 7

# Discussion and Related Work

In this chapter we present a discussion on related work with respect to a number of topics that are important to consider while putting our work in context.

## 7.1 Synthesis Techniques

Our work builds on that of the controller synthesis community and particularly on the generalised reactivity synthesis algorithm GR(1) proposed in [PPS06]. This line of work originates in updating Büchi, Landweber, and Rabin's work [BL69, Rab70] to modern terms [PR89]. While [PPS06] handles only a subset of the possible specifications, other recent work tries to bypass the hurdles involved in solving the general problem (e.g., [KV05] and its extension in [SF07]). The community has largely focused on controllers for

embedded systems and digital circuits (cf. [BGJ+07, SSR08]), hence adopting a shared memory model: The controller is aware of changes in the environment by querying the state space shared with the environment. For instance, GR(1) uses Kripke structures, state machines with propositional valuations on states, where the environment and the controller update and read respectively controlled and monitored propositions. However, in many settings such as requirements engineering, architectural design and self-adaptive systems, a message passing communication model in the context of a distributed system is typically considered. Hence, controller synthesis techniques require adaptation to the notion of event-based communicating machines [Hoa78]. This adaptation, specifically to Labelled Transition Systems (LTS) [Kel76] semantics and CSP-like parallel composition [Hoa78], is a contribution of this thesis.

The change from state-based to event-based semantics introduces the need for determinism of the environment to guarantee that the controller has sufficient information about the state of the environment to guarantee it satisfies its goals (see Definition 3.2 and Theorem 3.4). The change also introduces the need for a sound methodological approach to the definition of assumptions in order to avoid anomalous controllers.

Even though several behaviour model synthesis techniques have been studied (e.g. [DLvL06, YD06, BSL04]) these are restricted to user-defined safety requirements applying variations of the backward error propagation technique [RN95]. The exceptions that we are aware of relate to the self-adaptive systems and the planning communities.

In the self-adaptive systems community many architectural approaches for adaptation have been proposed. At the heart of many adaptation techniques, there is a component capable of designing at run-time a strategy for adapting to the changes in the environment, system and requirements (e.g. [DGM09, HGS04, GBMP97, KM07]). These architectures do not prescribe the mechanism for constructing adaptation strategies. The techniques we propose in this work could be used in the context of all of these architectures. In fact, we believe, that the methodological guidance that our approaches offer could help integrating the controller synthesis techniques into these architectures in a sound way. Furthermore, by providing support for explicitly modelling the environmental assumptions, our approaches allow to clearly understand what system goals are guaranteed to be satisfied.

## 7.2 Environment Assumptions

A few approaches to automated construction of adaptation strategies exist. Sykes et al. in [SHMK07, HSMK09b] build on the "Planning as Model Checking" framework [GT00] to construct plans that aim to guarantee reaching a particular goal state. Thus, this technique can handle certain liveness requirements. However, the execution of the plan is restarted every time the environment behaves unexpectedly. Hence, there is an implicit assumption that the environment behaves "well enough" for the system to eventually reach the goal state. Validating that the environment will behave "well enough" is not possible as the notion is not defined and it is not clear what

guarantees are provided by the plans. In the proposed approaches, assumptions are explicit and hence guarantees are clear. In addition, it is possible to validate if the assumptions needed in order to achieve the goals hold in the environment in which the controller is to be deployed. For instance, in the *Autonomous Vehicles* case study from Section 3.8.1, without any assumption on how the door behaves, there would not be a plan for the robot that guarantees bringing aid packages form south to north room infinitely often. The same occurs in the *Purchase & Delivery* case study (See Section 3.8.2), where without the assumption that if the user receives combination of furniture and delivery options often enough, he will acknowledge such combinations often enough.

More generally, the planning as model checking framework (e.g., [GT00]), supports CTL goals, requires a model in the form of a Kripke structure and does not consider the problem of composing the environment with a machine that is responsible for guaranteeing the system's goals. Consequently, it does not distinguish between controlled and monitored actions and the plans that are generated would not be realisable by the software system. Moreover, since planning as model checking synthesises from CTL formulas it is not possible to distinguish which are the assumptions on the environment behaviour required to guarantee the satisfaction of the controller goals.

Assumptions on environment behaviour are a key part of the synthesis problem. In general, without proper assumptions, controllers cannot be produced. In [CHJ08] a technique for correcting unrealisable specifications is proposed. Given an unrealisable specification $\varphi$, they compute an environ-

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

The notion of assumption compatibility was discovered independently in various contexts and used differently. In [CGG07], it is used in the context of vacuity detection (cf. [BBDER01] and a large body of other work [KV03, AFF+03, GC04, CGG07]) and for debugging environment models. They say that an environment model $E$ *guarantees* a property $\varphi$ iff for every possible controller $M$, $E\|M \models \varphi$. In other words, a property $\varphi$ is called an *environment guarantee* iff the environment satisfies $\varphi$ regardless of what the controller might do.

In the context of [CGG07] this is considered bad. Here, we consider it as a good thing. The fact that Chechik et al. consider CTL and not LTL is a minor difference. However, they implement only a pre-condition for

checking environment guarantees. Furthermore, it has since been discovered that implementing the complete check for the test suggested by Chechik et al. is in fact equivalent to realisability [GP09] and, as mentioned previously, unless restricted to a manageable fragment of the logic does not work well in practice. Our approaches are thorough (i.e. complete) but for applicability reasons we restrict the specifications to GR(1) formulas.

In [CGP03] a technique for automatic generation of assumptions in the context of assume-guarantee reasoning is proposed. Given a property and a parallel composition of two models, the technique produces the assumptions one of the model has in order to satisfy the property. If the second model satisfies these assumptions then it is possible to conclude that the composition does too. It could be argued that this technique is related to ours as the generated assumptions could be considered the controller. However, there are two main differences. First, the techniques used by Cobleigh et al. are inappropriate for usage in a context that distinguishes between monitored and controlled actions (essentially using model checking instead of game solving). Second, the pruning of unsafe states, can ensure safety but cannot handle liveness requirements. More recently, in [EGP08] Emmi et al. have presented a technique for assume-guarantee verification for interface automata. The technique distinguishes between controllable and monitored actions. However, liveness requirements are not supported.

## 7.3 Modelling Language

We use Labelled Transition Systems as our modelling framework. We distinguish between controlled and uncontrolled actions in the definition of the control problem. It could be argued that LTSs do not provide proper support for expressing the required environment models. There are a number of formalisms to describe behaviour distinguishing controlled from monitored actions, such as Input/Output Automata [LT89] or Interface Automata [dAH01] (IA). Since I/O automata require the input actions to be enabled from every state, they do not fit our environment model. There is no technical limitation that prevents us from using IA. Adapting our approach to IA is relatively simple. We treat IA output actions as controllable actions and input actions as monitored actions. The conversion of the control problem to a game remains as it is now with the slight difference of considering output actions as controllable actions. Computing the strategy and translating it to an interface automaton can be done straightforwardly. Finally, the generated controller might need to be slightly more restricted in order to satisfy the requirement of being a legal environment for the environment model according to IA. Indeed, our definition of Legal LTS is slightly less restrictive than that of Legal Environment for IA.

## 7.4 Fallible Environment Models

As we have already mentioned, in Sykes et al. setting the execution of the plan is restarted every time the environment behaves unexpectedly. Hence there is an implicit assumption that failures may occur, yet, the environment is assumed to be, somehow, well behaved. Similar assumptions are made in the planning as model checking approach. Hence, even though some notion of failures is supported by both techniques, no characterisation of what failures are is provided. Moreover, there are no guarantees as to when goals are actually achieved by plans.

