POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOTTORATO DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

Integrating formal methods with industrial standards in the development of flexible manufacturing systems

Doctoral Dissertation of:
Luca Ferrucci

Advisor:
Prof. Dino Mandrioli

Tutor:
Prof. Luciano Baresi

Supervisor of the Doctoral Program:
Prof. Carlo Fiorini

2013 – XXVI

ABSTRACT

This thesis presents a complete, innovative, formal verification approach to model certain class of manufacturing systems, the Flexible Manufacturing Systems (FMS). The thesis is the final product of a collaboration project between a group of automatic control engineers of the "Istituto di Tecnologie Industriali ed Automazione" (ITIA) of the CNR of Milano and a group of informatics engineers of the "Dipartimento di Elettronica, Informazione e Bioingegneria" (DEIB) of the Politecnico di Milano. A complete, integrated Model-Driven environment is the final target of the project; this environment should support the designer from the modelling phase to the formal verification, to obtain an agile, flexible development process.

To overcome the difficulties in the use of formal languages, the approach is based on widely-used, graphical but semi-formal high level modelling languages. The most part of these languages have Model-driven environments to help system designers in the development process and have simple and familiar notations, but lack of a rigorous semantics. We chose one of the typical languages adopted in such environments supporting the development of FMS, namely Stateflow, and we encoded it in terms of formulae of the metric temporal logic TRIO; by this way we provided a rigorous, compositional, run-to-completion semantics for Stateflow, based on concepts of micro and macro-steps. The formal model and the axiomatization of the semantics allow the formal verification of a set of real-time qualitative or quantitative user-defined properties, using a fully automatic model checker, $\mathbb{Z}$ot, which has been developed internally by the group of the DEIB. The approach is general enough to allow the use of different modelling languages, logical formalisms or model-checking tools. The approach is illustrated and validated through a realistic case study.

In the evolution of the work, a new metric temporal logic, X-TRIO, has been developed as an extension to TRIO, to overcome the limitations of the time model of TRIO. In effect, many formal semantics of high level graphical languages, such as the one of Stateflow, adopt the abstraction of "zero-time transitions", which does not consume time. These however have several drawbacks in terms of naturalness and logic consistency, as a system may be modelled to be in different states at the same time. X-TRIO exploits concepts from non-standard analysis to model micro and macro-steps. In the thesis, the

expressiveness and decidability properties of the new metric temporal logic have been studied and analysed. Also, a plugin for $\mathbb{Z}$ot is presented to allow the use of the new logical formalism.

SOMMARIO

Questa tesi presenta un completo ed innovativo approccio alla verifica formale e alla modellazione di una ben specifica classe di sistemi di produzione, i sistemi manifatturieri flessibili o Flexible Manufacturing Systems (FMS). La tesi è il prodotto finale di un progetto di collaborazione tra un gruppo di ingegneri automatici del controllo dell' "Istituto di Tecnologie Industriali Ed Automazione" (ITIA) del CNR di Milano e un gruppo di ingegneri informatici del "Dipartimento di Elettronica, Informazione e Bioingegneria" (DEIB) del Politecnico di Milano. L'obiettivo finale del progetto è un ambiente completamente integrato di modellazione, basato su un approccio orientato al modello; questo ambiente dovrebbe supportare il progettista dalla fase di modellazione a quella della verifica formale, per ottenere un processo di sviluppo agile e flessibile .

Per superare le difficoltà nell'utilizzo dei linguaggi formali, l'approccio è basato su linguaggi ampiamente diffusi di modellazione di alto livello, possibilemnte grafici, ma purtroppo soltanto semi-formali. La maggior parte di questi linguaggi hanno sia ambienti integrati adatti ad aiutare i progettisti di sistemi nel processo di sviluppo, sia notazioni semplici e familiari, ma mancano di una semantica rigorosa.

Per questa tesi, abbiamo scelto uno dei linguaggi più usati per la modellazione di FMS, Stateflow, e lo abbiamo codificato usando formule della logica metrica temporale TRIO; per ottenere ciò, abbiamo sviluppato una semantica per Stateflow rigorosa e composizionale, di tipo run-to-completion, basata sul concetto di micro- e macro- steps . Il modello e la rigorosa formalizzazione della semantica permettono la verifica formale di un insieme di proprietà in tempo reale, sia qualitative che quantitative, definite dall'utente, utilizzando un bounded model checker, $\mathbb{Z}$ot, che è stato sviluppato internamente dal gruppo del DEIB. L'approccio è sufficientemente generale da consentire l'uso di diversi linguaggi di modellazione, formalismi logici o strumenti di model-checking. L'approccio è illustrato e validato attraverso un caso di studio realistico.

In una successiva evoluzione del lavoro, una nuova logica metrica temporale, X-TRIO, è stata sviluppata come estensione di TRIO, per superare i limiti del suo approccio temporale. Infatti, molte semantiche formali per linguaggi grafici di modellazione ad alto livello, come ad esempio quella di Stateflow,

adottano l'astrazione delle "transizioni a tempo zero", che non consumano tempo reale. Queste però presentano alcuni inconvenienti in termini di naturalezza e coerenza logica, in quanto un sistema può essere modellato in modo da essere in diversi stati nello stesso istante temporale. X-TRIO sfrutta i concetti dell'Analisi Non-Standard per modellare micro e macro-steps. Nella tesi, sono state studiate e analizzate le proprietà di espressività e decidibilità della nuova logica temporale metrica. Inoltre, è stato sviluppato un plugin per $\mathbb{Z}$ot per consentire l'utilizzo pratico del nuovo formalismo logico.

ACKNOWLEDGMENTS

Thanks to my parents Antonia and Rolando, for their patience.
Thanks to my advisor Dino Mandrioli and to the components of my group at the DEIB of the Politecnico di Milano, Matteo Rossi and Angelo Morzenti, for their fundamental ideas and contributions on the theoretical part of this thesis.
Thanks to the components of the group of automatic engineers of the Institute ITIA of the CNR di Milano, Emanuele Carpanzano and Mauro Mazzolini, for their fundamental contributions.
Thanks to my reviewer Silvio Ghilardi of the Università degli Studi di Milano
Thanks to my sister Veronica to help me keeping my sanity.
Thanks to my colleagues at the DEIB of the Politecnico di Milano, for the infinite hours spent to hear me talking about boring and tedious, but (I hope) very interested arguments.
Thanks to all of them, simply for their existence

CONTENTS

1 INTRODUCTION 1
1.1 Content of the thesis: a formal verification method for FMS . . 3
1.2 Structure of the thesis 7

2 HIGH-LEVEL LANGUAGES AND STANDARDS FOR THE MODELLING OF FMS 11
2.1 Statechart: an high-level language for the modelling of FMS . . 13
2.1.1 Syntax 13
2.1.2 Semantics 17
2.2 The industrial standard IEC-61499 29

3 OVERVIEW OF THE METHODOLOGY APPROACH: A CASE STUDY 35
3.1 The case study: a robotic cell 35
3.2 A semantics for Simulink/Stateflow 44
3.3 Temporal logic encoding of the case study 51
3.3.1 TRIO: a metric temporal logic 51
3.3.2 TRIO encoding of the case study 52
3.4 System properties verification and experimental results 58
3.5 Limitations: towards a new metric temporal logic 62

4 X-TRIO: A METRIC TEMPORAL LOGIC FOR FMS 65
4.1 The general X-TRIO logic 65
4.1.1 An introduction to Non Standard Analysis 65
4.1.2 X-TRIO syntax and semantics 67
4.1.3 Examples of usage of X-TRIO 69
4.2 Towards decidable versions of X-TRIO 71
4.2.1 A propositional X-TRIO 72
4.2.2 Expressiveness of X-TRIO$\frac{P}{T}$ 73
4.2.3 Decidability of X-TRIO$\frac{P}{T}$ 77
4.3 A decidable fragment of X-TRIO 78
4.3.1 A decision procedure for X-TRIO$\frac{P\text{-}R}{\mathbb{N}_+}$ 82
4.4 X-TRIO encoding of the case study 92
4.4.1 Variations to the composition semantics 94

5 RELATED WORKS 97
5.1 MD approaches for the development process of FMS 98
5.2 Formal specifications for reactive systems 102

6 TOOLS 111
6.1 Zot verification tool 111
6.2 Sf2Trio tool: encoding Stateflow/Simulink into X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ . . . 115

7 CONCLUDING REMARKS 123
7.1 Future Work 124

BIBLIOGRAPHY 127

LIST OF FIGURES

Figure 1 Synthesis of the methodology approach 4
Figure 2 A Statechart example diagram 14
Figure 3 A Basic FB 31
Figure 4 An ECC of a Basic FB 31
Figure 5 A FB network 32
Figure 6 Robotic Cell. 38
Figure 7 Developed IEC 61499 control solution 39
Figure 8 ECC of the robot controller module 40
Figure 9 Stateflow diagram of the controller of the robotic cell of Figure 6. 41
Figure 10 Stateflow diagram of component *Machine 1* of Figure 6. 42
Figure 11 Simulink diagram of the robotic cell. 44
Figure 12 Example of stutter transitions 50
Figure 13 Deadlocked run returned by ℤot. 61
Figure 14 Part of trace representing counters. 81
Figure 15 An example of ρ_S (with $\delta_\phi = 1$). 86
Figure 16 Overall architecture of ℤot 113
Figure 17 Model of Machine 1 in the input syntax of the Sf2Trio tool. 119
Figure 18 Model of the composed module of the robotic cell in the input syntax of the Sf2Trio tool. 120

LIST OF TABLES

Table 1 TRIO derived temporal operators. 52
Table 2 Some variations of TRIO derived temporal operators. . 53
Table 3 Test results. 60
Table 4 Translation schema τ. 88

1 INTRODUCTION

In the last years, FMS systems are increasingly required to operate in dynamic environments characterized by quick changes of the demand, and to deliver highly customized products; this, in turn, calls for the agile and fast reconfiguration of production cells. As a consequence, the complexity of automation solutions for manufacturing systems has become considerable. The capability to operate in dynamic environments characterized by quick changes of the demand, the request for highly customized products and the need for agile and fast re-configuration of production cells are the main reasons for which the complexity of modern automation solutions for manufacturing systems is strongly increased. At the same time, features like interoperability, portability and scalability are the key to reduce the huge costs and times needed to design and realize a new production system, or to modify an existing one. This challenging context requires new paradigms, based on the distribution of control onto a network of embedded components, to make the design, modification, integration and reconfiguration of resulting solution more agile [1]. Furthermore, structured approaches to control system design that support design and testing of the whole automation system must be adopted [2]. Then, the guidelines, methods and tools of a comprehensive development methodology must be defined that allow developers to specify complex automation systems in an easy and safe way, to maintain the traceability along the different design phases, and to describe the behaviour of the target system [3, 4]. A commonly structured methodology consists of several steps which are briefly described hereafter:

- *Control system specification* in which the process to be automated is described and the functional activities to be performed, as well as the purpose of the complete system are defined
- *Control system architecture and functional design* in which the control system is conceived and developed exploiting concepts and paradigms provided by reference models and standards
- *Software implementation* in which the real software solution is deployed by means of appropriate programming languages

- *Verification and Validation (V&V)* in which the structural correctness of the control code and the compliance of the behaviour of the automation system with its requirements are verified

These phases are not strictly ordered, such as in the classical waterfall model; in particular, the V&V phase, if it is performed using formal methods, is usually executed immediately after the functional design phase, since it allows the designer to correct errors or unwanted behaviours of the system, before its implementation, with a considerable saving of time. It is well known that, effectively, in engineering the two phases of *verification* and *validation* are different and, often, confused, although they are usually performed together. The verification phase corresponds to check if the software implementation of the system, or its functional design or formal model, fulfils the requirements identified during the control system specification phase; in other words, it consists in answering to the question: "are we building the thing right?". Instead, the validation phase corresponds to check if the requirements and the system design or model are consistent with the needs and the informal conceptual idea of the whole system given by the commissioner; in other words, it consists in answering to the question: "are we building the right thing?". We can note that, if the set of requirements is complete and fulfils the need of the commissioner, verification and validation phases coincide perfectly, so in this thesis we refer to them as a unique V&V phase.

The V&V phase is crucial to obtain a robust and reliable automation solution, but there is no commonly adopted effective approach for this phase. In effect, in the current industrial practice, most operating conditions of the developed system are not properly verified, and several design and implementation errors often remain unresolved until the commissioning phase due to the considerable complexity of the control logic and to the limited development time available. Nonetheless, the lack of proper identification and correction of such errors before final commissioning critically impacts on ramp-up time and costs as well as on production downtimes [5]. V&V in control systems can be addressed through simulation or formal approaches. Simulation is currently the most widely known and adopted technique for V&V of industrial automation systems. The deployment and implementation of simulation frameworks is quite simple thanks to the tools available for software- and hardware-in-the-loop simulation [6]. The main open problem of such an approach is the definition of the test cases to achieve complete and exhaustive model analysis. Therefore, the quality of the simulation results closely depends on a good definition of testing scenarios and verifying any possible

behaviour of the system still remains a difficult task. To overcome this limitation, formal verification approaches, which are able to exhaustively explore the execution space of a system model, have been studied and proposed for the design of manufacturing systems [7]. A possible field of application is in the production of a good set of model coverage tests [8] to be successively performed over the formal model, sometimes saving time over the execution of the exhaustive static analysis of the model, which suffers of the well-known state-space explosion problem. In most instances, formal verification techniques are based on modelling notations that are separate from those normally used by practitioners in their design work, and the mapping from the concepts of one notation to those of the other one is often difficult. In the next section, we describe the path of the work on the development of a formal verification approach for the modelling of FMS.

1.1 CONTENT OF THE THESIS: A FORMAL VERIFICATION METHOD FOR FMS

This thesis is the result of a collaboration between a group of automatic control engineers of the "Istituto di Tecnologie Industriali ed Automazione" (ITIA) of the CNR of Milano and a group of informatics engineers of the "Dipartimento di Elettronica, Informazione e Bioingegneria" (DEIB) of the Politecnico di Milano. The aim of the work is to give a solution to the problem of specifying an integrated development process for the control software of FMSs, using the MDE (Model-Driven engineering) approach described in the last section. In particular, a way to perform the V&V phase using formal methods. The resulted IDE (Integrated Development Environment) should support the designer from the modelling phase to the formal verification phase.

In Figure 1, are shown the main ingredients of my formal verification approach. It is based on the use of *model checking* as a formal method to perform the V&V phase. Informally speaking, the *model checking* is a method to algorithmically check whether a program (the model) satisfies a specification. The model and the properties are usually expressed in a specification language with a precise, rigorous semantics, such as temporal logics or specific languages for model checker tools such as Promela, the input language for the model checker SPIN. The problem can be expressed mathematically as: given a temporal logic formula p and a model M, both expressed in the specification language, with initial state s, decide if $M, s \models p$.

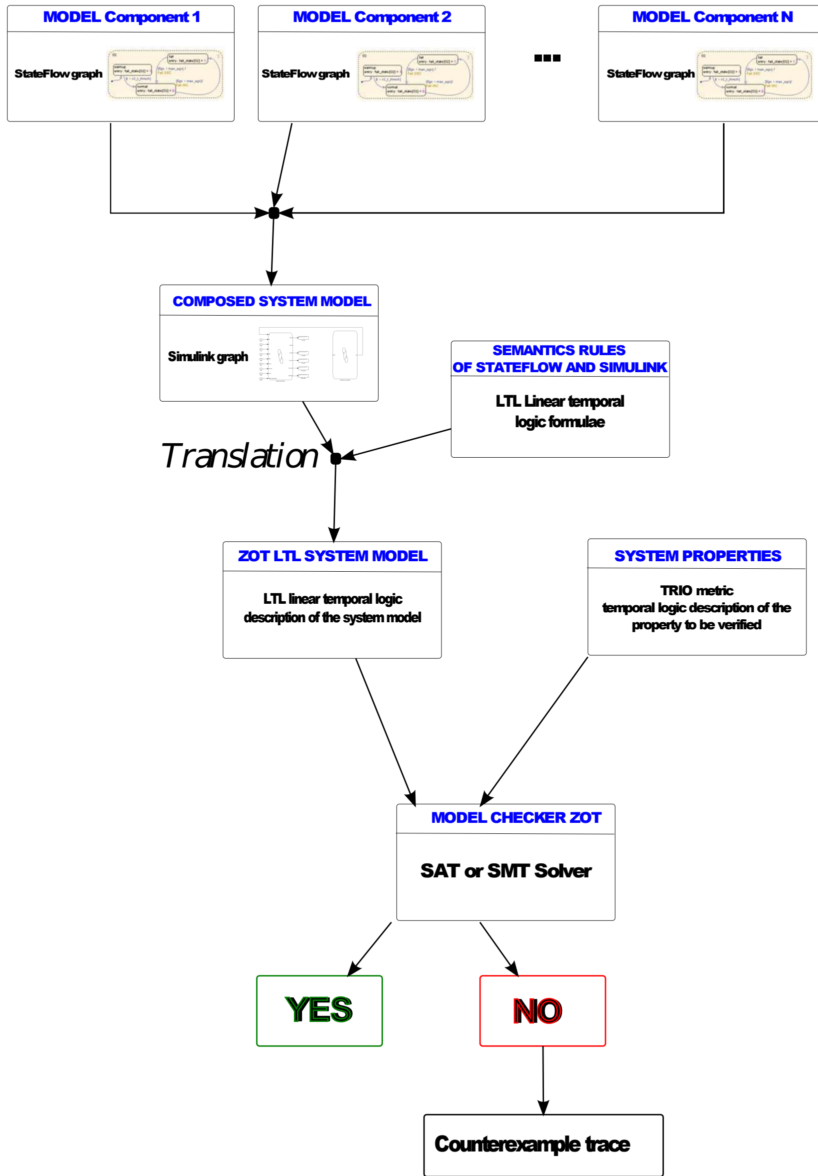

Figure 1: Synthesis of the methodology approach

The approach has been developed during the initial phase of the work. A crucial part of this phase was to identify a high-level formalism to model the system architecture and functional design of FMS, which accomplished to the following set of prerequisites identified by the group of the control engineers:

1. *easy-to-use*: the language should be easy to understand and use by practitioners or control engineers. Widely-used graphical languages are preferred, with an operational oriented notation similar to finite state machines, to provide them with a familiar notation.

