# Minimising Makespan of Discrete Controllers: A Qualitative Approach

by

**Ezequiel Gustavo Castellano**

## Dissertation

submitted to the Department of Informatics

in partial fulfillment of the requirements for the degree of

***Doctor of Philosophy***


The Graduate University for Advanced Studies, SOKENDAI

March 2020

# Acknowledgments

I would like to thank my advisor Prof. Nobukazu Yoshioka, as well as my subadvisors Prof. Shin Nakajima and Prof. Hiroyuki Kato of the Graduate University of Advanced Studies. I would also like to acknowledge the examiners Prof. Kenji Tei, Prof. Nicolás D'Ippolito and Prof. Fuyuki Ishikawa. I am gratefully indebted to them for their very valuable comments on this thesis.

I would also like to extend my deepest gratitude to Prof. Victor Braberman and Prof. Sebastián Uchitel of the University of Buenos Aires. Their ideas were the inspiration for my research direction. In particular, I would like to thank Victor for keep following my research closely and always giving me great advice.

I am also grateful to the National Institute of Informatics and the Japanese government for honoring me with a scholarship that made my studies possible. I would also like to extend this thanks to the University of Buenos Aires and the government of Argentina, who provided me with strong background to do my studies abroad through free education. Free education does not guarantee same opportunities for everyone, but it is one step forward to equality.

Huge thanks should also go to all my friends and people I met in the last years. Friends are always there to listen when you are having hard times, to celebrate victories, or to just be there. My most sincere thanks go to my friends from *El Pelle*, UBA courses, UBA futsal, LaFHIS, TIEC, NII and many other life friends. I would like to list each of them individually, but I do not want the acknowledgments to be longer than the rest of this thesis.

Special thanks to my family for providing me with unfailing support and continuous encouragement throughout my entire life. My deepest gratitude to *mi mamá* Silvia, *mi papá* Rafa, *mi hermana* Cyntia y *mi abuela* Dora. Through life,

the environment changes yourself, but your familiy are the ones that give you the basis to move through that environment.

Last but not least, I would also like to thank Linguang, *el amor de mi vida*, for always being next to me supporting me at times of stress, but most importantly for joining me in this adventure called life. This thesis would not be possible without her support.

Ezequiel Castellano

P.S.: Thanks to you that you are reading me now. Shall I say sorry instead?

# Committee

Advisor Dr. YOSHIOKA Nobukazu
Professor of National Institute of Informatics/SOKENDAI

Subadvisor Dr. NAKAJIMA Shin
Professor of National Institute of Informatics/SOKENDAI

Subadvisor Dr. KATO Hiroyuki
Professor of National Institute of Informatics/SOKENDAI

Examiner Dr. TEI Kenji
Professor of Waseda University

Examiner Dr. ISHIKAWA Fuyuki
Professor of National Institute of Informatics

Examiner Dr. D'IPPOLITO Nicolás
Professor of University of Buenos Aires

# Contents

# List of Figures

# Notation in Figures

$\longrightarrow$ transition or controllable transition (in a control problem setting)

$\dashrightarrow$ uncontrollable transition

$\xrightarrow{a}$ transition with action $a$

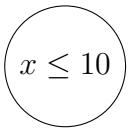
$\xrightarrow[a]{[x==10]}$ transition with action $a$ and guard $[x == 10]$

$\xrightarrow[\{x\}]{}$ transition with reset clocks $\{x\}$

 state or transient states (when non-transient states are distinguished)

 non-transient state

 final state or goal state (in a control problem setting)



state with invariant $x \leq 10$

# List of Tables

# List of Algorithms

# Abstract

Qualitative controller synthesis techniques produce controllers that guarantee to achieve a given goal in the presence of an adversarial environment. However, qualitative synthesis only produces one controller out of many possible solutions and typically does not provide support for expressing preferences over other alternatives.

Synthesis and planning techniques that allow expressing preferences exist, such as those regarding performance or reliability. Such quality attributes are modelled by introducing a quantitative aspect to the system specification, which imposes a preference order on the controllers that satisfy the qualitative part of the specification. However, from a practical perspective, these approaches require modelling quality attributes quantitatively, whereas in many cases, such detailed representation is not available, possible, or desired.

The main objective of this thesis is to present a formal approach to reason about preferences qualitatively, restricting attention to makespan of discrete event-based controllers for safety and reachability goals. Time is reasoned upon symbolically, which relieves the user from providing concrete quantitative measures. In particular, we study the scenario in which durations of individual activities are not known up-front.

First, we show how controllers can be symbolically and fairly compared by fixing the contingencies. Then, we present an algorithm to produce controllers that are makespan-minimising. The algorithm was implemented in the MTSA tool, as well as evaluated in case studies.

**Keywords:** Discrete Event Systems, Reactive Synthesis, Supervisory Control, Reachability, Makespan.

# 1 Introduction

## 1.1 Formal methods in software engineering

Specifying the requirements of the system is one of the first steps in the process of constructing a system [1]. In this process, the assumptions about the environment in which the system will be deployed are defined, as well as the guarantees that the system will provide. In software engineering, the requirements are divided into functional requirements and non-functional requirements. Functional requirements are intended to capture the behaviour that the system must have, while non-functional requirements describe quality attributes of the system, which define a preference order among possible solutions.

Modelling is a key part in the process of building a system. Models are used to build abstractions of the environment based on our assumptions and understanding of the world. These models are used to represent the behaviour of the system that we intend to build. Besides, models are useful to validate those abstractions and assumptions with relevant stakeholders and engineers. In particular, we are

interested in using mathematical models to describe the behaviour of both the environment and the system, because this enables the possibility of formally checking if our system is correct with respect to the specification. In addition, this type of models can be executed by a machine, reducing the gap between models and what it is executed.

In a world with constantly increasing interaction between systems and humans, the attention to techniques that can ensure the satisfaction of the requirements is rising. For instance, Model checking [2] is becoming a key instrument in critical systems such as flying control systems [3, 4], industrial robot systems [5], autonomous driving [6, 7], and medical systems [8]. Model checking is a technique that allows us to verify that our system satisfies the requirements if we specify them in a formal language, which means that we can automatically check that our system is correct according to the specification.

In this thesis, we are interested in modelling problems that involve reactive systems. Reactive systems [9] are computing systems which are interactive in nature. Their interaction could be with other computing systems and also with human beings. From the reactive system point of view, all these are external interactions with its environment. Reactive systems are among the hardest computing systems to program [10], because this type of systems needs to be prepared to react to any possible interaction with the environment that is defined by its interface.

Temporal logic [56] is a formalism that has shown to produce good results in specifying functional requirements of reactive systems [10]. By using this technique, we can specify the requirements in a formal language that clearly distinguishes the assumptions and guarantees of our system. One of the advantages of specifying our system in a formal language like temporal logic is that we can directly apply model checking techniques.

Specifying requirements in a formal language like temporal logic also enables a more ambitious goal, which is building solutions from the specification automatically. Given a specification in terms of assumptions and guarantees, discrete controller synthesis is a technique that can produce solutions that are correct by construction regarding the specification [11, 12]. General synthesis from temporal logic is known to be 2EXPTIME-complete [11]. However, there are polynomial algorithms for a

subset of this logic. For instance, there are applications of synthesis to robotic problems [13, 14, 15] that are specified with GR(1) formulas [16], which are a subset of linear temporal logic formulas that have polynomial synthesis algorithms.

Controllers can be synthesised offline and executed at run-time. This means that synthesis time is not a critical aspect, but termination is. Once a controller is synthesised, it is expected to be executed a large number of times. It will only be necessary to synthesise a new controller when changes in the environment or requirements arise. In these cases, updating controller techniques could be applied [17], which, however, is not within the scope of this work.

## 1.2 Qualitative controller synthesis

The problem of automatically synthesising event-based solutions from environment models and qualitative goal specifications has been widely studied [18, 11, 19, 16]. In these problems, the environment and the goals are specified by using a formal language. The environment is typically modelled as a state machine whose actions are partitioned into controllable and uncontrollable actions. The controller synthesis problem is to automatically produce a solution, i.e., a controller, that by only disabling controllable actions guarantees the satisfaction of the goals. In particular, we focus on reachability and safety goals which are of interest to supervisory control theory [18], conformant [20] and contingent [21] planning.

Qualitative control problems are boolean in the sense that a controller satisfies a set of goals, or it does not. When a qualitative control problem has a solution, we say it is realisable. Realisable control problems may allow for several possible solutions. Different solutions may differ in the strategy which they apply to satisfy the goals. Typically, based on the arrival of monitored actions, the strategies implemented by controllers decide which and when to start activities. For instance, regarding end-to-end makespan, a controller that starts several activities concurrently instead of executing them sequentially can be, intuitively, considered as a better strategy, no matter which the durations of the activities are. Unfortunately, qualitative synthesis procedures are, so far, oblivious to such considerations. The controller produced is one of the many alternatives and users cannot specify their preferences; e.g., lower-makespan controller. Thus, it is desired

to have the ability to express preferences and automatically compute a solution from a set of possible solutions to a control problem accordingly.

Synthesis and planning techniques that allow expressing preferences exist, such as those regarding performance or reliability. Such quality attributes are modelled by introducing a quantitative aspect to the system specification, which imposes a preference order on the controllers that satisfy the qualitative part of the specification [22, 23, 24]. However, from a practical perspective, these approaches require modelling quality attributes quantitatively, whereas in many cases, such detailed representation is not available, possible, or desired.

## 1.3 Proposed approach

In this thesis, we define lower-makespan as our preference and introduce a formal approach to qualitatively reason about makespan of discrete event-based controllers for safety and reachability goals. To reason qualitatively about makespan of controllers, we introduce a symbolic time metric derived from Parametric Timed Automata (PTA) [25] semantics. This metric requires modelling sub-tasks of the problem which take time as activities, but no quantitative information about the duration of the activities is required. Then, we define a mechanism to compare makespan of controllers under unknown durations of activities and event contingencies produced by uncontrollable behaviour. Such a comparison is made through exhaustive analysis by using a symbolic computation over the parameters of a PTA [26] and Satisfiability Modulo Theories (SMT) solving [27]). The parameters of the PTA represent the uncertain duration of the activities. Then, we define makespan-minimising controllers by using the symbolic comparison and we introduce an algorithm that produces a makespan-minimising controller qualitatively. The algorithm is implemented in the MTSA tool [28] and evaluated in case studies. The evaluation consists of comparing the output produced by our algorithm against the standard synthesis algorithm of the tool.

## 1.4 Objectives

The main objective is to present a formal approach to reason about preferences qualitatively, restricting attention to makespan of discrete event-based controllers for safety and reachability goals. We aim to provide a framework in which time is reasoned symbolically, which relieves the user from providing concrete quantitative measures. In particular, we study the scenario in which durations of individual activities are not known up-front. Our hypothesis is that it is possible to i) define a qualitative framework to compare controllers qualitatively regarding their makespan, ii) specify preferences in control problems qualitatively, and iii) produce controllers that reflect those preferences.

The intended output of this thesis is a qualitative framework to compare of makespan of controllers, and a qualitative algorithm that produces controllers with lower-makespan than the ones produced by standard qualitative synthesis algorithms. To validate our approach we provide formal definitions and proofs to show why reasoning about preferences by using our formalisation is valid. Besides, we implement our ideas in the MTSA tool [28], a qualitative synthesis tool, and compare the controllers produced by our algorithm against the ones produced by the tool.

## 1.5 Outline

This thesis is structured as follows. Chapter 2 surveys the main related areas. Chapter 3 introduces the background. Chapter 4 illustrates an example that is used throughout the rest of the thesis. Chapter 5 presents the general problem and the approach of this work. Chapter 6 introduces the control problem with activities, and it models the example as a control problem. Chapter 7 defines how to compare the makespan of a pair of discrete controllers qualitatively. Chapter 8 defines makespan-minimising controllers and presents an algorithm that produces a makespan-minimising controller. Chapter 9 shows the experimental results of the implementation of the algorithm in case studies. Chapter 10 discusses about the related work. Chapter 11 presents the conclusions, limitations of the proposed approach and future directions.

# 2 Survey of Related Areas

The problem of automatically producing solutions from formal specification has a long history. The first formulation of the problem was done by Church [29, 30] in the context of logical circuits. Church introduced the problem of synthesizing digital circuits, which have inputs and outputs signals, from logical specifications. Different communities proposed variations of this problem, as well as formulations and approaches that aim to solve the problem of automatically producing solutions from formal specification. Some of the research areas that tackled a similar problem are *supervisory control* [18] in control engineering community, *reactive synthesis* [11] in computer science community and *automated planning* [31] in artificial intelligence community. In the recent years, there are several works that are attempting to connect all these approaches to bring the knowledge gained in one community to the others [12, 32, 33, 34, 35, 36, 37, 38]. There are authors that are proposing unified views of some forms of automated planning and reactive synthesis [33] because of the similarities between these areas. Some examples of knowledge exchange between these communities are the use of planning heuristics

to obtain solutions in the supervisory control community [39], and the use of LTL to model temporally extended goals in the planning community [40, 41, 42].

## 2.1 Supervisory control

Supervisory control [18] problems of discrete-event systems are defined as a plant, a set of controllable actions and a set of marked states. The plant is typically modelled as a deterministic finite-state automata. As defined in this thesis, the actions of the automata are partitioned into controllable and uncontrollable actions. The marked states are a subset of states of the automata, which are the acceptance ones. The solution to this problem is a supervisor, which must guarantee achieving the marked states by disabling controllable actions. A supervisor is defined as a function that enables a subset of events based on the history of events that have happened so far. The subset of actions must include all the uncontrollable actions because the supervisor has no control over them. The supervisor and the plant form a closed-loop system, in which the supervisor chooses the control actions from the possible events enabled by the plant. Figure 2.1 illustrates the control-loop system. Then, the plant executes one of the actions enabled by the supervisor. However, the supervisor has no control over this choice.

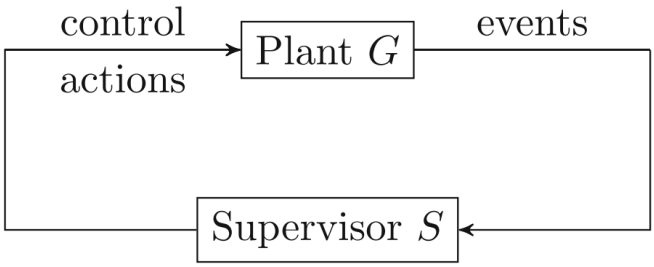



Figure 2.1: Control-loop system of a plant and its supervisor [12].

A supervisor is *non-blocking* if it can reach a marked state from any reachable state of the closed-loop system formed by the supervisor and the plant [12]. This means that the control-loop system never reaches a deadlock, unless it is a marked state, or never reaches a livelock that does not contain marked states.

A typical requirement for supervisors is maximal permissiveness. This means that a supervisor is expected to disturb the plant as little as possible. A unique maximally-permissive supervisor always exist, which is not generally the case in the reactive synthesis framework [12].

The problems presented in this work are similar to the ones of supervisory control when the plant is fully observable. In fully observable problems, the supervisor can observe all the events that happened and determine the current event of the plant uniquely, because the plant is deterministic.

## 2.2 Reactive synthesis

The reactive synthesis problem was introduced by Pnueli and Rosner in 1989 [11] by using linear temporal logic (LTL) [43] as the specification language for both the environment and the system goals. The propositions used to describe those formulas are partitioned into two disjoint sets: the input (uncontrollable) propositions and the output (controllable) propositions. A solution to this problem is a controller, which is an operational specification of a module that restricts the traces allowed in the specification of the environment to those traces satisfying the system goals. Controllers are open dynamical systems in the sense that its behaviour depends on the inputs that it receives. A reactive system has to satisfy a specification for every possible input to the system. The specification of reactive synthesis problems is considered to be more declarative, because it describes the desired behaviour of the controller on the interface level without setting requirements about the internal structure of the solution [12]. Figure 2.2 illustrates an example of a typical controller with one input and two outputs.

Classical definition of reactive synthesis does not have a notion of plant or environment modelled as state machine [44, 11]. However, in the controller synthesis community, there are variations of the original formulation that use plants to describe the behaviour of the environment [45, 46, 47, 48]. In these works, the interaction between the controller and the plant (environment) is done through events as in supervisory control problems. This is also the approach taken in this thesis. One advantage of modelling the environment as an automata is to reduce the size of the formulas used in the specification. This is important because

translating an LTL formula into an automata is at least exponential in the size of the formula [12, 11].

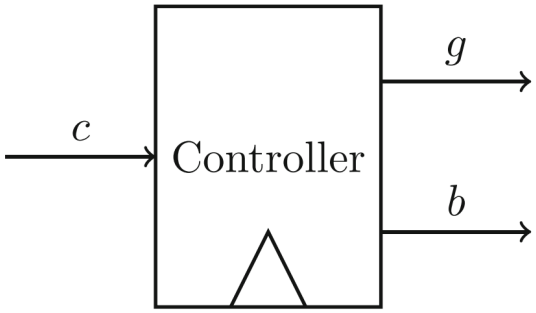



Figure 2.2: The interface of a controller for a coffee maker [12]. Input signal $c$ is the button to order coffee. Output signals are grinding $g$ and brewing $b$.

Solving a reactive synthesis problem typically involves transformations from LTL formulas into some type of word automaton such as Büchi automaton [12]. Then, the automaton is usually translated into a game between two players: the environment and the system. In this game, the system has to be able to win the game for every possible play of the environment [48]. If this is possible, there is a winning strategy, the problem is realisable and there is a controller.

The general problem of synthesis from LTL specifications was shown to be 2EXPTIME complete [11]. Thus, the reactive synthesis community focused on exploring subsets of LTL formulas that are expressive enough to describe interesting problems. For instance, GR(1) formulas are one of the most expressive and widespread formulas with polynomial synthesis algorithms [16].

## 2.3 Automated planning

Planning problems are typically defined as a finite set of propositions that can be modified by the use of actions [31]. Actions have preconditions and effects. The preconditions are the values that the propositions must hold to apply an action, while the effects define how the values of the propositions change. The goal is to generate a sequence of actions that modifies the propositions from their initial values to the goal values, which are called the goal propositions. This can also be

represented as a set of states, actions, and transitions, where each state represents a different valuation of the propositions. In this representation, the goal states are those that represent the values of the goal propositions. A plan, the solution, is a sequence of actions leading from the initial state to a goal state. Planning domains and problems are sometimes described in PDDL (planning domain definition languange) [49].

In classical planning problems, the effects of the actions are deterministic, which means that all the actions are controllable. The reactivity of the environment is typically not modelled, and in case of differences between the plan and the execution, the most common approach is to re-plan, i.e., to produce a new plan. Thus, the focus of this area is to produce efficient and optimal algorithms to generate plans. Efficiency is measured in time to produce the solution, and optimality typically refers to the number of actions of the plan. More compact plans are preferable. In this type of problems, heuristic search [50] is the approach that is often used to find a solution.

The use of temporal logics to specify problems is not limited to the reactive synthesis community. The planning community has also introduced the use of LTL to model temporally extended goals [51] for planning domains that have actions with both deterministic and non-deterministic effects. Most of the approaches propose to compile LTL specifications into classical planning [41, 42], while others propose specific planners for problems with LTL goals [40].

Planning domains with non-deterministic actions exist [52, 53] but with less attention than its deterministic counterpart. Fully Observable Non-Deterministic (FOND) Planning [54] models non-determinism by defining a set of propositions whose value can change over time regarding actions with non-deterministic effects. This makes FOND problems similar to reactive synthesis problems because the exact plan that will be executed is not known until execution time. Thus, FOND plans have to consider all possible contingencies that may appear during execution. Strong plans are those that guarantee that the goal is achieved regardless of contingencies. In the recent years, it has been shown that it is possible to transform reactive synthesis problems with reachability and safety goals into FOND planning problems [55].

# 3 Background

In this chapter we present the key underlying theories that are necessary to explain the rest of this work. Section 3.1 briefly introduces the concept of reactive systems. Section 3.2 and Section 3.3 describe the formalisms that are used to define control problems in Section 3.4. Section 3.5 and Section 3.6 presents the formalisms that are used to define the comparison between two LTS controllers.

## 3.1 Event-based reactive systems

Reactive systems [9] are machines that interact with their environment through sensors and actuators. This type of systems can modify the environment through their actions and perceive the changes through their sensors. The interaction between a reactive system and the environment is modelled by events (or actions). *Act* is a set of symbols that defines the set of observable actions. Some of these actions are executed by the machine, while other actions are events that the machine observes to decide its next actions. Both the environment and the machine

(the controller) will be modelled by using Labelled Transition Systems, which are defined in the next section.

## 3.2 Labeled transition systems

Labelled Transition System (LTS) [56] are one of the formalisms that is typically used for modelling the behaviour of a system. LTS is a transition system in which states are connected through transitions, which are labelled with actions. By using LTS for modelling behaviour, it is possible to decompose the behaviour of the system into several components. The interaction between these components (LTSs) is modelled by the parallel composition [56] of the LTSs. The parallel composition is defined as an LTS that models the asynchronous execution of composed models, interleaving non-shared actions while forcing synchronisation on shared actions. We use $q \xrightarrow{\ell} q'$ to denote a transition that $(q, \ell, q') \in \Delta$. Figure 3.1 illustrates two different LTSs and the parallel composition of them.

**Definition 3.1 (Labelled Transition Systems)** *A* labelled transition system *(LTS) is a tuple* $(Q, \Sigma, \Delta, q_0)$ *where*

- $Q$ *is a finite set of states,*
- $\Sigma \subseteq Act$ *is its alphabet, Act is the set of observable actions,*
- $\Delta \subseteq (Q \times \Sigma \times Q)$ *is a transition relation, and*
- $q_0 \in Q$ *is the initial state.*

**Definition 3.2 (Parallel Composition)** *The parallel composition* ($\|$) *of two LTSs* $M$ *and* $N$*, is a symmetric operator such that* $M \| N = (Q_M \times Q_N, \Sigma_M \cup \Sigma_N, \Delta_{M\|N}, (q_{M_0}, q_{N_0}))$*, where* $\Delta_{M\|N}$ *is the smallest relation that satisfies the following rules:*

$$\frac{q_M \xrightarrow{\ell}_M q_M'}{(q_M,q_N) \xrightarrow{\ell}_{M\|N} (q_M',q_N)}\ \ell \in \Sigma_M \backslash \Sigma_N \qquad \frac{q_N \xrightarrow{\ell}_N q_N'}{(q_M,q_N) \xrightarrow{\ell}_{M\|N} (q_M,q_N')}\ \ell \in \Sigma_N \backslash \Sigma_M$$

$$\frac{q_M \xrightarrow{\ell}_M q_M',\ q_N \xrightarrow{\ell}_N q_N'}{(q_M,q_N) \xrightarrow{\ell}_{M\|N} (q_M',q_N')}\ \ell \in \Sigma_M \cap \Sigma_N$$

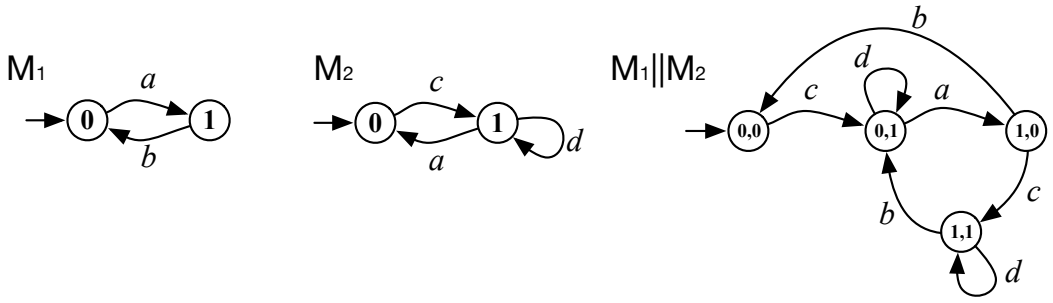



Figure 3.1: Generic example of parallel composition of two LTS.

The *paths* and *traces* of an LTS are used to describe possible behaviour during the execution. A path of an LTS is a possibly infinite sequence of states and actions, i.e., a valid sequence of transitions in the LTS. A trace is a sequence of actions that can be executed by the LTS. That is, that there exists a corresponding path in the LTS. We denote the set of infinite traces of an LTS $M$ as $Tr(M)$. The satisfaction of a formula in the LTS will be determined by its traces. The details will be explained in the next section. We also refer to the possibly infinite set of paths of an LTS that reach a state $q$ from the initial state as $Paths(M, q)$. For example: $q_0 \xrightarrow{a} q_1 \in Paths(M_1, q_1)$ and $q_0 \xrightarrow{c} q_1 \xrightarrow{d} q_1 \in Paths(M_2, q_1)$. Note that $q_{0,0} \xrightarrow{a} q_{0,1} \notin Paths(M_1 \| M_2, q_{0,1})$ because the parallel composition synchronizes on the shared action $a$.

**Definition 3.3 (Path)** *A* path $q_0 \xrightarrow{\ell_0} q_1 \xrightarrow{\ell_1} \ldots$ *of an LTS* $M = (Q, \Sigma, \Delta, q_0)$ *is a sequence of transitions of an LTS* $M$.

**Definition 3.4 (Trace)** *A* trace *of an LTS* $M = (Q, \Sigma, \Delta, q_0)$ *is a sequence of actions* $\pi = \ell_0, \ell_1, \ldots,$ *iff there exists a path* $q_0 \xrightarrow{\ell_0} q_1 \xrightarrow{\ell_1} \ldots$ *in the LTS* $M$.

LTS models can be deterministic or non-deterministic. A model is *deterministic* when given a state and an action of an LTS there is at most one successor. This means that given a trace there is a unique corresponding path. In contrast, if an LTS is *non-deterministic*, given a trace there may be multiple corresponding paths, because for a given action a state may have more than one successor. This work focuses only on deterministic models. Figure 3.2 illustrates some examples of deterministic and non-deterministic LTSs.

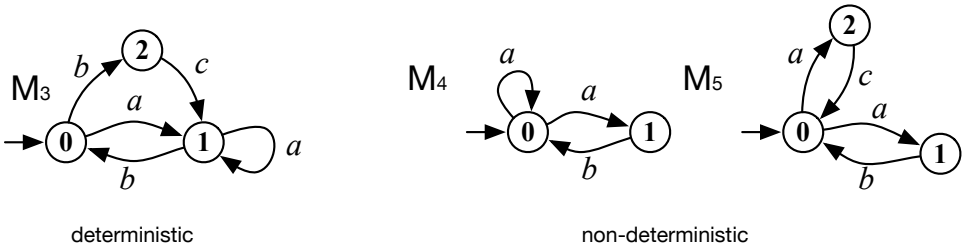



Figure 3.2: Examples of deterministic and non-deterministic LTSs.

**Definition 3.5 (Deterministic)** *An LTS* $(Q, \Sigma, \Delta, q_0)$ *is deterministic when* $\forall q \in Q\ \forall \ell \in \Sigma\ \{(q, \ell, q') \mid (q, \ell, q') \in \Delta\} \leq 1$.

We overload the symbol of the transition relation to define the set *enabled actions* in a state as $\Delta(q) = \{\ell \mid q \xrightarrow{\ell} q'\}$, and the *successors* of a state as $\Delta(q, \ell) = \{q' \mid q \xrightarrow{\ell} q'\}$. For example, in the LTS $M_3$, $\Delta(q_0) = \{a, b\}$ and $\Delta(q_0, a) = \{q_1\}$. Besides, when members of a tuple are not explicitly described we assume them to be indexed by the name of the tuple. For instance, given an LTS M we refer to its members as $M = (Q_M, \Sigma_M, \Delta_M, q_{M_0})$ with $q_M$ a state of $Q_M$.

## 3.3 Fluent linear temporal logic

Temporal logic [56] is one of the most commonly used formalism to express properties about correctness of the execution of a system. This logic extends propositional logic with modalities that permit to describe infinite behaviour that a system must have. Temporal logics can be linear or branching, depending on how time is modelled. From a linear-time perspective there is only one possible future, while from a branching-time perspective multiple alternative futures may exist at the same time. That is to say, branching logic can describe properties that may only hold in some futures, while linear logic describes properties that must hold in every possible future. This work uses an extension of Linear Temporal Logic (LTL) [56], which is the logic that it is used to describe linear-time properties.

Fluent Linear Temporal Logic (FLTL) [57] is the language that we use for describing properties. FLTL is a linear-time temporal logic for reasoning about fluents instead of state-based propositions. 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 $\dot{\ell} = \langle \ell, Act \setminus \{\ell\}, false \rangle$. The logic has the same expressiveness as standard LTL. However, as fluents can be used to overlay state-based propositions on an event-based model, FLTL allows to represent properties in a more compact manner.

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

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

where $Fl \in \mathcal{F}$, and $\psi$ and $\varphi$ are formulas FLTL. As well, we use the standard operators $\wedge$, $\Diamond$ (eventually), $\Box$ (always), and **W** (weak until). FLTL formula satisfaction is standard and it is computed over traces and fluents.

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:

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

In other words, a fluent holds at position $i$ if and only if it initially holds or some initiating action has occurred, but no terminating action has yet occurred. We denote that a possibly infinite trace $\pi$ satisfies an FLTL formula $\varphi$ for every position $i$ as $\pi \models \varphi$.

## 3.4 LTS control synthesis

The controller synthesis problem is the problem of automatically producing a solution from a formal specification. The solution to this problem is called a controller. The LTS control problem is an event-based control problem following the world-machine model [58]. In LTS control problems, the formal specification of the problem is decomposed into a description of the environment and a set of

functional goals [59]. The behaviour of the environment is described as an LTS (or the composition of several LTSs), and the goals are expressed as FLTL formulas. The environment describes when different actions can occur. The controller can decide when to execute some of these actions, but cannot control when the other actions happen. Thus, the actions $\Sigma$ of the environment are partitioned into *controllable actions* $\Sigma_c$ and *uncontrollable actions* $\Sigma_u$ ($\Sigma = \Sigma_c \cup \Sigma_u \wedge \Sigma_c \cap \Sigma_u = \emptyset$). When describing LTS models we may assume that this partition exists.