In [IT07] the problem of constructing an adaptation strategy is studied. However, it is limited to enforcing safety properties and uses a backward error propagation technique [RN95] to construct controllers. The lack of explicit live conditions makes failures and fairness conditions irrelevant.

## 7.5 Partially Specified Environment Models

Partial behaviour models have been extensively studied. A number of such modelling formalisms exist, such as multi-valued Kripke structures [Fit91], Modal Transition Systems (MTSs) [LT88] and variants such as Disjunctive MTS [LX90]. The results presented in this work would have to be revisited in the context of other partial behaviour formalisms. However, since many complexity results for MTS hold for extensions such as DMTS, we believe that our results could also extend naturally to these extensions.

The formal treatment of MTSs started with model checking, which received a lot of attention (cf. [BG99, BG00, GP09]). Initially, a version of three-valued model checking was defined [BG99] and shown to have the same complexity as that of model checking. Generalised model checking [BG00] improves the accuracy of model checking of partial specifications. Indeed, for example, three-valued model checking may yield that the answer is unknown even when no implementations of an MTS satisfy the formula. However, complexity of generalised model checking is much higher [GP09]. Furthermore, in order to solve generalised model checking one has to reason about games and not about transition systems. It is interesting that in the context of MTS control this does not happen. Indeed, our definition of control is more similar to generalised model checking than to three-valued model checking. However, we can reason about it in the context of LTS control and it has the same complexity as that of LTS control.

Another related subject is abstraction of games. For example, in [Ste98] abstraction is applied to games in order to enable reasoning about infinite games. Similarly, in [HJM03] abstraction refinement is generalised to the context of control in order to reason about larger games. Their main interest is in applying abstraction on existing games. Thus, they are able to make assumptions about which states are reasoned about together. We, on the other hand, are interested in the case that an MTS is used as an abstract model. In this case, the abstract MTS is given and we would like to reason about it.

Our work is also related to weakening of goals in order to make specifica-

tions realisable [KHB09]. This is related as our work answers what are the acceptable problem domains that can be actually controlled. We represent the set of acceptable environments as an MTS.

# Chapter 8

# Conclusions

The most general goal of the work in this thesis has been to develop synthesis techniques to aid the construction, elaboration, and analysis of models of concurrent and distributed systems. We started by studying controller synthesis techniques and existing planning approaches used in Software Engineering which has triggered the main goal of this work: to generate controller synthesis techniques that provide support for event-based behaviour models, expressive goal specifications and the application of Requirements Engineering best practices and processes.

## 8.1 Contributions

The main contribution of this thesis is a number of controller synthesis techniques that overcome limitations of existent synthesis techniques in the soft-

ware engineering community. The presented techniques work for expressive environment models and system goals, distinguish between controlled and monitored actions and, differentiate between prescriptive and descriptive aspects of the specification of system goals, environment behaviour, and environment assumptions.

More specific contributions of this thesis 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 Transition Systems and parallel composition in the definition of the *LTS control problem*; *(iii)* the definition of a restricted LTS control problem, named *SGR(1) LTS* that supports safety and GR(1)-like properties, and provide a polynomial time solution which builds, from a theoretical perspective on GR(1) games and from an implementation perspective on *(iv)* a rank-based algorithm which is suitable for explicit state space representation; *(v)* characterisation of non-anomalous controllers (best effort and assumption preserving) ;*(vi)* a sufficient condition, i.e. assumption compatibility, for an event-based setting to guarantee correctness of the synthesis procedure and to avoid anomalous controllers. *(vii)* A technique for synthesising event-based controllers when controllable actions may fail (i.e. fallible environment), *(viii)* a discussion of the fairness conditions required for such problem domains with failures. In particular the observation that strong fairness of successful controlled actions may be insufficient to guarantee that effective controllers are synthesised; *(ix)* novel fairness condi-

tions, i.e. t-strong fairness and strong independent fairness, that are stronger than strong fairness and are good fits for more realistic controller synthesis settings, *(x)* the definition of a polynomial time LTS control problem, named *RSGR(1)* that supports GR(1)-like goals under strong independent fairness assumption; *(xi)* the adaptation of the compatible assumptions notion to guarantee that RSGR(1) does not produce anomalous controllers; and *(xii)* 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 RSGR(1) can be applied. Finally, *(xiii)* we define the *MTS control problem* that given an MTS it answers whether *all*, *none* or *some* of the LTS implementations it describes admit an LTS controller that guarantees a given goal, *(xiv)* the technique yields an answer to the MTS control problem showing that, despite dealing with a potentially infinite number of LTS, the MTS control problem is actually in the same complexity class as the underlying LTS synthesis problem.

In addition, a tool supporting all the techniques presented in this work is reported. The tool improves and extends the MTSA tool set.

The evaluation of the presented techniques and tool has been conducted via a number of case studies from different domains, such as, robotics, web-services composition and industrial automation. The obtained results indicate that the discussed techniques are applicable to a wide range of problems, allowing for accurate descriptions and effective controllers.

## 8.2 Limitations

The techniques we propose in this thesis have been exemplified with case studies from several domains. However, we have found some limitations that are not being handled in this thesis. Such limitations can be grouped under three main topics: determinism of the environment model, the centralised nature of the controller, and expressiveness of the supported goals.

Our first limitation is the requirement that the environment model be deterministic. This issue arises from the distinction between partial and full information. Event-Based controllers can only observe the performed actions and not the actual state of the environment model. In deterministic environment models the controller has full information as the actual state of the environment model can be deduced from the sequence of actions observed. In non-deterministic environment models the controller has partial information as it can only deduce a set of possible states. Synthesis with partial information presents a greater challenge than synthesis with full information. Only in recent years have a few approaches towards partial information started to emerge. However, these have not been studied for event-based settings. We plan to investigate the adaptation of such techniques for the setting presented in this thesis.

The second limitation is the expressiveness of the supported goals. SGR(1) and RSGR(1) support GR(1)-like formulas, in other words safety and an expressive subset of liveness goals. We have shown that such goals are indeed expressive enough in a wide spectrum of problems. However, in some

situations more complex kinds of goals might be required. For instance, in Section 4.7 we report on a variation of the production cell that cannot be handled by our techniques as it requires strong fairness assumptions on the environment behaviour. Such requirements cannot be expressed with GR(1)-like formulas. Hence, techniques supporting more expressive goals are, in some cases, required. We plan to extend our techniques with effective algorithms supporting more expressive goals.

The third limitation is the centralised nature of the controllers our techniques produce. This is limiting in the context of distributed environments since it might be neither desirable nor possible to introduce a centralised controller due to unreliable communication or insufficient sensing for example. Synthesising distributed controllers is, in many cases, undecidable. Under certain assumptions about the interface of the components and the architecture in use, the problem is decidable yet its complexity is exponential. Hence, we plan to investigate how to apply our techniques to build centralised controllers and, based on the possibilities of the components' interfaces, to distribute the centralised controller as much as possible.

Although these limitations give us an indication of what range of problems may be tackled using our techniques, more investigation is necessary to delineate this class with greater precision.

## 8.3 Future Work

In the future we envisage working in two different directions. On the one hand, we aim at relaxing the requirement on determinism for the environment model that is currently in place for assuring the soundness of our approaches. In fact, this is closely related to non-observability of events controlled by the environment and imperfect information games, areas that we also intend to further investigate.