2. *compositional*: the language must allow the decomposition of complex and distributed systems into a set of independent modules, which can be developed separately. Each module component must be kept small and simple in order to be understandable and easy to be verified; it can be considered a black box, with an external interface and an internal behaviour whose details are not important for the development of other components.

3. *abstract*: the language should allow to abstract from the details of the implementation of a component when they are not important, or to refine it when they are necessary, i.e. it must allow a hierarchical decomposition.

4. *easy to be verified*: the language should be easy to be verified using formal methods; for this reason, it should be possible to verify each component separately, in an exhaustive manner without the interaction of the user.

Specification languages such as temporal logics or languages similar to Promela, do not satisfy all these prerequisites; in particular, they are not easy to use for control engineers, since the effort needed to properly master them is too high. This is the main reason why model checking is not widely-used by practitioners. In effect, the cost of staff training and the number of errors in the resulted specifications may be so high to affect the degree of penetration of this technique in the design of complex, distributed systems in industry. Furthermore, these languages are, in most part, no compositional; in contrast, they are easy to be verified formally, by nature.

For these reasons, the group decided to choose a toolbox of high-level graphical languages, Stateflow/Simulink, which are suitable to model *reactive systems*; they are compositional, widely-used in the international community and well supported by Model-driven environments. The choice of a particular

modelling language is not mandatory for the application of the approach: it is possible to use any language which satisfy the above prerequisites and is suitable to model reactive systems; Statechart [9], the SFC notation of the IEC 61131-3 [10], the IEC 61499 [11] and timed Petri Nets [12] are some examples of such type of languages.

The main drawbacks in the use of these languages is their lack of a rigorous semantics, which is mandatory to allow the application of formal methods. Most of them are complex languages with an intricate semantics, usually explained in an informal way in the official documentation. To overcome this limitation, the group developed a formal semantics of the Stateflow/Simulink based notation, given in terms of a set of rules specified in a metric temporal logic, TRIO; the rules are then translated in Linear Temporal Logic (LTL), a common input formalism for a large class of model checkers. To foster information hiding, the formal semantics is compositional, so the prerequisites 2 and 4 listed above are satisfied. The logic-based semantics precisely captures and resolves the intricacies (and possibly the hidden ambiguities) of the design notation, and it is used to formally check whether user-defined properties of interest are satisfied by the system model or not. In particular, thanks to the metric nature of the logic-based language underlying the approach, i.e. TRIO, Stateflow models are provided with a precise, metric, notion of time; this is exploited, on the one hand, to introduce metric constraints in the models, and on the other hand to allow users to analyse real-time properties.

To satisfy all above prerequisites, the idea was to translate a model of the FMS specified in Stateflow/Simulink into an equivalent LTL formal model, embedding the set of semantic rules specified in TRIO, and translated in LTL, into the model.

The guideline to apply the approach to the design of an FMS is the following:

- The system designer must provide the model of the manufacturing system, decomposing the architecture in a set of components, each of which should be specified by a Stateflow diagram; the external interface of the component (unless the system has a single component) must be specified as described in section 3.1
- The overall system can be composed, if needed, using a Simulink-like notation, exploiting the external interfaces of the components; the rules for the composition are described in section 3.1

- The obtained set of semantic rules, specified in TRIO, must be embedded in each component and integrated in the overall model of the system defined in Simulink/Stateflow, obtaining an LTL formal model
- The formal model of the system (or of a single component) and the set of real-time properties to be checked, are given as input to the bounded model checker ℤot to be analysed; ℤot[1] is described in section 6.1
- ℤot performs the checks by encoding temporal logic formulae into the input language of various solvers; ℤot supports two kinds of solvers: SAT solvers and SMT solvers. After the solver performs the analysis, if the property does not hold, a counterexample trace is returned to help the designer to identify and correct the error

In a successive evolution of the work, two further steps have been introduced in the methodology approach. First, the new standard modelling language, IEC 61499 [11], has been used to model a simple case study; it is largely used in literature and in industry to model FMSs. ISaGRAF6 [13] is a model-driven environment that is an implementation of this standard. We will see in chapter 3 that, exploiting the rules defined in [14], it is possible to translate in an easy and straightforward way an IEC 61499 model into a corresponding Stateflow/Simulink diagram, maintaining the semantics of the IEC 61499 standard.

Finally, a new metric temporal logic has been developed, called X-TRIO, which is an extension of TRIO; it exploits the concepts of Non-Standard Analysis [15] to introduce a new model of time, which captures and solves, at the semantic level, some theoretical limitations of the original TRIO formalization. A plugin for ℤot, which implements X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$, the decidable subset of X-TRIO described in section 4.3, has been developed during this part of the work. The motivations, and the limitations of the original formalization logic are given in section 3.5.

1.2 STRUCTURE OF THE THESIS

This thesis is structured as follows:

- In chapter 2 the most widely-used high level graphical languages for the modelling of FMS are described and analysed. In the first section,

1

the syntax and the various semantics of Statechart have been described; Statechart is a modelling language introduced by [16] and is the ancestor of Stateflow. The analysis of the various semantics shows the complexity and difficulties in the development of a rigorous semantics for a semi-formal complex language. In the second section, the syntax and the described part of the informal semantics of the IEC 61499 standard has been described, with a brief analysis of the most common differences between the various implementations of the standard, especially its execution model

- In chapter 3, a complete case study of a simple FMS, a robotic cell, is developed to demonstrate the validity of the approach. In the first section is described the case study, developed using the IEC 61499 standard and translating in the Simulink/Stateflow notation. In the second section, a formal semantics of Simulink/Stateflow is given, to allow the encoding of the case study in TRIO, described in the third section. Finally, a set of interested qualitative and quantitative real-time properties are analysed and formalized in TRIO, with a set of experimental results obtained using $\mathbb{Z}$ot on the case study

- In chapter 4 the new temporal logic X-TRIO is deeply analysed. In the first section, the main concepts of NSA are exposed. Then, the syntax and the structure semantics of the main (undecidable) version of the logic are given, with some examples of the usage. In the second section, to accomplish the not easy task of achieving a decidable but yet general enough version of the logic, a propositional syntax is given, simplifying the semantics; in the second part of the section, some expressiveness results are exposed, and the first demonstration of its undecidability is given. In the same section, a yet undecidable fragment of X-TRIO, called X-TRIO$^{P}_{\mathbb{N}_+}$, is defined, introducing a discrete temporal domain and the demonstration of the undecidability of X-TRIO$^{P}_{\mathbb{N}_+}$. Finally, in the third section, a sufficient condition to obtain a decidable fragment of the logic is given, with a decision procedure based on the syntax translation of the logic in PLTLB (Propositional Linear Temporal Logic with Both past and future operators). Then, the semantics of Stateflow is re-encoded exploiting the new logic

- In chapter 5 some related works are described and analysed deeply. The first section describes some works related to different model-driven engineering approaches to the development process of FMS using formal

methods, focusing on the main differences between them and this thesis. The second section, instead, describes alternative approaches to NSA to deal with zero-time transitions, and some works which exploit NSA in a different way

- In chapter 6 the tools that have been used in this thesis to model and perform the formal verification of the robotic cell case study are described. The first section presents the architecture of the model checker ℤot, and the main plugins, in particular the X-TRIO plugin. The second section describes the Sf2Trio tool, another plugin for ℤot that allows the manual translation of a Stateflow/Simulink model into a logic model in a simple and straightforward way, without writing the necessary TRIO formulae directly. It represents the first step towards an integrated Model-driven environment, which would be the final result of the collaboration project

2 HIGH-LEVEL LANGUAGES AND STANDARDS FOR THE MODELLING OF FMS

The design process for FMS has changed dramatically over the last few years. Increasingly, designers use a model-driven approach, exploiting model-based development environments; these allow the system, including its software, the plant that it will control and the environment in which it will operate, to be represented in graphical form at a high-level of abstraction. Model-based development environments provide extensive tools for validation through simulation, and code generators that can compile an executable controller from its graphical representation.

The evolution to model-based development has been driven by the increasing complexity of FMSs and of distributed control systems. Alongside these developments, there has been an increase in criticality, with regard to both human safety and the cost of faults. This increasing criticality creates a need for improved methods of analysis and verification, and this provides an opportunity for formal verification methods like model checking. However, a lot of graphical modelling languages to model FMS were not built with formal methods in mind, and do not appear to be well suited for formalization: in effect, they lack a formal and rigorous semantics. In this chapter, one of the most widely used graphical modelling languages has been described: the IEC 61499 standard [11]. The IEC 61499 is a recent international standard introduced by the IEC group, which is an advancement of the IEC 61131 standard [10]: it is well suited to develop distributed complex systems thanks to its compositional and distributed nature. For these properties, and mainly to its recent standardization, we have chosen the IEC 61499 standard to model a case study to prove the validity of the approach described in the introduction to this thesis.

The standard is based on the concept of *Function Blocks* (FB), that is the basic building block from which entire applications may be built. A basic function block executes an elementary control function, i.e. it is an abstraction of a single component of an FMS, like for example a manufacturing machine. Its behaviour is described by using a finite state machine known as *execution control chart* (ECC); this internal notation is similar to another well-known and widely used graphical notation language, Stateflow. Stateflow is

an extension of Statechart developed by Mathworks [9]; like Statechart, it has been introduced to overcome the limitations of state transition diagrams, to obtain a very succinct and intuitive representation. Stateflow is based on the Statemate semantics of Statechart [17].

In this thesis, it has been decided to translate the ECC notation into Stateflow, to provide to control and automatic engineers a familiar notation. In chapter 3, rules to translate an ECC into a Stateflow diagram are given. Not all the features of Stateflow have been exploited in this thesis, in particular the compositional state types (OR and AND-states), which are useless due to the flat nature of the ECC notation.

For completeness, and for a better comprehension of the features and the semantics difficulties in developed a rigorous semantics for Stateflow, in this chapter Statechart is described, which is the graphical notation language introduced by Harel in [16], while Stateflow is described in chapter 3. In the exposition, to simplify, only the basic elements of the original specification of Statechart have been taken in consideration, which include its most popular features, ignoring time actions, history, special events (e.g., events generated when a state is entered or exited), special actions (e.g. start action, history clear, deep clear), condition expressions and data items.

Condition expressions and data items have been introduced later in Statechart, as a natural evolution of the original language to increase its expressiveness. For this reason, a lot of subsequent extensions of Statechart, such as Stateflow itself, are based on them: in particular, the version of Stateflow described in this thesis, differently from the original one, implements the events using boolean variables. So, syntax and semantics of condition expressions and data items are described deeply in the chapter 3, introducing a suitable notation and all the obvious semantics intricacies. Instead, although not used in this version of Stateflow, for historical reasons compositional states are considered in the exposition of Statechart; another motivation is that, in the methodology described in this thesis, parallel and sequential decompositions are obtained exploiting Simulink diagrams, so a comparison between the two different approaches is possible.

The next two sections describe the main features of Statechart and the IEC 61499 standard, their syntax and the various semantics (especially for Statechart) that have been developed during last decades, to provide a look into the differences in the execution models and the difficulties in developing a complete formal verification process for them and, in the case of Statechart, their extensions.

2.1 STATECHART: AN HIGH-LEVEL LANGUAGE FOR THE MODELLING OF FMS

In this section, the syntax and the various semantics of Statechart are introduced and analysed.

Statechart has been proposed to overcome the limitations of state transition diagrams that are flat and unstructured, while preserving their visual nature. They exploit the notions of parallelism, hierarchy and broadcast communication. These features allow for very succinct representations and naturally support stepwise development. Thanks to these properties, and to the graphical simple notation, it has been widely used in industry, especially in the field of automation of large industrial processes or plants, such as in aviation systems, mechatronics systems, embedded systems or in the modelling of FMS. The main drawback is that its complexity has prevented the development of a rigorous semantics that allows the formal verification of systems modelled in this language: in last decades, various approaches have been investigated, which led to a proliferation of different semantics. They can be divided in three main groups, that are described in section 2.1.2. The existence of these different Statechart formalisations can lead to a Babel-like confusion, because the same Statechart can be interpreted in a completely different way under various semantics. This confusion impedes the communication of the meaning of Statechart designs, since that meaning largely depends on the actual semantics the viewer (not the designer) is using. In addition, it hampers the exchange of Statechart designs among different software tools.

2.1.1 *Syntax*

In the next subsection, the syntax of the graphical modelling language Statechart is described in an informal way. In subsection 2.1.1 the syntax is presented in a more rigorous, formal way.

Informal syntax

The most basic elements in Statechart are *states*. A state is usually represented by a variable sized rectangle with rounded edges. There are different kinds of states which are called *OR-states*, *AND-states* and *BASIC-states*. More specifically, a BASIC-state is a state such that it has no sub-OR-states, sub-AND-states and sub-BASIC-states. OR-states are complex states with one or more states defined internally; an OR-state represents an *abstraction* of its sub-states,

i.e. the internal states composed themselves a Statechart diagram which is a refinement of the OR-state, with a *default* state which represents the initial state of the sub-diagram; in particular, when the system is in an OR-state, it is in one and only one of its sub-states. AND-states are complex states which have two or more *orthogonal components*. Each of these components, as OR-states, are themselves Statechart diagrams which are executed concurrently; in other words, AND-states allow the parallel decomposition of a state. The orthogonal components of an AND-state are normally drawn with a dashed line. In this case, when the system is in an AND-state, it is also in one and only one states of any its components. In Figure 2, an example of a Statechart diagram is shown which is used in the rest of this chapter to explain the main features and the differences between the various semantics of Statechart.

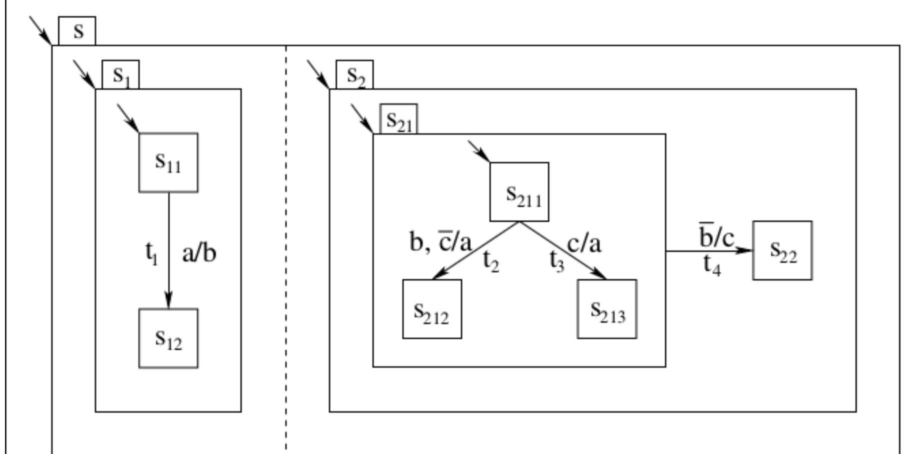

Figure 2: A Statechart example diagram

S_1, S_2 and S_{21} are OR-states, S is an AND-state and all the other ones are BASIC-states.

Using the concept of sub-states, the states of a Statechart diagram form a *hierarchy* of levels; the *parent* of a state is, by definition, the unique AND or OR-state that directly includes it. In Figure 2, S is the parent of S_1 and S_2, while S_1 is the parent of S_{11} and S_{12}. The state at the maximal level of the hierarchy, without a parent, is the *root* of the diagram. Usually, the root is an OR-state which is not always represented in the Statechart, as in Figure 2.

Transitions are used in order to specify system dynamics in a Statechart diagram. A *transition* is represented as an arrow which denotes that the system changes its current state from the one attached to the tail of the arrow, called *source state*, to the one attached to the head of the arrow, called the *destination state* of the transition. Transitions can leave and enter states on any level of the hierarchy.

Transitions are normally labelled with a construct of the form e/a; it denotes that the transition fires when the event e occurs. a is an action which can be carried out when the transition is taken. An *event* is a signal or message that can be generated by the external environment as input to the Statechart diagram, or generated internally as a result of the execution of an action. A special event is for example $\mathtt{entered}(S)$ that occurs (and hence causes the transition to take place) when state S is entered. An action, instead, can do one or more of the followings things:

- trigger an internal event
- launch an *activity*, which is an externally defined operation or algorithm which takes a certain quantity of time to be performed and is not under the direct control of Statechart. Statechart has the ability to start and stop activities to model behaviours of a system defined externally and treated as "black boxes".

Events can be combined with the usually boolean operators *or*, *and* and *not*. Events and actions are optional; a transition labelled with no events is always enabled and is forced to fire immediately.

Besides allowing actions to appear along transitions, they can also appear associated with the entry to or exit from a state. Actions associated with the entry to a state S are executed when S is entered, as if they appear on the transition leading into S. Similarly, actions associated with the exit from S are executed when S is exited, as if they appear on the transition exiting from S.

Each state can be associated with *static reactions*, which are of the same format of a transition label and are executed as long as the system is in (and is not exiting) the state in question.

In the Statechart diagram in Figure 2, t_2 is an example of a transition between the source state S_{211} and the destination state S_{212}, labelled with $b, \bar{c}/a$, where the comma is an abbreviation for the *and* operator and $\bar{c}$ is an abbreviation for $\mathtt{not}(c)$; the transition fires when the event b happens and the event c does not happen, triggering the event a. In the example diagram, there are neither actions entering or exiting states, nor static reactions.