The controller is also modelled as an LTS that restricts the ocurrence of controllable actions based on the observation of the events that have occurred. A controller $C$ is executed concurrently with the environment $E$. Since both of them are LTSs, this is modelled by the parallel composition of LTSs ($E\|C$). To be a solution, a controller must satisfy the following conditions. First, a controller must ensure that the goals $G$ are satisfied in every trace of its concurrent execution with the environment. We denote that a controller satisfies the goals as $E\|C \models G$, which means that every trace in the concurrent execution ($\forall \pi \in Tr(E\|C)$) satisfies the goals ($\pi \models G$). Second, a controller must not block the uncontrollable behaviour. The notion of a controller that does not block uncontrollable actions is built on the concept of the *legal environment* for Interface Automata [60]. Intuitively, a controller $C$ is a legal LTS for the environment $E$, when in every state $(q_E, q_C)$ of $E\|C$, an uncontrollable action is enabled in $(q_E, q_C)$ iff it is also enabled in $q_E$. Third, the concurrent execution $E\|C$ must be deadlock-free.

**Definition 3.6 (LTS Control Problem)** *Given an LTS $E = (Q_E, \Sigma, \Delta_E, q_{E_0})$, a goal $G$ expressed in FLTL, and a set of controllable actions $\Sigma_c \subseteq \Sigma$, the solution to the control problem $\mathcal{E} = \langle E, G, \Sigma_c \rangle$ is to find a deterministic LTS $C = (Q_C, \Sigma, \Delta_C, q_{C_0})$ such that i) $C$ is a legal LTS for $E$, ii) $E\|C$ is deadlock free, and iii) every infinite trace $\pi$ in $\mathrm{Tr}(E\|C)$ satisfies $G$ ($\pi \models G$).*

**Definition 3.7 (Legal LTS)** *Given an LTS $E = (Q_E, \Sigma, \Delta_E, q_{E_0})$, an LTS $C = (Q_C, \Sigma, \Delta_C, q_{C_0})$ and $\Sigma_u \subseteq \Sigma$, the LTS $C$ is a legal LTS for $E$ if $\forall (q_E, q_C) \in Q_{E\|C}$ holds that $\Delta_{E\|C}((q_E, q_C)) \cap \Sigma_u = \Delta_E(q_E) \cap \Sigma_u$, where $Q_{E\|C}$ are the states of $E\|C$.*

To distinguish between *enabled controllable actions* and *enabled uncontrollable actions* we use the following notation.

- *Enabled Controllable Actions*: $\Delta^c(q) = \Delta(q) \cap \Sigma_c$.

- *Enabled Uncontrollable Actions*: $\Delta^u(q) = \Delta(q) \cap \Sigma_u$.

## 3.5 Parametric timed automata

The formalisms mentioned above permit to describe properties about the behaviour of a system. However, by using those formalisms it is not possible to describe timing aspects. For instance, we are not able to model that an activity must finish within sixty seconds. Timed automata (TA) [56] is typically the formalism that is used to model the behaviour of time-critical systems. In fact, both LTS and TA are types of transition systems. The main difference is that TA comes with a finite set of clocks, which are real-valued variables. Note that this real-valued variables are used to represent time. Thus, in the transitions, clocks can be reset but cannot be assigned with a particular value. These clocks can be used to define time conditions in the automata. Conditions can be used to express invariants in the states or to describe guards in the transitions. That is to say, it permits to model conditions that the clocks of the system must satisfy to stay in a state or to transition from one state to another. For example, we could have a clock $x_1$ to measure when an activity is being performed. Then, we could use a state invariant $x_1 \leq 60$ to model that the activity must leave a state within sixty seconds, or a guard describing that some transition could only be taken within some time range $20 \leq x_1 \leq 60$. Automaton $T_1$ of Figure 3.3 describes a timed automaton with some of these conditions.

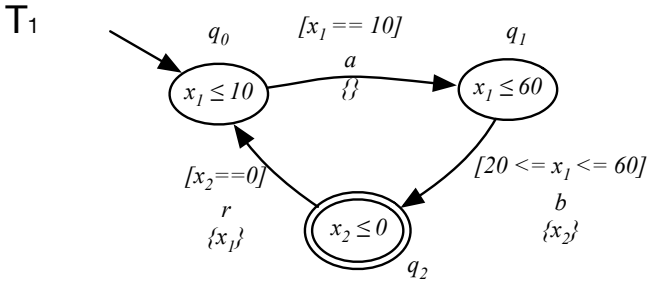



Figure 3.3: General example of timed automata.

**Definition 3.8 (Timed Automaton)** *A* timed automaton *(TA) is a tuple* $TA = (\Sigma, Q, Q_0, X, Q_f, I, \Theta)$ *where*

- $\Sigma$ *is a set of actions,*
- $Q$ *is a set of states,*
- $Q_0 \subseteq Q$ *is a set of initial states,*
- $X$ *is a finite set of clocks with domain* $\mathbb{R}_{\geq 0}$*,*
- $Q_f \subseteq Q$ *a set of final states*
- $I : Q \to XC(X)$ *a state-invariant function, and*
- $\Theta$ *is a set of edges s.t. each edge is a tuple* $(q_1, \ell, q_2, \lambda, \mu) \in \Theta$ *with* $q_1 \in Q$*,* $\ell \in \Sigma$*,* $q_2 \in Q$*, a set of clocks to be reset* $\lambda \subseteq X$*, and a guard* $\mu \in XC(X)$*.*

Conditions over clocks are usually referred as *clock constraints*. A valid clock constraint over the set of clocks $X$ is defined as follows:

$$xc ::= x < r \mid x \leq r \mid x \geq r \mid x > r \mid xc \wedge xc$$

where $x \in X$ and $r \in \mathbb{R}_{\geq 0}$. We will refer to the set of clock constraints over a set of clocks $X$ as $XC(X)$.

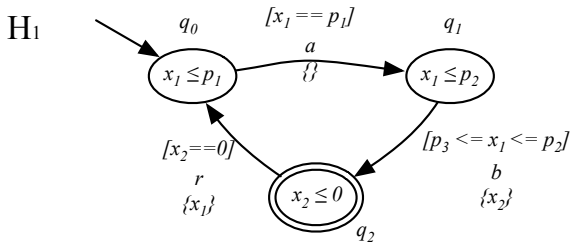
Timed automata is very powerful formalism to model time-critical systems, but it requires to have the precise time constraints that the system must satisfy. In this work, we are going to use time models to give a time semantic to models in which the time constraints are not available upfront. Therefore, we are going to use *parametric timed automata* (PTA) [25] to give a time semantic to LTS models. Parametric timed automata is an extension of timed automata in which *clock constraints* can be also defined over a finite set of *parameters* $P$ with domain $\mathbb{R}_{\geq 0}$. For instance, we would be able to express that an activity must finish within time $p$, where $p$ could be any real value. This is essential for us to model that an activity may take some time in an abstract way. Similarly to PTA, a valid clock constraint over the set of clocks $X$ and parameters $P$ is defined as follows:

$$xc ::= x < r \mid x \leq r \mid x \geq r \mid x > r \mid x < p \mid x \leq p \mid x \geq p \mid x > p \mid xc \wedge xc$$

where $x \in X$, $p \in P$ and $r \in \mathbb{R}_{\geq 0}$. We will refer to the set of clock constraints over a set of clocks $X$ as $XC(X, P)$.

**Definition 3.9 (Parametric Timed Automaton)** *A* parametric timed automaton *(PTA) is a tuple* $H = (\Sigma, Q, Q_0, X, P, Q_f, I, \Theta)$ *where*

- $\Sigma$ *is a set of actions,*
- $Q$ *is a set of states,*
- $Q_0 \subseteq Q$ *is a set of initial states,*
- $X$ *is a finite set of clocks with domain* $\mathbb{R}_{\geq 0}$*,*
- $P$ *is a finite set of parameters with domain* $\mathbb{R}_{\geq 0}$*,*
- $Q_f \subseteq Q$ *a set of final states,*
- $I : Q \to XC(X, P)$ *a state-invariant function, and*
- $\Theta$ *is a set of edges s.t. each edge is a tuple* $(q_1, \ell, q_2, \lambda, \mu) \in \Theta$ *with* $q_1 \in Q$*,* $\ell \in \Sigma$*,* $q_2 \in Q$*, a set of clocks to be reset* $\lambda \subseteq X$*, and a guard* $\mu \in XC(X, P)$*.*




$\Gamma(H_1) = p_1 <= p_2 \wedge p_3 <= p_2$

Figure 3.4: General examples of Parametric Timed Automata. $\Gamma(H_1)$ describes the conditions that parameters $\{p_1, p_2, p_3\}$ of $H_1$ need to satisfy to reach a final state.

The parameters in PTA can be instantiated into constant real-values. This means that every time that a parameter $p$ appears in a guard or an invariant, the parameter is replaced by a constant real value $r$. A *parameter valuation* $\gamma$ for

a set of parameters $P$ is function $\gamma : P \to \mathbb{R}_{\geq 0}$ that assigns to each parameter $p \in P$ a constant real-value $\gamma(p)$. Given a parameter valuation $\gamma$ and a PTA, we can obtain a TA by replacing the parameters with the constant real-values that are defined by $\gamma$. However, some parameter valuations might fix the guards and invariants with values that make impossible to take some transitions. For instance, if the invariant of the state becomes $x_1 \leq 60$ and the guard of the only outgoing transition becomes $x_1 \geq 100$. In this case, the invariant and the guard will never be valid at the same time. Thus, it is not possible to continue progressing from that state. When many transitions become invalid, there might not be a path that reaches a final state from the initial state. Then, the TA obtained may become invalid as a whole, since it is not describing any accepted behaviour. Figure 3.4 shows a parametric timed automaton $H_1$ that is similar to the timed automaton $T_1$ but with some parameters instead of constant numbers in the conditions. In this example, if parameter $p_3$ becomes 100, parameter $p_2$ becomes 60 and parameter $p_1$ becomes 10, the final state is not reachable. Thus, the parameters must satisfy certain conditions to reach a final state.

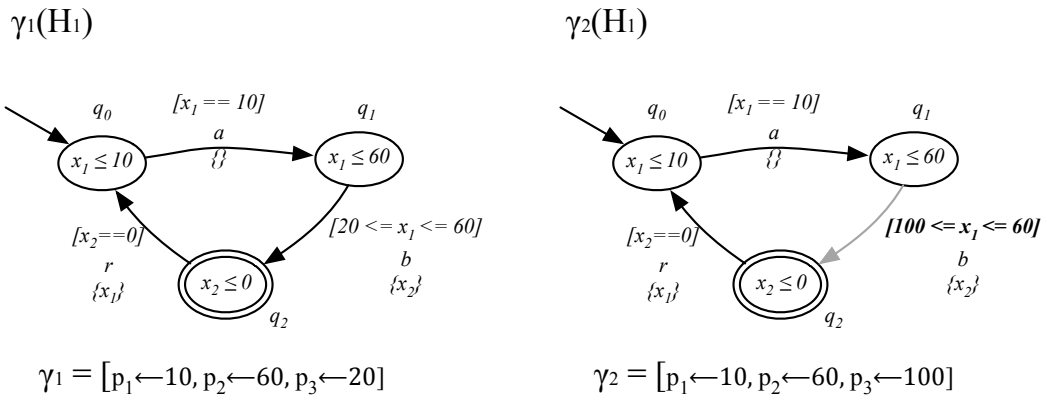



Figure 3.5: Timed automatons obtained by replacing the parameters of $H_1$ with the values indicated in each parameter valuation.

The assignment of values to the parameters is formalised as a *parameter valuation*. A parameter valuation is consistent with a PTA, if there exists at least one path that reaches a final state in the TA obtained from replacing the parameters with the constant values. In other words, the TA obtained describes a

non-empty language. We denote a parameter valuation $\gamma$, i.e., an assignment of real values $(r_1, \ldots, r_n) \in \mathbb{R}_{\geq 0}^{|P|}$ to the parameters $(p_1, \ldots, p_n) \in P$, that is consistent with a PTA $H$ as $\gamma \Vdash H$, or simply $(r_1, \ldots, r_n) \Vdash H$. The set of parameter valuations that is consistent with a PTA $H$ is denoted in the literature as $\Gamma(H)$ [25]. $\Gamma(H)$ is a set of constraints that a parameter valuation must satisfy to produce a valid TA from a given PTA. $\Gamma(H)$ can be expressed as a symbolic expression, which can be calculated by performing a fixed point iteration algorithm [26]. For instance, in the example of Figure 3.4, $\Gamma(H_1)$ is defined as $\Gamma(H_1) = p_1 \leq p_2 \wedge p_3 \leq p_2$. Also note that by applying the parameter valuation $\gamma_1 = [p_1 \leftarrow 10, p_2 \leftarrow 60, p_3 \leftarrow 20]$ to $H_1$ we obtain a timed automaton like $T_1$ from Figure 3.3. We denote it as $\gamma_1(H_1)$. As mentioned above, given a valuation $\gamma_2 = [p_1 \leftarrow 10, p_2 \leftarrow 60, p_3 \leftarrow 100]$, we obtain a timed automaton $\gamma_2(H_1)$, in which there is no trace that reaches a final state. Figure 3.5 shows these two timed automaton. The transition shown in gray cannot be traversed because no clocks would satisfy that condition.

In general, computing $\Gamma$ is undecidable, if there are more than three clocks in PTAs with cycles [25]. However, this work focuses on PTAs that do not have cycles, in which the symbolic procedure finishes [61]. In Chapter 7 we show the details to calculate $\Gamma(H)$ for the type of PTAs involved in this work.

## 3.6 Satisfiability modulo theories

The Boolean satisfiability problem (SAT) [62] is the problem of determining if a propositional boolean formula can be satisfiable or not. That is, to find an assignment of values (True or False) to the propositions of the formula that makes the formula be True. If such assignment exists, then the formula is satisfiable. For instance, the formula $p \vee q$ is satisfiable because the assignment $p = True$ and $q = False$ makes $p \vee q = True$. However, the formula $p \wedge \neg p$ is unsatisfiable because there is no possible assignment that makes this formula evaluate to $True$.

In the same vein, the satisfiability modulo theories (SMT) [27] problem is a decision problem for logical formulas of theories expressed in first order logic with equality. SMT formulas are more expressive than the ones of SAT. Among the supported theories are the theory of integers, the theory of real numbers, the theory of lists, the theory of arrays and theories of many other data structures. In

this work, we use SMT for solving decidability problems of the theory of real numbers. For instance, the formula $p_1 > p_2 \wedge p_1 \geq 0 \wedge p_2 \geq 0$ is satisfiable because there exists at least a pair of values $p_1 = 60$ and $p_2 = 20$ that evaluates the formula to $True$. This are the type of formulas (or expressions) that are produced by $\Gamma$. Checking whether these expressions are satisfiable or not is central to our approach.

# 4

# Industrial Automation Example

Assume an industrial automation setting, in which different product types are to be produced through a variety of processing activities according to given constraints. For simplicity, we assume that there are two product types. The machine has an inner tray where the raw elements of the products can be placed. The products have to be processed by using different tools that are available in the machine. Figure 4.1 shows a high level illustration of the configuration of the environment.

Figure 4.2 describes the list of requirements that the machine needs to satisfy to produce a product by using different tools. Some of the requirements are general, while others are specific to the product type to be processed.

According to the description from Figure 4.2, there are some tasks or activities that need to be performed to produce a product. First, the machine transforms the input-raw elements into processed elements by using some of the tools. Second, it checks that the processed elements fulfill the quality requirements by using a *QA Checker*. Third, if the quality of the product is not right, the machine repairs the products by using a *Repairing Tool*. Finally, it places the processed and quality

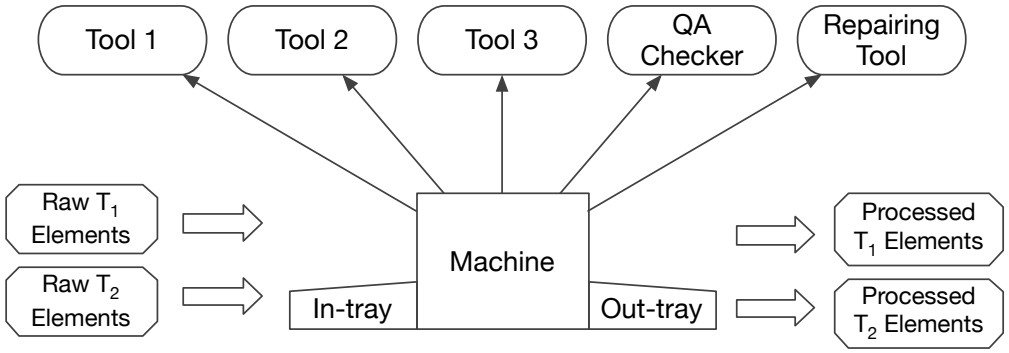



Figure 4.1: Illustration of an industrial automation plant with five tools and two types of input products to be processed.

General requirements:

1. Check the quality of a product before placing it in the output tray.
2. If the processed product does not pass the quality check, use the repairing tool to meet the quality requirements.
3. Emit a signal when the product is placed in the output tray.
4. Warning! Never start the tool 2 while the tool 1 is being used because it may produce a power outage.
5. Process one product at a time.

Product type specific requirements:

Type 1: Use the tool 3 to process them.

Type 2: Use the tool 1 and tool 2 to process them.
The tools can be applied in any order.

Figure 4.2: Requirements of the industrial automation example in natural language.

assured elements into the out-tray. The machine can process one element at a time, and it needs to use different tools depending on the product type.

Figure 4.3 shows an illustration of the activities that have to be done to process a product. The restrictions about when the tools cannot be used at the same time are not shown in this diagram. According to the requirements, there are three

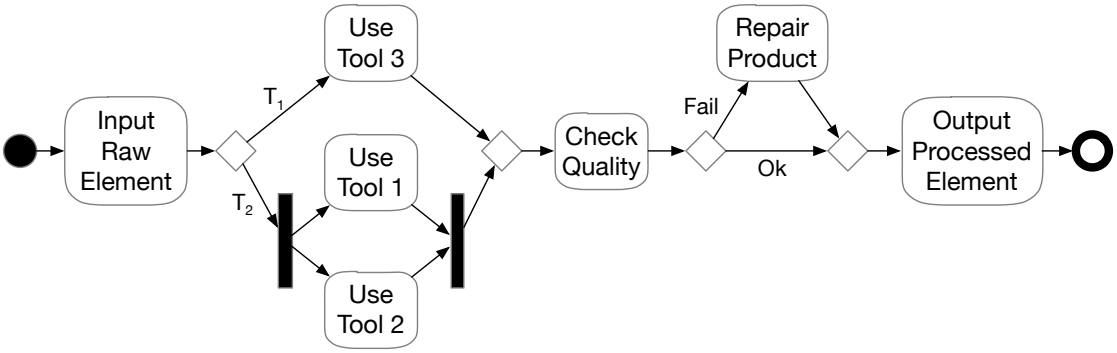



Figure 4.3: Diagram describing the activities that must be performed to process a product depending on the input type.

alternatives for using the tools on products of type 2, which are shown in Figure 4.4. All of these alternatives satisfy the requirements. However, *Alternative 1* seems to be preferable from a time perspective because it uses the tools simultaneously. This may save time in processing a product regardless the time it takes to use each of the tools. The main goal of this work is to find a way to express preference for solutions that may save execution time in a qualitative manner, i.e., without using numbers to represent the durations of the activities. We believe that it is possible to produce a comparison framework that can show that solutions like *Alternative 1* always have lower-makespan than *Alternative 2* or *Alternative 3*. Besides, we aim to produce synthesis algorithms that avoid sequential solutions like the ones show here, when concurrent alternatives are available.

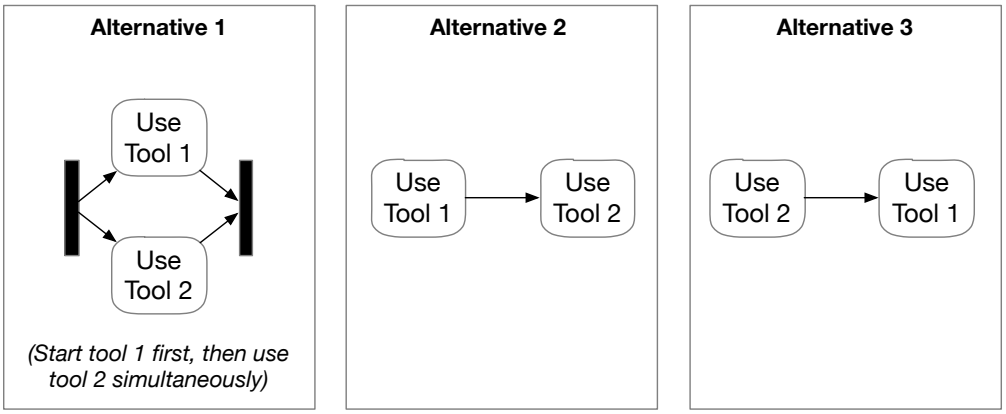



Figure 4.4: Alternatives for using the tools on products of type two.

# 5 A Qualitative Approach for Makespan

In this chapter we define the proposed approach from a high level perspective. Section 5.1 presents the two main problems that we aim to solve and the role that activities play in them. Section 5.2 defines the concept of makespan in qualitative problems with safety and reachability goals. Section 5.3 defines how to measure and Section 5.4 how to compare makespan of controllers. Section 5.5 defines how to deal with contigencies when comparing makespan. Section 5.6 defines makespan-minimising controllers, which are the type of controllers that we would like to generate. Section 5.7 introduces the main idea of the proposed algorithm to obtain a makespan-minimising controller. The details will be described in the following chapters.

## 5.1 The main parts of the problem

Our goal is to produce a framework to generate controllers that are preferable regarding a qualitative metric. The metric chosen in this work is makespan,

which is explained in the next section. Thus, one of the problems we aim to solve is *How to synthesise lower-makespan controllers?*. Typically, the inputs of a control problem are a description of the environment, a set of system goals and the actions that can be controlled. The *Controller Synthesiser* from Figure 5.1 illustrates this case. We propose to add extra information to the control problem to define algorithms that use this information to give more preferable controllers regarding makespan. This is illustrated in the second *Controller Synthesiser** of the same figure. In this approach, we relieve the user from providing quantitative estimations, but require additional qualitative description of the environment. This description is the *activities definition*, which is simply a description of the actions that are related in the environment. Details about activities are described in Section 5.3. In our approach, *Activities definition* must be specified at design time, because they are used to synthesise the controller.

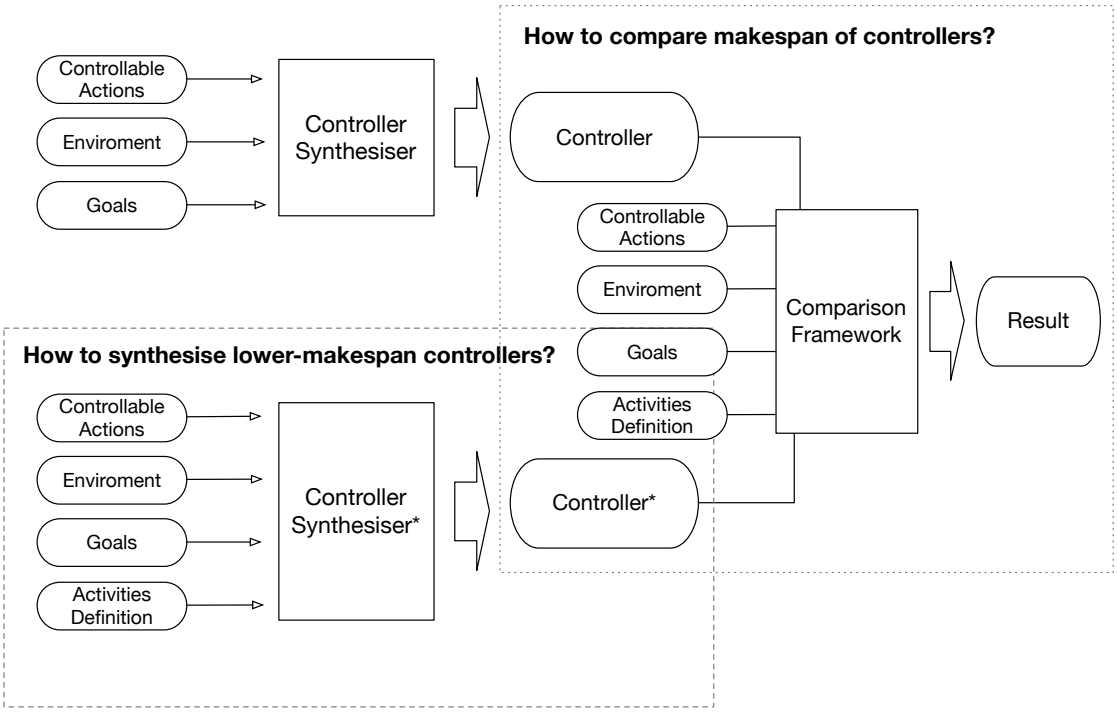



Figure 5.1: Two main problems and its different components.

*Activities definition* play also an important role in the other part of the

problem, which is *how to compare makespan of controllers? (qualitatively)*. The problem is to determine if any of the two controllers has lower-makespan than the other. This is also illustrated in Figure 5.1 as input of the comparison framework. Here activities are used to provide a timed semantics to controllers, which is defined as a parametric timed automata. By modelling activities, we can define a way to compare controllers qualitatively. Besides, activities are also used when defining schedulers, which are used to model different behaviour of the environment regarding contingencies. Details about this is described in Section 5.5.

## 5.2 Defining makespan

This research focuses on controllers with safety and reachability goals. Safety goals are used to define behaviour that should always be satisfied. It can be used to model bad behaviour that we want to avoid or to ensure some good behaviour throughout the whole execution. Reachability goals are typically used to model good behaviour that we want to achieve. Reachability goals can be seen as missions that must be accomplished. The satisfaction of the goals is evaluated when executing the controller with its environment. Figure 5.2 shows the satisfaction of these formulas over traces of the the $E \| C$. Note that a controller must continue ensuring the safety $S$ proposition after reaching a state that makes the proposition $P$ become True.

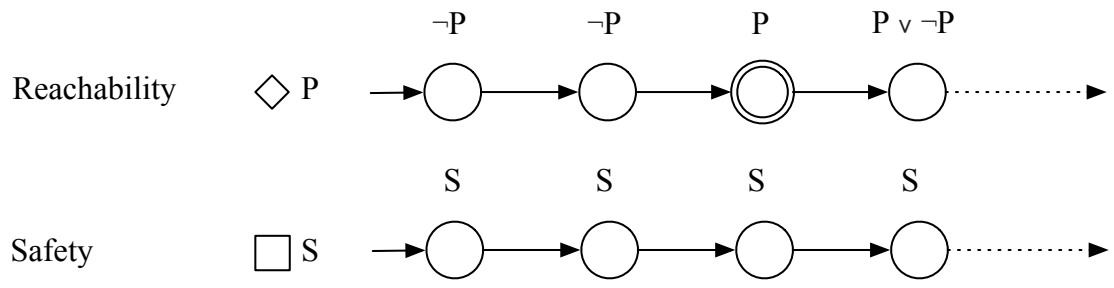



Figure 5.2: Satisfaction of safety ($\Box S$) and reachability ($\Diamond P$) goals over traces.

We define the *goal states* as those in which the reachability goal becomes accomplished, i.e., the mission is achieved. In Figure 5.2, the goal state is denoted

with double-lined circle. Achieving a reachability goal in the minimum time possible is a preference that is often desired. Thus, we define *makespan* as our metric to evaluate the quality of a controller. Makespan measures the time that elapses from the start of a task to the end. In our approach, makespan is measured from the initial state to the goal states. After reaching a goal state, a controller may do the minimum actions to keep itself safe, or it could continue to do other safe activities. The only condition after reaching a goal state is to respect the safety goals. This type of controllers is of interest in environments where a new mission can be assigned by updating the system with a new controller [17]. For example, a robot that is deployed in another planet and receives missions periodically. Figure 5.3 shows an abstract representation of how a safety and reachability controller would look like. In the following chapters, we will often illustrate the controllers only from initial state to goal states, because those are states considered when measuring makespan.

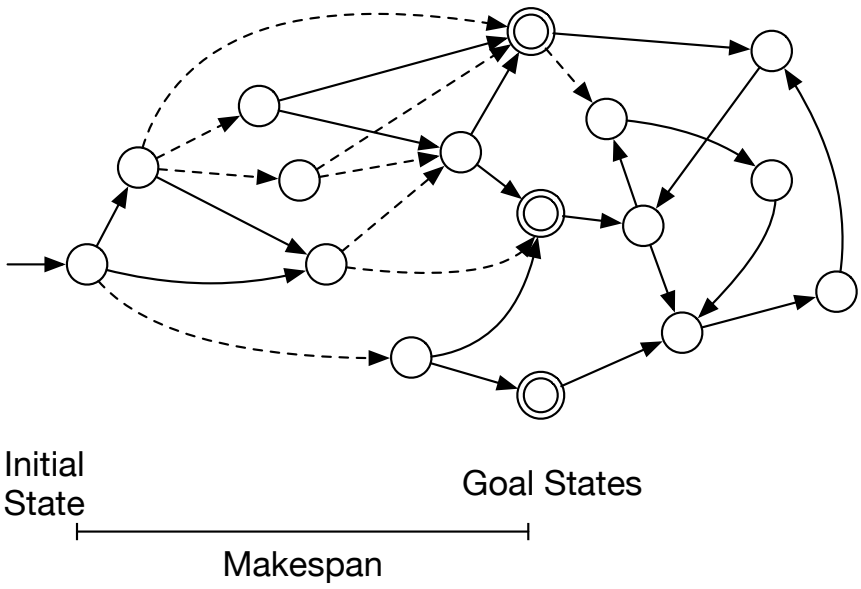



Figure 5.3: Abstract illustration of a controller for safety and reachability. Makespan is measured from the initial state to the goal states. Actions are omitted.