On the other hand, we intend to conduct research into possible ways of providing feedback when control problems are not realisable. In the case in which a controller that satisfies the specified goals does not exist, having feedback helping the user to identify the problem would be very helpful. We plan to investigate ways of providing feedback for such cases, perhaps, by providing a counter-strategy for the environment or applying debugging techniques for unrealisable synthesis problems.

## 8.4 Closing Remarks

The need for controller synthesis techniques has been considered by the software engineering community for many years. To this end, significant effort has been made in developing techniques that support different levels of expressiveness. However, the automatic synthesis of controllers in a framework that integrates requirements engineering best practices has been largely neglected. Controller synthesis techniques need to keep growing to support a

wider spectrum of engineering problems. To this end, the relation between requirements engineering practices and processes, and controller synthesis must be further explored. I strongly believe that such exploration would result in more effective and applicable synthesis techniques that would not only be interesting and challenging from a theoretical point of view, but would also have a deep impact in the application of controller synthesis techniques in the industry.

# Appendix A

# Proofs

**Theorem A.1.** *Given $\mathcal{R} = \langle \mathcal{E}, T \rangle$ 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 $\mathcal{R}$.*

*Proof.* $\Rightarrow$)

Given $\mathcal{E} = \langle E, \{(As, G)\}, A_C \rangle$, where $E = (S, A, \Delta, s_0)$, $G = \bigwedge_{i=1}^{m} \square \diamondsuit G_i$ and $As = \bigwedge_{i=1}^{n} \square \diamondsuit A_i$.

Let $T = \{(try_i, fail_i, suc_i \mid 0 \leq i \leq t\}$ be the set of try-response triplets, where $t$ is the number triplets in $\mathcal{R}$. Let $\dot{F}$ be the fluent that becomes true whenever a failure occurs, formally defined as $\dot{F} = \langle \{fail_i\}, A \backslash \{fail_i\}, false \rangle$.

This will be proven by contradiction. Suppose there is a trace $\pi \in tr(E \| M)$ such that

$$\pi \models \left( \bigwedge_{i=1}^{n} \Box \Diamond A_i \wedge \bigwedge_{i=1}^{t} (\Box \Diamond try_i \Rightarrow \Box \Diamond (suc_i \wedge \bigwedge_{q=1}^{n} \neg \dot{F}\ \ WA_q)) \right) \Rightarrow \bigwedge_{i=1}^{m} \Box \Diamond G_i$$

and

$$\pi \models \bigwedge_{i=1}^{n} \Box \Diamond A_i \wedge \neg \bigwedge_{i=1}^{m} \Box \Diamond (G_i \vee \dot{F})$$

Simplifying the negation in the front of the last conjunction

$$\pi \models \bigwedge_{i=1}^{n} \Box \Diamond A_i \wedge \neg \bigwedge_{i=1}^{m} \Box \Diamond G_i \wedge \neg \Box \Diamond \dot{F}$$

Since $\pi \models \bigwedge_{i=1}^{n} \Box \Diamond A_i$ holds the following cases arise. We analyse them in turn.

1. Case the antecedent $\bigwedge_{i=1}^{t}(\Box \Diamond try_i \Rightarrow \Box \Diamond(suc_i \wedge \bigwedge_{q=1}^{n} \neg \dot{F}\ \ WA_q))$ holds, then $\pi \models \bigwedge_{i=1}^{m} \Box \Diamond G_i$ which contradicts $\neg \bigwedge_{i=1}^{m} \Box \Diamond G_i$.

2. Case the antecedent $\bigwedge_{i=1}^{t}(\Box \Diamond try_i \Rightarrow \Box \Diamond(suc_i \wedge \bigwedge_{j=1}^{n} \dot{F}\ \ WA_j))$ does not hold, then there exists some $0 \leq k \leq t$ such that

$$\Box \Diamond try_k \wedge \neg(\Box \Diamond (suc_k \wedge \bigwedge_{q=1}^{n} (\neg \dot{F}\ \ WA_q)))$$

Simplifying the negation

$$\Box \diamondsuit try_k \wedge (\diamondsuit \Box \neg suc_k \vee \diamondsuit \Box \neg \bigwedge_{q=1}^{n} (\neg \dot{F}\ WA_q))$$

From the above the following cases arise.

(a) Case $\diamondsuit \Box \neg suc_k$ holds.

Then by try-response conditions (Definition4.1) and $\Box \diamondsuit try_k$ holding, it follows that $\Box \diamondsuit fail_k$ which contradicts $\neg \Box \diamondsuit \dot{F}$.

(b) Case $\diamondsuit \Box \neg \bigwedge_{q=1}^{n} (\neg \dot{F}\ WA_q)$ holds.

Then there exists $0 \leq j \leq n$ such that $\neg \dot{F} WA_j$ does not hold, which together with $\bigwedge_{i=1}^{n} \Box \diamondsuit A_i$ implies that $\pi \models \Box \diamondsuit \dot{F}$, which contradicts $\neg \Box \diamondsuit \dot{F}$.

$\Leftarrow$)

This will be proven by contradiction. Suppose there exists a trace $\pi \in tr(E \| M)$ such that

$$\pi \models \bigwedge_{i=1}^{n} \Box \diamondsuit A_i \Rightarrow \bigwedge_{i=1}^{m} \Box \diamondsuit (G_i \vee \dot{F})$$

and

$$\pi \models \bigwedge_{i=1}^{n} \Box \diamondsuit A_i \wedge \bigwedge_{i=1}^{t} (\Box \diamondsuit try_i \Rightarrow \Box \diamondsuit (suc_i \wedge \bigwedge_{q=1}^{n} (\neg \dot{F}\ WA_q))) \wedge \neg \bigwedge_{i=1}^{m} \Box \diamondsuit G_i$$

Since $\pi \models \bigwedge_{i=1}^{n} \Box \diamondsuit A_i$ it must be $\pi \models \bigwedge_{i=1}^{m} \Box \diamondsuit (G_i \vee \dot{F})$. If for every $0 \leq i \leq t$, $\Box \diamondsuit try_i$ does not hold, then by try-response conditions (Defini-

tion 4.1) $\Box \Diamond \dot{F}$ does not hold, from which follows that $\pi \models \bigwedge_{i=1}^{m} \Box \Diamond G_i$ which contradicts $\neg \bigwedge_{i=1}^{m} \Box \Diamond G_i$.