Formal syntax

In this section, a formal syntax for Statechart is introduced, using the method in [18]. It is useful to define and explain rigorously the differences in the various semantics of Statecharts that are analysed in the successive sections of this chapter.

In the formal syntax described in [18], a Statechart *SC* is a tuple $SC = \{S, T, E\}$ where:

- S is a finite set of *states*,
- T is a finite set of *transitions* that connect the states
- E is a finite set of *events*. Set E is divided into sets E_{ext} and E_{int}. Set E_{ext} contains all external events, which are generated by the environment, while set E_{int} contains all internal events, which are generated by transitions in T. They are not necessarily disjoint sets.

The environment is the external world which provides external events for the Statechart diagram. Function $children : S \to \wp(S)$ defines for each state s its immediate substates. If s is a child of s' , we call s' the parent of s, as defined informally in the previous section. By $children^*$ and $children^+$ we denote the reflexive-transitive and transitive closures of $children$, respectively. If $s \in children^*(s')$, we say that s is a descendant of s' and that s' is an ancestor of s. If s is ancestor or descendant of s' , then s and s' are ancestrally related. A state s is a basic state if $children(s) = \emptyset$, otherwise it is an OR or an AND-state. The function $type : S \to \{Basic, OR, AND\}$ assigns to each state its type. The root state is identified with the identifier $root$. Function $default : S \to S$ identifies for each OR-state s one of its children as the default state: $default(s) \in children(s)$. If a transition t enters s but does not explicitly enter any of its children, then t enters $default(s)$. For example, in Figure 2, $children(S_2) = \{S_{21}, S_{22}\}$, while $children^*(S_2) = \{S_{21}, S_{22}, S_{211}, S_{212}, S_{213}\}$; finally, $default(S_{21}) = S_{211}$.

For a set X of states, the least common ancestor (lca) of X , denoted $lca(X)$ is the state x such that:

- $X \subseteq children^*(x)$
- for every $y \in S$ such that $X \subseteq children^*(y)$, we have that $x \in children^*(y)$

Every set of states has a unique least common ancestor. Two states x and y are orthogonal, written $x \perp y$, if x and y are not ancestrally related, and their

lca is an AND-state. For example, in Figure 2, S_{11} and S_{212} are orthogonal since they are not ancestrally related and $lca(S_{11}, S_{212}) = S$, where $type(S) = AND$; instead, S_{211} and S_{22} are not orthogonal since, although they are not ancestrally related, $lca(\{S_{211}, S_{22}\}) = S_{21}$, where $type(S_{21}) = OR$, i.e. S_{21} is not and AND-state.

2.1.2 *Semantics*

Central to the question of formal verification of Statechart is its semantics. Unfortunately, defining the semantics of a complex notation like Statechart is not a straightforward task. The original Statechart paper [16] only hinted informally at how a semantics could be defined. In literature exists different formalizations, each of which uses its own assumptions in defining an execution semantics, but it is possible to identified a common set of basic ingredients and definitions. In this section, the common design decisions taken by the main semantics of Statechart are described, in the first part, in an informal way; instead, in the second part of the section, they are presented in a more rigorous, formal way. Then, in the next sections, the main theoretical differences between the various semantics are described and analysed deeply. This in-depth analysis will serve to guide and illustrate the choices made in the methodology presented in this thesis.

The semantics of a Statechart diagram is a set of *runs*, representing the reaction of the actual system to a sequence of events or stimuli. A run consists of a series of detailed snapshots of the system's situation; such a snapshot is called a *status* and is different from a state of the diagram. The first status of a run is called the *initial status*, while each subsequent one is obtained from its predecessor by executing a *step*. A status contains informations about active states and activities, values of data-items and conditions, generated events and scheduled actions, and some informations regarding the system's history (its past behaviour). At the beginning of each step, the environment supplies the system under description with a set of initial external events. These events, together with changes that occurred in the system during and since the previous step, trigger transitions between states and static reactions within states. As a result, the system moves into a new status. Some states are exited, and some are entered; values of conditions and data-items are modified; new events are generated; activities are started and stopped, and so on.

In Statechart, a status is formally represented by a *configuration*, which is a maximal consistent set of states that the system can be in simultaneously. The concepts of consistency and maximality are given formally later on in this section, using the definition given in section 2.1.1. More precisely, given a root state r, a configuration (relative to r) is a set of states C obeying the following rules:

- $r \in C$
- Given an OR-state $s \in C$, exactly one of s's substates is in C
- Given an AND-state $s \in C$, all of s's substates are in C
- The only states in C are those that are required by the above rules.

It follows that configurations are "closed upwards"; that is, when the system is in any state s, it must also be in s's parent state. Since it is enough to know its basic states to uniquely determine a configuration, we use the term *basic configuration* to refer to a maximal set of basic states that the system can be in simultaneously, or in other words, the set of basic states in a legal configuration. For example, if the actual configuration of the diagram of Figure 2 is $C = \{root, S, S_1, S_2, S_{11}, S_{21}, S_{211}\}$, the equivalent, unique and more concise basic configuration is $C' = \{S_{11}, S_{211}\}$.

In a *step*, the system typically carries out operations of four types: transitions, static reactions, actions performed when a state is entered, and actions performed when a state is exited. Similar to the definition of a configuration, a step is a maximal consistent set of *enabled* transitions. Again, concepts of consistency and maximality for a transition are given formally later on in this section. Given a system in a certain basic configuration C, once an external event e is generated, transitions which are labelled e/a become enabled if their source state is in C. Then the system proceeds to execute actions a and update the current configuration, adding the transition's target states of the enabled transitions to C and eliminating the source states from it. We need to distinguish between the case of leaving an OR or an AND-state. In the former case, the outgoing transition from the OR-state is enabled only if the sub-Statechart (the part of the Statechart diagram which is contained graphically in the OR-state) is in the final state. In the latter case, it is enabled only if *all* sub-Statecharts are in a final state; a final state is a state without outgoing transitions. For example, if the actual basic configuration of the diagram in Figure 2 is $C = \{S_{12}, S_{211}\}$ with an emply set of pending events, and the event

c is provided by the environment, then the transition t_3 becomes enabled and, when the system executes it (transitions are executed instantaneously, but the order in which the system performs one of the enabled transitions, if there are many, depends on the semantics), the internal event a is generated and added to the set of pending events; the next basic configuration of the system is $C' = \{S_{12}, S_{213}\}$, and the set of pending events becomes $E' = \{a\}$.

After executing a step, the system reaches a so called *stable configuration*, then it waits for the occurrence of a new external event. Once reached a stable configuration, every previous generated events become invalid. For example, the execution of the above transition t_3 is not a step, since in the basic configuration $C' = \{S_{12}, S_{213}\}$ with the set of pending events $E' = \{a\}$ the transition t_4 is enabled; when the system executes it, the system enters the basic configuration $C'' = \{S_{12}, S_{22}\}$ with an invariant set of pending events $E' = \{a\}$; C'' is a stable basic configuration, since there are no outgoing enabled transitions.

The different proposals semantics presented in literature can be classified in three main approaches:

1. Proposals of the first approach are based on the *fixpoint semantics* for Statechart, initially proposed in [19]
2. Proposals of the second approach focus on the semantics as implemented in the Statemate tool set, initially proposed in prose in [17]. It is called the *Statemate semantics* of Statechart.
3. The last group of formalisations focuses on Statechart for object-oriented systems. This group is expanding quickly due to the incorporation of Statechart in UML, the emerging de facto standard for modelling software systems. So it is called the *UML semantics* for Statechart, and it is officially proposed in prose in [20]

The various semantics differs in the step's execution, in the definition of a stable configuration, in the evaluation order of events (which influences the execution order of transitions) and other peculiarities described in the next sections.

In the rest of this section a set of definitions needed to define formally what is a step, a configuration and how to perform the update of the configuration is given; they are the common parts of the various semantics approaches.

Formally, a set X of states is *consistent* if for every $x, y \in X$, either x and y are ancestrally related or $x \perp y$. A consistent set X is *maximal* if for every state $s \in S \backslash X$, $\{s\} \cup X$ is not consistent. These definitions are needed to define a

configuration as a maximal consistent set of states, as mentioned above. For example, the set $X = \{S_{11}, S_{12}\}$ is not consistent, the set $Y = \{S_{11}, S_{21}\}$ is consistent but not maximal, while the set $Z = \{root, S, S_1, S_2, S_{11}, S_{21}, S_{211}\}$ is a maximal consistent set of states, i.e. it is a configuration.

Given a consistent set X of states, the *default completion dcomp(X)* is the smallest set D such that:

- $X \subseteq D$
- if $s \in D$ and $type(s) = AND$ then $children(s) \subseteq D$
- if $s \in D$ and $type(s) = OR$ and $children(s) \cap X = \emptyset$ then $default(s) \subseteq D$
- if $s \in D$ and $s \neq root$ then $parent(s) \in D$

For example, for the diagram in Figure 2, the default completion of the set of states $X = \{S_{11}\}$ is $dcomp(X) = \{S_{11}, S_1, S_2, S_{21}, S_{211}, S, root\}$. For each transition $t \in T$, $source(t)$ denotes the set of source states of t and $target(t)$ the set of target states: $source, target : T \rightarrow \wp(S)$. If the source state of a transition is an AND-state, then all the basic states in the AND-state that are active are in the set $source(t)$; on the contrary, if the target state of a transition is an AND-state, then all the basic states in the AND-state that are active are in the set $target(t)$. In all the other cases, the two sets are singletons.

To ensure that a transition can get enabled and enters a valid next configuration, we require that both $source(t)$ and $target(t)$ are consistent and non-empty.

The scope of a transition is the most nested OR-state that contains both $source(t)$ and $target(t)$. Thus, it equals $l = lca(source(t) \cup target(t))$ only if l has type OR, which is usually the case.

The event that triggers a transition t is denoted by $event(t)$. If a transition has no trigger event, we use the special event $null$.

We classify transitions according to their trigger events:

- A transition t is *external* if $event(t) \in E_{ext}$
- A transition t is *internal* if $event(t) \in E_{int} \setminus E_{ext}$
- A transition t is a *completion* transition if $event(t) = null$

The set of events generated by a transition t is denoted $action(t)$. We require that $action(t) \subseteq E_{int}$. The set of events generated by a set T of

transitions is denoted: $generated(T) = \bigcup_{t \in T} action(t)$. A transition t triggers transition t', written $t \gg t'$, if the trigger of t' is generated by t: $t \gg t' \Leftrightarrow event(t') \in action(t)$. We note that a transition can trigger itself.

Given a configuration $C \subseteq S$ and a set $I \subseteq E$ of external events, we assert that a transition is *relevant* if its sources are in C. The set of relevant transitions is defined as $relevant(C) = \{t \in T | source(t) \subseteq C\}$. Using these definitions, we can formally give the notion of enabling of a transition: a transition t is enabled if it is relevant in C and the trigger event of t is in I or $null$, i.e. the set of enabled transitions is defined as:

$$enabled(C, I) = \{t \in T | t \in relevant(C) \wedge event(t) \in I \cup \{null\}\}$$

Two transitions t_1 and t_2 are consistent if either they are the same transition, or their scopes are orthogonal:

$$consistent(t_1, t_2) \Leftrightarrow t_1 = t_2 \vee scope(t_1) \perp scope(t_2)$$

A set T of transitions is consistent if every pair of transitions in the set is consistent:

$$consistent(T) \Leftrightarrow \forall t_1, t_2 \in T : consistent(t_1, t_2)$$

Two transitions t_1 and t_2 conflict if $t_1 \neq t_2$, their sources are consistent and $scope(t_1)$ and $scope(t_2)$ are ancestrally related:

$$\begin{aligned} conflict(t_1, t_2) \quad \Leftrightarrow & \quad t_1 \neq t_2 \wedge consistent(source(t_1), source(t_2)) \\ \wedge & \quad scope(t_1) \text{ and } scope(t_2) \text{ are ancestrally related} \end{aligned}$$

Note that transitions t_1 and t_2 can be inconsistent yet not conflicting. In Figure 2 the transitions t_3 and t_4 are conflicting, since their source states are consistent, while transitions t_1 and t_2 are consistent and not conflicting; there are not inconsistent yet not conflicting transitions in the example diagram.

Finally, a set T of transitions is maximal if adding an enabled transition to T would result in an inconsistent set:

$$maximal(T, C, I) \Leftrightarrow \forall t \in enabled(C, I) \setminus T : \neg consistent(T \cup \{t\})$$

Using these auxiliary definitions, we can now formally define a step. A set of transitions St, given a configuration $C \subseteq S$ and a set $I \subseteq E$ of external events, is a step if and only if St is enabled, consistent and maximal:

$$isStep(St, C, I) \Leftrightarrow St \subseteq enabled(C, I) \wedge consistent(St) \wedge maximal(St, C, I)$$

For example, in Figure 2, given $St = \{t_3\}$, $C = \{root, S, S_1, S_2, S_{12}, S_{21}, S_{211}\}$ and $I = \{c\}$, $isStep(St, C, I)$ is true since the transition t_4, although enabled, is not consistent with t_3, while t_1 and t_2 are not enabled; instead, given $St = \{t_3\}$, $C = \{root, S, S_1, S_2, S_{11}, S_{21}, S_{213}\}$ and $I = \{a\}$, $isStep(St, C, I)$ is false since $maximal(St, C, I)$ is not true: in fact, the transition t_4 is enabled and consistent with t_3, but it does not belong to St.

To define the effect of taking a step, we need further definitions. First, we observe that by taking a transition t, only states below $scope(t)$ are left and entered. The states entered by t, denoted $enters(t)$, are the states below $scope(t)$ that are in $dcomp(target(h))$:

$$enters(t) = dcomp(target(t)) \cap children^*(scope(t))$$

Given a configuration C and step St, the function $nextConfig(C, St)$ defines the configuration reached by taking St:

$$nextConfig(C, St) = C \setminus \bigcup_{t \in St} children^*(scope(t)) \cup \bigcup_{t \in St} enters(t)$$

Thus, for each transition $t \in St$, the states in C that are below $scope(t)$ are left, and the states in $enters(t)$ are entered. For example, for the diagram in Figure 2, given the configuration $C = \{S, S_1, S_2, S_{21}, S_{211}, S_{12}\}$ and the step $St = \{t_3\}$, $nextConfig(C, St) = C \setminus children^*(scope(t_3)) \cup enters(t_3)$, where $children^*(scope(t_3)) = \{S_{211}, S_{212}, S_{213}\}$ and $enters(t_3) = \{S_{213}\}$; then, $nextConfig(C, St) = \{S, S_1, S_2, S_{21}, S_{213}, S_{12}\}$.

To define formally the fixpoint, Statemate, and UML semantics, we use a symbolic transition system (STS). A symbolic transition system is a tuple $STS = \{D, init, \rightarrow\}$, where:

- D is a finite set of typed variables on some typed data domain $dom(D)$. σ is a *evaluation* of the variables of D, i.e. a mapping $\sigma : D \rightarrow dom(D)$. We denote with $\Sigma(D)$ the set of evaluations on D

- $init$ is a first-order predicate over variables in D characterising the initial evaluation

- $\rightarrow$ is a first-order transition predicate over variables in D, D', where the unprimed variables in D refer to the current evaluation, while the primed ones in D' to the next evaluation. For example the predicate $x = x' + 1$ relates an evaluation σ to a next evaluation σ' if and only if $\sigma'(x) = \sigma(x) + 1$; we then write $\sigma \rightarrow \sigma'$.

A run of an STS is an infinite sequence of evaluations $\sigma_0\sigma_1\sigma_2\ldots$ such that σ_0 is the initial evaluation that satisfies $\mathtt{init}$, and for each pair σ_i, σ_{i+1} of evaluations, $\sigma_i \rightarrow \sigma_{i+1}$, where $i \geqslant 0$.

Fixpoint semantics

In the fixpoint semantics, a Statechart maps to a symbolic transition system STS^{FP} . Variables of STS^{FP} are the current configuration C and the current set of input events I.

In the initial evaluation, the configuration is the default completion of root, with no input events:

$$\mathtt{init} \Leftrightarrow C = \mathtt{dcomp}(\{\mathtt{root}\}) \wedge I = \emptyset$$

In the example in Figure 2, $\mathtt{init} \Leftrightarrow C = \{\mathtt{root}, S, S_1, S_2, S_{11}, S_{21}, S_{211}\} \wedge I = \emptyset$.

In the fixpoint semantics, the system waits in a stable evaluation for events to occur and takes a single step in response. To formalise this, two kinds of transition predicates are needed. The first predicate, denoted with $\longrightarrow^{FP}_{event}$, models the occurrence of external events in a stable evaluation:

$$\longrightarrow^{FP}_{event} \Leftrightarrow \mathtt{stable}^{FP}(C, I) \wedge C = C' \wedge \emptyset \subset I' \subseteq E.$$

An evaluation is defined to be stable if there are no input events to be processed:

$$\mathtt{stable}^{FP}(C, I) \Leftrightarrow I = \emptyset$$

Next, if events I have occurred, the system reacts by taking a step St, formalised by predicate $\longrightarrow^{FP}_{step}$. A peculiar feature of the fixpoint semantics is that events generated in the current step are sensed immediately. That is, transitions triggered by generated events are enabled immediately and are taken in the same step. Thus, generated internal events are additional input events for the isStep predicate.

$$\begin{aligned}\longrightarrow^{FP}_{step} \Leftrightarrow \quad & \neg \quad stable^{FP}(C, I) \wedge \exists St \subseteq T : isStep(St, C, I \cup generated(St)) \\ & \wedge \quad C' = nextConfig(C, St) \wedge I' = \emptyset\end{aligned}$$

This definition does not satisfy the causality principle. The causality principle requires that each transition in a step must be (in)directly triggered by an external event. This formalisation allows internal event generations that are not triggered by any external event, which violates causality. To solve this, it is sufficient to rules out statecharts violating causality, rendering an additional semantic definition of causality superfluous.