## 5.3 Measuring makespan

Makespan tends to be considered as a quantitate measure. Quantitative algorithms for makespan can produce optimal solutions when precise quantitative models are available. However, quantitative information are not always available or precise enough to be modelled. For instance, not always the duration of the activities are known in advance. In those cases, avoiding quantitative modelling may be desirable, because using quantitative algorithms on those models may produce solutions that are over-fitted to inaccurate numbers. For example, Figure 5.4 shows an abstract example of calculation of the average duration on a graph that has all the possible alternatives to reach goal states. A quantitative approach would choose a solution like *Option 1* because the value that represents makespan is lower than that of *Option 2*. However, if the estimation of those values were inaccurate we would be discarding the *Option 2* arbitrarily.

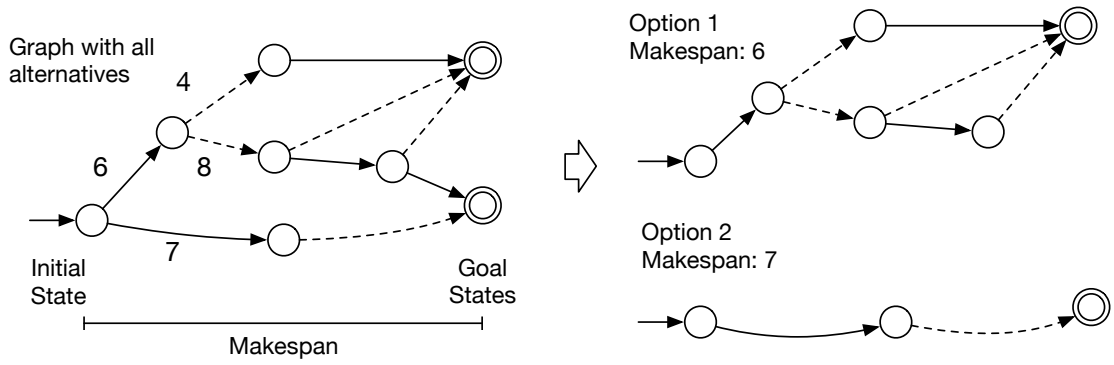



Figure 5.4: Illustrative example of using numbers to measure makespan. Numbers represent average duration to reach goal states. Actions are omitted.

In our approach, we define a qualitative way to compare controllers. This approach requires engineers to annotate the control problems with *activities definition*, which aggregate actions that are implicitly related. For instance, the action of *start doing something* and *finishing doing something* can be modelled as a *doing something* activity. This means that our method does not require estimating numbers that represent durations, but making the implicit relation between the actions become explicit as activities.

The way in which we measure makespan is similar to a logical description of the behaviour of the controllers regarding activities. For instance, the makespan of a controller is the time it takes the activity *doing something* plus that of activity *moving somewhere*. In more complex scenarios, the description would also have disjunctions depending on contingencies. For example, if the robot is in the right location the makespan of the controller is the time takes *doing something*, otherwise it is the time it takes *moving to location* plus that of *doing something*.

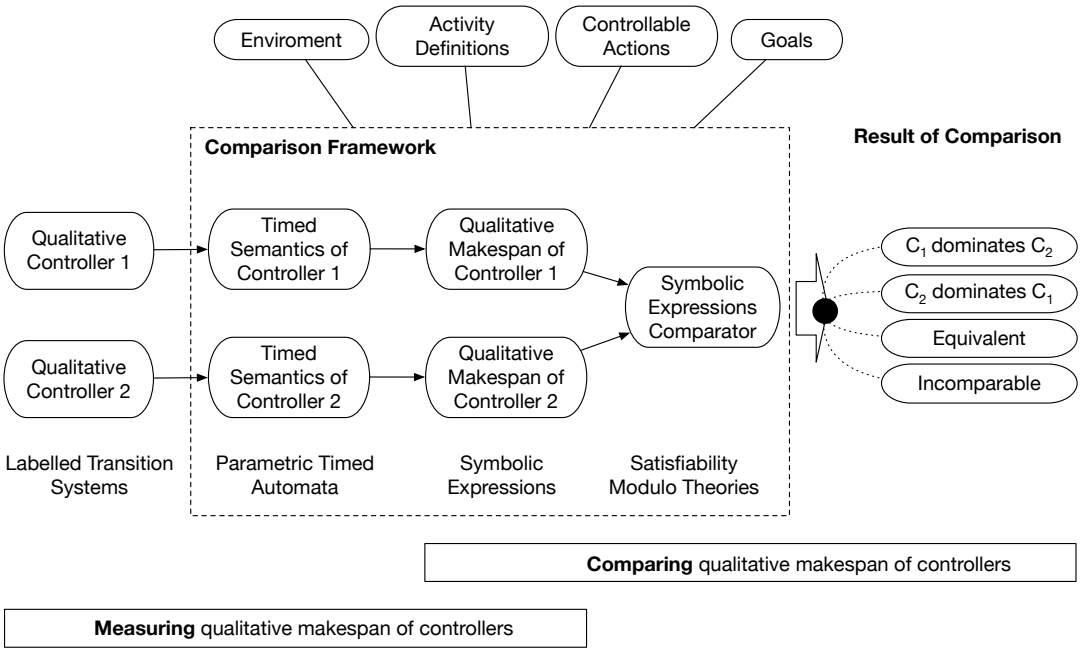



Figure 5.5: Formalisms that are used in each step of the transformation to solve the problem of measuring qualitative makespan and comparing it.

This is formalised by defining a timed semantics for LTS controllers with activities definition by using parametric timed automata. Activities are used to define symbols (clocks and parameters) of a PTA, which is the timed semantics of a controller. Then, we obtain a logical expression from the PTA, which is defined in terms of these symbols. This expression is the qualitative makespan of a controller based on activities definition. Figure 5.5 illustrates the process of this transformation. As we can observe in this figure, the next issue to address is how to compare qualitative makespan of two different controllers.

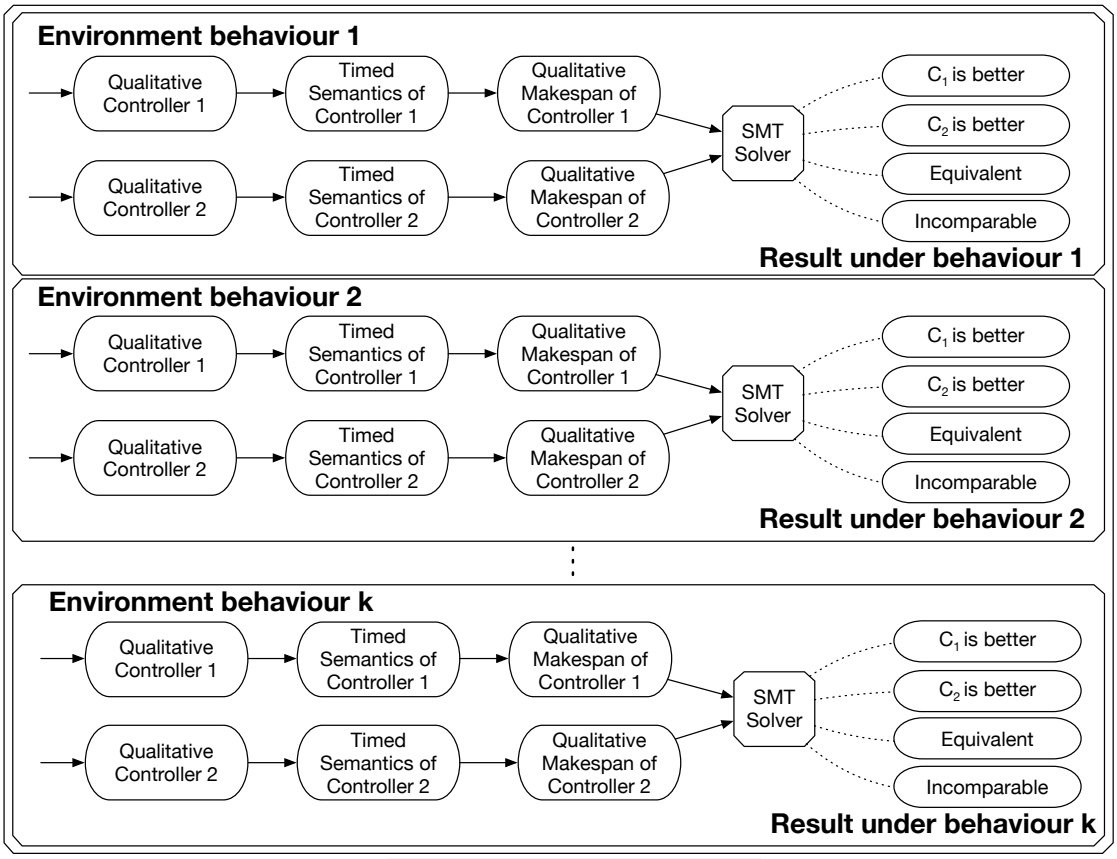
## 5.4 Comparing makespan

Given a qualitative control problem, there are different solutions (or controllers) that can be obtained by using different synthesis algorithms. In a quantitative approach, it would be easy to compare them by using average-case or worst-case makespan. In our qualitative approach, we need to define how to compare qualitative makespan of controllers, because they are logical expressions with parameters that represent activities durations, for which it is not defined how to compare them. Our goal is to establish a comparison relation between these expressions. For instance, it seems that *doing something* takes longer than *moving to location* plus *doing something*, but we need to formally show that.

To establish the relation between the qualitative makespan of two controllers, we are going to perform two comparisons, we are going to check if one controller can have a behaviour of higher makespan than the other, and the reverse. In Figure 5.5 this is modelled by the *Symblic Expressions Comparator*. Checking for a behaviour of higher makespan means to find a case in which the controller performs worse than the other. After checking that in both directions, the defined comparison produces one out of the four possible results, which are also shown in the same figure: $C_1$ dominates $C_2$, $C_2$ dominates $C_1$, equivalent or incomparable. A controller dominates the other, if it always behaves better or the same with the other. The details about how to measure and compare makespan are explained in Chapter 7.

## 5.5 Dealing with contingencies

The problems that are considered in this work are reactive, which means that the behaviour of the environment is not always the same. The reactiveness of the environment is modelled with uncontrollable actions. The controller cannot choose which or when uncontrollable actions happen. Instead, the environment chooses during the execution of the controller. It would be unfair to compare controllers assuming different behaviour of the environment. Thus, the comparison defined here fixes the behaviour of the environment and then compares the controllers under the same conditions.




**Aggregate Results**

| $C_1$ dominates $C_2$ | | | |
|---|---|---|---|
| $C_1$ is better | $C_2$ is better | Equivalent | Incomp. |
| > 1 | 0 | >= 0 | 0 |

| $C_2$ dominates $C_1$ | | | |
|---|---|---|---|
| $C_1$ is better | $C_2$ is better | Equivalent | Incomp. |
| 0 | > 1 | >= 0 | 0 |

| Equivalent | | | |
|---|---|---|---|
| $C_1$ is better | $C_2$ is better | Equivalent | Incomp. |
| 0 | 0 | k | 0 |

| Incomparable | | | |
|---|---|---|---|
| $C_1$ is better | $C_2$ is better | Equivalent | Incomp. |
| >= 0 | >= 0 | >= 0 | > 1 |
| > 1 | > 1 | >= 0 | >= 0 |

Figure 5.6: Comparison of two controllers under different environment behaviour during the execution.

Figure 5.6 illustrates the multiple comparisons that are needed to establish a relation between two controllers. Each of the comparisons considers a different behaviour of the environment. Also note that we use the word "dominates" because a controller dominates another controller only if the controller is always better than or equivalent to the other controller under any fixed behaviour of the environment. The term dominance is used in game theory to refer to strategies that provide greater utility to a player, regardless of what the other player does. In our case, the utility is to have lower makespan, and the other player is the environment, who can choose which and when to execute an uncontrollable action. The bottom of Figure 5.6 shows how different behaviours of the environment are aggregated to produce the final result of the comparison. Two controllers are equivalent if their makespan is always equivalent, and they are incomparable if there are cases in which one is better and cases in which the other one is better, or if there is a case in which their makespan is incomparable by itself. This could happen if two controllers perform different activities under the same behaviour of the environment. Section 7.1 formalizes how the different behaviours of the environment are modelled.

## 5.6 Makespan-minimising controllers

A makespan-minimising controller is our definition of a good controller. This concept is defined by using the qualitative comparison of makespan between controllers. A controller is makespan-minimising if it is not dominated by any other controller. In other words, there cannot be another controller that performs better than a makespan-minimising controller in every case.

Figure 5.7 illustrates the possible relations between a makespan-minimising controller and other controllers. A makespan minimising controller can dominate another controller, can be incomparable with other controller, or can be equivalent to other controller, which also implies that there could be more than one. Makespan-minimising controller is formalised in Section 8.1.

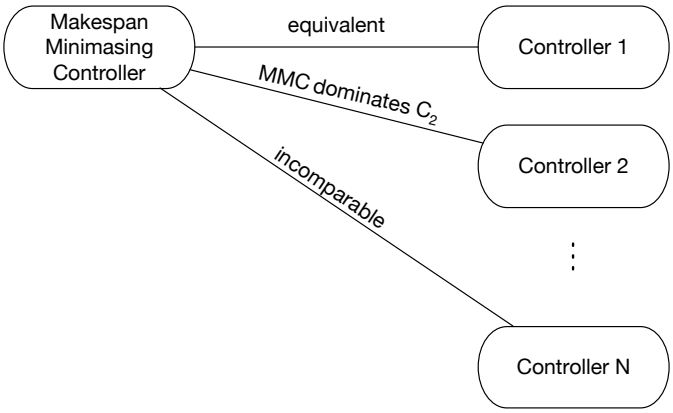



Figure 5.7: Possible relations between a makespan-minimising controller and other controllers.

## 5.7 Obtaining makespan-minimising controllers

To obtain a makespan-minimising controller we proposed an algorithm that takes as input a control problem (environment, controllable actions and goals) and the activities definition. This means that the only extra requirement to use this algorithm is to make explicit the *activities definition*. The idea of the algorithm is to first obtain a solution that contains all the possible alternatives to reach a goal. This is typically referred as universal controller. Then, the algorithm prunes transitions that lead to executions that are always worse than some of the other alternatives. To do that, it also uses the qualitative comparison of makespan between controllers.

Figure 5.8 illustrates the process of synthesising a controller with our approach. As mentioned above, the inputs of the algorithm are the input of the standard synthesis algorithm and the set of activity definitions. The activity definitions are necessary because the algorithm uses the comparison framework to prune transitions from a universal controller. The comparison framework requires activity definitions to be defined to give a timed semantics to the controller. The details of the algorithm are presented in Section 8.2.

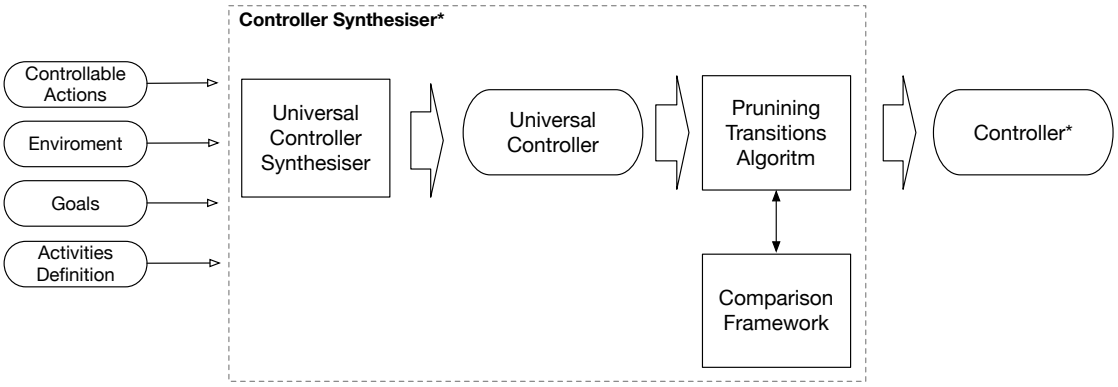



Figure 5.8: Proposed approach to synthesis a makespan-minimising controller.

# 6 Control Problems with Activities

This chapter introduces the control problem with activities. Section 6.1 models the industrial example from Chapter 4 as a control problem. Then, it shows alternative controllers to the same problem, and the limitations of standard qualitative synthesis techniques to express preferences. Section 6.2 describes how to interpret the passage of time in LTS Control. Section 6.3 shows how to model activities that take time in LTS. Finally, Section 6.4 introduces the formalization of the control problem with activities.

## 6.1 Industrial example as a control problem

The specification of the control problem is given as a set of LTSs describing the behaviour of the environment (Figure 6.1) and a set of FLTL formulas describing production constraints (Figure 6.2). In Figure 6.1, the environment is specified as several components, and it is the parallel composition of all of them ($E = Process_1 \| Process_2 \| Process_3 \| Choice \| QA \| RepairProcess$). $Process_i$ models the

use of a a tool $i \in \{1, 2, 3\}$ to process a raw element and it is described with the index $i$ for conciseness. Solid lines ($\rightarrow$) denote controllable transitions, and dashed lines ($\dashrightarrow$) denotes uncontrollable transitions. For instance, the action that starts a tool (e.g.: $s_{A_1}$) is controllable, and the action that determines the product type is uncontrollable ($type_1$ and $type_2$). This means that the machine does not control the input type nor when the tools finish processing a raw element.

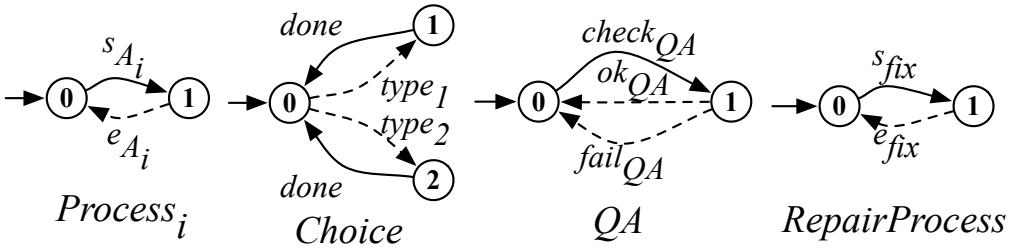



Figure 6.1: LTS components describing the behaviour of the of the environment $E = Process_1 \| Process_2 \| Process_3 \| Choice \| QA \| RepairProcess$.

As mentioned above, formulas in Figure 6.2 describe the production constraints. Type $T_1$ products require using the tool 3 (activity $A_3$), while $T_2$ type products require using the tool 1 and 2 (activity $A_1$ and $A_2$). After finishing the required activities, products must undergo a quality check (R1). When the quality check fails, it is required to do a repair activity (R2). Once the product has passed the quality requirements, it is signalled with the action *done* (R3). Additionally, for safety reasons, $A_2$ cannot be started while $A_1$ is ongoing (R4). The goal is to produce one product of the required type ($\Diamond \dot{done}$) while always satisfying the production constraints ($\Box(R_1 \wedge R_2 \wedge R_3 \wedge R_4)$).

R1) $\dot{checkQA} \implies ((TC_1 \wedge AC_3) \vee (TC_2 \wedge AC_1 \wedge AC_2))$
R2) $\dot{s_{fix}} \implies QAFailed$
R3) $\dot{done} \implies QAChecked$
R4) $\dot{s_{A_2}} \implies \neg OngoingA_1$

Figure 6.2: Production constraints described in FLTL.

The formulas described in Figure 6.2 are combinations of the fluents described in Figure 6.3 and fluents $\dot{\ell}$ induced by an action $\ell$. Fluents $TC_i$ and $AC_i$ are described with the index $i$ for brevity. $TC_i$ denotes the product type choice while

$AC_i$ denotes that the activity $A_i$ has finished. $QAChecked$ holds, when a product satisfies the quality requirements. $QAFailed$ holds when the quality check fails but the product is not repaired yet. $OngoingA_1$ is a fluent that holds when $A_1$ has started ($s_{A_1}$) but not ended yet ($e_{A_1}$).

a) $TC_i = \langle\{type_i\}, \{done\}, false\rangle$
b) $AC_i = \langle\{e_{A_i}\}, \{done\}, false\rangle$
c) $QAChecked = \langle\{okQA, e_{fix}\}, \{type_1, type_2\}, false\rangle$
d) $QAFailed = \langle\{failQA\}, \{e_{fix}\}, false\rangle$
e) $OngoingA_1 = \langle\{s_{A_1}\}, \{e_{A_1}\}, false\rangle$

Figure 6.3: Fluent definitions used to describe the formulas of Figure 6.2.

The synthesis problem is to build a controller that satisfies production requirements by controlling processing activities, while the choice of product type and the result of the quality check are unknown in advance. Various solutions to this problem exist, and current synthesis techniques are likely to yield the controller $C_2$ of Figure 6.4; whereas the controller $C_1$ in the same figure is another valid solution.

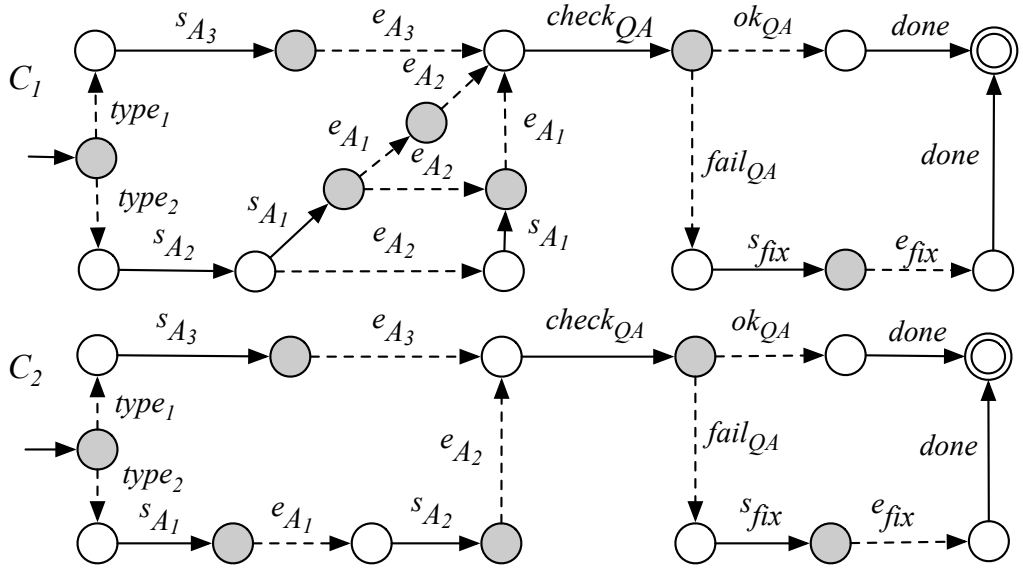



Figure 6.4: Possible controllers for the same problem.

Regarding makespan, $C_1$ seems to be better than $C_2$. It is because, for products of type $T_2$, $C_1$ will always try to run the two activities concurrently, while $C_2$ will

perform them sequentially. Intuitively, from a time point of view, $C_1$ will perform as well as or better than $C_2$ for any fixed choice of the environment (the type of product to be built and the result of the quality check).

## 6.2 Understanding time in LTS control

In LTS models, time passes only on states, and transitions change states instantaneously. In general, the amount of time that passes between two different actions is unknown. The standard way of modelling the passing of time in LTS is to incorporate actions that represent the start and the end of activities which take time. This is the approach we adopt. We assume that *all* activities that have duration will be modelled with start and end actions.

Another aspect to analyse is the behaviour of a controller when is concurrently executed with the environment. We assume that a controller is *proactive*. That means that the controller is always ready to take a controllable transition as soon as a controllable transition becomes enabled. Thus, states in which there is at least one outgoing controllable transition ($\Delta^c(q) \neq \emptyset$) are considered to be the ones in which time does not pass. Such states are called *transient states* [63]. Note that it does not mean that the controller is faster than the environment. In states in which both controllable and uncontrollable actions are enabled, any of the enabled actions may occur. No matter which of them occur first, we understand that no time has passed in those states. In Figure 6.4, the transient states are denoted in white color, while the non-transient are shown in grey color.

## 6.3 Modelling activities in LTSs

To formally specify the activities in our models, we require an LTSs to be annotated with an *activities definition* $\mathcal{AD} = (\mathcal{A}, Start, Ends)$. $\mathcal{A}$ is a set of symbols that represent each activity $\alpha$ uniquely. The function $Start : \mathcal{A} \to \Sigma_c$ defines the controllable action that starts an activity. The function $Ends : \mathcal{A} \to 2^{\Sigma_u}$ defines the uncontrollable actions that determine the end of an activity. These two functions must be defined for each activity $\alpha \in \mathcal{A}$. An action can be related to at

most one activity. Table 6.1 illustrates the activities definition of the industrial example.

**Definition 6.1 (Activities Definition)** *Given an LTS* $M = (Q, \Sigma, \Delta, q_0)$ *with* $\Sigma = \Sigma_c \cup \Sigma_u$*, an activities definition* $\mathcal{AD} = (\mathcal{A}, Start, Ends)$ *for* $M$ *is defined by*

- $\mathcal{A}$ *a set of symbols,*
- $Start : \mathcal{A} \to \Sigma_c$ *a function defining the starting actions of the activities, and*
- $Ends : \mathcal{A} \to 2^{\Sigma_u}$ *a function defining the ending actions of the activities,*

*where* $\forall \alpha \in \mathcal{A}\ Start(\alpha) \in \Sigma_c \wedge Ends(\alpha) \neq \emptyset \wedge \forall \alpha' \in \mathcal{A}\ \alpha \neq \alpha' \implies Start(\alpha) \neq Start(\alpha') \wedge Ends(\alpha) \cap Ends(\alpha') = \emptyset$.

| Activity ($\mathcal{A}$) | *Start* | *Ends* |
|---|---|---|
| Activity 1 ($A_1$) | $s_{A_1}$ | $e_{A_1}$ |
| Activity 2 ($A_2$) | $s_{A_2}$ | $e_{A_2}$ |
| Activity 3 ($A_3$) | $s_{A_3}$ | $e_{A_3}$ |
| Repair (Fix) | $s_{fix}$ | $e_{fix}$ |

Table 6.1: Activities definition for the industrial automation example.

An LTS $M$ must fulfill certain conditions to be *activity compatible* with an activities definition $\mathcal{AD}$. First, all paths that lead to a state must have the same set of running activities. We say that an activity $\alpha \in \mathcal{A}$ is *running* in a state if the activity has being started but not ended at the given state. For example, in a state $q_m$ that is reached by the trace $type_2, s_{A_1}, s_{A_2}$ the running activities are $A_1$ and $A_2$ ($\zeta(q_m) = \{A_1, A_2\}$). Second, when an activity $\alpha$ is *running* in a state $q_m$, the set of ending actions must be enabled ($Ends(q_m) \subseteq \Delta(q_m)$), and also the starting action must be disabled ($Start(q_m) \notin \Delta(q_m)$). This means that an activity cannot be started again while it is running. If we do not impose these restrictions, models with a trace like $s_{A_1}$,$s_{A_1}$,$e_{A_1}$,$e_{A_1}$would be possible. In this trace, it is not possible to know which is the corresponding end of activity of each of the starts of activity.

**Definition 6.2** *(Running Activity) Given a finite path* $\eta \in \text{Paths}(M)$ *of an LTS* $M = (Q, \Sigma, \Delta, q_0)$ *and an activity* $\alpha \in \mathcal{A}$ *defined in* $\mathcal{AD} = (\mathcal{A}, Start, Ends)$*, we say*

*that $\alpha$ is a* running activity *in the state $q_m$, according to the path $\eta = q_0 \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_k} \ldots \xrightarrow{\ell_i} \ldots \xrightarrow{\ell_m} q_m$, if $\exists k \leq m$ s.t. $\ell_k = Start(\alpha)$ and $\forall k < i \leq m\ \ell_i \notin Ends(\alpha)$. We use $\xi(\eta)$ to denote the set of running activities according to a path and we use $\zeta(q_m) = \bigcup_{\eta \in \text{Paths}(M,q_m)} \xi(\eta)$ to denote the set of running activities in a state.*

**Definition 6.3** *(Activity Compatible) An LTS $M = (Q, \Sigma, \Delta, q_0)$ is* activity compatible *with an activity definition $\mathcal{AD} = (\mathcal{A}, Start, Ends)$ if $\forall q \in Q$ and $\forall \alpha \in \mathcal{A}$ the following holds:*

1. $\forall \eta \in \text{Paths}(M, q)\ \xi(\eta) = \zeta(q)$
2. $\alpha \in \zeta(q) \Leftrightarrow Ends(\alpha) \subseteq \Delta(q)$
3. $Start(\alpha) \in \Delta(q) \Rightarrow \alpha \notin \zeta(q)$

## 6.4 Defining control problem with activities

We restrict attention to goals of the form $G = \Box S \wedge \Diamond P$, where $S$ (safety goals) and $P$ (reachability goals) are propositional FLTL formulas. A solution to problems with this type of goals must ensure that *i*) every infinite path of $E \| C$ satisfies the safety formula $S$, and *ii*) that it is always possible to reach a state in which the propositional formula $P$ becomes True from the initial state. Recall Figure 5.2 to observe how satisfaction over traces is defined. Similarly to Definition 3.6, we define a *control problem with activities* as an LTS control problem in which the LTS environment is *activity compatible* with an activity definitions, and the goals are *safety and reachability goals.*

**Definition 6.4** (Control Problem with Activities) *Given an activity definitions $\mathcal{AD} = (\mathcal{A}, Start, Ends)$, an LTS $E = (Q_E, \Sigma, \Delta_E, q_{E_0})$ that is activity compatible with $\mathcal{AD}$, a safety and reachability goal $G = \Box S \wedge \Diamond P$ expressed in FLTL, and a set of controllable actions $\Sigma_c \subseteq \Sigma$, the solution to the control problem $\mathcal{E} = \langle E, \mathcal{AD}, G, \Sigma_c \rangle$ is a deterministic LTS $C = (Q_C, \Sigma, \Delta_C, q_{C_0})$ such that i) $C$ is a legal environment for $E$, ii) $E \| C$ is deadlock free, and iii) every infinite trace $\pi$ in $\text{Tr}(E \| C)$ satisfies $G$ ($\pi \models G$).*

We distinguish the states in which the proposition $P$ is satisfied for the first time as the *goal states*. These states are a subset of the states of the controller executed in parallel with the environment $E\|C$, because the satisfaction of the goals is evaluated on the traces of this LTS. The *goal states* $Q_G$ are those in $Q_{E\|C}$ that can be reached through a path from the initial state that does not contain another state satisfying the reachability property.