Otherwise, there exists $0 \leq p < t$ such that

$$\pi \models \bigwedge_{i=1}^{p} \neg \Box \Diamond try_i \wedge \bigwedge_{i=p+1}^{t} \Box \Diamond try_i \wedge \bigwedge_{i=p+1}^{t} \Box \Diamond (suc_i \wedge \bigwedge_{q=1}^{n} (\neg \dot{F} \ WA_q))$$

and

$$\pi \models \bigwedge_{i=1}^{n} \Box \Diamond A_i \wedge \neg \bigwedge_{i=1}^{m} \Box \Diamond G_i \wedge \bigwedge_{i=1}^{g} \Box \Diamond (G_i \vee \dot{F})$$

Simplifying the expression.

$$\pi \models \bigwedge_{i=1}^{n} \Box \Diamond A_i \wedge \Box \Diamond \dot{F} \wedge \neg \bigwedge_{i=1}^{m} \Box \Diamond G_i$$

By finiteness of the model, the path $\pi$ ends in a cycle $C$. We are going to show that cycle $C$ is actually composed of smaller cycles and that one of the smaller cycles must violate the FMF property. As for every $1 \leq i \leq p$ we have $\pi \models \neg \Box \Diamond try_i$ it must be the case that in $C$ no $fail_i$ occur. By $\pi \models \Box \Diamond \dot{F}$ it must be the case that some fault occurs in $C$. Let $p < r \leq t$ be the minimal such that $fail_r$ appears in $C$. We know that the following holds.

$$C \models \Box \Diamond try_r \wedge \Box \Diamond fail_r \wedge \Box \Diamond (suc_r \wedge \bigwedge_{j=1}^{n} (\neg \dot{F} \ WA_j))$$

The transition on $try_r$ is unique. Hence, the cycle $C$ must have the form

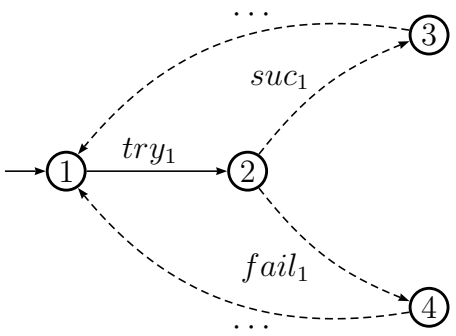



Figure A.1: Topology of a try-response cycle

shown in Figure A.1. Indeed, between every two visits to $try_r$ there must be a visit to either $suc_r$ or $fail_r$ and between a pair of $suc_r$ or $fail_r$ there must be a visit to $try_r$. It follows that there is a smaller cycle $C' \subseteq C$ such that $C'$ does not visit $fail_r$ and it preserves visits to $A_i$ for each $i$.

As $C'$ is a part of $C$ it must be the case that for all $j \leq p$ it still does not include either $try_j$ or $fail_j$. Proceeding by induction, we construct a cycle $C''$ such that $C''$ visits $A_i$ for each $i$, does not visit $fail_j$ for every $j$, and for some $k$ does not visit $G_k$. Hence, as $C''$ is part of the original cycle $C$ it contradicts the correctness of the controller for FMF. □

**Theorem A.2.** *Given an RSGR(1) LTS control problem $\mathcal{R} = \langle \mathcal{E}, T \rangle$ with an environment model $E$ and assumptions $As$, if $As$ is strongly compatible with $E$ according to $T$ then all solutions to $\mathcal{R}$ are strongly best effort controllers.*

*Proof.* Given $\mathcal{E} = \langle E, \{(As, G)\}, A_C \rangle$, where $E = (S, A, \Delta, s_0)$, $G = \bigwedge_{i=1}^{m} \Box \diamondsuit G_i$ and $As = \bigwedge_{i=1}^{n} \Box \diamondsuit A_i$.

Suppose that assumptions $As$ are strongly compatible with $E$ according to $T$.

Let $T = \{(try_i, fail_i, suc_i \mid 0 \leq i \leq t\}$ be the set of try-response triplets, where $t$ is the number triplets in $\mathcal{R}$. Let $\dot{F}$ be the fluent that becomes true whenever a failure occurs, formally defined as $\dot{F} = \langle \{fail_i\}, A \backslash \{fail_i\}, false \rangle$.

Let $M$ be a solution to $\mathcal{R}$.

Suppose by way of contradiction that $M$ is not best effort. Then there is a finite trace $\sigma \in tr(E \| M)$ and for all possible extensions $\sigma'$ of $\sigma$ in $E \| M$ we have either $\sigma.\sigma'$ is not strong independent fair or $\sigma.\sigma'$ satisfies $(\neg \bigwedge_{i=1}^{n} \Box \Diamond A_i)$.

Consider the SGR(1) control problem $\mathcal{R}' = \langle E_s, H', A_C \cup F \rangle$, where $s$ is the last state in $\sigma$ and $H'$, $E_s$, and $F$ are as in Definition 4.6. Consider now the controller $M'$ that is obtained from $M$ by disabling all transitions in $F$, i.e., disabling transitions of the type $fail_i$ in $T$, which are controllable in $\mathcal{R}'$. By Condition 6 of Definition 4.1 in every state in which $fail_i$ is enabled also $suc_i$ is enabled. It follows that $M'$ is a valid controller as in $E_s \| M'$ there are no dead ends.

By assumption, every trace in $E_s \| M'$ is either not strong independent fair or satisfies $(\neg \bigwedge_{i=1}^{n} \Box \Diamond A_i)$. However, recall that $M'$ disables all failures. As every trace in $E_s \| M'$ satisfies $\Box \neg \dot{F}$ and every two occurrences of $try_i$ must have either a $suc_i$ or $fail_i$ between them, it must be the case that every trace

in $E_s \| M'$ satisfies

$$\Box \diamondsuit try_i \Rightarrow \Box \diamondsuit (suc_i \wedge \Box \neg \dot{F}).$$

Hence, every trace in $E_s \| M'$ is in fact strong independent fair.

We conclude that all traces in $E_s \| M'$ satisfy $(\neg \bigwedge_{i=1}^{n} \Box \diamondsuit A_i)$. However, this contradicts assumption compatibility as in this case $M'$ is a solution to the SGR(1) control problem $\mathcal{R}'$, which we assumed has no solution. □

**Theorem A.3.** *Given an RSGR(1) problem $\mathcal{R}$ with an assumption 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 all fair scheduler $s$ of $E_p$ consistent with $M$, the following holds: the measure of the set $B = \{\pi \mid \pi$ is a trace of $E_p$ under scheduler $s$ and $\pi$ matches a trace of $E$ that satisfies assumptions infinitely often but is not strong independent fair in $M \| E\}$ is zero.*

*Proof.* In order to simplify presentation we prove the theorem for controllers with no mixed states. That is, controllable actions are only possible when no environment action is enabled. This is not a severe restriction since controllers are supposed to achieve goals even when races are always solved in favor of the environment. Furthermore, as $E$ is assumption compatible, none of the controllable actions are necessary for the environment to fulfill assumptions. Thus, focussing on this family of controllers does not alter feasibility of control problem. For such an $M$, turns in $E_p$ are always determined by the topology of $M$.

Thus, a scheduler $S$ for $M$ has no decisions to make. Then, we can think about the composition of $M$ and $E_p$ as a Markov chain denoted $M\|E_p$. The analysis is now reduced to measure the traces in $M\|E_p$ that satisfy assumptions infinitely often but are not strong independent fair traces. In what follows we will use $M\|E_p$ and $M\|E$ interchangeably when reasoning on paths and graph structure. Note that this is possible due to the requirement that $E_p{\downarrow}$ is simulation equivalent to $E$ and $E$ is deterministic. In a potential violating trace of $M\|E$ assumptions hold infinitely often and try transitions are performed infinitely often. As $E$ is assumption compatible, we conclude: (a) the trace is eventually generated by a terminal maximally strongly connected component (SCC) and, (b) that SCC exhibits a sequence of transitions that satisfy all assumptions with no failure and a success-labelled transition.

Conclusion (a) follows from the path being infinite, the composition $M\|E_p$ being probabilistic, and the fact that in a probabilistic system every transition that is enabled infinitely often is taken infinitely often. Hence, it must be the case that the SCC is maximal and terminal, i.e., no larger SCC containing it exists, and no other SCC is reachable from it.