Combining the two above predicates, we have that a reaction, in a stable evaluation σ_0, to a set of external events is always a finite sequence consisting of two transitions $\sigma_0 \longrightarrow^{FP}_{event} \sigma_1 \longrightarrow^{FP}_{step} \sigma_2$, where the first transition models the receiving of input events and the second transition the reaction to these input events. It is impossible that a Statechart diverges under the fixpoint semantics.

The following is an example of a complete step applying the fixpoint semantics for the diagram in Figure 2, starting from the initial stable evaluation, after the occurrence of the external event c, using basic configurations:

$$\begin{aligned}&\sigma_0 = C = \{S_{11}, S_{211}\}, I = \emptyset \longrightarrow^{FP}_{event} \sigma_1 = C = \{S_{11}, S_{211}\}, I = \{c\} \\ &\longrightarrow^{FP}_{step} \sigma_2 = C = \{S_{12}, S_{213}\}, I = \emptyset\end{aligned}$$

We can note that, in this example, the execution of the action on the enabled transition t_3 generates the internal event a, which is sensed immediately, triggering indirectly the transition t_1.

Now, the main features that differentiate the fixpoint semantics from the other ones are synthesized in the following:

- Events generated internally in a step are sensed immediately in the same step
- The *perfect synchrony hypothesis* holds: under this hypothesis, the system responds immediately to new input events. Further more, it responds infinitely fast, i.e. transitions take zero time to be executed
- At each step, a maximal non-conflict set of enabled transitions is executed
- Statecharts violating causality must be avoided
- The system cannot diverge

Statemate semantics

In the Statemate semantics, a Statechart maps to a symbolic transition system STS^{SM}. As in the fixpoint semantics, variables of STS^{SM} are the current configuration C and the current set of input events I.

The initial evaluation is defined the same as for the fixpoint semantics:

$$\mathtt{init} \Leftrightarrow C = \mathtt{dcomp}(\{\mathtt{root}\}) \land I = \emptyset$$

Like the fixpoint semantics, the Statemate semantics uses two transition predicates. On the surface, these are very similar to the ones defined for the fixpoint semantics, but as we will see, they differ subtly from them. The first predicate, $\longrightarrow^{SM}_{event}$, models the occurrence of one or more external events in a stable evaluation and is identical to the one defined for the fixpoint semantics:

$$\longrightarrow^{SM}_{event} \Leftrightarrow \mathtt{stable}^{SM}(C, I) \land C = C' \land \emptyset \subset I' \subseteq E.$$

However, the definition of stable evaluation is somewhat different. In Statemate, an evaluation is stable if there are no input events to be processed *and* there are no enabled transitions:

$$\mathtt{stable}^{SM}(C, I) \Leftrightarrow I = \emptyset \land \mathtt{enabled}(C, I) = \emptyset$$

The second transition predicate, $\longrightarrow^{SM}_{step}$, models again the taking of a step. A step is only taken if the current valuation is not stable, i.e. there are some input events or some enabled transitions. The effect of taking a step is that a next configuration is reached and that some internal events (actions of the transitions in the step) are generated. These generated events are put in I'. Transition relation $\longrightarrow^{SM}_{step}$ formalises this:

$$\begin{aligned} \longrightarrow^{SM}_{step} \Leftrightarrow \quad & \neg \mathtt{stable}^{SM}(C, I) \\ & \land \exists St \subseteq T : \mathtt{isStep}(St, C, I) \land C' = \mathtt{nextConfig}(C, St) \\ & \land I' = \mathtt{generated}(St) \end{aligned}$$

Again, we can note that this definition is similar to its counterpart in the fixpoint semantics. The major difference is that internally generated events are sensed in the next step only, while these are sensed immediately in the fixpoint semantics. Combining these transition predicates, we have that in Statemate a reaction to a set of external input events consists of a sequence of steps, called a *macro-step*: $\sigma_0 \longrightarrow^{SM}_{event} \sigma_1 \longrightarrow^{SM}_{step} \sigma_2 \ldots \sigma_{n-1} \longrightarrow^{SM}_{step}$

σ_n , where $\sigma_0, \sigma_n \models \mathtt{stable}^{SM}(C, I)$, and for every evaluation σ_i , where $0 < i < n$, $\sigma_i \not\models \mathtt{stable}^{SM}(C, I)$. A single $\longrightarrow^{SM}_{step}$ is called a *micro-step*. The sequence might be infinite, in which case the Statechart diverges. Then, for every valuation σ_i with $i > 0$, we have $\sigma_i \not\models \mathtt{stable}^{SM}(C, I)$. In literature, this is defined as a *Zeno behaviour*.

The following is an example of a complete step applying the Statemate semantics for the diagram in Figure 2, starting from the initial stable evaluation, after the occurrence of the external event c, using basic configurations:

$$\begin{aligned}
&\sigma_0 = C = \{S_{11}, S_{211}\}, I = \emptyset \longrightarrow^{SM}_{event} \sigma_1 = C = \{S_{11}, S_{211}\}, I = \{c\} \\
&\longrightarrow^{SM}_{step} \sigma_2 = C = \{S_{11}, S_{213}\}, I = \{a\} \\
&\longrightarrow^{SM}_{step} \sigma_3 = C = \{S_{12}, S_{22}\}, I = \{b, c\} \\
&\longrightarrow^{SM}_{step} \sigma_4 = C = \{S_{12}, S_{22}\}, I = \emptyset
\end{aligned}$$

We can note that, in this example, the execution of the action on the enabled transition t_3 generates the internal event a, which is sensed only at the beginning of the next step, triggering two transitions: t_1 and t_4, which are not conflicting; transition t_4 is not enabled in the fixpoint semantics, until a new external event occurred.

Now, as for the fixpoint semantics, the main features that differentiate the Statemate semantics from the other ones are synthesized in the following:

- Events generated internally in a step are sensed and processed only in the next step
- Two different models of time are supported: *synchronous* and *asynchronous*. In the synchronous time model, the system executes a single step every time unit (e.g. a clock tick), reacting to all the external changes that occur in the time unit that elapsed since the completion of the previous step. In the asynchronous model, the system reacts to an external event by performing a sequence of reactions (called steps). At each step, a maximal non-conflict set of enabled transitions is selected based on events and conditions generated in the previous step. While all events live for one step, external events are consulted only at the beginning of the first micro-step and are communicated to the environment after completion of the last micro-step of a macro-step. Micro-steps are executed infinitely fast, with the clock being incremented only at macro-step boundaries, i.e. transitions take zero time to be executed
- The system can diverge, causing Zeno behaviours

The asynchronous one is the model of time used in this description of the formal Statemate semantics, since it respects the perfect synchrony hypothesis.

UML semantics

In the UML semantics, a Statechart maps to a symbolic transition system STS^{UML} . Variables of STS^{UML} are the current configuration C and a current queue $q \in E^*$ which is a sequence of events.

For an event queue $q = e_1 \ldots e_n \in E^*$, we introduce the following notation:

- $head(q) = e_1$ denotes the first event of q if $q \neq \epsilon$, i.e. q is not empty.
- $tail(q) = e_2 \ldots e_n$, where $n \geqslant 2$, denotes q with the first element removed, and $tail(q) = \epsilon$ if $n < 2$.
- $enqueue(e, q) = qe$ denotes the result of appending event e to q, and $enqueue(Ev, q)$ denotes the result of appending all events in $Ev \subseteq E$ in some arbitrary order to q.

In the initial evaluation, the system is in the default completion of root and the queue has no input events:

$$init \Leftrightarrow C = dcomp(\{root\}) \wedge q = \epsilon.$$

For the UML semantics, three transition predicates are needed. The first predicate, denoted $\longrightarrow^{UML}_{event}$, models the occurrence of one or more external events, which are added to the queue. As in the other two semantics, such transitions do not change the current configuration:

$$\longrightarrow^{UML}_{event} \Leftrightarrow \exists Ev \subseteq E : Ev \neq \emptyset \wedge C = C' \wedge q' = enqueue(Ev, q)$$

Note that in this semantics, differently from the other two ones, external events can occur in both stable and unstable evaluations. In particular, they can occur while some other event is being processed. Events in the queue are processed one by one. An event is processed if the current evaluation is stable, i.e. there are no enabled completion transitions:

$$stable^{UML}(C, q) \Leftrightarrow enabled(C, \emptyset) = \emptyset.$$

In a stable evaluation, the system processes the first event from the queue by taking a step. Note that an event can be either external or internal, since generated events are also inserted in the queue:

$$\begin{aligned}\longrightarrow^{UML}_{step} \Leftrightarrow \quad & q \neq \epsilon \wedge stable^{UML}(C,q) \\ & \wedge \exists St \subseteq T : isStep(St, C, \{head(q)\}) \\ & \wedge C' = nextConfig(C, St) \\ & \wedge q' = enqueue(generated(St), tail(q))\end{aligned}$$

After the step has been taken, the current evaluation can be unstable: there could be some enabled completion transitions. However, the next event can only be processed in a stable evaluation. Therefore, the enabled completion transitions need to be taken first.

$$\begin{aligned}\longrightarrow^{UML}_{completionstep} \Leftrightarrow \quad & \neg stable^{UML}(C,q) \wedge \\ & \exists St \subseteq T : isStep(St, C, \emptyset) \wedge C' = nextConfig(C, St) \\ & \wedge q' = enqueue(generated(St), q)\end{aligned}$$

Combining these transition predicates, a reaction in configuration C to processing an event from the queue is typically a sequence:

$$\sigma_0 \longrightarrow^{UML}_{step} \sigma_1 \longrightarrow^{UML}_{completionstep} \sigma_2 \longrightarrow^{UML}_{completionstep} \sigma_3 \ldots \sigma_{n-1} \\ \longrightarrow^{UML}_{completionstep} \sigma_n$$

where $\sigma_0, \sigma_n \models stable^{UML}(C,q)$. If there is a cycle of completion transitions, the sequence is infinite: then for every evaluation σ_i , where $i > 0$, $\sigma_i \not\models stable^{UML}(C,q)$. Thus, a Statechart can diverge, as in the Statemate semantics.

The following is an example of three complete steps applying the UML semantics for the diagram in Figure 2, starting from the initial stable evaluation with an empty queue, after the occurrence of the external event c, using basic configurations:

$$\begin{aligned}& \sigma_0 = C = \{S_{11}, S_{211}\}, q = \epsilon \longrightarrow^{UML}_{event} \sigma_1 = C = \{S_{11}, S_{211}\}, q = c \\ & \longrightarrow^{UML}_{step} \sigma_2 = C = \{S_{11}, S_{213}\}, q = a \\ & \longrightarrow^{UML}_{completionstep} \sigma_3 = C = \{S_{11}, S_{22}\}, q = ac \\ & \longrightarrow^{UML}_{step} \sigma_4 = C = \{S_{12}, S_{22}\}, q = c \\ & \longrightarrow^{UML}_{step} \sigma_5 = C = \{S_{12}, S_{22}\}, q = \epsilon\end{aligned}$$

We can note that, differently from the Statemate semantics, the system cannot execute transitions t_4 and t_1 simultaneously, since it is not in a stable evaluation after the execution of the first step.

Now, as for the other semantics, the main features that differentiate the UML semantics from the other ones are synthesized in the following:

- Events generated internally in a step are sensed and processed only in successive steps (not necessarily the next)
- The *perfect synchrony hypothesis* does not hold: a step takes time and during this time the next events can already arrive
- A queue is used to avoid that events are lost
- The system processes events from the queue one by one and responds to each event by taking a step: all completion transitions must be executed before the next step is entered
- At each step, a maximal non-conflict set of enabled transitions is executed
- The system can diverge, causing Zeno behaviours

2.2 THE INDUSTRIAL STANDARD IEC-61499

In this section, the semi-formal international standard IEC 61499 is briefly described. IEC 61499 is an international standard for the modelling of distributed control systems. It is an advancement of the IEC 61131 standard [10], which deals mainly with the modelling of Programmable Logic Controllers (PLCs). As mentioned in the introduction of this chapter, the programming unit of the IEC 61499 is the *Function Block* (FB). It is the basic building block from which entire applications may be built. There are three types of function blocks: *basic function blocks*, *composite function blocks* and *service interface function blocks*. A basic function block executes an elementary control function, such as reading a sensor or setting the state of an actuator. A simple basic FB is shown in Figure 3. The behaviour of a basic function block is defined using a Moore-type finite state machine (FSM), known as the *execution control chart* (ECC), similar to the one shown in Figure 4. An ECC consists of *execution control (EC) states*, *EC transitions*, and *EC actions*. An EC transition has an associated Boolean expression that may contain event inputs, data inputs or internal variables. An EC state may be further associated with zero or more EC actions, which consist of an algorithm to be executed and/or an output event to be emitted after the execution of an algorithm. The initial state, that

shall have no associated EC actions, by convention is indicated with a double rectangle, as the Start state in Figure 4.

Basic function blocks may be combined together in a *composite function block*, to encapsulate an higher-level control function. The *service interface function block* is an implementation-dependent FB that provides an interface between the application and the underlying execution platform and a communication services among devices.

Regardless of the type, every function block has a well-defined input/output interface that consists of event and data ports, as illustrated in Figure 3. Event ports are drawn on the upper half of the block, while data ports are drawn on the lower half. Event-data associations may be specified at the interface to update internal data with new values from the interface whenever the associated event occurs. The WITH qualifier from IEC 61499 indicates that the corresponding event and data signals must be synchronized: it is represented as a vertical line between the event and the data signals associated to it, with a little square on each; it is possible to see an example in Figure 3, between the event *Init* and the data inputs *HLimit* and *XHLimit*. It is not possible to connect a data input signal to more than an event.

FBs adopt an *event-driven* model of execution. The transition conditions in an ECC are evaluated whenever the FB receives an input event. The input event triggers the scheduling of the FB. As soon as one of the conditions becomes true the corresponding EC transition fires. For states with more than one outgoing transition, the transition conditions will be evaluated according to the order in which they are declared in their XML representations. Actions associated with a given state will be executed once upon entry to the state.

Composite FBs enable the encapsulation of a network of FBs (Function block network, briefly FBN) within another block. Unlike basic FBs, the behaviour of composite FBs depends on the composite behaviour of the encapsulated network, rather than on an ECC. Networks consisting of these various types of blocks may be further grouped together within a functional unit of software known as a *resource*. Resources may be allocated to specific *devices*, and a device itself may have several resources allocated to it. In IEC 61499 parlance, a device is simply a PLC. A system is made up of various devices that implement a complete specification.

The run-to-completion semantics is assumed for the execution of the FB instance. The standard defines the execution of a basic FB as a sequence of eight (internal) events $t1 - t8$ as follows:

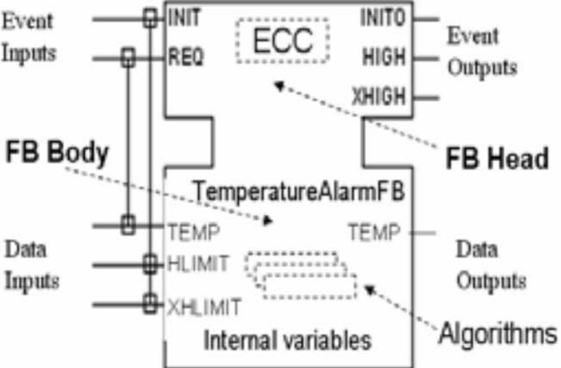

Figure 3: A Basic FB

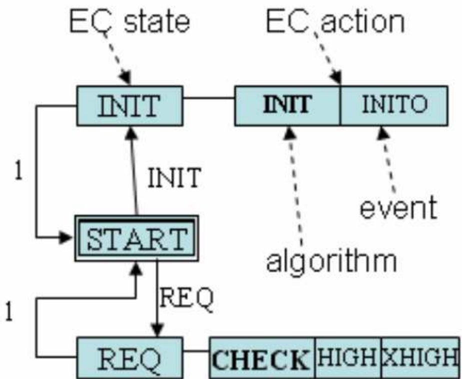

Figure 4: An ECC of a Basic FB

1. Relevant input variable values (i.e., those associated with the event input by the WITH qualifier) are made available.

2. The event at the event input occurs.

3. The execution control function notifies the resource scheduling function to schedule an algorithm for execution.

4. Algorithm execution begins.

Figure 5: A FB network

5. The algorithm completes the establishment of values for the output variables associated with the event output by the WITH qualifier.
6. The resource scheduling function is notified that algorithm execution has ended.
7. The scheduling function invokes the execution control function.
8. The execution control function signals an event at the event output.

These events are only specified in a informal way, without a detailed further description, so they are not sufficient for creating an execution model of function blocks and are not enough clear. Many of the discrepancies between existing function block implementations arise due to the standard's inadequate treatment of various fundamental aspects of the execution model. They are:

1. The lack of any notion of time. FBs do not have an explicit notion of time. Hence, the concept of simultaneous events and the lifetime of events within an ECC can be ambiguously interpreted.
2. The lack of any notion of composition. While individual function blocks may be connected to form networks, the standard does not define the composite behaviour of such a network. Nothing is said about the composite state when multiple ECCs are connected, or of the semantics for their communication. Hence, a variety of ad-hoc approaches rise.

3. The lack of an event-processing policy; in fact, an unreliable transition evaluation order is defined in the IEC 61499 standard. To avoid the unpredictable behaviour of the FB-network diagram, the event-processing policy should be defined at the design phase so as the control engineer is aware of the corresponding execution semantics of its design. There are three alternatives that can be supported by FB-based run-time environments for the processing of input events:
   - events are processed on a first come order. This is implemented by a traditional FIFO event queue.
   - events are processed on a priority based order. This can also be implemented by priority queues.
   - all pending input events are candidates for processing at the time the thread of the FB inserts the running state.
4. The lack of a possibility to define an evaluation order of transitions in the graphical notation, against of the evaluation order defined by the standard, based on the order of the transitions in the textual FB specification. This leads to a non deterministic execution. A possible solution is a definition of a priority label for the transitions at the design level.