**Definition 6.5 (Goal States)** *Given a control problem with activities* $\mathcal{E} = \langle E, \mathcal{AD}, G, \Sigma_c \rangle$ *and a controller* $C$ *that is solution to the control problem* $\mathcal{E}$*, we define the* goal states $Q_G$ *as those states* $q_G \in Q_{E\|C}$ *where* $\exists q_0 \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_m} q_m \xrightarrow{\ell_G} q_G \in \text{Paths}(E\|C, q_G)$ *such that* $\ell_1 \ldots \ell_m \not\models G$ *and* $\ell_1 \ldots \ell_m, \ell_G \models G$.

In controller synthesis problems with reachability goals, algorithms that produce memoryless solutions have linear complexity in the size of the problem [64]. These controllers are a sub-graph of the LTS defined by the parallel composition of the environment with the LTSs that represent the fluents used to describe the goals $(E\|M_{fl_1}\| \ldots \|M_{fl_k})$. From a game-theoretical perspective, this composition represents the arena of the game between the environment and the controller. Furthermore, universal controllers [65] are those that subsume any other memoryless controller [66]. If we restrict attention to the sub-graph resulting from removing all outgoing transitions from goal states, universal controllers have the form of directed acyclic graphs (DAG). This observation is important because some of the algorithms presented in this thesis assume an acyclic graph.

**Definition 6.6 (Universal Controller)** *Given a control problem with activities* $\mathcal{E} = \langle E, \mathcal{AD}, G, \Sigma_c \rangle$*, a universal controller* $\mathcal{U} = (Q_\mathcal{U}, \Sigma_\mathcal{U}, \Delta_\mathcal{U}, q_{\mathcal{U}_0})$ *for* $\mathcal{E}$ *is a solution that subsumes every memoryless controller* $C = (Q_C, \Sigma_C, \Delta_C, q_{C_0})$ *that is solution for* $\mathcal{E}$ $(Q_C \subseteq Q_\mathcal{U} \wedge \forall q \in Q_C \ \Delta_C(q) \subseteq \Delta_\mathcal{U}(q))$.

Figure 6.5 shows the universal controller for the industrial example. We can observe that the universal controller subsumes the controllers shown in Figure 6.4. States $q_0$, $q_1$, $q_2$, $q_3$, $q_4$, $q_6$, $q_8$, $q_9$, $q_{10}$, $q_{11}$, $q_{12}$, $q_{13}$, $q_{14}$, $q_{15}$, $q_{16}$, $q_{17}$ are the ones that subsume the controller $C_1$, while $q_0$, $q_1$, $q_2$, $q_3$, $q_5$, $q_7$, $q_{10}$, $q_{11}$, $q_{12}$, $q_{13}$, $q_{14}$, $q_{15}$, $q_{16}$, $q_{17}$ subsume the controller $C_2$.

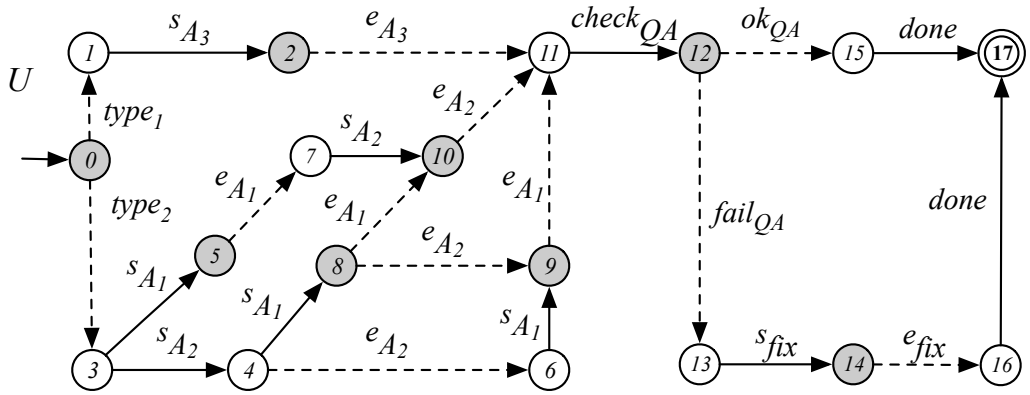



Figure 6.5: Universal controller for the industrial example.

# 7 A Qualitative Comparison Framework

This chapter focuses on how to compare solutions of a control problem with activities. We first formalise how to deal with contingencies by using schedulers (Section 7.1). Then, we give a timed semantics to LTSs based on PTA (Section 7.2) and show how to obtain a symbolic expression that represents the makespan of a controller using that PTA (Section 7.3). The parameters of the PTA stand for activity duration as well as a parameter representing the end-to-end makespan. By using parameters, we model that time durations are unknown a priori. Then, we define how to compare a pair of controllers by comparing their symbolic expressions (Section 7.4). Finally, we discuss the usage of real numbers to model the duration of activities that may not finish (Section 7.5).

## 7.1 Comparing under the same contingencies

When comparing controllers, we should pay attention to how uncontrollable behaviour occurs. For instance, let us consider the following unfair comparison

between the two controllers mentioned in Section 6.1. On the one hand, controller $C_1$ requires the final repair while constructing a product of type $T_1$. On the other hand, controller $C_2$ does not require the repair while building a product of type $T_2$. This comparison could suggest that controller $C_2$ has a lower makespan, but the conclusion is obviously flawed because the contingencies of which type of product is built and whether a repair is required or not do not depend on the controller itself and should not affect the comparison. Therefore, contingency scenarios should be consistent when a comparison is performed.

We use schedulers [67] to formalise how to resolve contingencies at a given state of the environment. We assume that schedulers are Markovian [67] and yield a decision only based on the current environment state regardless of the history of states traversed. Thus, we define schedulers as a function $\sigma : Q_E \times 2^{\Sigma} \to 2^{\Sigma}$ that picks a subset of enabled actions $A$ (K1), which stand for the actions that might be enabled by an arbitrary controller (Definition 7.1). The conditions defined aim to model schedulers that, we believe, ensure a fair comparison. This definition distinguishes actions into three categories: controllable, ending, and other uncontrollable actions.

**Definition 7.1 (Scheduler)** *Given an environment $E$ and a set of activity definitions $\mathcal{AD}$, a scheduler is a function $\sigma : Q_E \times 2^{\Sigma} \to 2^{\Sigma}$ that satisfies the following conditions:*

*K1)* $\sigma(q_E, A) \subseteq A \cap \Delta_E(q_E)$

*K2)* $(|\sigma(q_E, A)| = 1 \wedge \sigma(q_E, A) \subseteq \Sigma_c \cup (\Sigma_u \setminus \Sigma_e)) \vee$
$(\sigma(q_E, A) \cap \Sigma_e = \sigma(q_E, A))$

*K3)* $(\sigma(q_E, A) \subseteq \Sigma_u) \Rightarrow \forall A'(\sigma(q_E, A') = \sigma(q_E, A))$

*K4)* $\sigma(q_E, A) \subseteq \Sigma_c \Rightarrow \forall A' \subseteq \Delta_E(q_E)$
$((\sigma(q_E, A) \subseteq A' \Rightarrow \sigma(q_E, A') = \sigma(q_E, A)) \wedge$
$(\sigma(q_E, A) \not\subseteq A' \wedge A' \cap \Sigma_c \neq \emptyset \Rightarrow \sigma(q_E, A') \subseteq \Sigma_c) \wedge$
$(\sigma(q_E, A) \not\subseteq A' \wedge A' \cap \Sigma_c = \emptyset \Rightarrow \sigma(q_E, A') \subseteq \Sigma_u))$

*where* $A \subseteq \Delta_E(q_E)$, $q_E \in Q_E$, *and* $\Sigma_e = \{\ell \mid \ell \in \Sigma_u \wedge \exists \alpha \in \mathcal{A}\ \ell \in Ends(\alpha)\}$.

Above conditions require schedulers to either pick all enabled ending-actions or exactly one of the other enabled actions (K1 & K2). Since the duration of the activities is unknown, schedulers do not choose among the ending actions. Also,

schedulers have to be consistent, regarding the choices of uncontrollable (K3) and controllable actions (K4). That is, if possible, they pick the same actions or category.

For the sake of simplicity, to illustrate the behaviour of schedulers in the industrial example we will focus on the subset of states that overlap with the universal controller. We use the numbers shown in Figure 6.5 to refer to each of these states. One possible scheduler $\sigma_1$ is one that, amongst other things, determines the product type to be built is $T_2$ and that repair is not needed. Note that for $q_3$ and $q_4$, the output of the scheduler varies depending on which are the actions that controller has enabled. In state $q_3$, when a controller enables $\{s_{A_1}, s_{A_2}\}$ or $\{s_{A_2}\}$, the output is $\{s_{A_2}\}$. Otherwise, the output is the other controllable action $\{s_{A_1}\}$, which is the case of controller $C_2$. Similarly, in state $q_4$, when a controller enables the controllable action, i.e., $\{s_{A_1}, e_{A_2}\}$, the scheduler intends to enables the action $\{s_{A_1}\}$. Otherwise, it enables the uncontrollable action $\{e_{A_2}\}$, which should be enabled by every controller. This represents the case in which the end of activity action does not happen before a controller can execute a controllable action.

- $\sigma_1(q_0, \{type_1, type_2\}) = \{type_2\}$
- $\sigma_1(q_3, \{s_{A_1}, s_{A_2}\}) = \{s_{A_2}\}$, $\sigma_1(q_3, \{s_{A_2}\}) = \{s_{A_2}\}$, $\sigma_1(q_3, \{s_{A_1}\}) = \{s_{A_1}\}$
- $\sigma_1(q_4, \{s_{A_1}, e_{A_2}\}) = \{s_{A_1}\}$, $\sigma_1(q_3, \{e_{A_2}\}) = \{e_{A_2}\}$
- $\sigma_1(q_5, \{e_{A_1}\}) = \{e_{A_1}\}$
- $\sigma_1(q_6, \{s_{A_1}\}) = \{s_{A_1}\}$
- $\sigma_1(q_7, \{s_{A_2}\}) = \{s_{A_2}\}$
- $\sigma_1(q_8, \{e_{A_1}, e_{A_2}\}) = \{e_{A_1}, e_{A_2}\}$
- $\sigma_1(q_9, \{e_{A_1}\}) = \{e_{A_1}\}$
- $\sigma_1(q_{10}, \{e_{A_2}\}) = \{e_{A_2}\}$
- $\sigma_1(q_{11}, \{checkQA\}) = \{checkQA\}$
- $\sigma_1(q_{12}, \{failQA, okQA\}) = \{okQA\}$
- $\sigma_1(q_{15}, \{done\}) = \{done\}$

Another possible scheduler $\sigma_2$ could determine the product type to be built is $T_1$ and that quality check failed, which requires to repair the product. Note that these two schedulers represent two different behaviours of the environment. For products of $T_1$ there are no alternative solutions. Thus, the representation of

scheduler $\sigma_2$ is more compact than that of scheduler $\sigma_1$.

- $\sigma_2(q_0, \{type_1, type_2\}) = \{type_1\}$
- $\sigma_2(q_1, \{s_{A_3}\}) = \{s_{A_3}\}$
- $\sigma_2(q_2, \{e_{A_3}\}) = \{e_{A_3}\}$
- $\sigma_2(q_{11}, \{checkQA\}) = \{checkQA\}$
- $\sigma_2(q_{12}, \{failQA, okQA\}) = \{failQA\}$
- $\sigma_2(q_{13}, \{s_{fix}\}) = \{s_{fix}\}$
- $\sigma_2(q_{14}, \{e_{fix}\}) = \{e_{fix}\}$
- $\sigma_2(q_{16}, \{done\}) = \{done\}$

Schedulers are defined to consider the enabled actions in a state of the environment ($\Delta_E(q_E)$) and the actions a particular controller enables ($A$). Thus, we apply schedulers to a controller when executing in the environment $E\|_\sigma C$ (Definition 7.2). This produces an LTS $E\|_\sigma C$ that has a subset of the transitions defined by the standard parallel composition of $E\|C$.

**Definition 7.2 (Scheduled Composition)** *The* scheduled composition $E\|_\sigma C$ *is an asymmetric operator defined as the parallel composition ($\|$) changing the shared rule of $\Delta_{E\|_\sigma C}$ as follows:*

$$\frac{q_E \xrightarrow{\ell}_E q'_E,\ q_C \xrightarrow{\ell}_C q'_C,\ \ell \in \sigma(q_{E_1}, \Delta_C(q_{C_1}))}{(q_E, q_C) \xrightarrow{\ell}_{E\|_\sigma C} (q'_E, q'_C)}\ \ell \in \Sigma_E \cap \Sigma_C$$

In Figure 7.1, we show the scheduled composition of the controller $C_1$ and $C_2$ (in Figure 6.4) composed in parallel with the environment $E$ under a scheduler $\sigma_1$. Note that by using the scheduler, we can observe the controllers under similar behaviour of the environment.

## 7.2 Timed semantics of LTS

Our interpretation of time in a discrete controller is based on PTA [25], which is an extension of Timed Automata (TA) [56] that incorporates final states, clocks and parameters to LTSs. Clocks are real-valued variables that increase linearly, and parameters are unknown constants. We use PTA to interpret how time passes

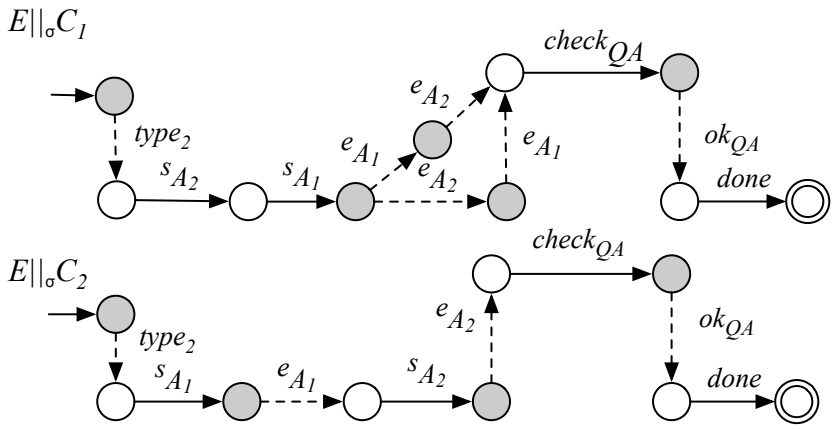



Figure 7.1: Scheduled composition of the controllers and environment defined in Section 6.1 under a scheduler $\sigma_1$.

on a controller, when the controller is executing in parallel with the environment under a given scheduler $\sigma$. Definition 7.3 presents the *Timed Semantics* of a controller by using a PTA. This PTA focuses on the paths between the initial state and goal states, because makespan is measured between those states.

**Definition 7.3 (Timed Semantics)** *Given a control problem $\mathcal{E} = \langle E, \mathcal{AD}, G, \Sigma_c \rangle$, a controller $C$ that is solution for $\mathcal{E}$, a scheduler $\sigma$ for the environment $E$ and goal states $Q_G$ in $E\|_\sigma C$, we define the timed semantics of $E\|_\sigma C$ as a parametric timed automaton $PTA(E\|_\sigma C) = (\Sigma', Q', Q'_0, X, P, Q_f, I, \Theta)$ as follows:*

- *a set of actions $\Sigma' = \Sigma \cup \{s_f, e_f\}$*
- *a set of states $Q' = Q_{E\|_\sigma C} \cup \{q_0{}', q'_f\}$*
- *a set of initial states $Q'_0 = \{q_0{}'\}$*
- *a set of clocks $X = \{x_\alpha \mid \alpha \in \mathcal{A}\} \cup \{x_{q_E} \mid q_E \in Q_E\} \cup \{x_u, x_f\}$*
- *a set of parameters $P = \{p_\alpha \mid \alpha \in \mathcal{A}\} \cup \{p_{q_E} \mid q_E \in Q_E\} \cup \{p_f\}$*
- *a set of final states $Q_f = \{q'_f\}$*
- *a state invariant $I$ defined as*

$$I(q) = \begin{cases} x_u \leq 0 & \text{if } q_C \text{ is transient} \\ \bigwedge_{\alpha \in \zeta(q)} x_\alpha \leq p_\alpha & \text{if } q_C \text{ is not transient} \\ & \text{and } \zeta(q) \neq \emptyset \\ x_{q_E} \leq p_{q_E} & \text{otherwise} \end{cases}$$

*for every state* $q = (q_E, q_C) \in Q_{E\|_\sigma C}$ *and* $I(q_0{}') = I(q'_f) = x_u \leq 0$

- *a set of edges* $\Theta$ *s.t.* $(q_1, \ell, q_2, \lambda, \mu) \in \Theta$ *iff either*
  - $(q_1 = (q_{E_1}, q_{C_1}), \ell, q_2 = (q_{E_2}, q_{C_2})) \in \Delta_{E\|_\sigma C}$ *and*
    *(*($q_{C_1}$ *is transient and has guard*

$$\mu = \begin{cases} x_u = 0 \wedge x_\alpha = p_\alpha & \text{if } \exists \alpha \in \mathcal{A}\ \ell \in Ends(\alpha) \\ x_u = 0 & \text{otherwise} \end{cases}$$

    *) or*
    *(*$q_{C_1}$ *is not transient and has guard*

$$\mu = \begin{cases} x_\alpha = p_\alpha & \text{if } \exists \alpha \in \mathcal{A}\ \ell \in Ends(\alpha) \\ x_{q_{E_1}} = p_{q_{E_1}} & \text{if } \zeta(s_1) = \emptyset \end{cases}$$

    *)) and reset clocks*

$$\lambda = \{x_u\} \cup \{x_{q_{E_2}}\} \cup \{x_\alpha | \exists \alpha \in \mathcal{A}\ \ell = Start(\alpha)\}$$

  - $q_1 \in Q_G$ *and* $q_2 = q'_f$ *and* $\ell = e_f$ *and has guard* $\mu = (x_f = p_f)$ *and reset clocks* $\lambda = \{x_u\}$
  - $q_1 = q_0{}'$ *and* $q_2 = q_{(E\|C)^\downarrow_{\sigma_0}}$ *and* $\ell = s_f$ *and has guard* $\mu = (x_u = 0)$ *and reset clocks* $\lambda = \{x_u, x_f\}$

The defined PTA features a clock $x_\alpha$ and parameter $p_\alpha$ for each activity $\alpha$. Each clock $x_\alpha$ measures the time elapsed from the start of the activity $\alpha$ to the end of it. The transitions that are start of activity reset the clock, and the transitions that are end of activity have a guard. This guard denotes that the transition can be taken, when the time elapsed by the clock of the activity is equal to its corresponding parameter ($p_\alpha = x_\alpha$). When the activity is being processed, there are invariants that restrict the clock of the activity to be greater than its parameter ($x_\alpha \leq p_\alpha$). This represents that the activity takes time $p_\alpha$. Similarly, there is a clock $x_{q_E}$ and parameter $p_{q_E}$ for each state $(q_E, q_C)$, in which all the outgoing transitions are uncontrollable actions and no activities are running. The clock $x_{q_E}$ measures the time spent at $(q_E, q_C)$, and $p_{q_E}$ stands for the total sojourn time. This models the time that the controller waits, when there are no

activities running. For instance, this happens in the initial state of the industrial automation example, when the controller is waiting for the product type to be processed. Finally, another clock $x_f$ and parameter $p_f$ are added to measure the end-to-end makespan. This clock works as an envelope that subsumes all the other clocks. That is to say, all the other parameters that appear in at least one path of the PTA are bounded by $p_f$. Besides, transient states, i.e., states that have controllable actions enabled, are forced to be abandoned in zero time through an invariant ($x_u < 0$). This is consistent with the definition of *transient states* in Section 6.2. Note that "$q_C$ is transient" refers to the enabled actions in $C$ without applying the scheduler ($\Delta^c_C(q_C) \neq \emptyset$). The final states are the goal states that appear in the scheduled composition.

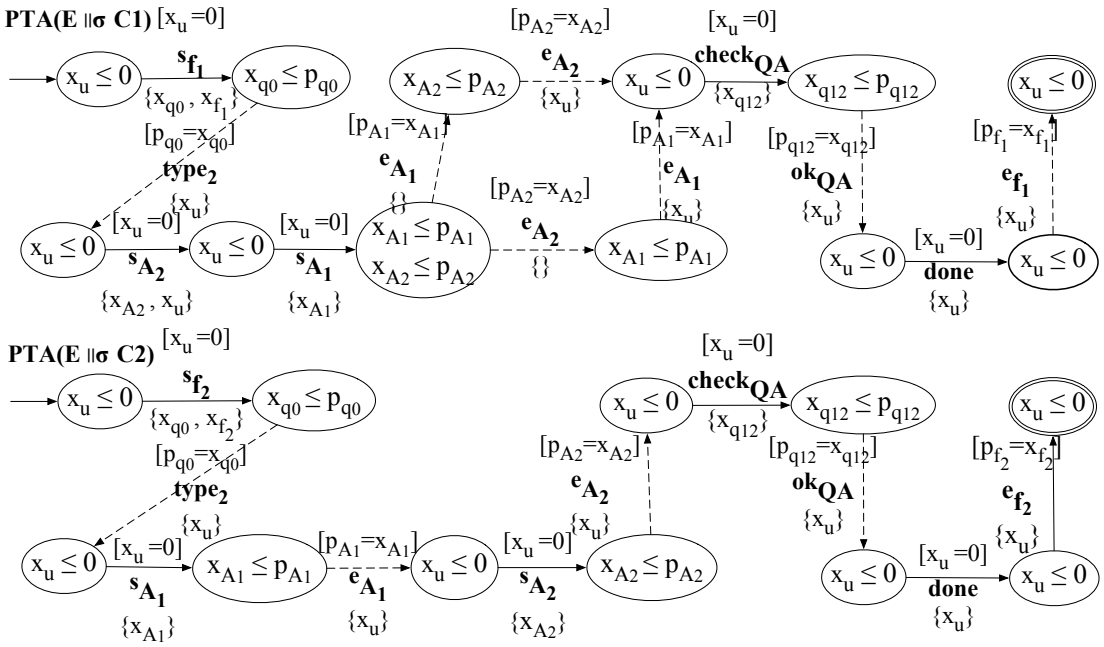



Figure 7.2: Timed semantics (PTAs) of the controllers and environment defined in Section 6.1 under the same scheduler $\sigma_1$ used in Figure 7.1.

In Figure 7.2, we show the timed semantics (PTAs) of the controller $C_1$ and $C_2$ (in Figure 6.4) composed in parallel with the environment $E$ under the same scheduler $\sigma_1$ used in Figure 7.1. Note that the clocks $x_{q_0}$ and parameters $p_{q_0}$ refer to a state of the environment $q_0 \in Q_E$, which is the initial state of the composition

$E$ of the LTSs from Figure 6.1. The other state $q_{12}$ corresponds to the state of the composition in which the LTS $QA$ is in the state 1, LTS *Choice* is in the state 2 and the other LTSs are in the initial state 0. We use the environment state to define the clocks and parameters in the states, because we want to model that two controllers have the same waiting time in the same state of the environment.

## 7.3 Using parameters to measure makespan

The parameters of the timed semantics (PTA) defined above represent the different activities and environment states that are on paths that reach a goal state from the initial state. These parameters can be instantiated into values, which models one possible duration of the activities. In PTA, a *parameter valuation* defines an assignment of values to the parameters. A parameter valuation has to satisfy certain constraints to produce a valid *timed automaton* (TA), when replacing the parameters of the PTA by the values that the parameter valuation defines. If a parameter valuation produces a valid TA, it means that those values are a possible duration for the activities to reach the goal. Thus, knowing which are the valid parameters for the duration of the activities is essential to know which are the possible values that the end-to-end parameter $p_f$ may take.

As mentioned in Chapter 3, there is a symbolic procedure to calculate the constraints of the parameters, which is defined in the literature as $\Gamma$. The general procedure [26] to compute $\Gamma$ is known to be undecidable, if there are more than three clocks in PTAs with cycles [25]. However, $\Gamma$ can be computed for the fragment of PTAs related to reachability controllers. The general procedure works symbolically through computing a fixed point of a precondition operator. Given that the PTAs generated are acyclic, the procedure terminates, and the fixpoint computation boils down into a one-pass backwards propagation of expressions. These expressions are changed at each edge by the precondition operator. Moreover, for the particular case of the PTAs from Definition 7.3, Algorithm 1 shows how to calculate $\Gamma$ by using the precondition operator shown in Definition 7.4. In Algorithm 1, we denote the edges from state q as $\Theta(q) = \{(q_1, \ell, q_2, \lambda, \mu) \mid (q_1, \ell, q_2, \lambda, \mu) \in \Theta \wedge q = q_1\}$.

[3]These PTAs have only one initial and one final state.

**Algorithm 1** Obtaining $\Gamma$ of the timed semantics of a controller (Definition 7.3)

1: **procedure** $\Gamma(H)$
2: $[q_0, \ldots, q_n, q_f] = \text{TOPOLOGICAL\_SORT}(H, Q_f)$
3: $\psi_{q_f} = True$[3]
4: **for** $q \in [q_n, \ldots, q_0]$ **do**
5: $\psi_q = \bigvee_{\varepsilon \in \Theta(q)} pre_\varepsilon(\psi_{q'})$
6: **return** $\psi_{q_0}[\forall x \in X(x := 0)]$[3]

The algorithm first initialises the formula $\psi_{q_f}$ of the final state with $True$ (line 3), and it traverses the other states in the inverse topological order while initialising the value of the formula in each state (line 4 - 5). The result of $\Gamma(H)$ is the formula of the initial state $\psi_{q_0}$ after resetting the clocks, i.e. replacing all the occurrences with the value 0. The precondition operator uses two sub-operators: transition-step $\triangleright$ and time-step $\Rightarrow$. The transition-step $\triangleright$ adds the guard $\mu$ to the formula and resets the clocks $x \in \lambda$ in $\psi$. The time-step $\Rightarrow$ adds the state invariant to the formula, but in the non-transient case it uses a new variable $\delta$ and an existential quantifier to represent the possible passage of time. The passage of time is done by replacing all the appearances of the clocks $x \in X$ in the formula $\psi$ and state invariant $I(q)$ with $x + \delta$. Note that the new variable $\delta$ will not appear in the formula produced by *pre*, because the algorithm applies quantifier elimination in each step to remove the existential quantifier.

**Definition 7.4 (Precondition Operator)** *Given an edge* $\varepsilon = (q_1, \ell, q_2, \lambda, \mu) \in \Theta$ *and the linear formula* $\psi$ *corresponding to the propagated conditions of the state* $q_2$*, the precondition operator* $pre_\varepsilon$ *is defined as follows:*

$$pre_\varepsilon(\psi) = (\Rightarrow (q_1, \triangleright(\varepsilon, \Rightarrow (q_2, \psi))))$$
$$\triangleright(\varepsilon, \psi) = (\mu \wedge \psi[\forall x \in \lambda \cdot x := 0])$$
$$\Rightarrow (q, \psi) = \begin{cases} \psi \wedge I(q) & \text{if } q \text{ is transient} \\ \exists \delta \geq 0 \; / & \text{otherwise} \\ (\psi \wedge I(q))[\forall x \in X(x := x + \delta)] & \end{cases}$$

For the PTAs in Definition 7.3, $\Gamma$ of the built PTA is a time constraint of the end-to-end makespan parameter as well as the parameters of the activities and environment states. $\Gamma$ is the relation among parameters that defines the

end-to-end makespan as a linear expression of time durations by using the parameters. For instance, for the PTAs in Figure 7.2, $\Gamma(PTA(E\|_\sigma C_1)) = ((p_{A_2} \geq p_{A_1} \wedge p_{f_1} = p_{A_2} + p_{q_0} + p_{q_{12}}) \vee (p_{A_1} \geq p_{A_2} \wedge p_{f_1} = p_{A_1} + p_{q_0} + p_{q_{12}}))$ whereas $\Gamma(PTA(E\|_\sigma C_2)) = (p_{f_2} = p_{A_1} + p_{A_2} + p_{q_0} + p_{q_{12}})$ are simplified versions of the symbolic expression that represent their makespan. The expression $\Gamma(PTA(E\|_\sigma C_1))$ states that, when the duration of activity $\alpha_1$ is greater than that of $\alpha_2$, the makespan ($p_{f_1}$) is the sum of durations of activity $\alpha_1$ plus the time spent in the states $q_0$ and $q_{12}$; otherwise, the makespan is the duration of activity $\alpha_2$ plus the time spent in the states $q_0$ and $q_{12}$. In contrast, $\Gamma(PTA(E\|_\sigma C_2))$ is the sum of the duration of the two activities plus the time spent in the states. Thus, no matter which values are assigned to the parameters, $p_{f_1}$ cannot be greater than $p_{f_2}$. Even though here it is easy to see, checking the existence of these parameters will require the use of an SMT-solver.

## 7.4 Comparing symbolic expressions

In the previous section we defined how to obtain $\Gamma$ of a controller when concurrently executed with the environment under a scheduler $\sigma$. Here we proceed to discuss how to compare controllers by using the obtained $\Gamma$. For the the PTAs in Figure 7.2, we explained that it was possible to find a set of values for the parameters that makes the end-to-end parameter $p_{f_1}$ of $\Gamma(PTA(E\|_\sigma C_1))$ be greater than the parameter $p_{f_2}$ of $\Gamma(PTA(E\|_\sigma C_1))$, i.e., $p_{f_2} > p_{f_1}$. When such a scheduler and a set of values for the parameters exist, we say that the controller $C_1$ shows a behaviour of higher makespan than $C_2$ (noted $C_2 \lhd C_1$). This means that there is a possible scenario in which the controller $C_1$ takes longer than the controller $C_2$. However, it is not possible to find an scheduler and a set of values that makes the controller $C_2$ show a behaviour of higher makespan than $C_1$.