Conclusion (b) follows from two observations: (b.1) assumption compatibility implies that, from every state, there is a path that satisfies assumptions without using failures, and (b.2) if that path does not include the success, it can be extended to reach the try transition in the same SCC again. Due to the topological restrictions on try-response tuples, the path necessarily traverses a success or a failure before the re-try. Moreover, since whenever a failure is enabled a success is also enabled, the path can be actually extended

to both satisfy assumptions and success without failures.

It follows that the SCC contains a finite path that does not traverse failures, traverses all assumptions, and visits the appropriate try transition. As before, in a probabilistic system, if a finite path is possible infinitely often, it is taken infinitely often with probability one.

Every infinite path in which the finite path we identified repeats infinitely often is strong independent fair. It follows that the measure of paths that are not strong independent fair is zero.

**Lemma A.1.** (All) *Given an MTS control problem* $\mathcal{E} = \langle E, \varphi, A_C \rangle$ *where* $E = (S, A, \Delta^r, \Delta^p, s_{0_E})$. *If* $E^I$ *is the LTS obtained by applying Definition 5.2 to* $E$, *then the following holds. The answer for* $\mathcal{E}$ *is* all *iff the LTS control problem* $\mathcal{E}_A^I = \langle E^I, \mathcal{X}_{\overline{A}}(\varphi), A_C \rangle$ *is realisable.*

*Proof.* $\Rightarrow$ Assume that the $\mathcal{E}_A^I$is unrealisable.

By Lemma 3.1 there exists an LTS $N = (S_N, A_N, \Delta_N, s_{0_N})$ such that for every possible controller $M$, $E^I \| M \| N \models \neg \mathcal{X}_{\overline{A_N}}(\varphi)$.

In the following we construct from $N$ an implementation $I \in \mathtt{I}^{det}[E]$ such that $I$ cannot be controlled to satisfy $\varphi$. Let $I = (S_I, A, \Delta_I, s_{0_I})$ with the following components. We set $s_{0_I} = (s_{0_E}, s_{0_N})$. Then, $S_I$ and $\Delta_I$ are constructed by induction as follows. Consider a state $(s_E, s_N) \in S_I$. Then, by definition, in $E^I$ the set $A_C \cap \Delta_{E^I}(s_E) = \emptyset$. Indeed, the only actions possible from $s \in S_{E^I}$ are the new actions. By definition there exists $\ell_i$ and $s'_N$ such that $(s_N, \ell_i, s'_N) \in \Delta_N$. It follows that $i \subseteq A$ is a set of actions enabled from $s$ in $E$. Furthermore, for every $\ell \in i$

there is a transition $(s, \ell, s_\ell)$ in $E$ and a transition $((s, i), \ell, s_\ell)$ in $E^I$. Now, for every choice of $i' \subseteq i$ that is possible by some controller for $E^I$ there is transition over an action $\ell$ that is enabled in $N$. In particular, if $i \subseteq A_C$ then for every possible controllable successor of $(s, i)$ there an option in $N$. Otherwise, there is an uncontrollable successor of $(s, i)$ for which $N$ has an option. By definition, there is some $s_{\ell_N}$ such that $(s_N, \ell, s_{\ell_N}) \in \Delta_N$ for every $\ell \in i$, let $s_{\ell_N}$ denote the successor of $s_N$ over $\ell$. Then, for every $\ell \in i$ we add to $I$ the transition $((s, s_N), \ell, (s_\ell, s_{\ell_N}))$. We can show that $E \preceq I$. Consider the refinement relation $H = \{(s, (s, s_N))\}$. By definition of $E^I$ and the construction above, for every state $(s, s_N)$ all the required actions enabled from $s$ are enabled from $(s, s_N)$ and every possible action enabled from $(s, s_N)$ is possible from $s$.

Suppose now that $I$ is controllable. That is, there is a controller $M$ such that $M \| I \models \varphi$. We construct from $M$ a controller $M'$ and a path in $E^I$ that is also a path in $N$. Correspondence between the path in $E^I$ and the path in $M \| I$ shows that the path in $E^I$ must satisfy $\mathcal{X}_{\overline{A}}(\varphi)$. Let $M = (S_M, A, \Delta_M, s_{0_M})$. We start from state $(s_{0_M}, (s_0, s_{0_N}))$ in $M \| I$ and state $s_0$ in $E^I$. Then from a state $(s_M, (s, s_N))$ in $M \| I$ and $s$ in $E^I$ we extend the path as follows. In $E^I$ we chooses a successor $(s, i)$ such that $i$ is exactly the set of transitions from $(s, s_N)$ in $I$. Then, $M$ dictates that either a controllable successor $(s'_M, (s', s'_N))$ of $(s_M, (s, s_N))$ is chosen in $M \| I$ or there is at least one enabled uncontrollable transition from $(s_M, (s, s_N))$. In that case, $N$ dictates that some uncontrollable successor $(s', s'_N)$ of $(s, s_N)$ is chosen in $E^I$. But

then, the successor $(s'_M, (s', s'_N))$ is enabled in $M \| I$. By construction of the two paths the first satisfies $\varphi$ iff the second satisfies $\mathcal{X}_{\overline{A}}(\varphi)$. As the first does not satisfy $\mathcal{X}_{\overline{A}}(\varphi)$ then the second does not satisfy $\mathcal{X}_{\overline{A}}(\varphi)$. This is in contradiction to $I$ being controllable.

$\Leftarrow$ Assume that there is a solution to $\mathcal{E}_A^I$.

Then there exists an LTS controller $M = (S_M, A_M, \Delta_M, s_{0_M})$ such that $E^I \| M \models \mathcal{X}_{\overline{A}}(\varphi)$.

Consider an implementation $I \in \mathtt{I}^{det}[E]$. We construct from $M$ a controller $N$ for $\mathcal{I} = \langle I, \varphi, A_C \rangle$.

As $E \preceq I$ there is a refinement relation $R \subseteq S \times I$. between $E$ and $I$. Based on $M$, we construct $N = (S_N, A_N, \Delta_N, s_{0_N})$ as follows: We first define the alphabet and initial state for $N$: $A_N = A$ and $s_{0_N} = (s_{0_E}, s_{0_M})$. Then, we construct the transition relation $\Delta_N$ and the set of states $S_N$ as follows:

Consider pairs $(s, s_I) \in R$, $(s, s_M) \in E^I \| M$ and $(s_I, (s, s_M)) \in I || N$. Let $i = \Delta_I(s_I)$. By definition of $E^I$, there exists a transition $(s, \ell_i, (s, \ell_i)) \in \Delta_{E^I}$. Since $s$ is uncontrollable in $E^I$, there exists $s'_M \in S_M$ such that $((s, s_M), \ell_i, ((s, \ell_i), s'_M)) \in \Delta_{E^I || M}$.

Let $j = \Delta_M(s'_M)$, i.e., the set of actions enabled from $s'_M$ in $M$. Clearly, $j \neq \emptyset$. For every $\ell \in j$ there is a transition $((s, s_M), \ell, (s', s''_M)) \in \Delta_{E^I \| M}$. For every such $\ell$ we add the state $(s', s''_M)$ to $S_N$. As $j \neq \emptyset$, it follows that $N$ enables at least one transition from $s_I$. Furthermore, by definition of $E^I$ all uncontrollable transitions from $s_I$ are included in $j$.

We now show that $N$ is indeed a solution to $\mathcal{I}$.