These have resulted in different interpretations and/or implementations of the standard. Such an implementation is the ISaGRAF6 environment [13], which has been used to model the case study developed in this thesis to demonstrate the validity of the approach.

3 OVERVIEW OF THE METHODOLOGY APPROACH: A CASE STUDY

In this chapter, a case study of a simple reactive system has been developed using the IEC 61499 standard as a modelling language to validate the effectiveness of the new process-development approach presented in this thesis. This simple case study, a robotic cell, is described in the section 3.1. Successively, the IEC 61499 model is translated into a Simulink/Stateflow diagram, exploiting a set of syntactic rules described in [14]. A rigorous semantics is given to the Simulink/Stateflow model, to allow a translation of the semantic rules into a temporal logical formalism, TRIO, described in 3.3.1. The logic-based semantics precisely captures and resolves the intricacies (and possibly the hidden ambiguities) of the design notation, and it is used to formally check whether user-defined properties of interest are satisfied by the system model or not. As described in the introduction, thanks to the metric nature of the logic-based language underlying the approach, Stateflow models are provided with a precise, metric, notion of time; this is exploited, on the one hand, to introduce metric constraints in the models (e.g., "the plant remains in state S no longer than 3 time units"), and on the other hand to allow users to analyse properties such as "does the plant terminate the processing within 10 time units of its start?". In fact, some interested qualitative and quantitative real-time properties are analysed in section 3.4, with a set of experimental results obtained using a bounded model checker, $\mathbb{Z}$ot, over the case study. In last section 3.5, the main weaknesses of the approach are described and analysed, especially the theoretical limitations of the metric temporal logic TRIO in the modelling of systems based on steps of different order of magnitude in their duration. For this reason, we have decided to develop a new metric temporal logic, called X-TRIO, as an extension of TRIO to overcome its limitations.

3.1 THE CASE STUDY: A ROBOTIC CELL

In this section, a simple case study in the field of FMS, modelled using the IEC 61499 standard [21, 11], is presented. The standard has been described in section 2.2. It is a simple robotic cell, which is one of the most typical part

of an FMS. An high level schema of the cell is shown in Figure 6. This case study, although very simple, is adapt to show the validity of the new process-development approach presented in this thesis, since it is a compositional system, with five different basic components, which interact by sending and receiving signals, so all the various features of the language are exploited. To simplify the model, all variables and signals are of boolean type.

The cell has been developed using a closed-loop methodology, typical of system engineering: all the components of the cell and the external environment are modelled, to obtain a complete specification of the behaviour of the system; in this way, in the last section of this chapter some relevant real-time properties have been checked. The FBN is composed of 6 FBs:

1. A conveyor belt (*Conveyor_in*), which provides pallets of two types, A and B, to be processed by two different machines. To simplify the implementation of the logic controller of the *Conveyor_in*, every time unit only a pallet, randomly chosen between the two types, is present on the conveyor belt to be processed. It remains on the conveyor until a robot arm, which is the main component of the robotic cell, moves it to the suitable machine. The *Conveyor_in* sends a signal to the robot arm to indicate the presence of a new pallet. The model of the *Conveyor_in* is not shown.

2. Two different machines, (*Machine 1* and *Machine 2*), which can only process pallets of types A and B, respectively, and produce final artefacts without a particular type. The working time of the two machines is hardcoded in the ECC of the FBs and translated in the logical model, so it is possible to analyse real-time properties such as "Is it possible to finish processing three pallets of type A in 10 units of time?"; the exact specification of the working time is not important, but in the implementation has been fixed to two time units. A machine controller sends to the robot two different signals: a signal to indicate that the machine has finished processing a piece, and another signal to indicate that the piece has been removed by the robot arm and the machine is free to process a new pallet. During the initialization of the system, each machine controller sends to the robot arm the signal that its corresponding controlled machine is free.

3. Another conveyor belt (*Conveyor_out*), which receives from the robot arm the finished artefacts to be released out of the cell. *The Conveyor_out*

does not send any signal to the robot arm; it is only an abstract component of the cell, without an important behaviour, so it has not been really implemented as FB.

4. A robot arm (*Robot*), which loads and unloads the pallets of the two types A ad B from the *Conveyor_in* to the suitable machine, and loads and unloads the finished artefacts from one of the machines to the *Conveyor_out*. The developed IEC 61499 control solution of the controller of the robot arm is shown in Figure 7. The FB of the controller of the robot arm has 10 input signals and 6 output signals, which are represented on the left and on the right respectively in Figure 7. The two FBs, on the left and on the right, are service interface FBs needed to interface the controller with the rest of the components: the FB on the left receives signals from the rest of the components of the cell (input events and data), instead the FB on the right sends the commands generated by the robot controller (output events and data) to them. The FB called *E_RESTART* is needed to fire the initialization of the system. During the initialization, the robot is in a predefined position, the position called P_0, where it waits for a signal from the (*Conveyor_in*), or from a machine. The internal behaviour of the robot arm is modelled by the ECC depicted in Figure 8. The ECC of the robot controller sends command to the robot arm such as "go to the *Machine 1*", or "go to the *Conveyor_in*", and receives the reactions of the robot arm as input signals such as "position of the *Machine 1* reached" or "position P_0 reached"; these reactions are modelled by the FB described in the next point.

5. The last component models the reactions of the environment: the main part of this FB models the reaction of the robot arm to the commands sent by the robot controller. In particular, in a way similar to the *Conveyor_in*, the time needed to move the robot arm between the various stations of the cell is hardcoded in the ECC of the FB and embedded in the logical model; again, the exact specification of this time duration is not important, but in the implementation it has been fixed to one time units. The model of the environment is not shown.

In FMSs, non-deterministic choices within a component must be avoided. In the case study, to deal with multiple requests from different components, it has been statically assigned priorities to the operations performed by the robot: the unloading of final artefacts from the machines has higher priority

than their loading; also, unloading final artefacts from *Machine 1* has precedence over unloading them from *Machine 2*. Finally, at any time, the robot arm can switch from automatic to manual mode, where an operator can send commands directly to the robot when the need arises to perform operations outside the production cycle. The system switches back to automatic mode through a suitable command.

The entire case study is implemented using the ISaGRAF6 environment [13], which completely supports the development of control applications with the IEC 61499 standard.

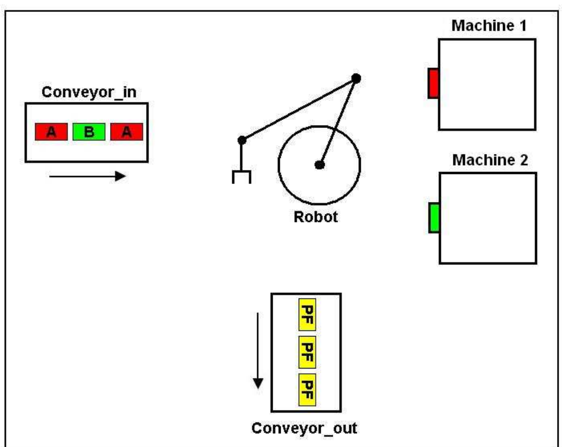

Figure 6: Robotic Cell.

The adoption of the IEC 61499 standard as reference model for the control system of the robotic cell fosters the definition of reusable control modules and the re-configurability of the solution, since the principles of modularity, encapsulation and standardization of interfaces are effectively supported. The control logic of each component of the FMS, i.e. the ECC of each FBs, is translated into Stateflow diagrams, from the developed IEC 61499-compliant control solution, for its formal verification. To guarantee that the properties and the features of the IEC 61499 control solution described above are maintained in the Stateflow diagrams, the rules defined in [14] to translate an IEC 61499 model into a corresponding Stateflow are exploited. More precisely, the Simulink/Stateflow description of an IEC 61499 model is obtained by translating each FB into a Simulink block constituted by a Stateflow model, where:

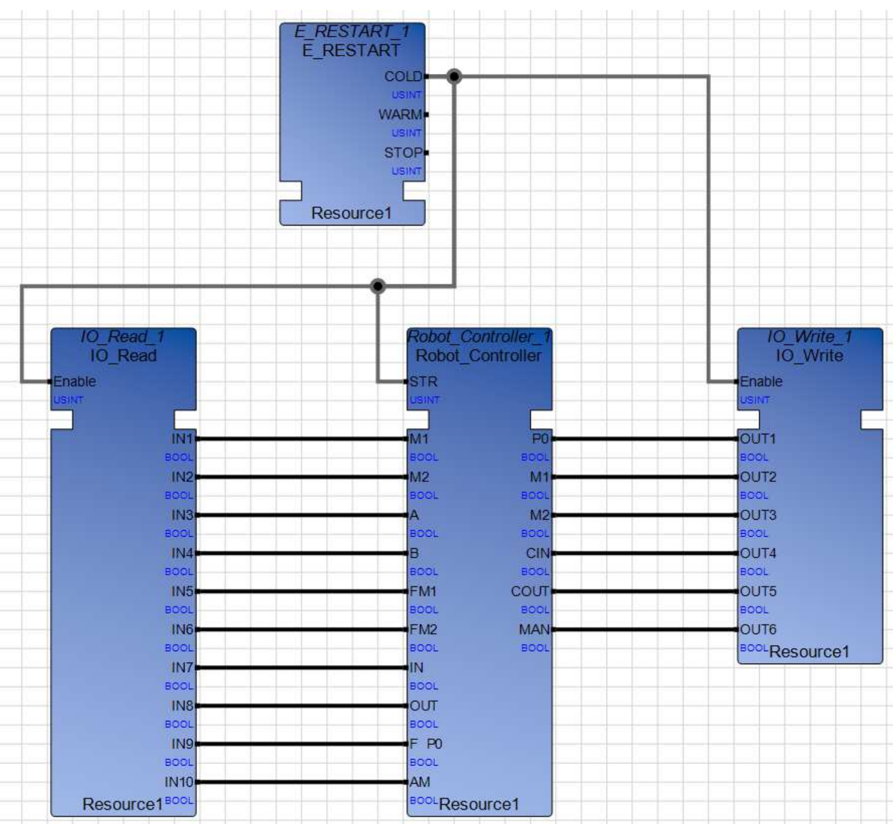

Figure 7: Developed IEC 61499 control solution

- Input and output data are represented as input and output signals of Boolean or Double types in the corresponding Simulink block.
- Input and output events are represented as rising or falling edges of input and output signals of Boolean type in the corresponding Simulink block.
- The ECC and related algorithms are represented by means of Stateflow diagrams, in a straightforward way; for example, each state and transition of the ECC is translating in a state and a transition with similar conditions, events and actions of the corresponding Stateflow diagram. The exact way to perform this translation is not given here in details,

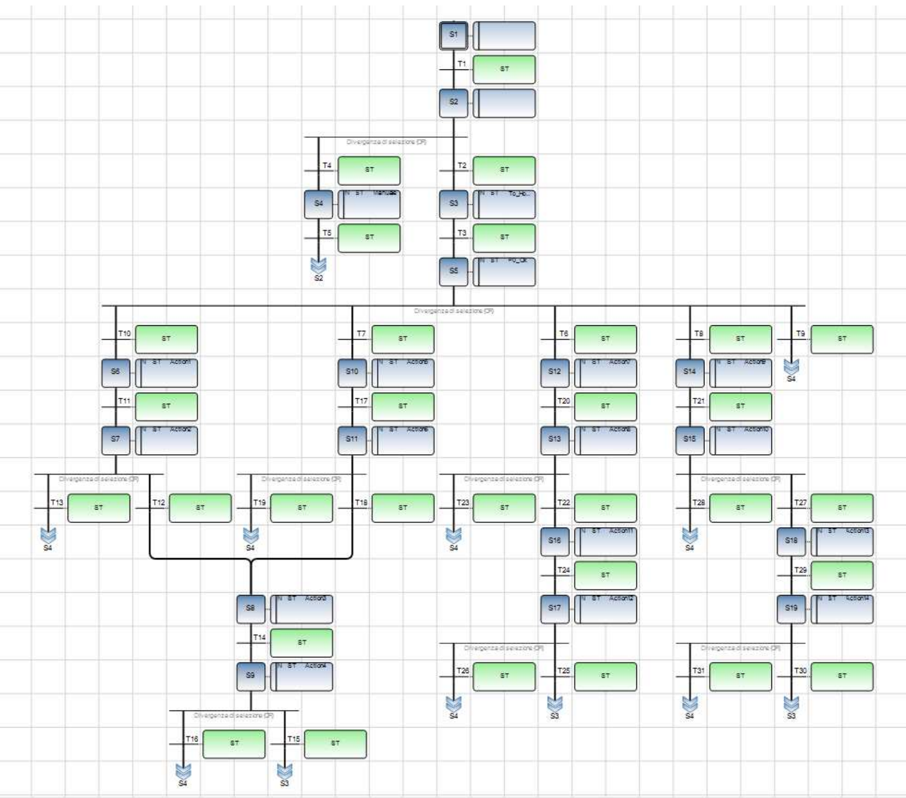

Figure 8: ECC of the robot controller module

since it is not mandatory for this thesis, and is performed by hand right now.

- Internal data is represented by means of local Simulink variables.

Finally, the Simulink/Stateflow model is obtained by connecting the Simulink blocks according to the structure of the original IEC 61499 model, such as the Simulink graph of Figure 11, which is the composed model of the robotic cell obtained translating the IEC 61499 model of Figure 8.

Simulink and Stateflow are two toolboxes of the Matlab environment integrated with each other, that support dynamic systems analysis through time-based and discrete-event simulation.

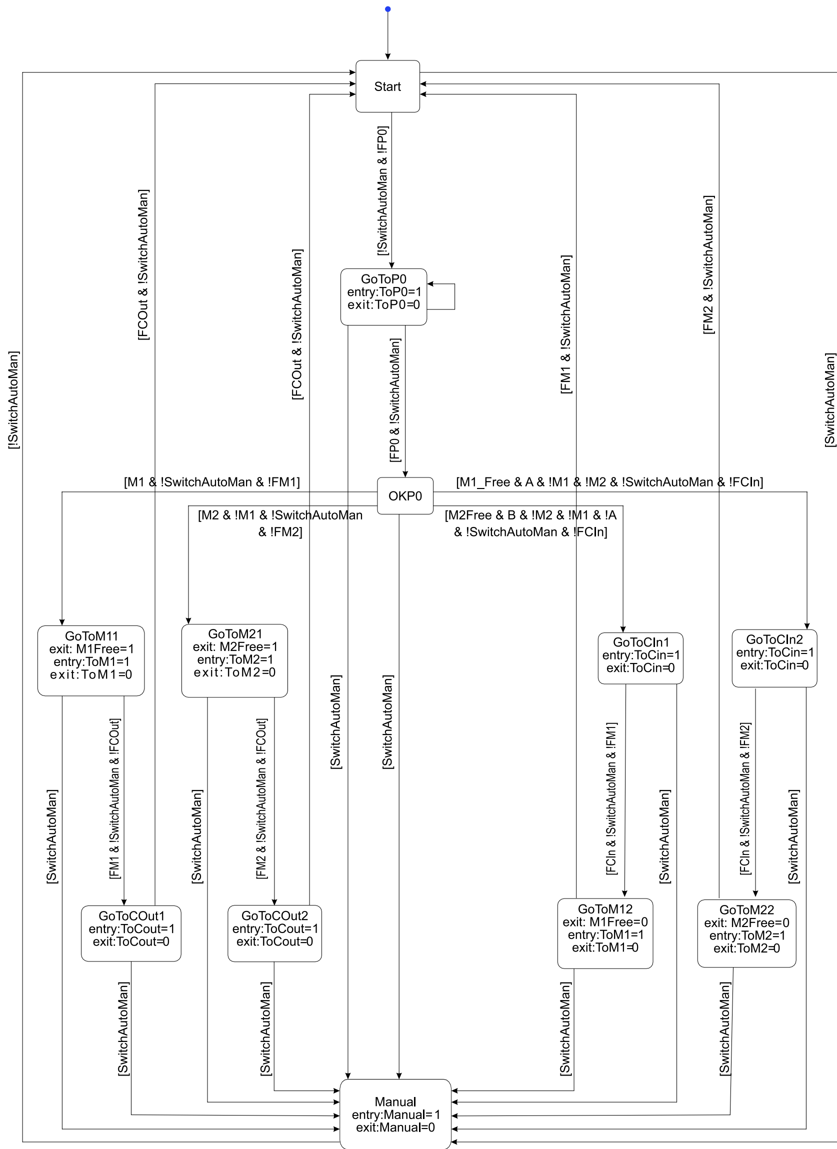

Figure 9: Stateflow diagram of the controller of the robotic cell of Figure 6.

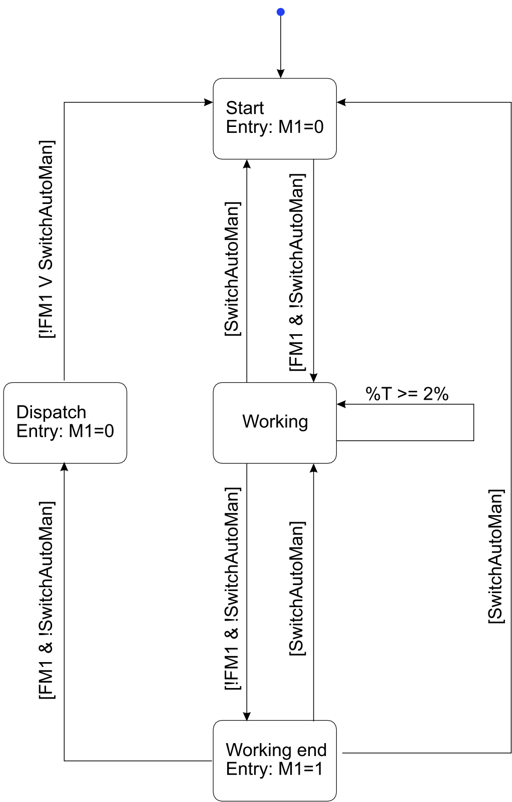

Figure 10: Stateflow diagram of component *Machine 1* of Figure 6.