**Definition 7.5 (Behaviour of Higher Makespan)** *Given two controllers $C_1$ and $C_2$, $C_1$ shows a behaviour of higher makespan (noted $C_2 \lhd C_1$) than $C_2$ iff* $\exists\sigma\ \exists(r_1..r_n, r_{f1}, r_{f2}) \in \mathbb{R}^{n+2}_{\geq 0}\ ((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge (r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f1} > r_{f2})$.

To compare two controllers, we evaluate their makespan under every possible

scheduler. We say that a controller $C_1$ shows a behaviour of higher makespan than $C_2$ ($C_2 \lhd C_1$), if there exists a parameter valuation and a scheduler such that the makespan of $C_1$ is strictly higher than that of $C_2$. That is to say, $C_1$ shows a behaviour of higher makespan than $C_2$ iff $\exists\sigma\ \exists(r_1..r_n, r_{f1}, r_{f2}) \in \mathbb{R}^{n+2}_{\geq 0}$ $((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge (r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f1} > r_{f2})$. It means that there exists at least one case, in which for the same scheduler $\sigma$ and assignment of values $(r_1, \ldots, r_n)$ to the parameters $(p_1 \ldots p_n)$, $C_1$ performs worse than $C_2$ ($r_{f1} > r_{f2}$). These parameters stand for activity durations and environment states that are common to the two controllers. The values $r_{f1}$ and $r_{f2}$ are assigned to the end-to-end makespan parameters $p_{f1}$ and $p_{f2}$ of the controllers $C_1$ and $C_2$ respectively. These two values are conditioned by the values $(r_1, \ldots, r_n)$.

We define the relationship between two controllers based on the $\lhd$ comparison in both ways, $C_1 \lhd C_2$ and $C_2 \lhd C_1$. Table 7.1 shows the four possible outcomes of this comparison. Two controllers are: i) *Incomparable* when there are circumstances in which both controllers show a behaviour of higher makespan than the other, ii) *Equivalent* when there is no scheduler that allows one controller to show a behaviour of higher makespan than the other, or iii) one *dominates* the other when one controller does not show a behaviour of higher makespan than the other, while the other shows a behaviour of higher makespan. Note that this comparison takes into account all possible schedulers of an environment.

| $C_2 \lhd C_1$ | $C_1 \lhd C_2$ | Conclusion |
|---|---|---|
| Sat | Sat | Incomparable |
| Unsat | Unsat | Equivalent ($C_1$ *as good as* $C_2$) |
| Sat | Unsat | $C_2$ dominates $C_1$ |
| Unsat | Sat | $C_1$ dominates $C_2$ |

Table 7.1: Possible results of comparing two controllers

## 7.5 Beyond the horizon

In this work, the horizon of the comparison framework is defined by the goal states, which are the states where the reachability proposition is satisfied for the first time. Typically, in these states there are no running activities. In this case, the duration of the activities is bounded by the main parameter. However, there are solutions to control problems in which there are running activities in the goal states. This means that some end of activity events may occur after reaching a goal state, or may never occur. This is possible because we are using LTS models that are over infinite traces, and the end of activity actions are uncontrollable. That is to say, there may be infinite traces in which the end of activity never occurs, i.e., some activities may not finish. Then, is it reasonable to define the comparison by using real numbers? What should be the duration of the activities when they do not finish? Section 7.5.1 presents an example in which this phenomena occurs, and Section 7.5.2 shows how to interpret the parameters of the activities that do not finish by using the comparison framework defined above.

### 7.5.1 Motivating example

Let us consider an example in which there are two activities $\alpha_1$ and $\alpha_2$. These activities start with actions $s_1$ and $s_2$, and finish with actions $e_1$ and $e_2$ respectively. The goal is to finish one of the two activities (i.e.: $\Diamond(\dot{e_1} \vee \dot{e_2})$). Figure 7.3 shows the LTSs that model this environment.

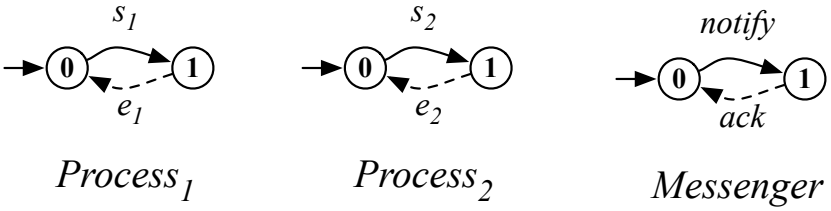



Figure 7.3: Environment model of two process activities and one communication process ($E = Process_1 \| Process_2 \| Messenger$ ).

One possible controller would be the one that starts doing both activities as soon as possible. Figure 7.4 illustrates the behaviour of this controller. Note that

in this figure, we are also modelling the behaviour after reaching the goal states. This controller first starts the activity $\alpha_1$, then starts activity $\alpha_2$ (if activity $\alpha_1$ did not finish immediately), and waits for the activities to finish. When one of the two activities finishes, the controller starts notifying that the goal was achieved. A possible trace in the execution of the controller with the environment is $s_1, s_2, e_2, notify, ack, notify, ack \ldots$. Then, there is an infinite trace in which the end of one of the two activities never occurs because other actions may occur infinitely many times. This can happen if do not have any fairness assumption. Note that the LTS shown in Figure 7.4 also illustrates $E \| C$.

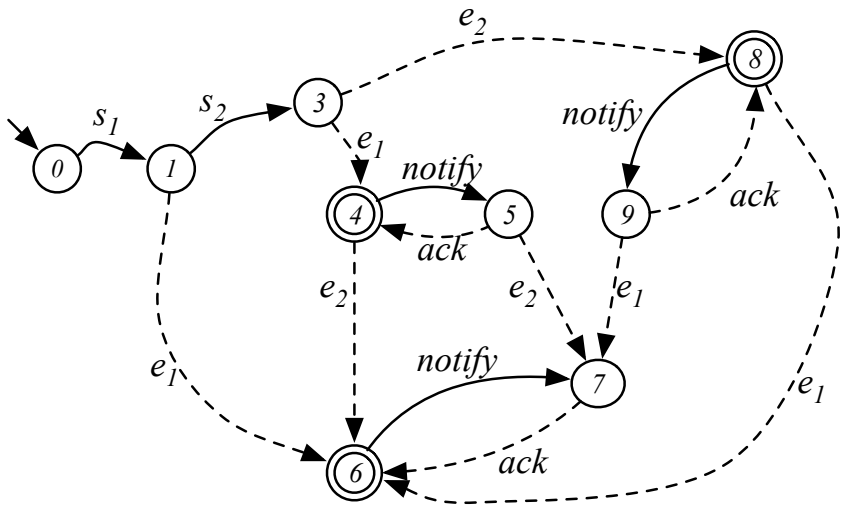



Figure 7.4: A possible controller executed with the environment from Figure 7.3 that satisfies the goal $\Diamond(\dot{e_1} \vee \dot{e_2})$. Goal states are states 4, 6 and 8.

To represent that an activity does not finish, we should permit parameter valuations that assign $\infty$ to the parameters of the activities. Consider a scheduler $\sigma$ that permits the controller to start the two activities. The consistent parameter valuations under this scheduler are $\Gamma^*(E\|_\sigma C_1) \equiv ((p_f = p_{\alpha_1} \wedge p_{\alpha_1} \leq p_{\alpha_2})) \vee (p_f = p_{\alpha_2} \wedge p_{\alpha_2} \leq p_{\alpha_1}) \wedge p_f \neq \infty)$, where $\Gamma^*$ is an extension of $\Gamma$ that allows parameter valuations to be $\infty$. The details about $\Gamma^*$ are explained in Section 7.5.2. Note that $p_f \neq \infty$ becomes a necessary parameter constraint to reach a goal state, which was implicit before. This condition will indirectly require to one of the activity parameters ($p_{\alpha_1}$ or $p_{\alpha_2}$) not to become infinity. However, there is no bound for the other activity parameter, which may not finish before reaching a goal state.

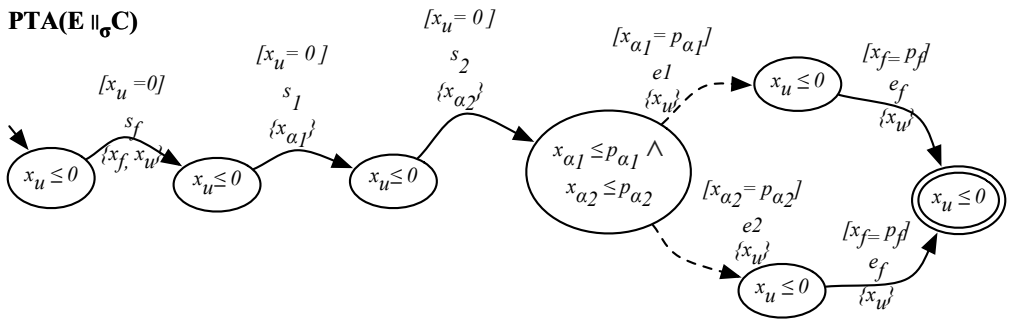



Figure 7.5: Timed semantics of the controller of Figure 7.4 when executed in the environment of Figure 7.3, under a scheduler that allows the controller to start the two activities.

Let us assume a valuation $\gamma^* : P \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ that assigns infinity to activity $\alpha_2$ and a real value to activity $\alpha_1$ ($\gamma^* : [p_f \leftarrow N, p_{\alpha_1} \leftarrow N, p_{\alpha_2} \leftarrow \infty]$ with $N \in \mathbb{R}_{\geq 0}$). This represents a valid trace in the execution of the controller in the environment that never takes the final event of the activity $\alpha_2$. Nonetheless, the parameter valuation with infinity cannot directly apply to a PTA because TA is defined over the reals without infinity. Thus, to apply this valuation of the parameters to the PTA, we need to pre-process the invariants and guards of the PTA. First, we produce an intermediate PTA by replacing the conditions where parameters assigned to infinity appear with conditions that do not involve these parameters. Then, we produce a valuation over the reals for the other parameters. Finally, we apply this parameter valuation to the intermediate PTA.

### 7.5.2 Interpreting the infinity in PTA

As mentioned above, the timed semantics defined in Section 7.2 does not allow parameters to be instantiated as infinity, because valuations of PTAs are defined over $\mathbb{R}_{\geq 0}$. However, the timed semantics is a particular case of PTA, which has invariants that are of the form $(\bigwedge x \leq p \wedge \bigwedge x \leq k)$ and guards of the form $(\bigwedge x = p \wedge \bigwedge x = k)$ with $x \in X$, $p \in P$ and $k \in \mathbb{R}_{\geq 0}$. Interpreting $\infty$ for these conditions is possible, but it requires some extra steps. Parameters in the invariants always appear as the upper bound of a clock. A clock will never reach the value of the parameter when the parameter becomes infinity. Thus, these conditions (e.g.: $x_{\alpha_2} \leq p_{\alpha_2}$) can be replaced by $True$, because it is true that the clock will never

reach the value $\infty$ (e.g.: $x_{\alpha_2} \leq \infty$). Parameters in the guards always appear as an equality with the clock. By using the same reasoning, this condition can never be satisfied. Thus, the parts of these conditions that have an equality with a parameter to become infinity (e.g.: $x_{\alpha_2} = p_{\alpha_2}$) are interpreted as $False$, because a clock cannot be equal to $\infty$ (e.g.: $x_{\alpha_2} = \infty$).

First, we need to extend the semantics of PTA valuations to support positive reals and infinity for this subset of PTAs. Given a timed semantics PTA $H = (\Sigma, Q, Q_0, X, P, Q_f, I, \Theta)$, an *infinity parameter valuation* $\gamma^* : P \to \mathbb{R}^+ \cup \{\infty\}$ is an assignment of values in $\mathbb{R}_{\geq 0} \cup \{\infty\}$ to the parameters in $P$. These infinity parameter valuations are only defined for the timed semantics. Allowing comparisons between infinite values for general PTAs may lead to critical problems. For instance, an inequality comparing a set of parameters may become infeasible if more than one parameter assigned to infinity by a valuation $\gamma^*$ (e.g.: $p_1 \leq p_3 * p_2$ and $\gamma^* = [p_1 \leftarrow \infty, p_2 \leftarrow \infty, p_3 \leftarrow \infty]$). However, the conditions of timed semantics PTAs are simpler. They are conjunctions of inequalities of the form $x \leq p$ or $x \leq 0$ (state invariants) and $x = p$ or $x = 0$ (guards) with no relationship between parameters. Thus, it is possible to define a language that accepts infinity as a valuation in terms of $\mathbb{R}_{\geq 0}$ by replacing some conditions.

To formalize this transformation, we define an *interpreter automata* and an *interpreter valuation*. The *interpreter automata* $\mathcal{I}(H)$ is a PTA that replaces parts of the invariants and guards which have a parameter assigned to infinity by a $\gamma^*$ with $True$ or $False$. The *interpreter valuation* is a parameter valuation over the reals for the parameters that are not assigned to $\infty$ by an infinity valuation $\gamma^*$.

**Definition 7.6 (Interpreter Automata)** *Given a PTA* $H = (\Sigma, Q, Q_0, X, P, Q_f, I, \Theta)$ *and an infinity parameter valuation* $\gamma^*$*, we define the interpreter automata* $\mathcal{I}(H) = (\Sigma, Q, Q_0, X, P', Q_f, I', \Theta')$ *as a PTA such that*

- $P' = \{p \mid p \in P \wedge \gamma^*(p) \neq \infty\}$,
- $\forall q \in Q : I'(q) = \nabla_{\gamma^*}(I(q))$, *and*
- $(q_1, \ell, q_2, \lambda, \nabla_{\gamma^*}(\mu)) \in \Theta' \Leftrightarrow (q_1, \ell, q_2, \lambda, \mu) \in \Theta$,

*where* $\nabla_{\gamma^*}(\psi)$ *is a function that recursively substitutes all the atoms that contain parameters that are assigned to infinity in* $\psi$ *as follows:*

$$\nabla_{\gamma^*}(\psi) = \begin{cases} \Xi_{\gamma^*}(x \bowtie p) \wedge \nabla_{\gamma^*}(\varphi) & \text{if } \psi \equiv x \bowtie p \wedge \varphi \\ x \bowtie k \wedge \nabla_{\gamma^*}(\varphi) & \text{if } \psi \equiv x \bowtie k \wedge \varphi \\ \Xi_{\gamma^*}(x \bowtie p) & \text{if } \psi \equiv x \bowtie p \\ x \bowtie k & \text{if } \psi \equiv x \bowtie k \end{cases}$$

$$\Xi_{\gamma^*}(x \bowtie p) = \begin{cases} True & \text{if } \gamma^*(p) = \infty \wedge \bowtie \in \{<, \leq\} \\ False & \text{if } \gamma^*(p) = \infty \wedge \bowtie \in \{>, \geq, =\} \\ x \bowtie p & \text{otherwise} \end{cases}$$

*where* $k \in \mathbb{R}_{\geq 0}$, $p \in P$ *and* $\bowtie \in \{<, \leq, >, \geq, =\}$.

**Definition 7.7 (Interpreter valuation)** *Given an infinity parameter valuation* $\gamma^* : P \to \mathbb{R}_{\geq 0} \cup \{\infty\}$, *we define the interpreter valuation of* $\gamma^{\mathcal{I}}$ *as a parameter valuation* $\gamma^{\mathcal{I}} : P' \to \mathbb{R}_{\geq 0}$ *s.t* $\forall p \in P' \gamma^{\mathcal{I}}(p) = \gamma^*(p)$, *where* $P' = \{p \mid p \in P \wedge \gamma^*(p) \neq \infty\}$.

Let us recall the motivating example. In Figure 7.6, we show the intermediate PTA obtained after replacing the parts of the conditions where the parameter $p_{\alpha_2}$ appeared with values $True$ and $False$, depending on how the parameter is used. We can see that a clock being less than or equal to infinity ($x_2 \leq \infty$) is interpreted as *True*, while a clock being equal ($x_2 = \infty$) is interpreted as *False*.

After interpreting the parameters that are assigned to infinity in the original PTA, we need to assign the rest of the values defined in the valuation $\gamma^*$ to the PTA from Figure 7.7. To this end, we define the interpreter valuation $\gamma^{\mathcal{I}} : [p_f \leftarrow N, p_{\alpha_1} \leftarrow N]$ as a valuation over the real numbers. The interpreter valuation is only defined for the parameters from $\gamma^*$ that are not assigned to $\infty$. Figure 7.7 shows the result of applying the $\gamma^{\mathcal{I}}$ to the PTA. In this Figure, the transitions and states that are not reachable were removed for clarity.

Analogously, we use $\Gamma^*(H)$ to describe the set of *infinity parameter valuations* that are consistent with a PTA $H$, and $\gamma^*(H)$ to represent the TA obtained from applying an *infinity parameter valuation*. The timed-behaviour defined by the timed automaton $\gamma^*(H)$ is equivalent to the timed automaton obtained after

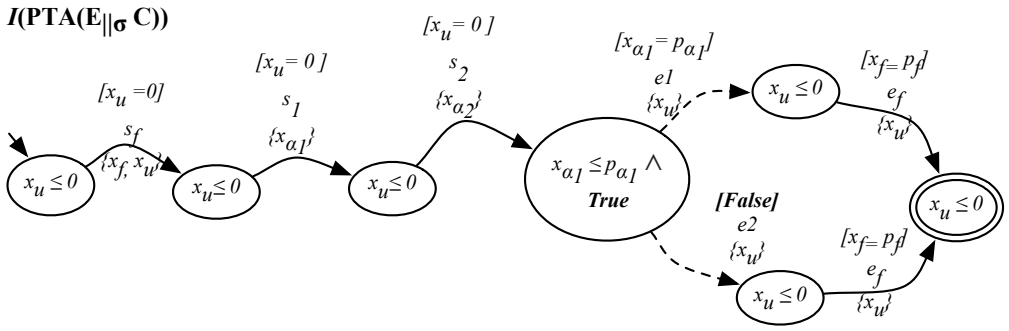



Figure 7.6: Interpreter automata obtained by replacing the parts of the conditions related to the parameter $p_{\alpha_2}$, which was assigned to $\infty$ by the valuation $\gamma^* : [p_f \leftarrow N, p_{\alpha_1} \leftarrow N, p_{\alpha_2} \leftarrow \infty]$ with $N \in \mathbb{R}_{\geq 0}$.

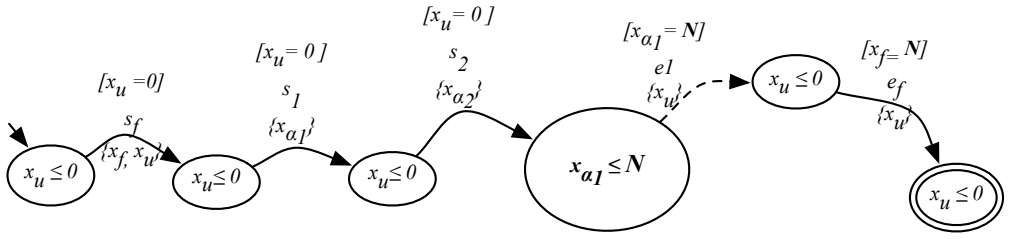



Figure 7.7: Timed Automaton obtained after replacing the values of the Intermediate PTA from Figure 7.6 with the value $N$. This was the value defined by the parameter valuation $\gamma^{\mathcal{I}} : [p_f \leftarrow N, p_{\alpha_1} \leftarrow N]$ with $N \in \mathbb{R}_{\geq 0}$.

substituting each non-infinity parameter of the interpreter valuation $\gamma^{\mathcal{I}}$ in the PTA $\mathcal{I}(H)$ as described above (i.e.: $\gamma^{\mathcal{I}}(\mathcal{I}(H))$).

This interpretation may also be used for general PTAs that have conditions and invariants similar to the ones of the timed semantics. In Figure 7.8 we show an example of a PTA $H$ that has the following restriction over the parameters $\Gamma(H) \equiv ((p_1 \leq p_2 \wedge p_1 = p_3) \vee (p_2 \leq p_1 \wedge p_2 = p_3))$. Given a valuation over the reals $\gamma$, the trace $abd^*$ would not be a possible trace in $H$, because the clock $x_2$ must eventually reach $p_2$ and take the transition $c$. However, by using the infinity consistent parameter valuations $\Gamma^*(H) \equiv ((p_1 \leq p_2 \wedge p_1 = p_3) \vee (p_2 \leq p_1 \wedge p_2 = p_3) \wedge (p_3 \neq \infty))$, the trace $abd^*$ becomes a possible trace in $H$, because the infinity parameter valuation $\gamma^* = [p_1 \leftarrow 5, p_2 \leftarrow \infty, p_3 \leftarrow 5]$ is characterised by $\Gamma^*(H)$. In Figure 7.9, we show the interpreted automata $\mathcal{I}(H)$, which is the result of replacing the conditions that have parameters that are mapped to infinity ($p_2 \leq \infty$

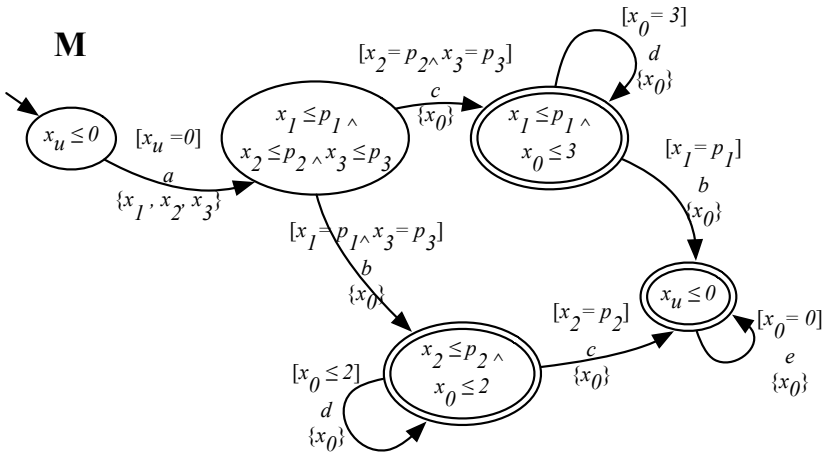



Figure 7.8: PTA that does not accept $abd^*$, when considering valuations over the reals.

is replaced by $True$ and $p_2 = \infty$ by $False$). This interpretation may also be of interest when giving a timed semantics to an LTS without pruning the outgoing transitions of the goal states, but this is out of the scope of this work.

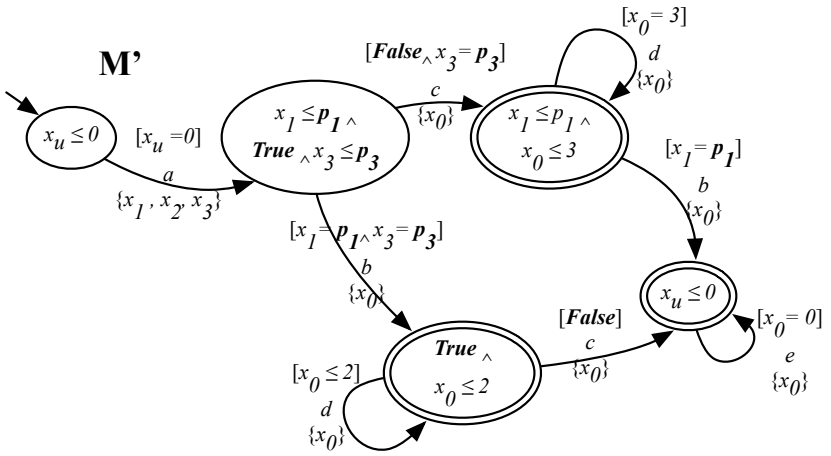



Figure 7.9: Interpreting the infinity parameters valuation $\gamma^* = [p_1 \leftarrow 5, p_2 \leftarrow \infty, p_3 \leftarrow 5]$ on the PTA from Figure 7.8.

# 8 A Makespan-Minimising Controller

In this chapter we formalize the concept of makespan-minimising controller and define an algorithm to produce a solution that is makespan-minimising.

## 8.1 Defining makespan-minimising controllers

A quantitative approach would define a controller with minimum makespan as the one that can reach goal in the least time possible, where the time is measured by a number that typically represents the average run. In this work, we define a *makespan-minimising controller* in a qualitative manner. As briefly explained in Section 5.6, the definition of makespan-minimising controller is based on the *dominance* relationship defined in Section 7.4. A controller $C$ is makespan-minimising if it is a *non-dominated* controller. In other words, a controller is makespan-minimising if there is no other controller $C'$ that *dominates* the controller $C$. This is formalised in the Definition 8.1.

**Definition 8.1 (Non-dominated)** *Given a control problem with activities* $\mathcal{E} = \langle E, \mathcal{AD}, G, \Sigma_c \rangle$ *and the set of solutions* $\mathcal{C}$ *for* $\mathcal{E}$*, we say that* $C \in \mathcal{C}$ *is non-dominated iff* $\nexists C' \in \mathcal{C}$ ($C'$ *dominates* $C$).

Note that when a control problem is realisable, there is at least one non-dominated controller. Without loss of generality, we restrict to the subset of memoryless solutions [66], because solutions that have additional memory would also have additional makespan. Since this set is finite, it is not possible for all of them to be dominated by another controller. This is because $C$ *dominates* $C'$ is a transitive and asymmetric relation. Also note that there can be more than one non-dominated controller.

**Lemma 8.2 (Transitivity)** *Given controllers* $C_1, C_2, C_3$, $C_1$ *dominates* $C_2 \wedge$ ($C_2$ *dominates* $C_3 \vee C_2$ *as good as* $C_3$) $\iff$ $C_1$ *dominates* $C_3$.

**Proof.** It can be proven from definition of behaviour of higher makespan (Definition 7.5), *dominates* and *as good as* (Table 7.1).

1. $C_1$ *dominates* $C_2 \iff$ (Definition 7.5 and Table 7.1)
   *a.* $\exists\sigma\ \exists(r_1..r_n, r_{f1}, r_{f2}) \in \mathbb{R}^{n+2}_{\geq 0}$
   $((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge$
   $(r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f1} < r_{f2}) \wedge$
   *b.* $\forall\sigma\ \forall(r_1..r_n, r_{f1}, r_{f2}) \in \mathbb{R}^{n+2}_{\geq 0}$
   $((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge$
   $(r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f1} \leq r_{f2})$
2. $C_2$ *dominates* $C_3 \iff$ (Definition 7.5 and Table 7.1)
   *a.* $\exists\sigma\ \exists(r_1..r_n, r_{f2}, r_{f3}) \in \mathbb{R}^{n+2}_{\geq 0}$
   $((r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge$
   $(r_1..r_n, r_{f3}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f2} < r_{f3}) \wedge$
   *b.* $\forall\sigma\ \forall(r_1..r_n, r_{f2}, r_{f3}) \in \mathbb{R}^{n+2}_{\geq 0}$
   $((r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge$
   $(r_1..r_n, r_{f3}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f2} \leq r_{f3})$
3. $C_2$ *as good as* $C_3 \iff$ (Definition 7.5 and Table 7.1)
   *a.* $\forall\sigma\ \forall(r_1..r_n, r_{f2}, r_{f3}) \in \mathbb{R}^{n+2}_{\geq 0}$
   $((r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge$
   $(r_1..r_n, r_{f3}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f2} = r_{f3})$

4. $C_1$ *dominates* $C_2 \wedge C_2$ *dominates* $C_3 \iff$ (1 and 2.b)
   *a.* $\exists\sigma\ \exists(r_1..r_n, r_{f1}, r_{f3}) \in \mathbb{R}^{n+2}_{\geq 0}$
   $((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge$
   $(r_1..r_n, r_{f3}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f1} < r_{f3}) \wedge$
   *b.* $\forall\sigma\ \forall(r_1..r_n, r_{f1}, r_{f3}) \in \mathbb{R}^{n+2}_{\geq 0}$
   $((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge$
   $(r_1..r_n, r_{f3}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f1} \leq r_{f3})$
   $\iff$ (Definition 7.5 and Table 7.1) $C_1$ *dominates* $C_3$
5. $C_1$ *dominates* $C_2 \wedge C_2$ *as good as* $C_3 \iff$ (1 and 3.a)
   *a.* $\exists\sigma\ \exists(r_1..r_n, r_{f1}, r_{f3}) \in \mathbb{R}^{n+2}_{\geq 0}$
   $((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge$
   $(r_1..r_n, r_{f3}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f1} < r_{f3}) \wedge$
   *b.* $\forall\sigma\ \forall(r_1..r_n, r_{f1}, r_{f3}) \in \mathbb{R}^{n+2}_{\geq 0}$
   $((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge$
   $(r_1..r_n, r_{f3}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f1} \leq r_{f3}))$
   $\iff$ (Definition 7.5 and Table 7.1) $C_1$ *dominates* $C_3$
6. $C_1$ *dominates* $C_2 \ \wedge\ (C_2$ *dominates* $C_3 \ \vee\ C_2$ *as good as* $C_3)$
   $\iff$ (4 and 5) $C_1$ *dominates* $C_3$

□

**Lemma 8.3 (Asymmetry)** *Given controllers* $C_1$ *and* $C_2$, $C_1$ *dominates* $C_2 \implies \neg(C_2$ *dominates* $C_1)$.