We can show that every trace $\pi$ of $I \| N$ corresponds to a trace $\pi'$ of $E^I \| M$. The trace of $E^I \| M$ is obtained from the trace of $I \| N$ by inserting intermediate states that are determined by the set of actions in every state of $I$. As the action trace $\pi'$ satisfies $\mathcal{X}_{\overline{A}}(\varphi)$, it follows that the action trace of $\pi$ must satisfy $\varphi$.

**Lemma A.2.** ***(None)*** *Given an MTS control problem $\mathcal{E} = \langle E, \varphi, A_C \rangle$ where $E = (S, A, \Delta^r, \Delta^p, s_0)$. If $E^I$ is the LTS obtained by applying Definition 5.2 to $E$, then the following holds.*

*There exists a solution for $\mathcal{E}_N^I = \langle E^I, \mathcal{X}_{\overline{A}}(\varphi), A_C \cup \{i_\ell \mid i \subseteq A\} \rangle$ iff there exists $I \in \mathcal{I}^{det}[E]$ such that the there exists a solution to the LTS control problem $\mathcal{I} = \langle I, \varphi, A_C \rangle$*

*Proof.* $\Rightarrow$ Intuitively, to prove this side of the lemma, from a solution to $\mathcal{E}_N^I$, we build both an implementation $I$ and a controller $N$ for the control problem $\mathcal{I}$.

Let $M = (S_M, A_M, \Delta_M, s_{0_M})$ be a solution to $\mathcal{E}_N^I$. We define $I$ as follows:

- $S_I = S_E \cup (S_E \times S_M)$
- $A_I = A$,

– In order to define $\Delta_I$ (and $\Delta_N$ below) we define the following sets.

$$\Delta_1 = \left\{ ((s, s_M), \ell, (s', s''_M)) \;\middle|\; \begin{array}{l} \text{Exists } i \subseteq A \text{ s.t. } \ell \in i, \\ ((s, s_M), \ell_i, ((s, i), s'_M)) \in \Delta_{E^I \| M}, \\ \text{and} (((s, i), s'_M), \ell, (s', s''_M)) \in \Delta_{E^I \| M} \end{array} \right\}$$

$$\Delta_2 = \left\{ ((s, s_M), \ell, s') \;\middle|\; \begin{array}{l} \text{Exists } i \subseteq A \text{s.t.} \ell \in i, ((s, i), \ell, s') \in \Delta_{E^I}, \\ ((s, s_M), \ell_i, ((s, i), s'_M)) \in \Delta_{E^I \| M}, \text{and} \\ (((s, i), s'_M), \ell, (s', s''_M)) \notin \Delta_{E^I \| M} \text{for all} s''_M \end{array} \right\}$$

Then, $\Delta_I = \Delta_E \cup \Delta_1 \cup \Delta_2$

– $s_{0_I} = (s_0, s_{0_M})$.

We define $N$ as follows:

– $S_N = S_E \times S_M$,

– $A_N = A$,

– $\Delta_N = \Delta_1$,

– $s_{0_N} = (s_0, s_{0_M})$,

We have to show that $E \preceq I$, that $N$ is a valid controller for $I$, and that $I \| N \models \varphi$.

To see that $E \preceq I$ we use the refinement relation $H = \{((s, (s, s_M)), (s, s)) \mid s \in S_E \text{ and } s_M \in S_M\}$. Then from every state of the form $(s, s_M)$ the controller $M$ defines set $i$ that includes all required transitions from $s$. Furthermore, every $\ell \in i$ is enabled from $(s, s_M)$, going to either a state of the form $(s', s'_M)$ or a state of the form $s'$. Thus, all required transitions from $s$ are present from $(s, s_M)$. Clearly, all transitions from $(s, s_M)$ are possible from $s$. Also, clearly every state

$s \in E$ refines itself.

We have to show that $N$ enables at least one transition from every state of $I$. This follows from $M$ being a controller of $E^I$ and the composition $E^I \| M$ having no deadlocks.

Finally, every trace of $I \| N$ corresponds to a trace of $E^I \| M$ with additional actions that are not in $A$. Thus, a trace of $I \| N$ satisfies $\varphi$ iff the corresponding trace of $E^I \| M$ satisfies $\mathcal{X}_{\overline{A}}(\varphi)$.

$\Leftarrow$ Suppose that there is an implementation $I \in \mathtt{I}^{det}[E]$ and controller $N$ that solves $\mathcal{I} = \langle I, \varphi, A_C \rangle$. We construct an LTS $M$ that solves $\mathcal{E}_N^I$.

We define $M$ as follows:

- $S_M = S_{I\|N} \cup \{(s_I, \Delta_I(s_I))\}$
- $A_M = A \cup \{\ell_i \mid i \subseteq A\}$,
- $\Delta_M = \{((s_I, s_N), \ell_{\Delta_I(s_I)}, ((s_I, \Delta_I(s_I)), s_N))\} \quad \cup$
$$\left\{ (((s_I, \Delta_I(s_I)), s_N), \ell, (s'_I, s'_N)) \;\middle|\; \begin{array}{r} \ell \in I,\ (s_I, \ell, s'_I) \in \Delta_E, \text{ and} \\ (s_N, \ell, s'_N) \in \Delta_N \end{array} \right\}$$
- $s_{0_M} = (s_{0_I}, s_{0_N})$.

We have to show that $M$ is a valid controller and that $E^I \| M \models \mathcal{X}_{\overline{A}}(\varphi)$. In order to show that $M$ is a valid controller we have to show that it does not restrict uncontrollable transitions and that $E^I \| M$ does not have deadlocks. Let $H \subseteq S_E \times S_I$ be a refinement relation between $E$ and $I$. Consider the initial state $(s_{0_I}, s_{0_N})$. By definition $(s_{0_E}, s_{0_I}) \in H$. Thus, every required transition from $s_{0_E}$ is implemented also from $s_{0_I}$. It follows that $\ell_{\Delta(s_{0_I})}$ is an enabled transition from $s_{0_E}$ in $E^I$. Thus, the state $(s_{0_E}, (s_{0_I}, s_{0_N}))$ is not a deadlock in $E^I \| M$. The action $\ell_{\Delta_I(s_{0_I})}$ leads from $s_{0_E}$ to $(s_{0_E}, \Delta_I(s_{0_I}))$. Now, as

$N$ is a valid controller of $I$, every uncontrollable transition from $s_{0_I}$ is not restricted by $s_{0_N}$. It follows that all uncontrollable transitions from $(s_{0_E}, \Delta_I(s_{0_I}))$ are enabled. Furthermore, consider a transition $\Big(\big((s_{0_E}, \Delta_I(s_{0_I})), ((s_{0_I}, \Delta_I(s_{0_I})), s_{0_N})\big), \ell, \big(s_E, (s_I, s_N)\big)\Big)$ appearing in $\Delta_{E^I \| M}$. Then, by determinism of $E$ and $I$, it must be the case that $(s_E, s_I) \in H$. It follows that we can continue in the same way by induction showing that $M$ does not restrict uncontrollable actions in $E^I$.

In order to show that $E^I \| M \models \mathcal{X}_{\overline{A}}(\varphi)$ we note that a trace in $E^I \| M$ corresponds to a trace in $I \| N$ where the new actions $\ell_i$ for $i \subseteq A$ are factored out. Thus, as every trace of $I \| N$ satisfies $\varphi$ it follows that every trace of $E^I \| M$ satisfies $\mathcal{X}_{\overline{A}}(\varphi)$.