Now, the syntax of the graphical modelling language Stateflow and an introduction to the use of Simulink as a compositional notation for composed Stateflow diagrams are briefly described.

As mentioned in the previous chapter 2, the Stateflow notation is a variation of Statechart, so the description is limited only to the differences between the two notations.

Simulink and Stateflow [9] are two toolboxes of the Matlab environment integrated with each other, that support dynamic systems analysis through time-based and discrete-event simulation.

A Stateflow diagram, similar to a Statechart one, is a tuple $SF = \{S, s_0, D, T\}$ where:

1. $D = D_I \cup D_O \cup D_L$. D is a finite set of typed variables partitioned into input variables D_I, output variables D_O, and local variables D_L. D_I and D_O include Boolean variables used to represent input and output events: a variable v_i (resp. v_o) modelling an input (resp. output) event is set to $\mathtt{true}$ when the event is received from (resp. notified to) the environment. This is a variation to simplify the original notation of Stateflow, where events and variables are separated.

2. S is a finite set of states. A state can be associated with three kinds of *actions*: *entry*, *exit* and *during* actions; they are executed, respectively, when the state is entered, is exited, or during the permanence of the system in the state. An *action* is the assignment of the evaluation of an expression e over constants and variables of D to a non-input variable v: it is denoted with $v = e$. A *during* action corresponds to a static reaction of a Statechart diagram.

3. s_0 is the unique initial state of the Stateflow diagram

4. T is a finite set of transitions, that may include conditions (i.e., constraints) and actions. The graphical Stateflow notation of the label of a transition is similar to the Statechart notation, $[c]/a$, without the event part. A condition is a boolean expression over the variables of D, that can be composed using the usual logical operators $\mathtt{or}$, $\mathtt{and}$ and $\mathtt{not}$.

Actions over both states and transitions allow one to write Stateflow diagrams in a more concise way, since it is possible to build a semantically equivalent Stateflow diagram with actions over transitions only. For example, an *entry* action of a state is equivalent to an action on any transition entering the state; a during action of a state s with a transition from s to itself is equivalent to a *during* action on the transition.

Such as in Statechart, in Stateflow there are also advanced features such as time actions, history, special events, special actions, junctions and so on. In this thesis, to simplify the exposition, only the basic features are used. So, OR and AND-states are not used since the ECC of the IEC 61499 has a flat, sequential structure: as a consequence, only a state of the ECC and, consequently, of the corresponding Stateflow diagram, is active.

As mentioned above, an FB is translated into a Simulink block, which represents the *basic* component of a Simulink/Stateflow representation of the model. The public interface of the basic component, which represents the set of input and output data and events of the FB, comprises the set of variables $D_{Int} = D_I \cup D_O$ of the Stateflow diagram that models its behaviour.

An FBN, instead, is translated into a Simulink diagram: this Simulink diagram, composed of one or more basic components (i.e. Simulink blocks), represents a *composed* component of the Simulink/Stateflow representation. Its public interface is the union of the input and output variables of its components, while its behaviour is described by the Stateflow diagrams of its basic components, whose communication channels are represented graphically through *links*. Each link corresponds to a flow of messages (signals or data) sent from a component to another one, and models the corresponding link in the IEC 61499 FBN. Simulink diagrams can, in turn, be composed to obtain higher-level components. The detailed features of communication are explained in the coming Section 3.2.

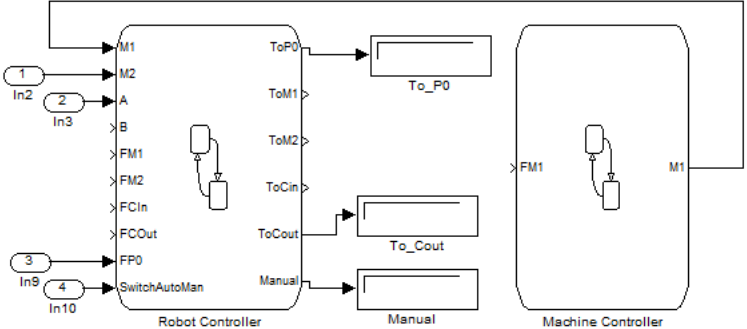

Figure 11: Simulink diagram of the robotic cell.

3.2 A SEMANTICS FOR SIMULINK/STATEFLOW

Stateflow is a complex language with numerous, complicated, and often overlapping features lacking any formal definition. Its documentation describes the semantics in informal operational terms, supported by numerous examples, but the actual definition of the language is the "simulation semantics" given by its behaviour when simulated in the Matlab environment. A complete formal verification process for Stateflow requires first to give it a formal definition. As it is possible to understand from the documentation of Mathworks, the semantics of Stateflow is similar to the Statemate semantics of Statecharts described in section 2.1.2. Furthermore, in this thesis a composition operator $\|$ is defined to build hierarchical, modular models from simpler

ones: it is useful to define the behaviour of a composed component of a Simulik/Stateflow diagram, according to its Simulink graph.

Now some basic concepts and definitions of the semantics of Statechart have been formally redefined to adapt them to Stateflow, in particular for the presence of variables with different domains. Since, given a Stateflow diagram, it is possible to build another semantically equivalent Stateflow diagram with actions over transitions only, the semantics is given only for this type of Stateflow diagrams.

Such as for Statechart, the semantics of a Stateflow diagram is a set of *runs*, representing the reaction of the system to a sequence of input events, represented by a subset of input boolean variables of the Stateflow diagram.

A run is a sequence of *configurations* $\{C_i\}_{i \geqslant 0}$ such that, for each $i > 0$, C_i is obtained from C_{i-1} by executing a step.

We remind that a configuration is a maximal consistent set of states that the system can be in simultaneously; but, since all states of this particular version of Stateflow are basic states, there is only a single active state. For this reason, in the semantics of this version of Stateflow, a configuration C_i is a pair $\langle s_i, \mu_i \rangle$ where $s_i \in S$ is the currently active state and μ_i is an *evaluation* of the variables of D, i.e. a mapping $\mu_i : D \to dom(D)$. An expression e on the variables of D is evaluated in the active configuration by the function $eval(e, \mu)$, which return a value in the common domain of all the variables in e. The initial configuration is always of the form $C_0 = \langle s_0, \mu_0 \rangle$, where μ_0 is the initial evaluation for all the variables.

Since any state of a Stateflow diagram is a basic state, $children(s) = \emptyset$ for every state s, so functions $children$ and its reflexive-transitive function $children^*$ and transitive closure $children^+$ become useless; the same for the $default$ function. Instead, every state s has the same $parent(s) = r$, where r is the root of the Stateflow diagram; r is the OR-state (not indicated in the graphical notation) which represents the entire Stateflow diagram and is the unique common ancestor of any set of states, and every other state is its children. For this reason, every couple of states is not orthogonal and every set of states X is not consistent, except if it is a singleton.

A transition in a Stateflow diagram is denoted with $s \xrightarrow{[c]/a} s'$, where s is the (unique) $source$ state, s' is the (unique) target state, c is a condition expression and a is an action. In this semantics, the scope of any transition $scope(t)$ is always the root r.

The effect of executing an action a of the form $v = e$ in a certain configuration $C = \langle s, \mu \rangle$ is a new evaluation function $\mu' = \mu[v/eval(a, \mu)]$, where the

value of v is substituted with the value $eval(a, \mu)$, where $eval$ is a function that evaluates the expressions in a using the evaluation function μ. As for Statechart, a transition is relevant in C if its unique source state is s.

Given all this new definitions, a Stateflow transition is *enabled* in a configuration C only if it is relevant in C and the condition c is true under the evaluation function μ. So, the set of enabled transitions in C is defined as:

$$enabled(C) = \{t \in T | t = [c]/a \in relevant(C) \wedge eval(c, \mu) = true\}$$

Since the scope of a transition is always r, two transitions are always inconsistent, and are conflicting if their source states are the same. So, a set of transitions is maximal if it is a singleton or is empty, i.e. the set of enabled transitions is always a singleton or is empty. The problem of conflicting transitions has been solved by assigning them a priority.

Taking in consideration that a step, in Stateflow, is constituted by the single transition t that is enabled in the configuration C, i.e. $isStep(t, C) \Leftrightarrow t \in enabled(C)$, executing a step leads the system in the new configuration $nextConfig(C, t) = \langle s', \mu' \rangle$, where $s' = target(t)$ and $\mu' = \mu[v/eval(a, \mu)]$. The system reaches a *stable* configuration when the set of enabled transitions is empty, i.e. $stable^{SF}(C) \Leftrightarrow enabled(C) = \emptyset$.

Before given the definition of a Stateflow step, we need to redefine the predicate $\longrightarrow^{SF}_{event}$, which models the occurrence of one or more external events in a stable configuration $C = \langle S, \mu \rangle$. So, denoted with E_{D_I} an evaluation function that returns the new values assigned externally to the input variables in D_I, where at least one of the input variables has changed its value respect to the value returned by μ, the redefined predicate is:

$$\begin{aligned} \longrightarrow^{SF}_{event} \Leftrightarrow \quad & stable^{SF}(C) \wedge \exists v \in D_I : \mu(v) \neq E_{D_I}(v) \\ & \wedge C' = \langle S, \mu' = \mu[v_1/E_{D_I}(v_1), \ldots, v_n/E_{D_I}(v_n)] \rangle \end{aligned}$$

where $v_1, \ldots, v_n \in D_I$.

Finally, the definition of a Stateflow step is the following:

$$\begin{aligned} \longrightarrow^{SF}_{step} \Leftrightarrow \quad & \neg stable^{SF}(C) \wedge \exists t \subseteq T : isStep(t, C) \\ & \wedge C' = nextConfig(C, t) \end{aligned}$$

Combining these transition predicates, we have that a reaction to a set of external input events consists of a sequence of steps, called a *macro-step*: $\sigma_0 \longrightarrow^{SF}_{event} \sigma_1 \longrightarrow^{SF}_{step} \sigma_2 \ldots \sigma_{n-1} \longrightarrow^{SF}_{step} \sigma_n$, where $\sigma_0, \sigma_n \models stable^{SF}(C)$,

and for every valuation σ_i , where $0 < i < n$, $\sigma_i \nvDash stable^{SF}(C)$. A single $\longrightarrow^{SF}_{step}$ is called a *micro-step*.

Such as the Statemate semantics for Statechart, the semantics for Stateflow defined in this thesis is of the so-called *run-to-completion* type, with an asynchronous time model. We briefly remind the main features:

- Events generated internally in a micro-step are sensed and processed only in the next micro-step. It means that, when an action update the value of a variable in the set $D_O \cup D_L$, the update is processed only at the beginning of the next micro-step.
- The system senses input events, i.e. update input variables, only when it is in a stable configuration, i.e. at the beginning of a macro-step. Output events, i.e. the value of output variables, are communicated to the environment after completion of the macro-step, when the system is in a new stable configuration
- The perfect synchrony hypothesis holds: micro-steps are executed infinitely fast, with the clock being incremented only at macro-step boundaries, i.e. transitions take zero time to be executed
- The system can diverge, causing Zeno behaviours. A run has Zeno behaviour if infinitely many actions are executed in a finite amount of time. In Stateflow, it corresponds to the situation in which infinitely many micro-steps are executed during a single macro-step, which in turn occurs when the run enters a loop of micro-steps that is never exited, thus never triggering the advancement of time. Zeno runs must be avoided in models because they represent unfeasible behaviours. Section 3.4 shows how Zeno runs can be detected for the formal semantics of Stateflow described in this section, using a simple formula written in the TRIO metric temporal logic.

In summary, the semantics of a macro-step is the following:

1. When a macro-step begins, input data and events are assigned to the corresponding variables of set D_I. Suppose for example that the current configuration C_i of the Robot Stateflow of Fig. 9 is $C_i = \langle OkP0, \{M1Free = 1, M2Free = 0, FM2 = 0, M1 = 0, \ldots\} \rangle$, and that the input event $In2$ occurs, meaning that there is a completed workpiece on *Machine 2*. Then the input variables are updated to *false* except $M2$, producing a new configuration C' with the current time and state unchanged.

2. As long as there are enabled transitions, micro-steps are executed in zero time. For example, the transition enabled in configuration C_{i+1} is the one with condition [M2 & !M1 & !$SwitchAutoMan$ & !FM2]; the transition is immediately taken and the system executes the action $entry$: ToM2 = 1 of the destination state GoToM21, leading to the following new configuration:

   $C_{i+2} = \langle \text{GoToM21}, \{\text{ToM2} = 1, \ldots\} \rangle$

   As before, time does not advance.

3. When there are no more enabled transitions to execute, a stable configuration is reached. At this point the macro-step is completed, time advances of one unit, and output events and data produced during the macro-step are communicated to the environment. In the example above, no transitions are enabled in configuration C_{i+2}, so time advances, the values of the variables and the current state do not change and the new event ToM2 is produced according to the Simulink graph of Figure 11.

A run identifies a sequence of time instants $\{T_i\}_{i \in \mathbb{N}}$, one for each macro-step, hence the time domain is discrete. This is consistent with the underlying physical model, since the PLCs on which FMS control solutions are built, are governed by discrete clocks, i.e. each macro-step corresponds to a *clock cycle* of the modelled PLC.

The last part of the semantics concerns the composition of two or more components, according to the Simulink graph. Given two Stateflow diagrams $G_1 = \langle S_1, s_{1_0}, D_1, T_1 \rangle$ and $G_2 = \langle S_2, s_{2_0}, D_2, T_2 \rangle$, we introduce a compositional binary operator $\|$ whose result is a new composed component $G = \langle S, s_0, D, T \rangle$ where $D = D_1 \cup D_2$, $S = S_1 \times S_2$ and $s_0 = \langle s_{1_0}, s_{2_0} \rangle$.

The sets of all possible configurations of G_1 and G_2 are denoted with C_1 and C_2, respectively. The evaluation functions of the input variables in D_{I_1} and D_{I_2}, given externally, are denoted with $E_{D_{I_1}}$ and $E_{D_{I_2}}$. A global configuration of G is a couple $c = \langle c_1, c_2 \rangle$ with $c_1 \in C_1$ and $c_2 \in C_2$.

A global configuration $c =< c_1, c_2 >$ is stable if the two local configurations $c_1 = \langle s_1, \mu_1 \rangle \in C_1$ and $c_2 = \langle s_2, \mu_2 \rangle \in C_2$ are stable:

$$stable^{SL}(c_1, c_2) \Leftrightarrow stable^{SF}(c_1) \wedge stable^{SF}(c_2)$$

Given the new concept of global stability through the new $stable^{SL}$ predicate, we must introduce two new global predicates $\longrightarrow^{SL}_{event}$ and $\longrightarrow^{SL}_{step}$

which are based on it. The predicate $\longrightarrow^{SF}_{event}$ asserts that the composed component is sensible to new external events only when a global stable configuration is reached:

$$\begin{aligned}\longrightarrow^{SL}_{event} \Leftrightarrow\ & stable^{SL}(c_1, c_2)\\ \wedge\ & (\exists v \in D_{I_1} : \mu(v) \neq E_{D_{I_1}}(v) \vee \exists v \in D_{I_2} : \mu(v) \neq E_{D_{I_2}}(v))\\ \wedge\ & c' = \langle c'_1, c'_2\rangle : c'_1 = \langle s_1, \mu' = \mu[v_1/E_{D_{I_1}}(v_1), \dots, v_n/E_{D_{I_1}}(v_n)]\rangle\\ \wedge\ & c'_2 = \langle s_2, \mu' = \mu[v_2/E_{D_{I_2}}(v_2), \dots, v_n/E_{D_{I_2}}(v_n)]\rangle\end{aligned}$$

where $v_1, \dots, v_n \in D_{I_1} \cup D_{I_2}$. We note that the global configuration C is updated only if at least one of the input variables of one of the two components has changed its value respect to the value returned by μ.

Finally, the definition of a step of the composed component G, i.e. a global micro-step, is the following:

$$\begin{aligned}\longrightarrow^{SL}_{step} \Leftrightarrow\ \neg\ & stable^{SL}(c_1, c_2)\\ \wedge\ & (\exists t_1, t_2 \subseteq T : (isStep(t_1, c_1) \wedge isStep(t_2, c_2)\\ \wedge\ & c' = \langle nextConfig(c_1, t_1), nextConfig(c_2, t_2)\rangle)\\ \vee\ & \exists t_1 \subseteq T : (isStep(t_1, c_1) \wedge c' = \langle nextConfig(c_1, t_1), c_2\rangle)\\ \vee\ & \exists t_2 \subseteq T : (isStep(t_2, c_2) \wedge c' = \langle c_1, nextConfig(c_2, t_2)\rangle)))\end{aligned}$$

From the definition of a global step, we can note that the system can evolve in two different ways:

1. Suppose that G is in configuration $c = \langle c_1, c_2\rangle$, with $c_1 = \langle s_1, \mu_1\rangle \in C_1$ and $c_2 = \langle s_2, \mu_2\rangle \in C_2$. If there are two transitions $t_1 : s_1 \xrightarrow{g_1/a_1} s'_1$ and $t_2 : s_2 \xrightarrow{g_2/a_2} s'_2$ of, respectively, G_1 and G_2 that are enabled in c_1 and c_2, then the next configuration of the composed component is $c' = \langle c'_1, c'_2\rangle$ where $c'_1 = \langle s'_1, \mu'_1\rangle$ and $c'_2 = \langle s'_2, \mu'_2\rangle$. μ'_1 and μ'_2 are the new evaluations of the variables according to the execution of actions a_1 and a_2. In this case both components execute their transitions in a real parallel manner, instead of force them to interleave.