**Proof.** It can be proven by contradiction. Assume $C_2$ *dominates* $C_1$.

1. $C_1$ *dominates* $C_2 \implies$ (Definition 7.5 and Table 7.1)
   *a.* $\exists\sigma\ \exists(r_1..r_n, r_{f1}, r_{f2}) \in \mathbb{R}^{n+2}_{\geq 0}$
   $((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge$
   $(r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f1} < r_{f2}) \wedge$
   *b.* $\forall\sigma\ \forall(r_1..r_n, r_{f1}, r_{f2}) \in \mathbb{R}^{n+2}_{\geq 0}$
   $((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge$
   $(r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f1} \leq r_{f2})$
2. $C_2$ *dominates* $C_1 \implies$ (Definition 7.5 and Table 7.1)
   *a.* $\exists\sigma\ \exists(r_1..r_n, r_{f2}, r_{f1}) \in \mathbb{R}^{n+2}_{\geq 0}$
   $((r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge$

$(r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f2} < r_{f1}) \wedge$
*b.* $\forall\sigma\ \forall(r_1..r_n, r_{f2}, r_{f1}) \in \mathbb{R}^{n+2}_{\geq 0}$
$((r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge$
$(r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f2} \leq r_{f1})$

3. There is a contradiction between 1.b. and 2.a.. Then, $\neg(C_2$ *dominates* $C_1)$.

□

## 8.2 Synthesising makespan-minimising controllers

There are algorithms for controller synthesis problems that produce memoryless controllers [66, 64, 68]. Such algorithms have linear complexity in the size of the problem, and they build controllers in the form of directed acyclic graphs (DAG), which are subgraphs of the given problem. These controllers are universal in the sense that any other memoryless controller is a subgraph of the universal controller.

**Algorithm 2** Non-dominated controller algorithm.

```
1: procedure NON_DOMINATED(E, 𝒟, G, Σ_c)
2:     𝒰 = UNIVERSAL_SYNTHESIS(E, Σ_c, G)
3:     K = E‖𝒰
4:     [q_0, ..., q_n] = TOPOLOGICAL_SORT(K, Q_G)
5:     for q ∈ sorted([q_n, ..., q_0]) do
6:         Alt = {{c} | c ∈ Δ^c_K(q)}
7:         if Δ^u_K(q) ≠ ∅ then Alt= Alt∪{∅}
8:         if |Alt|> 1 then
9:             𝒟 = ∅
10:            for A ∈ Alt ∧ A' ∈ Alt \ {A} do
11:                if K_⟨q,A'⟩ dominates K_⟨q,A⟩ then 𝒟 = 𝒟 ∪ A
12:            Δ_K = Δ_K \ {(q, c, q') | (q, c, q') ∈ Δ_K ∧ c ∈ 𝒟}
    return K
```

Algorithm 2 produces a non-dominated controller. This algorithm first obtains a universal controller. Then, it traverses the states of the universal controller in the inverse topological order. In the states that have more than one controllable action enabled, the algorithm uses the dominance comparison to disable some of the enabled controllable actions. The dominance comparison initially only

compares two controllers from the initial state. However, by defining a sub-LTS (Definition 8.4), the comparison can be made from any states, not limiting to the initial state. It then generates the set of alternatives $Alt$ of a state. An alternative can be one of the controllable actions $c \in \Delta^c(q)$ or $\emptyset$ that represents waiting for uncontrollable events to happen. It compares an alternative $A \in Alt$ against all the other alternatives $A'_1 \dots A'_n \in Alt$, by comparing sub-LTS $M_{\langle q,A\rangle}$ against sub-LTSs $M_{\langle q,A'_1\rangle} \dots M_{\langle q,A'_n\rangle}$. If any of the other alternatives in $M_{\langle q,A'_1\rangle} \dots M_{\langle q,A'_n\rangle}$ dominates $M_{\langle q,A\rangle}$, it removes the controllable transition of alternative $A$ from $\Delta_K$.

**Definition 8.4 (sub-LTS)** *Given an LTS $M = (Q, \Sigma, \Delta, q_0)$, a state $q \in Q$ and $A \subseteq \Delta^c(q)$, we define $M_{\langle q,A\rangle} = (Q, \Sigma, \Delta', q)$ as a sub-LTS of $M$, whose initial state is $q$, and the transition function as $\Delta' = \Delta \setminus \{(q, \ell', q') \mid (q, \ell', q') \in \Delta^c(q) \wedge \ell' \notin A\}$.*

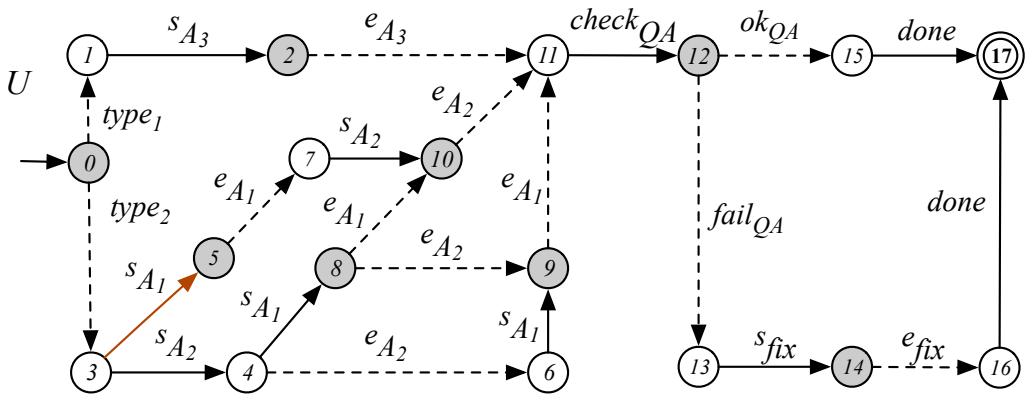



Figure 8.1: Universal controller for the example of Section 6.1. The red transition is the transition removed by the Algorithm 2.

Figure 8.1 shows the universal controller of the example presented in Section 6.1. Note that $K = E \| \mathcal{U}$ is structurally equivalent to $\mathcal{U}$. In this example, the algorithm traverses the states in the order specified in the states of the figure. Here, the states with more than one alternative are only two. One is the state 4 which has alternatives $\emptyset$ and $\{s_{A_1}\}$. Figure 8.2 illustrates these two subLTSs. Since the alternative of waiting for $e_{A_2}$ does not dominate the one that enables the action $s_{A_1}$ (i.e. $K_{\langle q_3,\{s_{A_1}\}\rangle}$ *dominates* $K_{\langle q_3,\emptyset\rangle}$), the algorithm does not remove any controllable transition here.

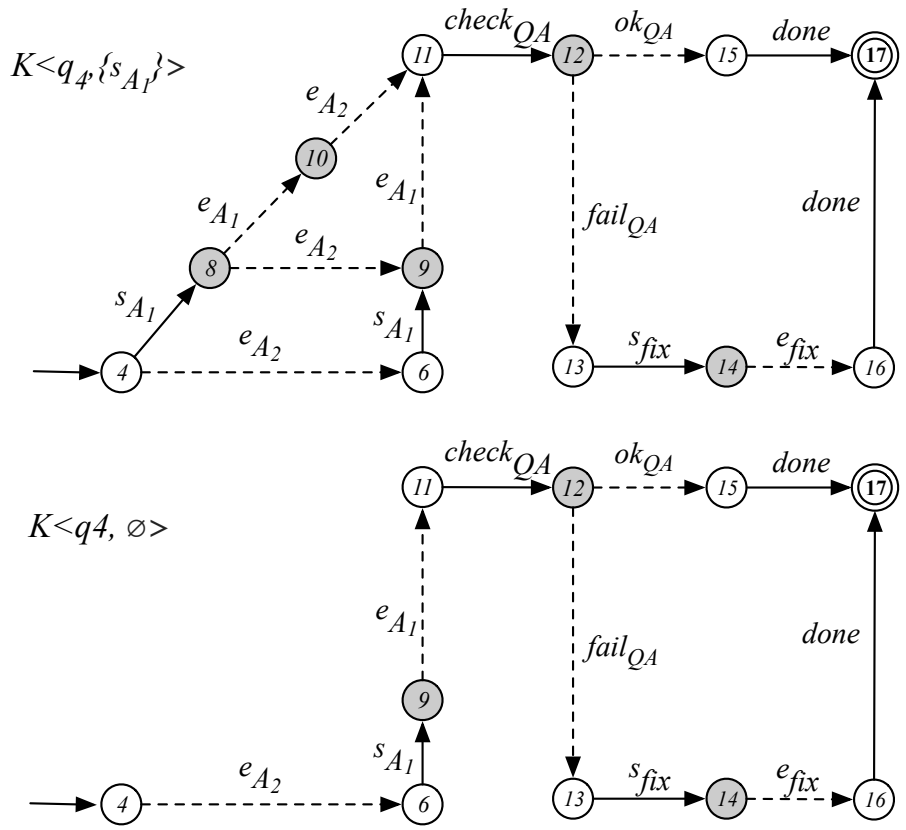



Figure 8.2: SubLTSs of the universal controller from the state $q_4$.

The other state in which there are alternatives is the state 3. In these case, there are also two subLTSs which are shown in Figure 8.3. Here, the alternative $\{s_{A_1}\}$ is dominated by the alternative $\{s_{A_2}\}$ ($K_{\langle q_4,\{s_{A_2}\}\rangle}$ *dominates* $K_{\langle q_4,\{s_{A_1}\}\rangle}$). Then, the algorithm removes the transition $(q_3, s_{A_1}, q_5)$ from the universal controller, which is shown in red in Figure 8.1. Finally, the resulting controller would be equivalent to $C_1$ of Figure 6.4, if we consider only the states that are reachable from the initial state.

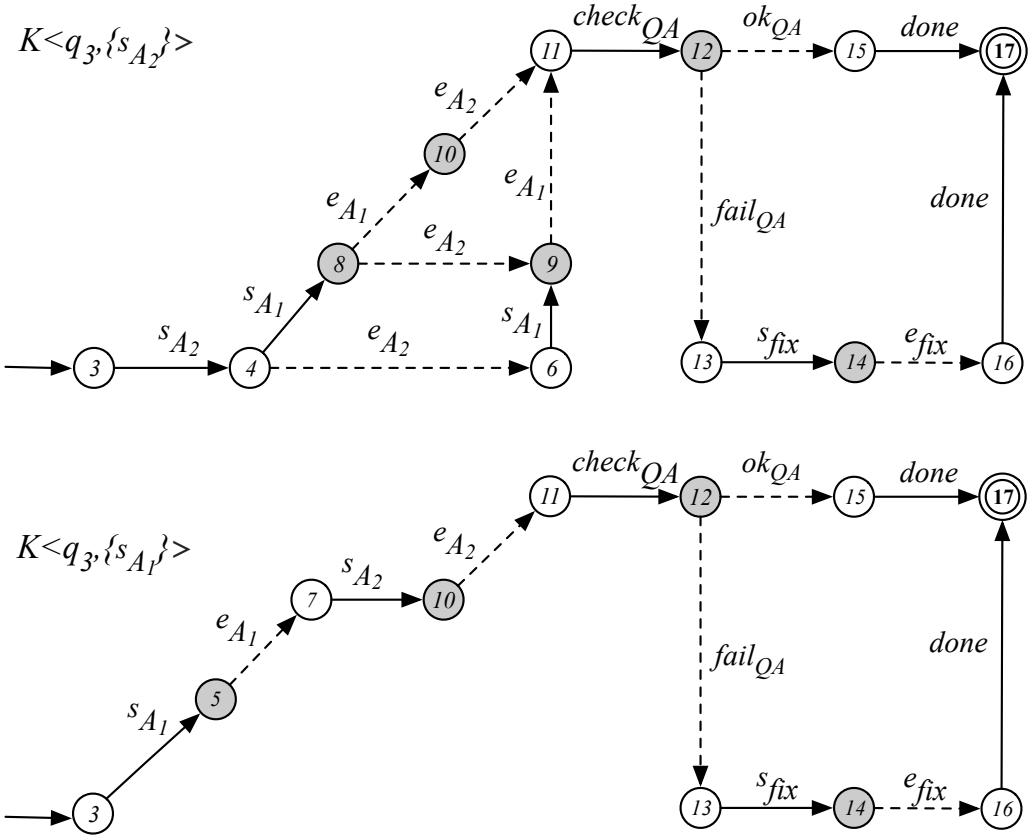



Figure 8.3: SubLTSs of the universal controller from the state $q_3$.

The following theorem shows that the Algorithm 2 produces a controller $K$ that is non-dominated.

**Theorem 8.5 (K is non-dominated)** *Given an activity control problem* $\mathcal{E} = \langle E, \mathcal{AD}, G, \Sigma_c \rangle$*, a* $K = \text{NON\_DOMINATED}(E, \mathcal{AD}, G, \Sigma_c)$*, and the set of solutions* $\mathcal{C}$ *for* $\mathcal{E}$*,* $K$ *is a controller and* $\nexists C \in \mathcal{C}$ *(*$C$ *dominates* $K$*).*

**Proof.**

i) The algorithm produces a controller. This comes from the fact that it is not possible to have a circular relationship of dominance. Thus, the algorithm can never remove all the transitions of a state.

ii) The non-dominance can be proved by induction on the topological order. The inductive step requires reasoning on properties which relate valuations satisfying

$\Gamma$ in the sub-LTS of a state to valuations satisfying $\Gamma$ of successor sub-LTSs. Universality of $\mathcal{U}$ is also required to prove that all memoryless controllers are considered. Also note that all non-dominated memoryless controllers are included in the result yielded by the algorithm. □

# 9

# Evaluation

In this chapter, we report the details of the implementation of Algorithm 2 and some of the experiments done. The algorithm was implemented in an extension of MTSA [28] that uses Z3 [69] as SMT-Solver, which is necessary in the dominance comparison. MTSA is a tool to specify discrete-event controls problems described in FLTL and LTS. We extended this tool to specify control problems with activities. The implementation of Algorithm 2 was validated on case studies from the literature. The evaluation consists in comparing the result produced by the Algorithm 2 against the one produced by the standard algorithm of MTSA. In addition, the synthesis time of the algorithm was evaluated on different instances of the job scheduling case study, which is one of the examples used in the previous experiment. In those case studies, the activities are inferred from the actions that informally denoted start and end of activity.

## 9.1 Extending MTSA

In this section, we first show how to specify control problems in the MTSA Tool. Then, we explain how we extended the tool to specify control problems with activities. In particular, we show how the syntax of the specification language is extended and how to compare the makespan of two controllers by using Z3.

### 9.1.1 MTSA

MTSA [28] is a tool to specify qualitative control problems described with LTSs and FLTL formulas. The tool is written in Java language and has an graphical interface to define control problems. The interface can be used to specify problems, to visualize the controllers produced by the tool, and to enact the controllers [70] on robots, by previously defining the interface with the robot. Figure 9.1 shows the interface of MTSA displaying the specification of the industrial automation example.

In MTSA, control problems are specified in a simple process algebra notation called Finite State Process (FSP) [71]. FSP specification language allows for a compact textual representation of LTSs. In Figure 9.1 we can see the *LTS definitions* of the industrial automation example. For instance, the *QA* LTS illustrated in Figure 6.1 is described as QA = (query -> (fail -> QA | ok -> QA). The different processes can be composed into another process by using the standard parallel composition. This is shown in the last line under *LTS definitions* as ||Environment = (PROCESS(1) || PROCESS(2) || PROCESS(3) || FIX || QA || PROD). To make the specification clearer, some ranges and sets are defined at the top of the file. Those definitions are not necessary, but can be used to specify processes and formulas in a more compact way. They are also helpful in reducing the number of errors when specifying control problems in the tool. The different components can be visualized in the tool for users validation. Figure 9.2 shows the tool displaying the different components.

Fluents and FLTL properties are defined in a syntax similar to the one used in Section 6.1. In the safety properties definitions, the symbol [] represents the temporal operator $\Box$. The symbols &&, ||, ! and $-$ > represent the logical

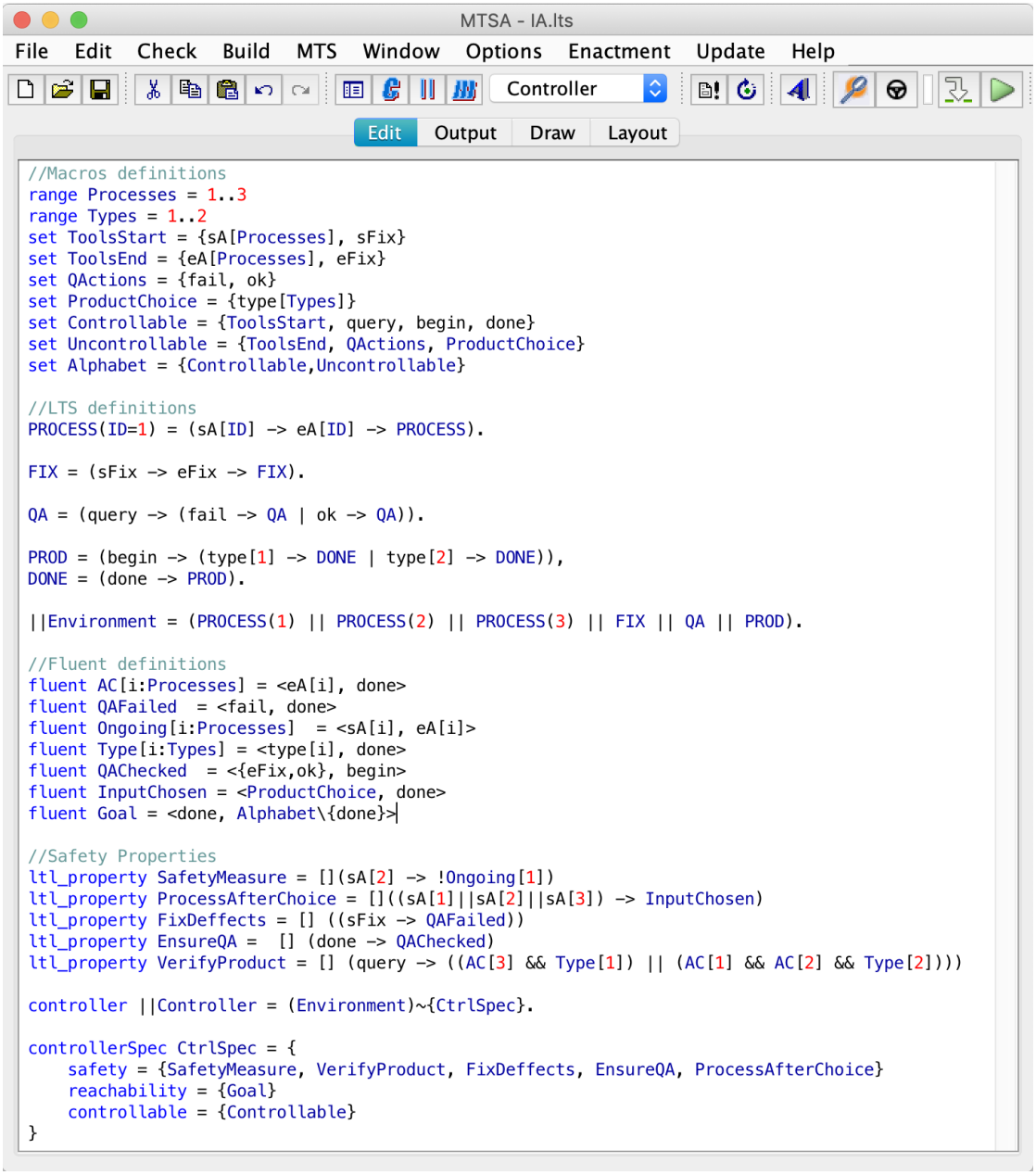



Figure 9.1: Interface of MTSA displaying the specification of the industrial automation example.

operators $\wedge$, $\vee$, $\neg$ and $\Longrightarrow$ respectively. These properties are equivalent to the ones shown when defining the control problem with activities.

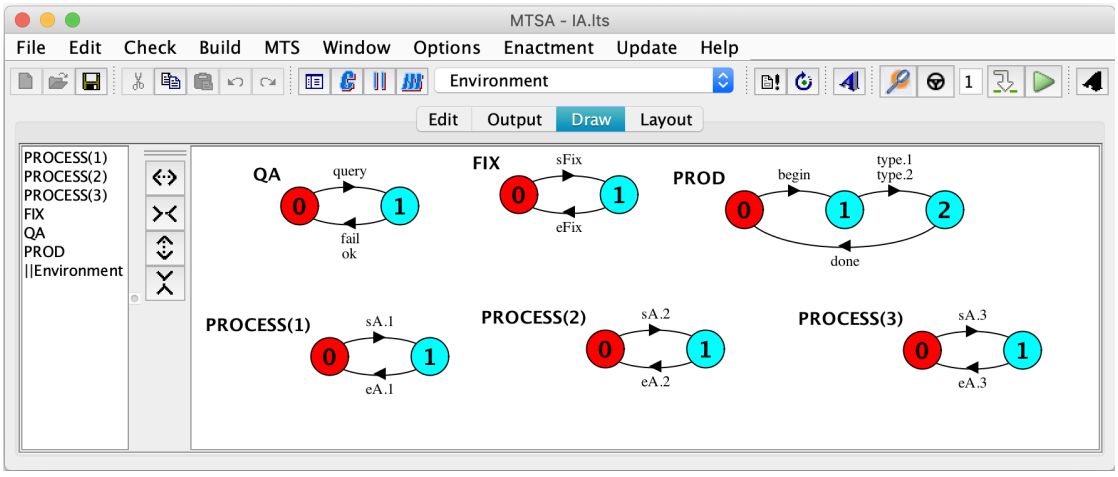



Figure 9.2: MTSA displaying the components of the environment of the industrial automation example.

To generate a controller we need an LTS representing the environment and the controller specification (*controllerSpec*). The specification includes a list of safety propositions, the set of controllable actions, and the reachability goal, which is defined as a fluent. Then, the controller is defined as controller || Controller = (Environment)~{CtrlSpec}. If the problem is realisable, the controller can synthesise a solution, which can be visualized as in Figure 9.3.

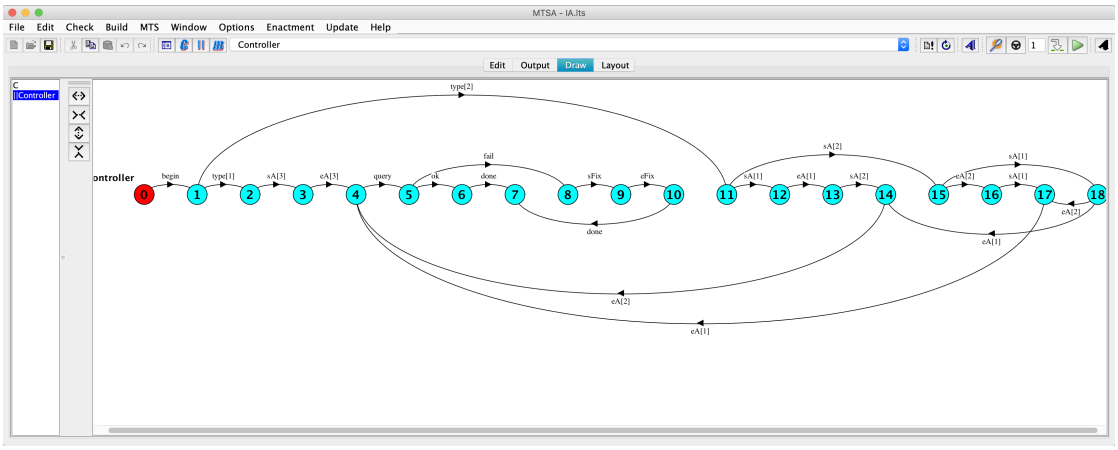



Figure 9.3: Visualization of the controller synthesised by the standard synthesis algorithm of MTSA.

### 9.1.2 Specifying control problems with activities in MTSA

We extended the definition of controller specification to define control problems with activities. Figure 9.4 shows the MTSA interface with the additional inputs. The controller specification has now two optional attributes: *algorithm* and *activityDefinitions*. The keyword algorithm enables the possibility of using a different algorithm to synthesise a controller, which in our case is a non-dominated controller (*NON_DOMINATED*). The keyword *activityDefinitions* takes as input a set of activities, which are defined as fluents in the tool. By doing that, we enforce the first rule to be activity compatible.

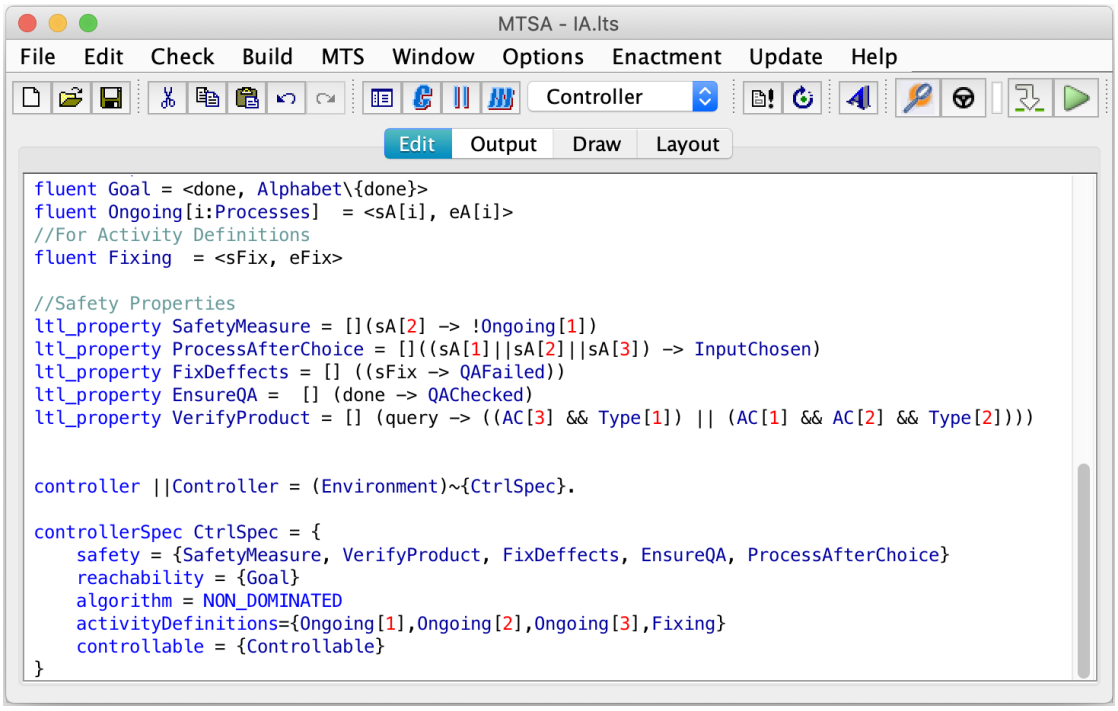



Figure 9.4: Interface of MTSA displaying the lower part of the specification of the industrial automation example with the activities definition and a keyword to enable Algorithm 2.

Figure 9.4 shows that the only extra requirements to use our approach are to specify one extra activity as a fluent, which is *Fixing*, and to add the keyword algorithm = NON_DOMINATED and the activities definition in the controller specification.By adding this extra information the tool can synthesise the controller

shown in Figure 9.5, which does not have the sequential alternative on products of type two. In state $q_{10}$ the only controllable action enabled is sA[2], while in the corresponding state of Figure 9.5, which is state $q_{11}$, the action sA[1] is also enabled. Note that the numbers shown in the states are automatically created by the tool for visualization, but do not have any correlation to the original states of the control problem. In Section 9.2.1 we explain why this solution is better than the standard one.

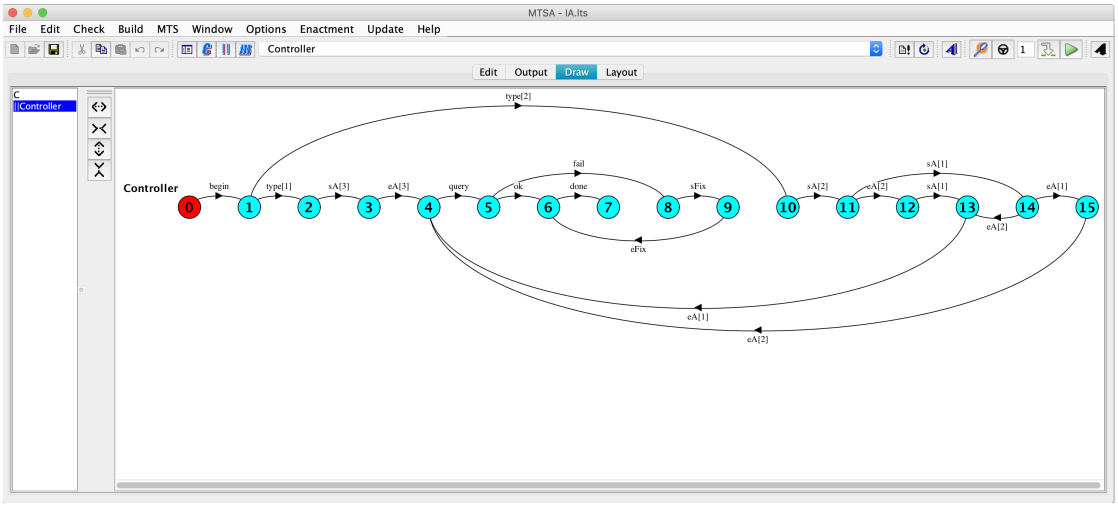



Figure 9.5: Visualization of the controller synthesised by the Algorithm 2.

### 9.1.3 Implementing the comparison framework

As defined in Section 7.4, comparing two controllers qualitatively requires to compare two controllers under every possible scheduler. In the implementation, we separated the problem of generating schedulers from the problem of finding a possible set of values to the parameters. The schedulers are generated from the LTS model of the environment. For each state of the LTS, we define the possible candidates according to the Definition 7.1. Then, we combine them to generate different schedulers. Since the number of states is finite, the number of schedulers is also finite. Given a scheduler $\sigma$, we say that a controller has a behaviour of higher makespan under the scheduler $\sigma$ as $C_1 \lhd_\sigma C_2 : \exists (r_1..r_n, r_{f1}, r_{f2}) \in \mathbb{R}^{n+2}_{\geq 0}$

$((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge (r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f1} > r_{f2})$. Similarly, we define the outcome of the comparison for a particular $\sigma$ based on the bidirectional comparison of behaviour of higher makespan ($C_1 \lhd_\sigma C_2$ and $C_2 \lhd_\sigma C_1$). Table 9.1 shows the four possible outcomes of this bidirectional comparison. We say that they are *equivalent* when they perform the same for every possible value assignment to the parameters, and that two controllers are *incomparable* when it is possible to find a set of values that shows a behaviour of higher makespan in both cases. For instance, two controllers would be incomparable if they perform different activities to reach a goal state.

| $C_2 \lhd_\sigma C_1$ | $C_1 \lhd_\sigma C_2$ | Conclusion (under $\sigma$) |
|---|---|---|
| Sat | Sat | Incomparable ($C_1 \not\approx_\sigma C_2$) |
| Unsat | Unsat | Equivalent ($C_1 \approx_\sigma C_2$) |
| Sat | Unsat | $C_2$ is better than $C_1$ ($C_2 \preceq_\sigma C_1$) |
| Unsat | Sat | $C_1$ is better than $C_2$ ($C_1 \preceq_\sigma C_2$) |

Table 9.1: Possible results of comparing two controllers under a scheduler $\sigma$.