# Bibliography

[AFF+03] Roy Armoni, Limor Fix, Alon Flaisher, Orna Grumberg, Nir Piterman, Andreas Tiemeyer, and Moshe Vardi. Enhanced vacuity detection in linear temporal logic. In Rebecca Wright, editor, *Financial Cryptography*, volume 2742 of *Lecture Notes in Computer Science*, pages 368–380. Springer Berlin / Heidelberg, 2003.

[AITG04] Marco Autili, Paola Inverardi, Massimo Tivoli, and David Garlan. Synthesis of "correct" adaptors for protocol enhancement in component-based systems. In *Specification and Verification of Component-Based Systems*, page 79. ACM, 2004.

[AMPS98] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *Proceedings of the IFAC Symposium on System Structure and Control*, 1998.

[BB94] Derek L. Beatty and Randal E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proceedings*

*of the 31st annual Design Automation Conference*, DAC '94, pages 596–602, New York, NY, USA, 1994. ACM.

[BBDER97] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in actl formulas. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 279–290, London, UK, 1997. Springer-Verlag.

[BBDER01] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18:141–163, March 2001.

[BCP+01] Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. MBP: a model based planner. In *Proceedings of the IJCAI01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[Bel57] R. Bellman. A Markovian decision process. *Journal of Mathematics and Mechanics.*, 6:679–684, 1957.

[BG99] Glenn Bruns and Patrice Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *11th International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 274–287. Springer, 1999.

[BG00] Glenn Bruns and Patrice Godefroid. Generalized model checking: Reasoning about partial state spaces. In *11th Interna-*

*tional Conference on Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2000.

[BGJ+07] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, pages 1188–1193, San Jose, CA, USA, 2007. EDA Consortium.

[BIPT09] Antonia Bertolino, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 141–150, New York, NY, USA, 2009. ACM.

[BL69] J.R. Buchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, pages 295–311, 1969.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, August 1986.

[BSL04] Yves Bontemps, Pierre-Yves Schobbens, and Christof Löding. Synthesis of open reactive systems from scenario-based specifi-

cations. *Fundamenta Informaticae - Application of Concurrency to System Design (ACSD'03)*, 62:139–169, February 2004.

[BT04] B. Boem and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Person Education, 2004.

[CGG07] Marsha Chechik, Mihaela Gheorghiu, and Arie Gurfinkel. Finding environment guarantees. In *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, FASE'07, pages 352–367, Berlin, Heidelberg, 2007. Springer-Verlag.

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 331–346, Berlin, Heidelberg, 2003. Springer-Verlag.

[CHJ08] Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Environment assumptions for synthesis. In *Proceedings of the 19th international conference on Concurrency Theory*, CONCUR '08, pages 147–161, Berlin, Heidelberg, 2008. Springer-Verlag.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international sym-*

*posium on Foundations of software engineering*, ESEC/FSE-9, pages 109–120, New York, NY, USA, 2001. ACM.

[DBPU10] Nicolás Roque D'Ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. Synthesis of live behaviour models. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 77–86, New York, NY, USA, 2010. ACM.

[DBPU11] Nicolás Roque D'Ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. Synthesis of live behavior models for fallible domains. In *to appear in the 33rd International Conference of Software Engineering*, ICSE '11, New York, NY, USA, 2011. ACM.

[DBPU12] Nicolás D'Ippolito, Víctor A. Braberman, Nir Piterman, and Sebastián Uchitel. The modal transition system control problem. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 2012.

[DBPU13] Nicolás D'Ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. Synthesising non-anomalous event-based controllers for liveness goals. *ACM Tran. Softw. Eng. Methodol.*, 22, 2013.

[DFCU08] Nicolas D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastian Uchitel. Mtsa: The modal transition system analyser. In

*Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 475–476, Washington, DC, USA, 2008. IEEE Computer Society.

[DFG+10] Nicolás D'Ippolito, Marcelo F. Frias, Juan P. Galeotti, Esteban Lanzarotti, and Sergio Mera. Alloy+hotcore: A fast approximation to unsat core. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *ASM*, volume 5977 of *Lecture Notes in Computer Science*, pages 160–173. Springer, 2010.

[DGM09] Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. An architecture for requirements-driven self-reconfiguration. In *Proceedings of the 21st International Conference on Advanced Information Systems Engineering*, CAiSE '09, pages 246–260, Berlin, Heidelberg, 2009. Springer-Verlag.

[D'I] Nicolás Roque D'Ippolito. Thesis Resources. http://www.doc.ic.ac.uk/ srdipi/thesis-resources.

[D'I12] Nicolás D'Ippolito. Synthesis of event-based controllers: A software engineering challenge. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *ICSE*, pages 1547–1550. IEEE, 2012.

[DLvL06] Christophe Damas, Bernard Lambeau, and Axel van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software*

*engineering*, SIGSOFT '06/FSE-14, pages 197–207, New York, NY, USA, 2006. ACM.

[EGP08] Michael Emmi, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Assume-guarantee verification for interface automata. In *Proceedings of the 15th international symposium on Formal Methods*, FM '08, pages 116–131, Berlin, Heidelberg, 2008. Springer-Verlag.

[EJ88] E.A. Emerson and C.S. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th Symposium on Foundations of Computer Science*. IEEE, 1988.

[EWS05] Kousha Etessami, Thomas Wilke, and Rebecca A. Schuller. Fair simulation relations, parity games, and state space reduction for büchi automata. *SIAM Journal on Computing*, 34:1159–1175, May 2005.

[FDB+10] Dario Fischbein, Nicolás Roque D'Ippolito, Greg Brunet, Marsha Chechik, and Sebastián Uchitel. Weak alphabet merging of partial behaviour models. In *IEEE Transactions on software engineering and Methodology*, to appear, New York, NY, USA, 2010. ACM.

[Fit91] Melvin Fitting. "Many-Valued Modal Logics". *Fundamenta Informaticae*, 15(3-4):335–350, 1991.

[Fra86] Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[GBMP97] Erann Gat, R. Peter Bonnasso, Robin Murphy, and Aaai Press. On three-layer architectures. In *Artificial Intelligence and Mobile Robots*, pages 195–210. AAAI Press, 1997.

[GC04] A. Gurfinkel and M. Chechik. How vacuous is vacuous? *Tools and Algorithms for the Construction and Analysis of Systems*, pages 451–466, 2004.

[GH82] Yuri Gurevich and Leo Harrington. Trees, automata, and games. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 60–65, New York, NY, USA, 1982. ACM.

[GM03] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 257–266, New York, NY, USA, 2003. ACM.

[GP09] Patrice Godefroid and Nir Piterman. Ltl generalized model checking revisited. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 89–104, Berlin, Heidelberg, 2009. Springer-Verlag.

[GT00] Fausto Giunchiglia and Paolo Traverso. Planning as model checking. In *Proceedings of the 5th European Conference on*

*Planning: Recent Advances in AI Planning*, pages 1–20, London, UK, 2000. Springer-Verlag.

[GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata logics, and infinite games: a guide to current research*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[HGS04] An-Cheng Huang, David Garlan, and Bradley Schmerl. Rainbow: Architecture-based self-adaptation with reusable infrastructure. In *Proceedings of the First International Conference on Autonomic Computing*, pages 276–277, Washington, DC, USA, 2004. IEEE Computer Society.

[HJM03] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Counterexample-guided control. In *30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 886–902. Springer, 2003.