2. Suppose that G is in configuration $c = \langle c_1, c_2\rangle$, but only the transition $s_1 \xrightarrow{g_1/a_1} s'_1$ is enabled. In this case, the next configuration of the composed component is $c' = \langle c'_1, c_2\rangle$ (with $c'_1 = \langle s'_1, \mu'_1\rangle$); we can note that the second component does not change its local configuration, i.e. it executes a so-called *stutter* transition (usually denoted with ε). Time does not advance, since the first component executes a non-stutter transition.

Instead of using the global semantics, which substitutes the local semantics, in the next section 3.3 we show that it is possible to construct a compositional global semantics exploiting the local one, avoiding the use of the global predicate $\mathtt{stable}^{SL}$. $\mathtt{stable}^{SL}$ models a global clock, which ticks every time all components reach a local stable configuration, but it exposes to every component informations about the other ones. Using the global compositional semantics, each component can be considered as a black box that an engineer can design and verify separately through a compositional verification approach.

From the above description, it follows that the global clock of the composed component G synchronizes the clocks of its components; so, it is possible that a component can reach a local stable configuration before the others, while the system does not reach yet a global stable configuration. Then, each component that is in a stable configuration must perform stutter transitions until all other components also reach their local stable configuration. This mechanism is exemplified in Figure 12, which shows the fragments of the runs of two modules A and B that are composed to realize a third component C. The

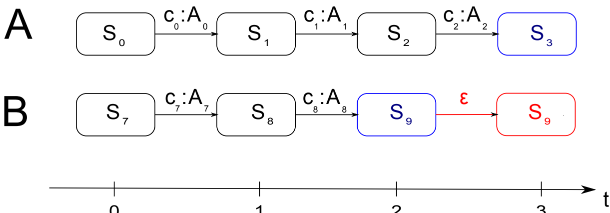

Figure 12: Example of stutter transitions

figure shows the first macro-step of the two runs: for component A the macro-step begins in state S_0 and ends in state S_3; similarly for component B. The x axis shows the number of micro-steps executed from the beginning of the macro-step: component B reaches a stable configuration in state S_9, in fewer micro-steps than A, so a stutter ε transition is introduced to synchronize the clocks of the two components. After the fourth micro-step, both components have reached a stable configuration and the clock of component C advances to the next time unit.

The composition operator $\|$ is such that, when two components of any type are composed, their input and output variables become input and output variables of the composition, i.e., they do not become local to the composed component, hence they cannot be hidden. In addition, an input variable cannot be linked to more than one output variable, to guarantee the uniqueness of the

value assigned to the input variable (on the other hand, an output variable can be linked to more than one input variable, as in a "multicast" communication). It is easy to prove that, thanks to this further restrictions, the parallel composition operator becomes associative.

3.3 temporal logic encoding of the case study

3.3.1 *TRIO: a metric temporal logic*

The original TRIO [22] language is a general-purpose formal specification language suitable for describing complex real-time systems. TRIO is a first-order linear temporal logic that supports a metric on time. TRIO formulae are built out of the usual first-order connectives, operators, and quantifiers, as well as a single basic modal operator, called Dist, that relates the current time, which is left implicit in the formula, to another time instant: given a time-dependent formula ϕ (i.e., a term representing a mapping from the time domain to truth values) and a (arithmetic) term t indicating a time distance (either positive or negative), the formula $\mathrm{Dist}(\phi, t)$ specifies that ϕ holds at a time instant whose distance is exactly t time units from the current one. TRIO formulae can be interpreted both in discrete and dense time domains.

The following is the TRIO syntax (where v, k, f and p are, respectively, a variable, a constant, a function, and a predicate; functions and predicates can have arity 0):

$$\begin{array}{rcl} \phi & := & p(\tau_1, \ldots, \tau_n) \,|\, \neg\phi \,|\, \phi_1 \wedge \phi_2 \,|\, \mathrm{Dist}(\phi, \tau) \,|\, \forall v.\phi \\ \tau & := & v \,|\, k \,|\, f(\tau_1, \ldots, \tau_n) \end{array} \qquad (1)$$

Functions and predicates can be either interpreted (they can be for example arithmetic operations or comparisons) or uninterpreted; for example, one could introduce a predicate $connected(c_1, c_2)$ that holds for all those pairs of components that are connected with each other. In its full generality, TRIO allows users to write arithmetic formulae, hence it is trivially undecidable.

TRIO also defines, and allows the user to define, a large set of derived operators to make formulae simpler and more intuitive. For instance, $\mathrm{Futr}(\phi, t)$ is equivalent to $t \geqslant 0 \wedge \mathrm{Dist}(\phi, t)$, while $\mathrm{Past}(\phi, t)$ is equivalent to $t \geqslant 0 \wedge \mathrm{Dist}(\phi, -t)$. Table 1 presents a meaningful sample of TRIO derived operators, whereas Table 2 shows some useful variations thereof. For convenience, TRIO items (variables, functions, predicates) are distinguished into time-dependent

(TD) and time-independent ones (TI) with the obvious meaning of the two terms [22].

The goal of the verification phase is to ensure that the system S satisfies some desired property R, that is, that $S \models R$. In the TRIO approach S and R are both expressed as logic formulae Σ and ρ, respectively; then, showing that $S \models R$ amounts to proving that $\Sigma \Rightarrow \rho$ is valid. TRIO is supported by a variety of verification techniques implemented in prototype tools. In this thesis, we use $\mathbb{Z}$ot [23], a bounded satisfiability checker which supports verification of discrete-time TRIO models. $\mathbb{Z}$ot is described in section 6.1 in further details.

OPERATOR	DEFINITION
$\mathrm{Futr}(\phi, d)$	$d \geqslant 0 \wedge \mathrm{Dist}(\phi, d)$
$\mathrm{Past}(\phi, d)$	$d \geqslant 0 \wedge \mathrm{Dist}(\phi, -d)$
$\mathrm{AlwF}(\phi)$	$\forall d(0 \leqslant d \rightarrow \mathrm{Futr}(\phi, d))$
$\mathrm{AlwP}(\phi)$	$\forall d(0 \leqslant d \rightarrow \mathrm{Past}(\phi, d))$
$\mathrm{SomF}(\phi)$	$\exists d(0 \leqslant d \wedge \mathrm{Futr}(\phi, d))$
$\mathrm{SomP}(\phi)$	$\exists d(0 \leqslant d \wedge \mathrm{Past}(\phi, d))$
$\mathrm{Lasts}(\phi, \delta)$	$\forall d(0 \leqslant d \leqslant \delta \rightarrow \mathrm{Futr}(\phi, d))$
$\mathrm{Lasted}(\phi, \delta)$	$\forall d(0 \leqslant d \leqslant \delta \rightarrow \mathrm{Past}(\phi, d))$
$\mathrm{WithinF}(\phi, \delta)$	$\exists d(0 \leqslant d \leqslant \delta \wedge \mathrm{Futr}(\phi, d))$
$\mathrm{WithinP}(\phi, \delta)$	$\exists d(0 \leqslant d \leqslant \delta \wedge \mathrm{Past}(\phi, d))$
$\mathrm{Until}(\phi, \psi)$	$\exists d(\mathrm{Futr}(\psi, d) \wedge \forall \nu(0 \leqslant \nu < d \rightarrow \mathrm{Futr}(\phi, \nu)))$
$\mathrm{Since}(\phi, \psi)$	$\exists d(\mathrm{Past}(\psi, d) \wedge \forall \nu(0 \leqslant \nu < d \rightarrow \mathrm{Past}(\phi, \nu)))$

Table 1: TRIO derived temporal operators.

3.3.2 *TRIO encoding of the case study*

In this section, the semantics of Stateflow is formalized using the TRIO temporal logic. The usual qualitative temporal operators of the classical LTL [24] are used to describe the sequence of micro-steps: the idea is to exploit the $\bigcirc$ operator to model a transition execution, i.e. the passage from a configuration to the next one, which corresponds to a micro-step and could be interpreted as a discrete "logical" time model; $\bigcirc$ is the usual LTL *next state* operator, i.e., $\bigcirc F$ holds in the current "state" iff F holds in the next "state".

OPERATOR	DEFINITION
$\text{SomF}_{(}(\phi)$	$\exists d(0 < d \wedge \text{Futr}(\phi, d))$
$\text{SomP}_{(}(\phi)$	$\exists d(0 < d \wedge \text{Past}(\phi, d))$
$\text{WithinF}_{()}(\phi, \delta)$	$\exists d(0 < d < \delta \wedge \text{Futr}(\phi, d))$
$\text{WithinF}_{(]}(\phi, \delta)$	$\exists d(0 < d \leqslant \delta \wedge \text{Futr}(\phi, d))$
$\text{WithinF}_{[)}(\phi, \delta)$	$\exists d(0 \leqslant d < \delta \wedge \text{Futr}(\phi, d))$
$\text{WithinP}_{(]}(\phi, \delta)$	$\exists d(0 < d \leqslant \delta \wedge \text{Past}(\phi, d))$

Table 2: Some variations of TRIO derived temporal operators.

Instead, the real time advancement is modelled by a special tick predicate, which holds any time a component is in a stable configuration: the tick predicate models the advancement of the clock of a PLC, using a discrete "real" time model. The metric operators specific of TRIO are used to express the quantitative properties of macro-steps, exploiting the tick predicate to specify real-time metric temporal properties.

The TRIO logic formulae are written in *Lisp*, which is the input language of the $\mathbb{Z}$ot tool, thus allowing users to perform automatic verification of Stateflow models. For each variable $V \in D$ of the Stateflow model, we introduce a corresponding $\mathbb{Z}$ot variable with finite domain $dom(V)$. When variables have the same domain, we group them in $\mathbb{Z}$ot *arrays* (i.e., finite sequences of variables that are accessed through an index). In the case of the controller of the robotic cell of Figure 9 we introduce three arrays, InputCRO (of 10 elements), OutputCRO (of 6 elements) and LocalCRO (of 2 elements), corresponding, respectively, to the sets D_I, D_O and D_L of the Stateflow model. We also introduce a $\mathbb{Z}$ot variable s representing the current state of the Stateflow diagram, whose domain $dom(s)$ corresponds to the set S of states. In the case of the diagram of Figure 9, StateCRO is a variable with domain $[0, \dots, 11]$, where each value corresponds to a different state and 0 is the initial state. We use temporal logic formulae to express constraints defining valid sequences of micro-steps. For readability, we write $v = k$ to mean that the actual value returned by the evaluation function of the current configuration for the variable v is k.

Given a Stateflow diagram representing the behaviour of a module m, for each transition $H_{m,i} : s_{m,i} \overset{g_{m,i}/a_{m,i}}{\longrightarrow} t_{m,i}$ with source state $s_{m,i}$ and target

state $t_{m,i}$, with condition expression $g_{m,i}$, we introduce the following formula:

$$(\gamma_{m,i} \wedge s_m = s_{m,i}) \rightarrow ((\bigcirc(s_m = t_{m,i})) \wedge \alpha_{m,i}) \quad (2)$$

where $\gamma_{m,i}$ is a Boolean formula encoding condition expression $g_{m,i}$, and $\alpha_{m,i}$ is a temporal logic formula encoding the transition action $a_{m,i}$.

The formula (2) asserts that if the current state is $s_{m,i}$ and the transition condition $\gamma_{m,i}$ holds in the current configuration, then in the next micro-step the new configuration active state must be $t_{m,i}$ and the action $a_{m,i}$ must be executed, updating the output and local variables at the beginning of the next micro-step. The left part of the formula (2) checks if the transition is enabled, implementing the predicate $enabled$, while the right part executes the transition, implementing the $\Longrightarrow^{SF}_{step}$ predicate, i.e. the formula models the execution of a micro-step.

If no transition is enabled, the system reaches a stable configuration, so the next configuration is exactly the same of the actual one, which is captured by the following formula:

$$(\bigwedge_{i=1}^{|H_m|} \neg(\gamma_i \wedge s_i = s_{m,i})) \rightarrow NOCHANGE \quad (3)$$

where $|H_m|$ is the number of transitions of the Stateflow diagram of the module m, and subformula $NOCHANGE$, which is not detailed here to simplify the exposition, asserts that in the next micro-step the current state and the values of all output and local variables do not change, except the input variables. The formula (3) models the execution of stutter transitions in stable configurations, waiting for the occurrence of an external event.

The complete definition of the behavior of the transitions of the Stateflow diagram of a module m is given by $\left(\bigwedge_{i=1}^{|H_m|} (2)_i\right) \wedge (3)$.

The time advancement of the run-to-completion Stateflow semantics is modeled, as mentioned in the introduction to this section, by a predicate called $tick$, which is added to the encoding of each Stateflow diagram. Predicate $tick$ holds in each micro-step following the one in which the system has reached a stable configuration. When predicate $tick$ holds, time advances to the next clock cycle of the PLC.

The behaviour of predicate $tick$ is captured by the following formula:

$$(\bigwedge_{i=1}^{|H_m|} \neg(\gamma_i \wedge s_m = s_{m,i}))) \Leftrightarrow (\bigcirc tick) \quad (4)$$

To foster information hiding, instead of relying on a single global predicate modelling the time advancement, in this encoding each module has its local tick predicate. As explained in section 3.2, additional predicates and formulae are introduced in the composed system model for the synchronization of the local ticks.

We introduce a formula asserting that when predicate tick does not hold, the values of the input variables D_I must be the same of the preceding microstep:

$$\Box(\neg \mathtt{tick} \rightarrow (\bigwedge_{\nu \in D_I} (\forall x((\overleftarrow{\bigcirc}(\nu = x)) \rightarrow (\nu = x))))) \quad (5)$$

where the $\overleftarrow{\bigcirc}$ and $\Box$ are, respectively, the *yesterday* and *globally* LTL operators: $\overleftarrow{\bigcirc}F$ holds if formula F held in the previous state, while $\Box F$ holds if F is true in the current and in all future state. The formula (5) models the fact that the system is sensible to external events only when it is in a stable configuration, forcing the input variables to not change their value in all other configurations. In this way, an error in a formula which encode an action which tries to update an input variable lead to an inconsistent logic model and can be immediately identified by a counterexample trace of the model checker tool.

The conjunction of formulae $MOD \equiv ((\bigwedge_{i=1}^{|H_m|} (2)_i) \wedge ((3) - (5)))$, is a formula that characterizes all the runs of a Stateflow diagram, i.e. it encodes the behaviour of a basic component.

Next, we encode the semantics of module composition described through Simulink graphs. We use a modular approach to hide the details of the time advancement of a module to the other components, exploiting the local semantics encoded by the formulae (2), (3), (4) and (5). A special integrator module M must be added to access the data of the public interfaces of each component; it embeds a number of axioms needed to ensure the correct behaviour of the composed module, as represented in the Simulink graph.

As described in Section 3.2, the local clock of a component that is part of a composed component advances only when the system has reached a global stable configuration. However, the local tick predicate of the module does not convey the information on when such event occurs, since it is not directly related to the state of the other components. To avoid the use of a global tick predicate shared between components, which would break compositionality, we add two new local predicates to the interface of each component module, called stable and $\mathtt{tick}^{ext}$. $\mathtt{stable}_m$ holds when the module m reaches a locally

stable configuration and it replaces predicate $\mathtt{tick}$ in formula (4), thus giving the new formula:

$$(\bigwedge_{i=1}^{|H_m|} (\neg(\gamma_i \wedge (s_m = s_{m,i})))) \leftrightarrow (\bigcirc \mathtt{stable}_m) \tag{6}$$

We use predicate $\mathtt{tick}^{ext}$ to convey to single modules the information about the overall system state. This predicate is set to true by the integrator module M when all its components reach a stable configuration. In the compositional semantics, the local $\mathtt{tick}_m$ of a component m holds iff both $\mathtt{stable}$ and $\mathtt{tick}^{ext}$ are true, which is captured by the following formula:

$$\Box(\mathtt{tick}_m \Leftrightarrow (\mathtt{stable}_m \wedge \mathtt{tick}^{ext}_m)) \tag{7}$$

Module M has also its own local clock predicates $\mathtt{tick}_M$, $\mathtt{stable}_M$ and $\mathtt{tick}^{ext}_M$, so the semantics is compositional in that M itself can be part of other composed modules, but this does not affect its formulae. If M is the composition of N_M modules, to obtain the synchronization of the clocks of the component modules, we impose the following three conditions:

1. Predicate $\mathtt{stable}_M$ is true iff all the predicates $\mathtt{stable}_m$ of the component modules are true. This condition implies that the local clock of each composed module advances iff the overall system is in a stable configuration.

2. Each predicate $\mathtt{tick}^{ext}_m$ of each component module must be equivalent to $\mathtt{tick}_M$.

3. Predicate $\mathtt{tick}_M$ is true iff predicates $\mathtt{stable}_M$ and $\mathtt{tick}^{ext}_M$ hold, such as for the local components.

The three conditions above are formalized through the following formulae embedded in the module M:

$$\Box(\mathtt{stable}_M \Leftrightarrow (\bigwedge_{i=1}^{n_M} \mathtt{stable}_i)) \tag{8}$$

$$\Box(\bigwedge_{i=1}^{n_M} (\mathtt{tick}_M \Leftrightarrow \mathtt{tick}^{ext}_i)) \tag{9}$$

$$\Box(\mathtt{tick}_M \Leftrightarrow (\mathtt{stable}_M \wedge \mathtt{tick}^{ext}_M)) \tag{10}$$

Formulae (8), (9) and (10) lead to a behaviour that is equivalent to implement the global $\mathtt{stable}^{SL}$ predicate.