Figure 9.6 shows the comparison framework for two controllers. As mentioned above, the framework compares two controllers under every possible scheduler. This means that for each scheduler $\sigma \in \{\sigma_1 \dots \sigma_k\}$, there are two queries to be resolved. These queries are $C_2 \lhd_\sigma C_1 : \exists(r_1..r_n, r_{f1}, r_{f2}) \in \mathbb{R}^{n+2}_{\geq 0}$ $((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge (r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f1} > r_{f2})$ and $C_1 \lhd_\sigma C_2 : \exists(r_1..r_n, r_{f1}, r_{f2}) \in \mathbb{R}^{n+2}_{\geq 0}$ $((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge (r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f2} > r_{f1})$. Given a scheduler $\sigma$, the comparison framework resolves $C_2 \lhd_\sigma C_1$ by using Z3 SMT-solver [69]. The SMT-Solver is able to determine if these expressions are satisfiable or not. Besides, when the expression is satisfiable, it can also return a possible assignment of values to the parameters. By aggregating the result of the individual queries to the SMT-Solver, it is possible to determine that *i*) a controller $C_1$ dominates $C_2$ if $(\forall\sigma\ C_1 \preceq_\sigma C_2 \vee C_1 \approx_\sigma C_2) \wedge (\exists\sigma\ C_1 \preceq_\sigma C_2)$, *ii*) that they are equivalent if $\forall\sigma C_1 \approx_\sigma C_2$, or *iii*) that they are incomparable if $(\exists\sigma C_1 \not\approx_\sigma C_2) \vee (\exists\sigma\exists\sigma'\ C_1 \preceq_\sigma C_2 \wedge C_2 \preceq_{\sigma'} C_1)$.

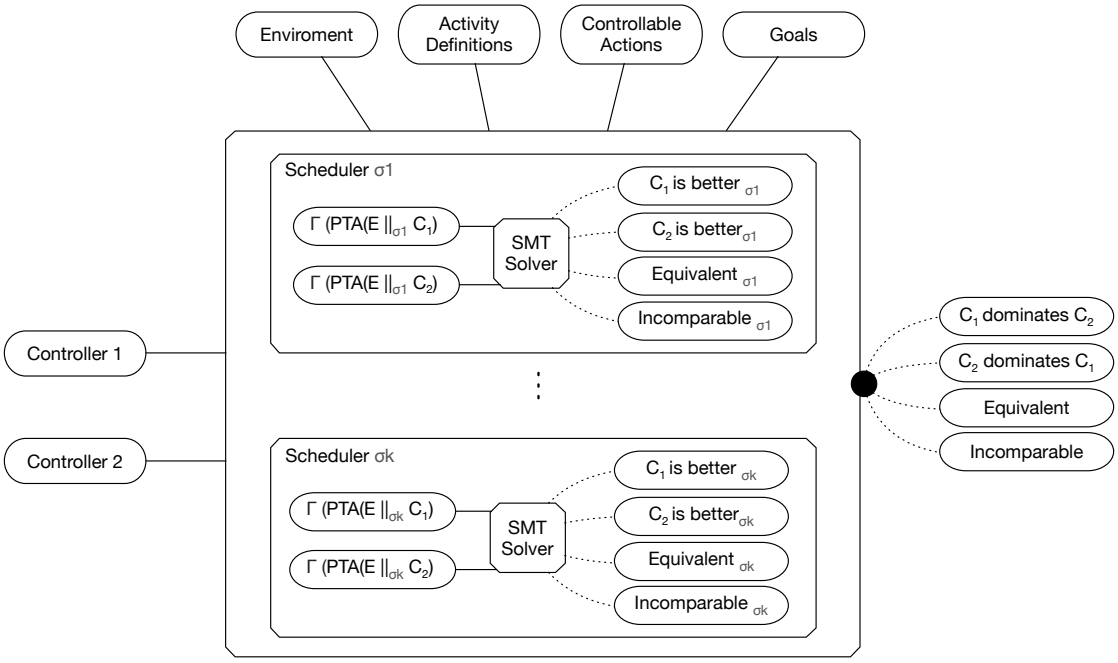



Figure 9.6: Comparison framework for a pair of controllers $C_1$ and $C_2$. The number of SMT queries depends on the number schedulers ($\sigma_1 \dots \sigma_k$).

### 9.1.4 Comparing Γ expressions with Z3

In order to implement the comparison framework in MTSA, we integrated the tool with the SMT-Solver Z3 by using the Java API [72]. Among other classes, we use BoolExpr, RealExpr, Context, and Solvers. Most of the operations that involve using Z3 are wrapped as functions in the class *MTSSynthesis.controller.gr.time.comparator.MySMT*. Figure 9.7 illustrates the fragment of the code that compares two Γ expressions. This expressions are obtained from the timed semantics of the scheduled composition of a controller. Each of this comparisons corresponds to one of the comparisons per secheduler. This means that those expressions are equivalent to each pair $\Gamma(PTA(E\|_\sigma C_1))$ and $\Gamma(PTA(E\|_\sigma C_2))$ shown in Figure 9.6.

In Figure 9.7, lines 6-13 initialize the expressions that are common to the two queries that are done to Z3 in each step. That is the Γ expressions of the two controllers (e.g.:*gamma1.getExpression()*) and some expressions that express that variables involved are positive reals ($\mathbb{R}_{\geq 0}$). This is necessary because it is

```
package MTSSynthesis.controller.gr.time.comparator;
import ...
public class GammaComparator<A, S> {
  ...
  public Result compareGammas(Gamma gamma1, Gamma gamma2){
   List<BoolExpr> expressions = new ArrayList<BoolExpr>();
   expressions.add(positive);
   expressions.add(smt.nonNegative(gamma1.getParameter()));
   expressions.add(smt.nonNegative(gamma1.getClock()));
   expressions.add(smt.nonNegative(gamma2.getParameter()));
   expressions.add(smt.nonNegative(gamma2.getClock()));
   expressions.add(gamma1.getExpression());
   expressions.add(gamma2.getExpression());

   BoolExpr T2_GT_T1 = smt.greaterThan(gamma2.getParameter(),
       gamma1.getParameter());
   BoolExpr T1_GT_T2 = smt.greaterThan(gamma1.getParameter(),
       gamma2.getParameter());

   //isT2_GT_T1 is satisfiable if exists a valuation that makes T2 > T1.
   Pair<Status,Model> isT2_GT_T1 = smt.check(expressions, T2_GT_T1);
   Pair<Status,Model> isT1_GT_T2 = smt.check(expressions, T1_GT_T2);

   Result result;
   if(isT1_GT_T2.getFirst().equals(Status.SATISFIABLE)){
      if(isT2_GT_T1.getFirst().equals(Status.SATISFIABLE)){
         result = Result.INCOMPARABLES;
      }else{
         result = Result.WORSE;//T1 is WORSE
      }
   }else{
      if(isT2_GT_T1.getFirst().equals(Status.UNSATISFIABLE)){
         result = Result.EQUALLYGOOD;
      }else {
         result = Result.BETTER;//T1 is BETTER
      }
   }
   return result;
  }
   ...
}
```

Figure 9.7: Comparing two Γ expressions with Z3.

.

not possible to directly define positive real variables in Z3. Otherwise, variables that represent activities durations may be instantiated into negative values. Lines 15-16 create two *BoolExpr* that represent the parts that differ in the two queries described in the previous section: $T2 > T1$ and $T1 < T2$. Lines 19-20 execute this two queries. These queries are $C_2 \lhd_\sigma C_1 : \exists(r_1..r_n, r_{f1}, r_{f2}) \in \mathbb{R}^{n+2}_{\geq 0}$ $((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge (r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f1} > r_{f2})$ and $C_1 \lhd_\sigma C_2 : \exists(r_1..r_n, r_{f1}, r_{f2}) \in \mathbb{R}^{n+2}_{\geq 0}$ $((r_1..r_n, r_{f1}) \Vdash \Gamma(PTA(E\|_\sigma C_1)) \wedge (r_1..r_n, r_{f2}) \Vdash \Gamma(PTA(E\|_\sigma C_2)) \wedge r_{f2} > r_{f1})$. The result of these queries is a pair that contains the status and a model. The status can be satisfiable or unsatisfiable. When the result is satisfiable the model contains a set of possible values to explain why it is satisfiable. Otherwise, the model is *null*. The rest of the code shown in this figure is to calculate the result of the comparison according to Table 9.1, which is shown from the perspective of $\Gamma_1$. The code is available in the repository of MTSA [73].

## 9.2 Experiments

In this section, we report two experiments that were performed to evaluate the Algorithm 2. The purpose of the first experiment is to show that it is possible to produce controllers that are better regarding makespan by using only qualitative information. In this experiment, we compare the controller produced by the standard synthesis algorithm of MTSA against the one produced by the Algorithm 2. The objective of the second experiment is to evaluate the performance of the algorithm on a variation of a job scheduling problem (JOB($n$)) [74], where $n$ jobs must be executed while satisfying dependencies related to the start and end of activities. The number of jobs involved in the problem was increased in order to evaluate the performance of the algorithm regarding the size of the problem.

### 9.2.1 Comparison against standard synthesis algorithm

The experiment consists in synthesising a controller with the standard algorithm of MTSA and the Algorithm 2 and in comparing those controllers by using the comparison framework. For each case study, the experiment is performed as shown in Figure 9.8. The standard synthesis algorithm takes as input an environment, a

set of controllable actions, and the goals. The non-dominated controller algorithm takes those inputs and the set of activity definitions. The controllers generated are compared by using the comparison framework, which also uses the same inputs as the Algorithm 2.

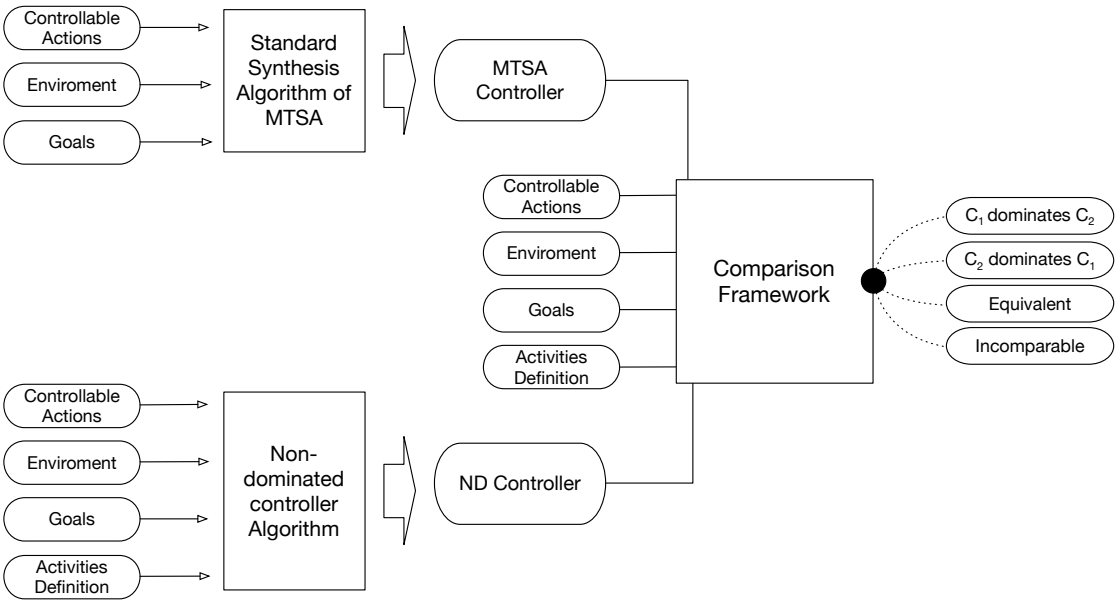



Figure 9.8: Experimental setting to compare the solution generated by Algorithm 2 against the ones produced by standard synthesis algorithm of MTSA.

For the evaluation we selected the following case studies because they involve executing a set of activities in a reactive environment.

- Furniture Delivery Service (FDS), a web service composition problem [75]. Requests to web services are modeled as activities that can end as a success or as a failure. The goal is to successfully arrange a purchase and delivery, or failure after two re-attempts.
- Industrial Automation (IA) [48]. The motivating example described in Section 6.1. Contingencies are related to the type of product to be processed and the need for a final repair action, while the goal is to produce a product.
- Medical (MED) [76]. A sick patient has one of five possible sickness and a doctor has two tests to discover wich of them is. The two tests are the check of *stain* and *count cells*. Each sickness requires a different medicine. If the doctor applies the correct medicine the patient is cured, otherwise the patient dies. The goal is to cure the patient, and the contingencies are

product of the test results.

- Job scheduling problem (JOB($n$)) [74] where $n$ jobs must be executed while satisfying dependencies related to the relative start and end of activities.

These problems have alternative solutions, and some of them seem to be better than the other in terms of makespan. Most of them may also require different activities to fulfill the goal depending on the contingencies that occur. Besides, for this case studies, the solution generated by the standard synthesis algorithm of MTSA is a universal controller.

### Results

Table 9.2 shows the results of obtaining a non-dominated controller for the case studies FDS, MED, IA and JOB(4). For each case study we report the number of states of the universal controller ($\#Q_{\mathcal{U}}$), the number of transitions of the universal controller separated in controllables (C), end of activity (EA), and other uncontrollables (U), the number of transitions removed by the algorithm ($\#\mathcal{D}$ ), the number of reachable states from the initial state ($\#Q_{C^*}$) and the result of the comparison between the two controllers (Result). The universal controller $\mathcal{U}$ is the solution produced by the standard synthesis algorithm and controller $C^*$ the one produced by Algorithm 2.

| Example | $\#Q_{\mathcal{U}}$ | #Transitions ($\#\Delta_{\mathcal{U}}$) | | | $\#\mathcal{D}$ | $\#Q_{C^*}$ | Result |
|---|---|---|---|---|---|---|---|
| | | C | EA | U | | | |
| FDS | 69 | 72 | 24 | 5 | 0 | 69 | Same Controller |
| MED | 50 | 40 | 24 | 1 | 0 | 50 | Same Controller |
| IA | 25 | 18 | 10 | 2 | 1 | 23 | $C^*$ dom. $\mathcal{U}$ |
| JOB(4) | 34 | 30 | 23 | 0 | 1 | 20 | $C^*$ dom. $\mathcal{U}$s |

Table 9.2: Results of the evaluation of Algorithm 2 on cases studies from different fields that involve the execution of activities to reach a goal.

The results show that for the case studies FDS and MED the number of transitions removed is 0. This means that the controller produced by both algorithms is the same. We inspected the case studies and noticed that despite the existence of different alternatives to reach the goal in a concurrent manner, the order in which the controllable actions are executed do not affect the makespan in reaching the goal. Then, it is reasonable that Algorithm 2 does not remove

any transition and produces the same result as the standard synthesis algorithm. In contrast, for the case studies JOB(4) and IA transitions are removed from the universal controller. By removing those transitions, the controller produced by Algorithm 2 avoids those taking transitions that lead to behaviour of higher makespan independently of the durations of the activities and contingencies. Table 9.2 shows that controller produced by Algorithm 2 dominates the one produced by the standard synthesis algorithm. In the next section, we inspect the industrial automation example to explain why $C^*$ dominates $\mathcal{U}$.

### Inspecting the industrial automation example

Figure 9.9 shows the controllers produced by the standard algorithm of MTSA and the Algorithm 2. The only difference in the input is the activities definition. The controller synthesised by MTSA is equivalent to the universal controller from Figure 6.5, while the Algorithm 2 produces the same controller as in Figure 8.1, after removing the red transition.

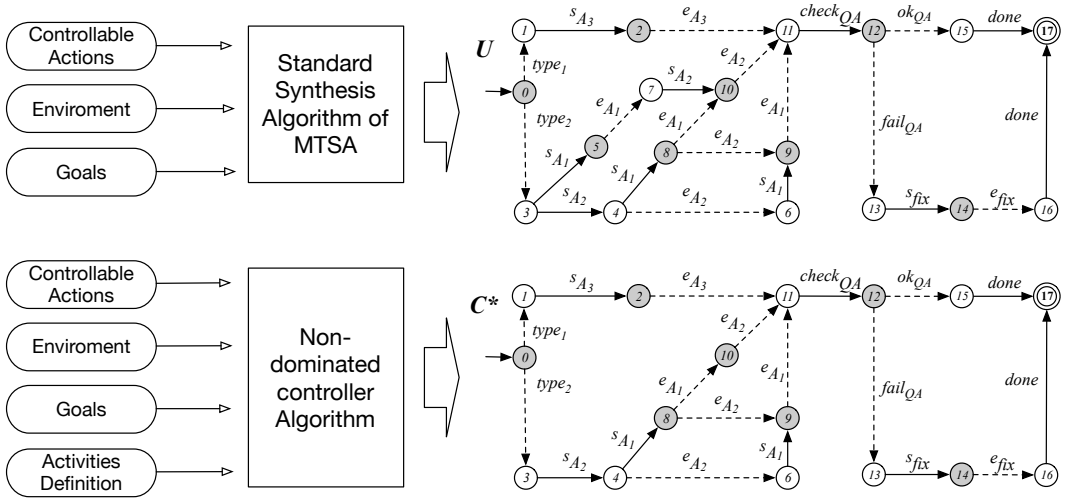



Figure 9.9: Controllers produced by standard algorithm of MTSA and Algorithm 2.

The controller $C^*$ dominates $\mathcal{U}$ because $C^*$ does not have transition $sA1$ in the state $q_3$, which leads to a sequential execution of the activities $A_1$ and $A_2$. In particular, the controller $C^*$ is better than $C$ for the following scheduler $\sigma_3$, because the controller $C*$ does not enable the action $s_{A_1}$ in the state $q_3$.

- $\sigma_3(q_0, \{type_1, type_2\}) = \{type_2\}$
- $\sigma_3(q_3, \{s_{A_1}, s_{A_2}\}) = \{s_{A_1}\}$, $\sigma_3(q_3, \{s_{A_2}\}) = \{s_{A_2}\}$, $\sigma_3(q_3, \{s_{A_1}\}) = \{s_{A_1}\}$

- $\sigma_3(q_4, \{s_{A_1}, e_{A_2}\}) = \{s_{A_1}\}$, $\sigma_3(q_3, \{e_{A_2}\}) = \{e_{A_2}\}$
- $\sigma_3(q_5, \{e_{A_1}\}) = \{e_{A_1}\}$
- $\sigma_3(q_6, \{s_{A_1}\}) = \{s_{A_1}\}$
- $\sigma_3(q_7, \{s_{A_2}\}) = \{s_{A_2}\}$
- $\sigma_3(q_8, \{e_{A_1}, e_{A_2}\}) = \{e_{A_1}, e_{A_2}\}$
- $\sigma_3(q_9, \{e_{A_1}\}) = \{e_{A_1}\}$
- $\sigma_3(q_{10}, \{e_{A_2}\}) = \{e_{A_2}\}$
- $\sigma_3(q_{11}, \{checkQA\}) = \{checkQA\}$
- $\sigma_3(q_{12}, \{failQA, okQA\}) = \{okQA\}$
- $\sigma_3(q_{15}, \{done\}) = \{done\}$

This scheduler would lead to a comparison between the scheduled composition of the controllers like the one shown in Figure 9.10. This two scheduled controllers are analogous to the ones shown in Figure 7.1. In Section 7.3, we show that it is possible to find a set of parameters that makes the makespan of $\mathcal{U}$ by higher than that of $C^*$. For instance, this happens when all the parameters are assigned with value 1. This is because for this scheduler, the scheduled composition of the universal controller is sequential, while the one of the controller $C^*$ is concurrent.

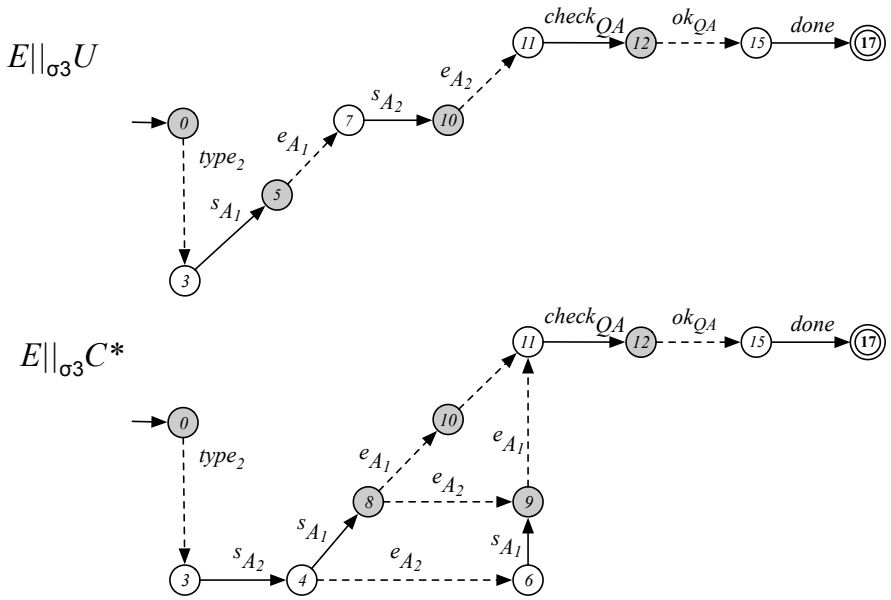



Figure 9.10: Scheduled parallel composition of controllers generated by the standard synthesis algorithm and the Algorithm 2 on the IA example.

### 9.2.2 Scalability in the job scheduling example

The experiment takes as input a control problem with activities. The goal is to evaluate the synthesis time of the Algorithm 2. We distinguish the synthesis time of the universal controller from the total time to identify the time it takes to apply the comparison for pruning transitions in each state. We report number of transitions removed and the number of reachable states after removing the transitions. The number of reachable states is of interest because more compact controllers are easier to validate. Additionally, we compare the solution synthesised by Algorithm 2 with the one generated with the standard algorithm of MTSA.

The experiment is performed on different instances of the JOB($n$) case study with increasing number of jobs to be executed ($n \in 4, 5, 6, 7$). In this case study, there are $n$ jobs to be executed and the goal is to finish all of them. To exemplify this case study, we use the JOB(4) instance. In this instance, there are four jobs: $job_1$, $job_2$, $job_3$ and $job_4$. Each of the jobs consumes the amount of memory indicated by the number. There are two type of restrictions regarding when jobs can be executed. First, there is a memory limit that does not allow to execute all the jobs at the same time. For simplicity, the memory limit is set to be $n + 1$. For example, $job_4$ and $job_2$ cannot be executed at the same time because there is not enough memory ($4 + 2 > 5$). Second, there are also dependencies between the jobs. For instance, $job_2$ has to be executed after $job_1$ finishes and $job_1$ must start after either $job_4$ or $job_3$ have started. These dependencies are modelled as safety constraints in the model. The number of dependencies in each instance of the case study is $n - 2$ and the dependencies are chosen arbitrarily .

#### Results

Table 9.3 shows the results of obtaining a non-dominated controller for JOB($n$) case study with $n \in \{4, 5, 6, 7\}$. For each instance, we report the number of states of the universal controller ($\#Q_{\mathcal{U}}$), synthesis time of the universal controller (Synth. Time $C^*$), the number of transitions of the universal controller separated in controllables (C), end of activity (EA), and other uncontrollables (U), the number of transitions removed by the algorithm ($\#\mathcal{D}$ ), the number of reachable states from the initial state ($\#Q_{C^*}$), the synthesis time of the Algorithm 2 (Synth.

Time $C^*$) and the result of the comparison between the universal controller and the controller $C^*$, which is the one produced by Algorithm 2. As in the previous experiment, the universal controller $\mathcal{U}$ also referst to the solution produced by the standard synthesis algorithm.

| Case Study | $\#Q_{\mathcal{U}}$ | Synth. Time $\mathcal{U}$ | $\#\Delta_{\mathcal{U}}$ | | | $\#\mathcal{D}$ | $\#Q_{C^*}$ | Synth. Time $C^*$ | Result |
|---|---|---|---|---|---|---|---|---|---|
| | | | C | EA | U | | | | |
| JOB(4) | 34 | 0.006s | 30 | 23 | 0 | 1 | 20 | 3.81s | $C^*$ dom. $\mathcal{U}$ |
| JOB(5) | 63 | 0.012s | 58 | 49 | 0 | 3 | 39 | 21.73s | $C^*$ dom. $\mathcal{U}$ |
| JOB(6) | 101 | 0.029s | 103 | 81 | 0 | 5 | 90 | 240.26s | $C^*$ dom. $\mathcal{U}$ |
| JOB(7) | 259 | 0.329s | 288 | 219 | 0 | 22 | 211 | 18833.77s | $C^*$ dom. $\mathcal{U}$ |

Table 9.3: Results of the performance of Algorithm 2 on the JOB case study with increasing number of tasks.

The results of the experiment show that synthesis time of the Algorithm 2 in the JOB($n$) increments when the size of the problem increases regarding states and transitions. The reason is that each local comparison generates the set of possible schedulers, which size grows with respect to the number of actions and states. In contrast, we can observe that the synthesis time of the universal controller stays below a second. The case study JOB(8) was not reported in the table because of time-out, which was defined as 24h (86400 seconds). The number of transitions pruned by the algorithm increases in larger problems, as well as the number of states of the non-dominated controller is reduced compared to that of the universal controller. In all these cases, the controller produced by the standard algorithm of MTSA was equivalent to the universal controller. These controllers were dominated by the ones produced by the Algorithm 2.

### 9.2.3 Discussion

The results of the experiments show that the Algorithm 2 can prune those transitions that lead to sequential behaviour from a universal control. This shows that it is possible to qualitatively define a preference, and to define an algorithm that can take it into account. However, the algorithm does not scale up to large problems, because the number of schedulers increases as the size of the problem does. On the positive side, we can observe that many transitions are removed

by the algorithm when the size of the problem increases. In some cases, these decisions also reduce the number of reachable states in the controller making it more compact and easier to validate by an engineer or stakeholder. For instance, in the JOB(5) case study, the number of states decreased by 38% (from 63 states to 39 states). The impact in the number of states of the controller changes depending on how close this transition is in terms of steps to the initial state. Removing a transition that is closer to the initial states produces a bigger impact in the size of states.

The fact that no transitions are removed in the case studies FDS and MED shows that not every problem has a solution that is better than other. Sometimes the order in which the actions are executed is not necessarily relevant. Our hypothesis that the order of the actions matter in this case studies was not correct. For instance, in the MED case study, we expected that the order in which the tests were done would affect the makespan of the solution. This is because one of the two tests could directly indicate which is the sickness that the patient has. In order to understand the details, let us explain more about this case study. As mentioned before, there is a patient that needs to be diagnosed among five possible diseases (*disease 1*... *disease 5*). There are two possible medical tests to evaluate the condition of the patient: *stains* and *count cells*. The stain test has three possible results: A (*disease 3 or 4*), B (*disease 1 or 2*) and C (*disease 5*). The count cells test has two possible results: Normal (*disease 2 or 4*) or High (*disease 1, 3 or 5*). The fact that the stain test can directly diagnose *disease 5* made us believe that the makespan of a controller doing this test first would be less than that of a controller doing count cells test first. However, when analysing the results, we realised that the order in which tests are done does not matter. The only thing that matters is that the controller tries to execute the two tests as soon as possible. If the result of the test that diagnoses the sicknesses directly comes first, the controller would diagnose the patient without waiting for the result of the other test. This means that in the goal states there are still activities running. For instance, the following trace would be a possible: *sCountCells*, *sStain*, *eStainResultC*, *useMedicineC*. Here, in the goal state, the activity of *counting cells test* is still running. This is the reason why in those cases, the algorithm does not remove any controllable transitions, which is the expected output.

# 10 Related Work

In this chapter, we summarize some papers from the literature that are relevant to our work. These works are from the areas described in Chapter 2, which are supervisory control, reactive synthesis, and automated planning. Section 10.1 presents quantitative approaches to preferences, while Section 10.2 shows qualitative approaches. Section 10.3 discusses time and uncertainty, which is also a relevant topic in this thesis. Each section finishes with a comparison between approaches taken in the literature and our approach.

## 10.1 Quantitative approaches for preferences

In the supervisory control community, optimal control for quantitative discrete-event systems studies the performance of the controller [77, 78]. A solution to this problem should not only achieve a goal, but also achieve it in an optimal manner. The performance of the controller is measured by introducing cost functions, and the goal is to reach a set of desired states while optimising the

cost. Passino et al. [78] proposed a heuristic search algorithm to reach one of the target states optimally. In the work of Brave et al. [77], the authors presented a solution that uses optimal attractors to generate a supervisor that reaches a set of target states and stays there indefinitely. Pantelic et al. [79] studied a synthesis method for probabilistic models. Minimising makespan was studied by Su et al., who proposed a solution by modelling the problem as an optimal supervisor problem [81]. Pruekprasert et al. [82] proposed an algorithm to compute an optimal solution to infinite sequences problems modelled by weighted automata, which is based on two-player games.

In the reactive synthesis community, formally reasoning about the quality of the solutions has been a relevant topic in the recent years [22, 83]. Bloem et al. [22] proposed a method that uses quantitative properties to measure how good is an implementation. They also presented an optimal synthesis method, which solves games with quantitative objectives. In particular, their solution uses lexicographic mean-payoff conditions to express quantitative properties for reactive systems. Those algorithms are to solve problems with safety requirements, which can be solved by using lexicographic mean-payoff games [23], and to solve problems with liveness requirements, which need using both lexicographic mean-payoff games and parity objectives. Chaterjee et al. [84] studied multi-dimensional quantitative objectives on multidimensional mean-payoff and energy games. They proposed a symbolic and incremental synthesis method to solve multi-dimensional energy parity games, which computes a finite-memory winning strategy, if exists.