[HKLN12] Keijo Heljanko, Misa Kein?nen, Martin Lange, and Ilkka Niemel?. Solving parity games by a reduction to sat. *Journal of Computer and System Sciences*, 78(2):430 – 440, 2012. ¡ce:title¿Games in Verification¡/ce:title¿.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, August 1978.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[HSMK09a] William Heaven, Daniel Sykes, Jeff Magee, and Jeff Kramer. A Case Study in Goal-Driven Architectural Adaptation. *Software Engineering for Self-Adaptive Systems*, 2009.

[HSMK09b] William Heaven, Daniel Sykes, Jeff Magee, and Jeff Kramer. Software engineering for self-adaptive systems. chapter A Case Study in Goal-Driven Architectural Adaptation, pages 109–127. Springer-Verlag, Berlin, Heidelberg, 2009.

[IT07] Paola Inverardi and Massimo Tivoli. A reuse-based approach to the correct and automatic composition of web-services. In *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*, ESSPE '07, pages 29–33, New York, NY, USA, 2007. ACM.

[Jac95a] Michael Jackson. *Software requirements & specifications: a lexicon of practice, principles and prejudices*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[Jac95b] Michael Jackson. The world and the machine. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 283–292, New York, NY, USA, 1995. ACM.

[JP06] Sudeep Juvekar and Nir Piterman. Minimizing generalized bchi automata. In Thomas Ball and Robert Jones, editors, *Proceed-*

*ings 18th International Conference on Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 45–58. Springer Berlin / Heidelberg, 2006.

[Jur00] Marcin Jurdziński. Small progress measures for solving parity games. In Horst Reichel and Sophie Tison, editors, *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301, Lille, France, February 2000. Springer-Verlag.

[Kec95] AS Kechris. *Classical Descriptive Set Theory*, volume 156. Springer, 1995.

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19:371–384, July 1976.

[KHB09] R. Konighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 152–159. IEEE, 2009.

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*, FOSE '07, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[KPP05] Yonit Kesten, Nir Piterman, and Amir Pnueli. Bridging the gap between fair simulation and trace inclusion. *Information and Computation*, 200:35–61, July 2005.

[KPR04] Raman Kazhamiakin, Marco Pistore, and Marco Roveri. Formal verification of requirements using spin: A case study on web services. In *Proceedings of the Software Engineering and Formal Methods, Second International Conference*, SEFM '04, pages 406–415, Washington, DC, USA, 2004. IEEE Computer Society.

[Kri59] S.A. Kripke. A completeness theorem in modal logic. *The Journal of Symbolic Logic*, 24(1):1–14, 1959.

[KV03] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):224–233, 2003.

[KV05] Orna Kupferman and Moshe Y. Vardi. Safraless decision procedures. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '05, pages 531–542, Washington, DC, USA, 2005. IEEE Computer Society.

[Lam01] Axel Van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, volume 0, page 0249, Los Alamitos, CA, USA, 2001. IEEE Computer Society Washington, DC, USA, IEEE Computer Society.

[le2]

[LKMU05] Emmanuel Letier, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Fluent temporal logic for discrete-time event-based models. *SIGSOFT Softw. Eng. Notes*, 30(5):70–79, September 2005.

[LKMU08] Emmanuel Letier, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *Automated Software Engineering*, 15:175–206, June 2008.

[LL95] Claus Lewerentz and Thomas Lindner, editors. *Formal Development of Reactive Systems - Case Study Production Cell*, volume 891 of *Lecture Notes in Computer Science*, London, UK, 1995. Springer-Verlag.

[LT88] K.G. Larsen and B. Thomsen. "A Modal Process Logic". In *Proceedings of 3rd Annual Symposium on Logic in Computer Science (LICS'88)*, pages 203–210. IEEE Computer Society Press, 1988.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. volume 2, pages 219–246, 1989.

[LvL02] Emmanuel Letier and Axel van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 83–93, New York, NY, USA, 2002. ACM.

[LX90] Kim Guldstrand Larsen and Liu Xinxin. Equation solving using modal transition systems. In *LICS*, pages 108–117. IEEE Computer Society, 1990.

[Mar75] D.A. Martin. Borel determinacy. *Annals of Mathematics*, pages 363–371, 1975.

[MK06] J. Magee and J. Kramer. *Concurrency: state models & Java programs*. Wiley New York, 2006.

[MKG97] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Analysing the behaviour of distributed software architectures: a case study. In *FTDCS*, pages 240–247. IEEE Computer Society, 1997.

[MNP07] Oded Maler, Dejan Nickovic, and Amir Pnueli. On synthesizing controllers from bounded-response properties. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 95–107, Berlin, Heidelberg, 2007. Springer-Verlag.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[MPS95] Oded Maler, Amir Pnueli, and Josef Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*, volume 95, pages 229–242. Springer, 1995.

[PBB+04] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. In Christoph Bussler and Dieter Fensel, editors, *Artificial Intelligence: Methodology, Systems, and Applications*, volume 3192 of *Lecture Notes in Computer Science*, pages 106–115. Springer Berlin, 2004. 10.1007/978-3-540-30106-6_11.

[PM95] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41 – 61, 1995.

[PP06] N. Piterman and A. Pnueli. Faster solutions of Rabin and Streett games. In *Logic in Computer Science, 2006 21st Annual IEEE Symposium on*, pages 275–284. IEEE, 2006.

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive (1) designs. *Lecture notes in computer science*, 3855:364–380, 2006.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.

[Rab70] M.O. Rabin. Weakly definable relations and special automata. In *Proc. Symp. Math. Logic and Foundations of Set Theory*, pages 1–23, 1970.

[RCDH07] Jean-François Raskin, Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Algorithms for omega-regular games with imperfect information. *Logical Methods in Computer Science*, 3(3), 2007.

[RN95] S. Russell and P. Norvig. Artificial intelligence: a modern approach. *New Jersey*, 1995.

[RW89] P.J.G Ramadge and W.M Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[SF07] Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In *Proceedings of the 5th international conference on Automated technology for verification and analysis*, ATVA'07, pages 474–488, Berlin, Heidelberg, 2007. Springer-Verlag.

[SHMK07] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Plan-directed architectural change for autonomous systems. In Arnd Poetzsch-Heffter, editor, *SAVCBS*, pages 15–21. ACM, 2007.

[SSR08] S. Sohail, F. Somenzi, and K. Ravi. A hybrid algorithm for ltl games. In *9th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 4905 of *Lecture Notes in Computer Science*, pages 309–323. Springer, 2008.

[Ste98] Perdita Stevens. Abstract games for infinite state processes. In *9th International Conference on Concurrency Theory*, vol-

ume 1466 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 1998.

[UBC09a] Sebastian Uchitel, Greg Brunet, and Marsha Chechik. Synthesis of partial behavior models from properties and scenarios. *IEEE Transactions on Software Engineering*, 35:384–406, May 2009.

[UBC09b] Sebastián Uchitel, Greg Brunet, and Marsha Chechik. Synthesis of partial behavior models from properties and scenarios. *IEEE Trans. Software Eng.*, 35(3):384–406, 2009.

[UKM04] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13(1):37–85, 2004.

[vL09] Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[vLL00] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, 26:978–1005, October 2000.

[YD06] Michal Young and Premkumar T. Devanbu, editors. *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2005, Portland, Oregon, USA, November 5-11, 2006*. ACM, 2006.

[ZJ97] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6:1–30, January 1997.