There are also tools for quantiative synthesis. For instance, QUASY [85] is the first tool for quantitative synthesis that can model both adversarial environments and probabilistic environments. This tool can solve these two types of quantitative synthesis problems by using two-player games and Markov Decision Processes (MDPs) with quantitative winning objectives.

In the work of Amalgor et al. [83], the authors presented a different approach by defining a specification formalism to measure how well the system satisfies the specification. They define propositional quality and temporal quality for temporal logic by adding quantitative information that is used as a metric of satisfaction. A similar method has also been applied in branches of planning that use LTL synthesis to generate plans [86].

In the planning community, preferences are added in the third version of planning domain definition language (PDDL) [87]. This version of PDDL permits defining preferences, hard constraints, as well as a metric function that defines the quality of the plan. For instance, it is possible to specify for example *when a robot leaves the room, it would be better if it turns off the light.* The metric function is an arithmetic function, which allows to define how to measure the quality of a plan in terms of length of the plan and number of violations of the preferences [88]. Some authors proposed that this type of preferences could be compiled into classical planning with action costs [89], which is also a quantitative approach.

Decision theoretic-planning [90] solves planning problems with non-deterministic effects by obtaining an optimal policy from an MDP. A policy in the MDP is a function that returns an action based on the history of actions and observations perceived. Finding optimal policies can be seen as a form of preference-based planning. Preferences can be specified by using a reward functions. However, defining a utility function that reflects the preferences of the user is not trivial [91].

Typically, to compare solutions, quantitative techniques must aggregate values of different executions by using the worst or average case. As mentioned above, this requires modelling costs and probabilities in the models. This means that quantitative measures are sensitive to the numbers that users assign in the models. In contrast, our approach does not require probabilistic modelling or quantitative information about the environment. Instead, we establish a way to compare controllers qualitatively by using a symbolic comparison. The synthesis method proposed does not guarantee an optimal solution, but relieves the user from providing quantitative information. Our work is currently limited to reachability and safety goals with a makespan preference, while some quantitative methods can generate optimal solutions for problems with liveness goals [23, 82] and multiple preferences [84], which are aspects we intend to develop in our future work.

## 10.2 Qualitative approaches for preferences

Qualitative approaches to express preferences exist in the planning community [91]. One of these approaches is conditional preference networks (CP-nets) [92] , which allows users to specify conditional preference relations among variables. For

instance, a user could specify that email is preferred over SMS under general circumstances, but if the signal is low SMS is preferred over email. This type of relations induces a partial order, which means that incomparability is also a possible outcome of CP-net. TCP-nets [93] is an extension of CP-nets that allows modelling tradeoff preferences between certain variables. For example, delivering priority packages is more important than fuel economy. Despite that incomparable is a possible outcome of TCP-nets as well, this form of preferences permits reducing incomparable results.

Other approaches focus on defining temporal preferences qualitatively. That is, allowing the user to specify temporal aspects in the execution of a plan [91], which hold under certain preconditions. One of these approaches proposes a framework for temporal preferences that is expressed in Boolean language and arithmetic operators [94]. The arithmetic operators are introduced to define quantified operations over steps of the plan. For example, the will of delivering package $A$ prior to package $B$ can modelled as deliver $A$ in step $k$ of the execution, and deliver $B$ in step $p$ with be $p > k$. This form of preference also defines a partial order, which allows for incomparability. In this framework, aggregation of preferences is possible by defining a preference relation in quantitative terms, which means that it is not purely qualitative.

Another interesting approach in temporal preferences is a language named LPP [95], which alllows expressing temporal preferences in some form of LTL that allows quantification and aggregation preferences. For instance, it is possible to define preferences such as $\psi_1$ : *eventually return to pick-up more mail*, $\psi_2$: *always deliver priority mail before standard one*, or $\psi_3$: *do not deliver mail A until you deliver mail B*. The user can also define levels of satisfaction, e.g., excellent > good > bad. Then it is possible to define how preference formulas affect the levels of satisfaction. For instance, satisfying $\psi_1$ and $\psi_2$ is excellent, satisfying only $\psi_1$ is good, and satisfying only $\psi_3$ is bad. Then the plan is evaluated by using those levels of satisfaction, which makes this framework have a total order, i.e., there are no incomparables.

Some works implemented planners for qualitative languages on top of general-purpose solvers such as SAT, ASP or CSP solvers [96, 97, 98]. For instance, PREF-PLAN [98] is a planner that uses TCP-net [93] specification language and a

constraint satisfaction problem solver as part of its planning engine. As most of the planning techniques, they do not represent the entire problem as a graph and use heuristics to find next actions for the plan. In this step, they create a graph of depth $k$ and define a constraint satisfaction problem that produces the most preferred plan for the next k actions. However, this does not guarantee that the solution produced is optimal.

To the best of our knowledge, both supervisory control and reactive synthesis community have not explored modelling preferences qualitatively. As mentioned in the previous section, preferences in these two areas are mainly studied quantitatively in the search of optimality. Our approach intends to be the first step in a similar direction to that of qualitative approaches in planning. Our approach generates a partial order among alternative solutions to a control problem. Currently we focus only on makespan, while qualitative preferences in planning can support for multiple preferences, which is one of the branches that we aim to develop as future directions. Nonetheless, the proposed framework to compare controllers is different from the works that we previously mentioned.

Those approaches focus on expressing preferences regarding the order in which events occur (e.g.: *deliver a prior to b*) or preference among alternative ways of doing something (e.g.: *deliver by email is preferred over deliver by SMS*). We believe that some of these forms of preferences can be captured as temporal logic properties of our models, which is similar to what is done in temporal preferences. The main difference is that preferences are not required to be satisfied by the solution produced while properties in a control problem must be satisfied. Another important difference is that those approaches are designed for classical planning, which means that those problems do not involve contingencies. In addition, none of those approaches seem to focus on modelling time qualitatively. Our form of comparison is a novel approach, and it is the only one that relies on PTA and SMT-Solving to produce a qualitative result in the context of controller synthesis.

## 10.3 Temporal problems and uncertainty

Minimising time in reaching a goal is not only considered as a quality measure, but also as a requirement that the system must statisfy. In fact, the main goal of

temporal planning problems is to minimise the makespan in reaching a deadline [99], while satisfying a set of quantitative temporal constraints about the order in which activities have to be executed. PDDL is capable of expressing temporal and numeric properties of planning domains [100].

In the planning community, there has been strong interest in the uncertainty of durations, i.e., achieving the goal for any possible value of uncontrollable durations [101, 102]. This problem is named strong temporal planning with uncontrollable durations (STPUD). A strong plan must always reach the goal. In STPUD problems, the planner can choose when to start an action, and the environment chooses when it finishes within bounds that are known to the planner. Uncertain durations are bounded between concrete values $[\delta_{min}, \delta_{max}]$. One of these works [103] uses SMT to encode time constraints and determine if a temporal problem is solvable. The SMT-Solver is used to find examples of uncontrollable time points, given an assignment to the controllable points. This type of problems extends strong temporal planning problems, which are plans that do not deal with contingencies. In contrast, conditional simple temporal networks with uncertainty and decisions (CSTNUD) [104] allow to represent both dynamic controllability and uncertain durations. Zavatteri [104] proposes to encode the problem as timed game automata and presents synthesis methdos for CSTNUD.

The problem of how to synthesise a controller has also been defined for timed automata [105]. This problem extends the controller synthesis problem for discrete event dynamic systems by allowing the controller to choose between executing an action or letting the time pass. The problem is formulated by using timed games, from which a strategy is built. Works mainly focus on safety [105] and reachability [106] games, because solving a reachability-time game is known to be already EXPTIME-complete for timed automata with at least two clocks [106].

Parametric timed automata extends timed automata with parameters to model uncertainty of durations in the system. Parametric timed models have been studied since the seminal work of Alur [25]. Parameters are the basis to study robust schedulability conditions [107, 108]. There are approaches that tackle parametric timed reachability games for fragments of parametric timed automata [109]. Synthesis approaches for parametric timed automata are scarce due to decidability problems. For this reason, a subclass of PTA called lower

bound / upper bound automata was characterized. For this type of PTA, there are only semi-algorithms to compute the corresponding parameter valuations. A solution to these problems guarantees reaching a goal state, but not necessarily in minimum time. Recent studies [61, 110] focus on the *parameter synthesis* problem. A solution to this problem is to obtain a set of values for the parameters that guarantee reaching the goal in minimal time.

Both temporal planning problems and synthesis for timed models focus on the temporal aspects of the specification, while the focus of our approach is on behaviour. These approaches require to model a quantitative part of the specification because time is the focus of the study. In contrast, we solve a reactive problem against an adversarial environment, where contingencies are a key aspect of the problem, and time is only a preference. In our work, parameters do not need to be specified by the user. They are inferred from the behavioural model by using activity definitions. To the best of our knowledge, the problem of directly comparing parametric timed automatons has not been explored. Thus, our form of comparison based on symbolic expressions can also be a contribution to this community.

# 11 Conclusions

In this chapter, we summarize the contributions of this thesis and its limitations. Some of these limitations may be a good start for possible extensions.

## 11.1 Contributions

The main contribution of this work is building a framework to symbolically compare makespan of controllers for safety and reachability goals. The results of the comparison establish a dominance relation between controllers. Moreover, we provide a qualitative definition of makespan-minimising controllers, which are controllers that cannot be dominated by any other controller. We also present a sound algorithm to produce a makespan-minimising controller. In addition, we conduct experiments to compare the controllers generated by our algorithm against the ones produced by the standard synthesis algorithm of MTSA. The experiment shows that our algorithm removes those transitions that lead to sequential execution from a universal controller. By removing those transitions, we

can generate controllers that perform better than the ones produced by standard qualitative synthesis algorithms. In some cases, the algorithm produces the same solution as the standard synthesis algorithm, because in those cases the order in which the actions are executed do not alter the makespan of the controller. The current implementation can produce makespan-minimising controllers for small to medium size problems, but it does not scale up to large size problems.

## 11.2 Limitations and potential extensions

### Reachability and safety goals

Although in this work we focus on reachability goals and makespan analysis, supporting more general goals, such as GR(1) [16], would be of interest. However, this has a key challenge which is to deal with cycles, which are product of uncontrollable behaviour that can be executed an unbounded number of times.

### Recurrence of activities

The qualitative measure of makespan is defined in terms of the duration of the activities. In our models, activities do not occur twice in a trace between the initial and goal states because we are dealing with memoryless solutions. However, if the activity occurs more than once, the same parameter will be used to model the duration of the activity. This means that different instances of the same activity take the same parametric duration. One alternative is to enumerate the instances of the activities and assign a different parameter to each of them. Another alternative is to model the duration of an activity by using parametric intervals instead of a single parameter.

### Relation between the activity durations

Currently, we make no assumption on possible relations among activity durations, this technique can be easily extended to support symbolic constraints of activity durations (e.g., $p_{\alpha_1} \leq p_{\alpha_2} + p_{\alpha_3}$). These constraints can be added to the comparison framework, which would also be reflected in the algorithm. Adding this to the

framework may also help to reduce the number of incomparable results in the comparison, which may also improve the performance of the algorithm.

### Compare standard PTAs with the comparison framework

The comparison framework is defined to compare controllers generated by qualitative synthesis techniques. In our framework, activities must be specified to be able to define a timed semantics for controllers, which is based on PTAs. In the last step, the comparison framework is comparing symbolic expressions that are genereated from PTAs. This means that it may be also possible to apply this form of comparison directly on PTAs. However, these PTAs may need to satisfy some conditions. For instance, it should be possible to calculate $\Gamma$. Besides, a different form of schedulers may need to be defined, because the schedulers that are defined in this work distinguishes the end of activity actions, other uncontrollable actions, and controllable ones.

It would be interesting to evaluate if the comparison without the schedulers can produce relevant results when comparing makespan of standard PTAs for reachability. Otherwise, it might be necessary to define different type of schedulers. One important fact to consider is that when comparing PTAs, those PTAs need to share some parameters. Otherwise, the results of the comparison would always be *incomparable*. Thorough analysis need to be done to characterize the conditions that PTAs need to satisfy to use these framework, and the significance that may have for this community.

### Multiple qualitative preferences

In this work, we use the symbolic constraints over the parameters as a metric of makespan. However, parameters could be given a different semantic such as energy consumption. For example, we could have a symbolic expression $b_f = (b_{\alpha_1} + b_{\alpha_2} \vee b_{\alpha_3})$ that defines the energy consumption regarding and activities, and another expression $p_f = (p_{\alpha_1} + p_{\alpha_2} \vee p_{\alpha_3})$. This enables the possibility of combining relations between activity durations and energy consumption. Nonetheless, it requires to develop a way to express our main preference, and how multiple preferences should be prioritized. Otherwise, most of the results may be

incomparable.

### Encoding the schedulers in the symbolic expression

The current implementation of the schedulers has shown performance issues regarding execution time of the algorithm. Alternatively, encoding a scheduler in the symbolic expression may reduce the time it takes to produce a controller with our technique. However, defining an encoding for the scheduler may not be trivial, because schedulers need to consider alternative transitions when a controller does not have a particular controllable action enabled.

### Contextual schedulers

The schedulers defined in this work are defined regarding the states of the environment. This means that when executing two different controllers, the environment behaves the same (or similar) in a particular state. Another option is to define schedulers that do not depend on the states of the environment but only on the currently running and finished activities. However, defining how to model this type of schedulers requires thorough analysis.

# Bibliography

[1] Axel Van Lamsweerde. *Requirements engineering: From system goals to UML models to software*, volume 10. Chichester, UK: John Wiley & Sons, 2009.

[2] Orna Grumberg, EM Clarke, and Doron Peled. Model checking, 1999.

[3] Thomas Bochot, Pierre Virelizier, Hélene Waeselynck, and Virginie Wiels. Model checking flight control systems: The airbus experience. In *2009 31st International Conference on Software Engineering-Companion Volume*, pages 18–27. IEEE, 2009.

[4] Pierre-Alain Bourdil, Bernard Berthomieu, and Eric Jenn. Model-checking real-time properties of an auto flight control system function. In *25th IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Naples, Italy, November 3-6, 2014*, pages 120–123. IEEE Computer Society, 2014.

[5] Markus Weißmann, Stefan Bedenk, Christian Buckl, and Alois Knoll. Model checking industrial robot systems. In *International SPIN Workshop on Model Checking of Software*, pages 161–176. Springer, 2011.

[6] Johan Arcile, Raymond Devillers, and Hanna Klaudel. Verifcar: a framework for modeling and model checking communicating autonomous vehicles. *Autonomous Agents and Multi-Agent Systems*, 33(3):353–381, 2019.

[7] Tichakorn Wongpiromsarn and Richard M Murray. Formal verification of an autonomous vehicle system. In *Conference on Decision and Control*, 2008.

[8] Fujio Miyawaki, Ken Masamune, Satoshi Suzuki, Kitaro Yoshimitsu, and Juri Vain. Scrub nurse robot system-intraoperative motion analysis of a scrub nurse and timed-automata-based model for surgery. *IEEE Transactions on Industrial Electronics*, 52(5):1227–1235, 2005.

[9] David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer, 1985.

[10] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer Science & Business Media, 2012.

[11] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190. ACM, 1989.

[12] Rüdiger Ehlers, Stéphane Lafortune, Stavros Tripakis, and Moshe Y Vardi. Supervisory control and reactive synthesis: a comparative introduction. *Discrete Event Dynamic Systems*, 27(2):209–260, 2017.

[13] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Where's waldo? sensor-based temporal logic motion planning. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 3116–3121. IEEE, 2007.

[14] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE transactions on robotics*, 25(6):1370–1381, 2009.

[15] Hadas Kress-Gazit, Tichakorn Wongpiromsarn, and Ufuk Topcu. Correct, reactive, high-level robot control. *IEEE Robotics & Automation Magazine*, 18(3):65–74, 2011.

[16] Nir Piterman, Amir Pnueli, and Yaniv SaâĂŹar. Synthesis of reactive (1) designs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 364–380. Springer, 2006.

[17] L. Nahabedian, V. Braberman, N. D'Ippolito, S. Honiden, J. Kramer, K. Tei, and S. Uchitel. Assured and correct dynamic update of controllers. In *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '16, pages 96–107, New York, NY, USA, 2016. ACM.

[18] PJ Ramadge and WM Wonham. Supervisory control of a class of discrete event processes. In *Analysis and Optimization of Systems*, pages 475–498. Springer, 1984.

[19] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

[20] Robert P. Goldman and Mark S. Boddy. Expressive planning and explicit knowledge. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, May 29-31, 1996*, pages 110–117, 1996.

[21] Louise Pryor and Gregg Collins. Planning for contingencies: A decision-based approach. *J. Artif. Intell. Res. (JAIR)*, 4:287–339, 1996.

[22] Roderick Bloem, Krishnendu Chatterjee, Thomas A Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In *Computer Aided Verification*, pages 140–156. Springer, 2009.

[23] Krishnendu Chatterjee, Thomas A. Henzinger, and Marcin Jurdzinski. Mean-payoff parity games. In *LICS*, pages 178–187. IEEE Computer Society, 2005.

[24] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[25] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *Proc. of the ACM Symp. on Theory of Computing*, STOC '93, 1993.

[26] Thomas A Henzinger. *The theory of hybrid automata*. Springer, 2000.

[27] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[28] Nicolás D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastián Uchitel. MTSA: the modal transition system analyser. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 475–476. IEEE, 2008.

[29] A Church. Applications of recursive arithmetic to the problem of circuit synthesis–summaries of talks. *Institute for Symbolic Logic, Cornell University*, 1957.

[30] Alonzo Church. Logic, arithmetic and automata. In *Proceedings of the international congress of mathematicians*, volume 1962, pages 23–35, 1962.

[31] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004.

[32] Giuseppe De Giacomo and Moshe Vardi. Synthesis for ltl and ldl on finite traces. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[33] Alberto Camacho, Meghyn Bienvenu, and Sheila A. McIlraith. Towards a unified view of AI planning and reactive synthesis. In J. Benton, Nir Lipovetzky, Eva Onaindia, David E. Smith, and Siddharth Srivastava, editors, *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019.*, pages 58–67. AAAI Press, 2019.

[34] Alberto Camacho, Jorge A. Baier, Christian J. Muise, and Sheila A. McIlraith. Finite LTL synthesis as planning. In Mathijs de Weerdt, Sven Koenig, Gabriele Röger, and Matthijs T. J. Spaan, editors, *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018.*, pages 29–38. AAAI Press, 2018.

[35] Nicolás D'Ippolito, Natalia RodrıÌĄguez, and Sebastian Sardina. Fully observable non-deterministic planning as assumption-based reactive synthesis. *Journal of Artificial Intelligence Research*, 61:593–621, 2018.

[36] Giuseppe De Giacomo, Paolo Felli, Fabio Patrizi, and Sebastian Sardina. Two-player game structures for generalized planning and agent composition. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[37] Rune M Jensen. Des controller synthesis and fault tolerant control. Technical report, Citeseer, 2003.

[38] Ruediger Ehlers, Stephane Lafortune, Stavros Tripakis, and Moshe Vardi. Reactive synthesis vs. supervisory control: Bridging the gap. *EECS Department, Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2013-162*, 2013.

[39] Daniel Ciolek, Víctor A. Braberman, Nicolás D'Ippolito, and Sebastián Uchitel. Directed controller synthesis of discrete event systems: Taming composition with heuristics. In *55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, NV, USA, December 12-14, 2016*, pages 4764–4769. IEEE, 2016.

[40] Marco Pistore and Paolo Traverso. Planning as model checking for extended goals in non-deterministic domains. In *IJCAI*, volume 1, pages 479–486, 2001.

[41] Jorge A Baier and Sheila A McIlraith. Planning with temporally extended goals using heuristic search. In *ICAPS*, pages 342–345, 2006.

[42] Stephen Cresswell and Alexandra Coddington. Compilation of ltl goal formulas into pddl. In *ECAI*, volume 16, page 985. Citeseer, 2004.

[43] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[44] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, January 1984.

[45] P Madhusudan. Control and synthesis of open reactive systems. 2009.

[46] Nicolás D'Ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. Synthesis of live behaviour models. In *Proc. of the int. symposium on Foundations of soft. eng.*, FSE '10, 2010.

[47] Nicolás D'Ippolito, Víctor A. Braberman, Nir Piterman, and Sebastián Uchitel. Synthesis of live behaviour models for fallible domains. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 211–220. ACM, 2011.

[48] Nicolás D'Ippolito, Víctor A. Braberman, Nir Piterman, and Sebastián Uchitel. Synthesizing nonanomalous event-based controllers for liveness goals. *ACM Trans. Softw. Eng. Methodol.*, 22(1):9:1–9:36, 2013.

[49] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language, 1998.

[50] Blai Bonet and Hector Geffner. Planning as heuristic search: New results. In *Proceedings of the 5th European Conference on Planning: Recent Advances in AI Planning*, ECP '99, pages 360–372, Berlin, Heidelberg, 2000. Springer-Verlag.

[51] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial intelligence*, 116(1-2):123–191, 2000.

[52] Sven Koenig and Reid G. Simmons. Real-time search in non-deterministic domains. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1660–1667, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[53] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pages 809–815. AAAI Press, 1992.

[54] Hector Geffner and Blai Bonet. A concise introduction to models and methods for automated planning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(1):1–141, 2013.

[55] Alberto Camacho, Christian J Muise, Jorge A Baier, and Sheila A McIlraith. Ltl realizability via safety and reachability games. In *IJCAI*, pages 4683–4691, 2018.

[56] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[57] D. Giannakopoulou and J. Magee. Fluent Model Checking for Event-Based Systems. In *ESEC/FSE*, pages 257–266, Helsinki, Finland, September 2003.

[58] Michael Jackson. The world and the machine. In *1995 17th International Conference on Software Engineering*, pages 283–283. IEEE, 1995.

[59] Nicolas D'Ippolito. *Synthesis of event-based controllers for software engineering*. PhD thesis, Citeseer, 2013.

[60] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120. ACM, 2001.

[61] Laura Bozzelli and Salvatore La Torre. Decision problems for lower/upper bound parametric timed automata. *Formal Methods in System Design*, 35(2):121, 2009.

[62] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of model checking*, volume 10. Springer, 2018.

[63] Emmanuel Letier and William Heaven. Requirements modelling by synthesis of deontic input-output automata. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 592–601. IEEE Press, 2013.

[64] Erich Gradel and Wolfgang Thomas. *Automata, logics, and infinite games: a guide to current research*, volume 2500. Springer Science & Business Media, 2002.

[65] Walter Murray Wonham. Supervisory control of discrete-event systems. *Encyclopedia of systems and control*, pages 1396–1404, 2015.

[66] Wolfgang Thomas. On the synthesis of strategies in infinite games. In *STACS*, pages 1–13, 1995.

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

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

[69] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[70] Víctor Braberman, Nicolas D'Ippolito, Nir Piterman, Daniel Sykes, and Sebastian Ucriitel. Controller synthesis: From modelling to enactment. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1347–1350. IEEE, 2013.

[71] Jeff Magee and Jeff Kramer. *State models and java programs*. wiley Hoboken, 1999.

[72] Z3 api documentation. http://z3prover.github.io/api/html/index.html. Accessed: 2020-01-29.

[73] The modal transition system analyser official website. http://mtsa.dc.uba.ar/. Accessed: 2020-01-29.

[74] Laurent Fribourg, Romain Soulat, David Lesens, and Pierre Moro. Robustness analysis for scheduling problems using the inverse method. In *Temporal*

*Representation and Reasoning (TIME), 2012 19th International Symposium on*, pages 73–80. IEEE, 2012.

[75] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. In *Artificial Intelligence: Methodology, Systems, and Applications*, Lecture Notes in Computer Science. 2004.

[76] Blai Bonet and Hector Geffner. Gpt: A tool for planning with uncertainty and partial information. In *In Proc. IJCAI01 Workshop on Planning with Uncertainty and Incomplete Information*, pages 82–87, 2001.

[77] Yitzhak Brave and Michael Heymann. On optimal attraction in discrete-event processes. *Information sciences*, 67(3):245–276, 1993.

[78] KM Passino and PJ Antsaklis. On the optimal control of discrete event systems. In *Proceedings of the 28th IEEE Conference on Decision and Control,*, pages 2713–2718. IEEE, 1989.

[79] Vera Pantelic and Mark Lawford. Optimal supervisory control of probabilistic discrete event systems. *IEEE transactions on automatic control*, 57(5):1110–1124, 2011.

[80] Hervé Marchand, Olivier Boivineau, and Stéphane Lafortune. Optimal control of discrete event systems under partial observation. In *Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No. 01CH37228)*, volume 3, pages 2335–2340. IEEE, 2001.

[81] Rong Su, Jan H Van Schuppen, and Jacobus E Rooda. The synthesis of time optimal supervisors by using heaps-of-pieces. *IEEE Transactions on Automatic Control*, 57(1):105–118, 2012.

[82] Sasinee Pruekprasert, Toshimitsu Ushio, and Takafumi Kanazawa. Quantitative supervisory control game for discrete event systems. *IEEE Transactions on Automatic Control*, 61(10):2987–3000, 2015.

[83] Shaull Almagor, Udi Boker, and Orna Kupferman. Formally reasoning about quality. *Journal of the ACM (JACM)*, 63(3):24, 2016.

[84] Krishnendu Chatterjee, Mickael Randour, and Jean-François Raskin. Strategy synthesis for multi-dimensional quantitative objectives. *Acta Informatica*, 51(3-4):129–163, 2014.

[85] Krishnendu Chatterjee, Thomas A Henzinger, Barbara Jobstmann, and Rohit Singh. Quasy: Quantitative synthesis tool. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 267–271. Springer, 2011.

[86] Alberto Camacho, Meghyn Bienvenu, and Sheila A McIlraith. Finite ltl synthesis with environment assumptions and quality measures. In *Sixteenth International Conference on Principles of Knowledge Representation and Reasoning*, 2018.

[87] Alfonso Gerevini and Derek Long. Plan constraints and preferences in pddl3. Technical report, Technical Report 2005-08-07, Department of Electronics for Automation âĂę, 2005.

[88] Jorge A Baier, Fahiem Bacchus, and Sheila A McIlraith. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence*, 173(5-6):593–618, 2009.

[89] Emil Keyder and Hector Geffner. Soft goals can be compiled away. *Journal of Artificial Intelligence Research*, 36:547–556, 2009.

[90] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[91] Jorge A Baier, Sheila A McIlraith, et al. Planning with preferences. *AI Magazine*, 29(4):25–25, 2008.

[92] Craig Boutilier, Ronen I Brafman, Carmel Domshlak, Holger H Hoos, and David Poole. Cp-nets: A tool for representing and reasoning withconditional ceteris paribus preference statements. *Journal of artificial intelligence research*, 21:135–191, 2004.

[93] Ronen I Brafman, Carmel Domshlak, and Solomon Eyal Shimony. On graphical modeling of preference and importance. *Journal of Artificial Intelligence Research*, 25:389–424, 2006.

[94] James P Delgrande, Torsten Schaub, and Hans Tompits. A general framework for expressing preferences in causal reasoning and planning. *Journal of Logic and Computation*, 17(5):871–907, 2007.

[95] Meghyn Bienvenu, Christian Fritz, and Sheila A McIlraith. Planning with qualitative temporal preferences. *KR*, 6:134–144, 2006.

[96] Enrico Giunchiglia and Marco Maratea. Planning as satisfiability with preferences. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 22, page 987. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[97] Tran Cao Son and Enrico Pontelli. Planning with preferences using logic programming. *Theory and Practice of Logic Programming*, 6(5):559–607, 2006.

[98] Ronen I Brafman and Yuri Chernyavsky. Planning with goal preferences and constraints. In *ICAPS*, pages 182–191, 2005.

[99] William Cushing, Subbarao Kambhampati, Mausam, and Daniel S. Weld. When is temporal planning really temporal? In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 1852–1859, 2007.

[100] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[101] Alessandro Cimatti, Minh Do, Andrea Micheli, Marco Roveri, and David E Smith. Strong temporal planning with uncontrollable durations. *Artificial Intelligence*, 256:1–34, 2018.

[102] Alessandro Cimatti, Andrea Micheli, and Marco Roveri. Strong temporal planning with uncontrollable durations: A state-space approach. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 3254–3260, 2015.

[103] Alessandro Cimatti, Andrea Micheli, and Marco Roveri. Solving strong controllability of temporal problems with uncertainty using smt. *Constraints*, 20(1):1–29, 2015.

[104] Carlo Combi, Roberto Posenato, Luca Viganò, and Matteo Zavatteri. Conditional simple temporal networks with uncertainty and resources. *J. Artif. Int. Res.*, 64(1):931âĂŞ985, January 2019.

[105] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. *IFAC Proceedings Volumes*, 31(18):447–452, 1998.

[106] Marcin Jurdziński and Ashutosh Trivedi. Reachability-time games on timed automata. In *International Colloquium on Automata, Languages, and Programming*, pages 838–849. Springer, 2007.

[107] A. Cimatti, , L. Palopoli, and Y. Ramadian. Symbolic computation of schedulability regions using parametric timed automata. In *Real-Time Systems Symposium*, RTSS'08, pages 80–89. IEEE Press, 2008.

[108] Manas Chandra Saksena. *Parametric Scheduling for Hard Real-time Systems*. PhD thesis, College Park, MD, USA, 1994. UMI Order No. GAX95-14577.

[109] A Jovanović, Sébastien Faucou, Didier Lime, and Olivier H Roux. Real-time control with parametric timed reachability games. *IFAC Proceedings Volumes*, 45(29):323–330, 2012.

[110] Étienne André, Vincent Bloemen, Laure Petrucci, and Jaco van de Pol. Minimal-time synthesis for parametric timed automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 211–228. Springer, 2019.