Finally, we formalize the semantics of the communication between components, represented as links in the Simulink graph, as described in section 3. A link between an output variable of a component m_1 and an input variable of a component m_2 means that the corresponding data or event produced by m_1 is communicated and received by m_2. This corresponds to synchronizing the value of the input variable of m_2, to the value of the output variable of m_1 only when the overall system has reached a stable configuration, i.e. when the predicate $\mathtt{tick}_M$ holds. This is captured by the following formula:

$$\Box(\mathtt{tick}_M \rightarrow (\nu_{m_1,out} = \nu_{m_2,in})) \tag{11}$$

where $\nu_{m_1,out}$ is an output variable of component m_1 linked to the $\nu_{m_2,in}$ input variable of the component m_2. M contains an instance of formula (11) for each link of the Simulink graph.

In the compositional semantics, the formula SYS encoding the behavior of the composed system is given by the conjunction of the local formulae $SYS \equiv ((\bigwedge_{i=1}^{n_M} (2)_i) \wedge ((5) - (6)))$, for each component, plus the formulae ((7)-(11)) embedded in module M.

We remark that, since in the run-to-completion semantics the real time advances only when macro-steps are performed, to express the metric properties of a system we need to redefine some TRIO operators to reflect this notion of time advancement. First, we redefine the TRIO Dist operator, where $\text{Dist}(F, K)$ holds in each instant t such that formula F holds at the instant $t + K$. The following formula defines the meaning of the new operator $\text{Dist}_{new}(F, 1)$, to take into account that time advances only at the occurrence of $\mathtt{tick}$, starting from the operators predicating on micro-steps:

$$\text{Dist}_{new}(F, 1) = \bigcirc(\neg\mathtt{tick} \cup (\mathtt{tick} \wedge \bigcirc(\neg\mathtt{tick} \cup ((\bigcirc\mathtt{tick}) \wedge F)))) \tag{12}$$

where $\cup$ is the usual binary LTL operator *until*, and $A \cup B$ holds in those states such that there is a future state in which B holds, and A holds in all states up to that one (excluded).

Formula (12) asserts that $\text{Dist}_{new}(F, 1)$ holds if, at the end of the macro-step following the current one, F holds. The end of the current macro-step occurs in the micro-step right before the next state in which $\mathtt{tick}$ holds (i.e., the future micro-step in which $\bigcirc\mathtt{tick}$ is true and such that $\mathtt{tick}$ is false in all states in-between). $\text{Dist}_{new}(F, K)$ is therefore defined as follows:

$$\mathrm{Dist_{new}}(\ldots\mathrm{Dist_{new}}(\mathrm{Dist_{new}}(F,1),1\ldots,1)) \text{ (k times).}$$

Finally, we also redefine the Until TRIO operator, to predicate only on macro-steps. The new $\mathrm{Until_{new}}(A, B)$ is defined by the following formula:

$$\mathrm{Until_{new}}(A, B) = \bigcirc(((\bigcirc\mathtt{tick}) \rightarrow A) \cup ((\bigcirc\mathtt{tick}) \wedge B)) \tag{13}$$

Formula (13) asserts that $\mathrm{Until_{new}}(A, B)$ holds if there is a future state in which the clock ticks, B holds at the end of that macro-step, and A holds at the end of all macro-steps in between. Notice that formulae A and B are evaluated only in the last micro-step of a macro-step, when all system variables have certainly been updated.

3.4 SYSTEM PROPERTIES VERIFICATION AND EXPERIMENTAL RESULTS

In this section, it is illustrated how the encoding presented in section 3.3.2 can be exploited to check some relevant properties of the robotic cell of Figure 6. The formulae analysed in this section capture but a portion of the kinds of properties that can be checked using TRIO on a Simulink/Stateflow model; they show how the technique presented in this thesis can be applied to study a wide range of features of modelled systems. As mentioned in the introduction section of this chapter, we used the bound model checker ℤot to perform model checking on the TRIO model of the robotic cell. ℤot is described deeply in section 6.1; it performs checking of the properties by encoding temporal logic formulae into the input language of various solvers; for this tests, ℤot exploits the widely used SMT solvers Microsoft Z3 [25], so it is possible to represent also arithmetic variables and quantitative properties over them. To avoid infinite state systems, every variable has a finite domain, so it is possible to apply bounded model checking techniques without further decidability issues.

We first check that the modelled system does not have Zeno runs, which would make it unfeasible. We remember that the system shows a Zeno behaviour if, from a certain point on, time does not advance, i.e. predicate $\mathtt{tick}$ does not hold. The *presence* of Zeno runs is formalized by the following simple LTL formula:

$$\Diamond(\Box(\neg\mathtt{tick}_M)) \tag{14}$$

where $\mathtt{tick}_M$ is the global tick predicate of the robotic cell, and $\Diamond$ is the *eventually* LTL operator. $\Diamond F$ holds in a state if there is a future state in which

formula F holds. Formula (14) holds if, from a certain micro-step on, the clock does not tick anymore. Using the $\mathbb{Z}$ot tool, it has been checked that the formula $SYS \wedge (14)$ is *unsatisfiable*, which means that there are no runs of the system that also show property (14), hence the model does not exhibit Zeno runs.

Since we are now guaranteed that time advances in the modelled system, we can use the TRIO temporal operators to predicate on actual time instants (i.e., on macro-steps), and state metric properties such as "operation *OP* terminates within *K* time units", etc.

Next, we check for the existence of deadlocks in the system model. A model is *deadlock-free* if it cannot reach a configuration after which its state does not progress anymore. The usual definition of deadlock requires that the model *never* leaves its state s; in this case, however, we only consider what happens at the end of macro-steps, and while the intermediate micro-steps are ignored. In other words, the deadlock is defined over macro-steps only, considering internal micro-steps states as transient states, non-observable outside the module. Different analyses would have been possible with a simple tweak of the formulae checked. The presence of deadlock does not depend on the value of the input data since we have a closed-loop system. The system is in deadlock if *all* of its components are in a deadlock state. The following TRIO formula captures this notion of deadlock: it holds if all components $c \in C$ (with C the set of system components) can reach a deadlock state:

$$\bigwedge_{c \in C} \bigvee_{x \in \mathrm{dom}(s_c)} \text{SomF}(\text{AlwF}(s_c = x)) \tag{15}$$

where $\mathrm{dom}(s_c)$ is the set of states of component c, SomF and AlwF are the TRIO counterparts of the *eventually* and *globally* LTL operators; they are defined in Table 1.

Finally, a property concerning the possibility to produce and deliver one final artefact within T time units from the system start-up is analysed. The property is captured by the following formula:

$$\text{WithinF}((s_{Rob} = \mathrm{GoToCo1}) \vee (s_{Rob} = \mathrm{GoToCo2}), T) \tag{16}$$

The formula states that, within T time units from the start of the system, one of the two states $\mathrm{GoToCo1}$ or $\mathrm{GoToCo2}$ of Figure 9 is reachable. The Stateflow diagram reaches state $\mathrm{GoToCo1}$ if a workpiece of any type has been produced

by *Machine 1*, and similarly for GoToCo2 and *Machine 2*. WithinF is a TRIO operator derived from Dist:

$$\text{WithinF}(F, T) \stackrel{\text{def}}{=} \bigvee_{0 \leqslant t \leqslant T} \text{Dist}(F, T)\,.$$

In the tests for the formula (16), we used two values, 15 and 20, for T. Formula (16) does not hold if $T = 15$, but it does if $T = 20$. Analysing the output of the $\mathbb{Z}$ot tool, in the latter case we found that 16 is the minimum number of time units to satisfy the formula (which can be confirmed by checking formula (16) with $T = 16$).

Some performance results obtained during the verification of properties (14) and ((15)-(16)) with the two values of T mentioned above, are shown in Table 3. Table 3 shows the time required by the tool to check the property, the memory occupation and the result, i.e. whether the property holds or not. All tests have been carried out on a 3.3 Ghz quad core PC with 4 Gbytes of Ram.

Table 3: Test results.

Formula	Time (sec)	Memory (Mb)	Result
Zeno Paths detection (14)	85	264	No
Deadlock detection (15)	17991	268	No
Production, T=15 (16)	407	260	No
Production, T=20 (16)	89	272	Yes

The Stateflow diagram of Figure 9 has $12 \cdot 2^{18}$ possible configurations (corresponding to the state space of cardinality $|S| \cdot 2^{|D|}$); the overall system model, which also includes diagrams for all the other components, is considerably larger. As a consequence, deadlock detection analysis (formula (15)) takes a long time, as the tool must exhaustively analyse all possible runs.

Formulae (14) and (16), instead, formalize *reachability* properties; their analysis is much faster, since the tool stops as soon as it finds a run that satisfies the formula. Verifications was performed with a bound of 70 time units, which is a user-defined parameter that corresponds to the maximal length of runs analysed by $\mathbb{Z}$ot. It is well known that, for each LTL formula, exists a *completeness threshold* CT for the bounded model checking problem: it represents the minimum value for the bound such that, if there is no counterexample to the formula of length CT or less, then the formula is satisfiable. We checked

automatically, using a particular parameter of ℤot called textitloop-free, that 70 is a valid CT for the formulae (14), (15) and (16).

To conclude this section, we briefly illustrate an example of verification that allowed us to detect and correct errors in a previous version of the model. By feeding ℤot formula (15) on an earlier model of the robotic cell, not shown here to simplify the exposition, the tool determined that deadlock configurations did exist, and it returned a counterexample of a deadlocked run. By studying this run, we discovered that the system model remained forever in configurations with state GoToCIn1, (see Figure 9). The run, which is summarized in Figure 13, shows a problem in the communication protocol between the *Robot* component and the *Machine 1* component, which also affects the cell *Controller*.

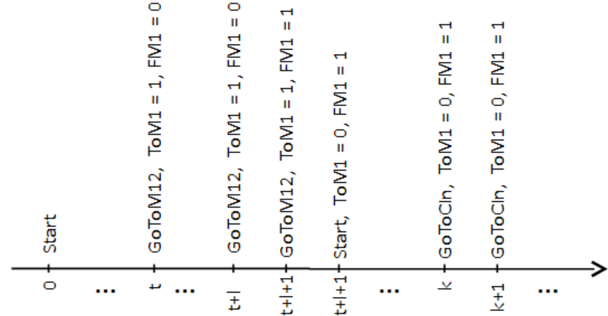

Figure 13: Deadlocked run returned by ℤot.

The run is represented as a timeline, starting from 0. Over each micro-step we report the partial configuration with the name of the active state of the *Controller* component of Figure 9, and the values of the input variable FM1 and the output variable ToM1 (the other variables are not shown). The label below the micro-step (e.g., t) identifies the macro-step to which it belongs. When the *Controller* component enters state GoToM12 at time t, it signals to *Machine 1*, sending an output event modelled by variable ToM1, that the *Robot* has just arrived. After working for l instants of time, at time $t + l + 1$ *Machine1* signals to the *Controller* the "work termination" event, which is mapped to variable FM1. The transition between states GoToM12 and Start becomes enabled and the *Controller* component returns to the initial state, resetting FM1 to signal the end of the communication. The problem occurs if the *Controller* component reaches state GoToCIn a second time, such as at

time k in Figure 13. In fact, starting from the time instant $k+1$, the *Controller* component cannot leave the state GoToCIn anymore since variable FM1 has not been reset by *Machine 1*. Hence, no outgoing transitions are enabled. After correcting the error, a new check of property (15) showed that the modified system model is deadlock-free.

3.5 LIMITATIONS: TOWARDS A NEW METRIC TEMPORAL LOGIC

The main drawback of the approach is mainly due to the nature of the metric temporal logic TRIO. In fact, the original TRIO language is well suited to deal both with continuous systems that evolve in a continuous time domain and with discrete systems where each step takes exactly one time unit; finally, it can deal also with heterogeneous systems that combine both continuous and discrete components through suitable approximations [26]. More complex systems, however, evolve through discrete steps, whether in continuous or discrete time and state domains; furthermore, different steps may require time durations that differ even by orders of magnitude from each other. An example of these systems are reactive systems, such as the simple robotic cell used as a case study in this thesis.

These type of systems are, usually, modelled through high level graphical languages such as Stateflow, whose run-to-completion semantics is based on the synchrony hypothesis: under this hypothesis, the system responds immediately to new input events. Further more, it responds infinitely fast. This hypothesis is suitable for reactive systems, such as the abstract model of a PLC cycle, since it holds under the following assumptions:

1. the environment can be described as a discrete process, namely as an infinite sequence of inputs $\{I_i\}_{i \in \mathbb{N}}$, occurring at successive instants of time

2. the system is infinitely faster than the environment, so its reaction to inputs I_i is completed before inputs I_{i+1} are produced

Systems under this hypothesis, usually, are modelled assuming that their evolution is a sequence of steps of two different types (and durations), micro- and macro-steps, of which, normally, only the latter ones "consume time": micro-step durations, being negligible w.r.t macro ones, is roughly assimilated to zero-time. This assumption simplifies models and their analysis but it is not without drawbacks. In fact, in temporal logics such as TRIO, that

adopt a view of system state as a function of time, such a notion of zero-time transition is however counterintuitive (when a zero-time micro-step is performed more than one state is associated with a single time instant), and may lead to logical contradiction. For example, to overcome this contradiction, in the encoding of the semantics of Stateflow, the $\bigcirc$ operator is exploited to model a zero-time transition, executing a micro-step in one "logical" time unit; to distinguish between "logical" time and "real" time, we use an explicit "tick" predicate. This approach is similar to some temporal logics, which adopt a time structure where more than one system state may be associated with a single time instant (an example, reported in chapter 5, is the super-dense time model) and therefore must provide distinct notations to refer to time and state change. In these notations, the progress of time and state evolution are fully decoupled, which is rather unnatural in most practical cases.

To accurately and naturally model such systems, we have decided to follow a different path: the idea is to develop a new metric temporal logic, enriching TRIO with a new "next-step" operator; whereas normally in metric frameworks this operator is associated with exactly one time unit (in TRIO terms this would mean $\bigcirc\phi \equiv \text{Dist}(\phi, 1)$), here we do not associate a fixed time duration to the execution of a step; rather, we distinguish between micro-steps that occur in a negligible but non-null time duration, and macro-steps that take a finite, but in general not *a priori* fixed time, using two different versions of the new operator. We borrow from Nonstandard Analysis (NSA) [15] the concept of infinitesimal number to formalize the non-null but negligible duration of micro steps, as opposed to that of macro-steps which is represented by standard numbers. This is not a completely novel approach, since in literature other works exploited it to model reactive or discrete-event systems; some of them are reported in chapter 5, with a brief description of their differences respect to the approach described in this thesis.

We called this new metric temporal logic *X-TRIO*; its main features are described in the next chapter. From a mere experimental viewpoint, the new logic does not add performance improvements respect to the TRIO encoding described in this chapter. Furthermore, although the full version of new logic, in the same way as TRIO, is Touring equivalent (since it includes first-order arithmetic), its actual decidable fragment, called X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$, is not expressive more than the classic LTL since, in the chapter 4, a complete procedure to translate formulae written in X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ to equisatisfiable LTL formulae is given.

4 X-TRIO: A METRIC TEMPORAL LOGIC FOR FMS

In this chapter is described deeply the new metric temporal logic *X-TRIO*. In its full generality, X-TRIO includes full arithmetic and is, therefore, undecidable; thus, we look for a suitable restriction that makes it decidable, but still general enough to formalize and analyse the main properties of various systems of industrial relevance. Among the many possible ones, the approach we used is based on a syntax inspired by decidable versions of Metric Temporal Logic (MTL) [27] and a "fine tuning" of the temporal domain over which X-TRIO formulae are interpreted. The decidability of the chosen X-TRIO subset is then shown through translation into LTL, which enables the use of any LTL-based satisfiability solver. Finally, to show the practical applicability and generality of X-TRIO, we use it to formalize the semantics of Stateflow, in a similar way to the one described in 3.3.

4.1 THE GENERAL X-TRIO LOGIC

In this section, we introduce X-TRIO in its full generality. After a short summary on the main concepts of NSA, we give the syntax and the model-theoretic semantics of the full version of the X-TRIO logic. We conclude this section with a few preliminary examples of application of the logic.

4.1.1 *An introduction to Non Standard Analysis*

In this section, we introduce the main concepts of the modern theory of infinitesimals founded by A. Robinson [15], the Non-Standard Analysis (NSA). We provide only a minimum background that is needed to explain our application of this theory to the modelling of reactive systems, by changing the underline model of time.

The main idea that facilitates practical application of NSA is given in [28]: the author defines a theory, called Internal Set Theory (IST), which includes a typical axiomatization of arithmetic and extends it through the predicate *standard* (briefly $\mathtt{st}$), which is deliberately left undefined, plus three additional axiom schemes.

The predicate *standard* help us to introduce the concept of *infinitesimal* number in a standard temporal domain T, such as the sets $\mathbb{R}$ or $\mathbb{N}$. Given a domain T, the number ϵ is infinitesimal respect to T if $\epsilon \geqslant 0$ and ϵ is smaller than any number in $T_{>0}$.

The original values of T are classified as *standard* and are characterized by the above predicate st, that is, x is standard iff $st(x)$ holds. T is, by this way, augmented with infinitesimal numbers and all numbers resulting from adding and subtracting infinitesimal non-zero numbers to and from standard ones, respectively.

Predicate $ns(x)$ denotes that x is *non-standard*. For each x, $st(x)$ holds if and only if $ns(x)$ does not hold. Note that 0 is the only infinitesimal standard number. It is possible to demonstrate that all non-standard numbers are of the form $\nu \pm \epsilon$, where $st(\nu)$ holds, and ϵ is infinitesimal greater than 0; since we use this form for non-standard numbers in the rest of the thesis, we give the demonstration of the decomposition over, for example, the set of the hyperreals $\overline{\mathbb{R}}$:

Theorem 4.1.1. *Any finite non-standard real number $x \in \overline{\mathbb{R}}$, can be uniquely decomposed as a sum $x = \nu \pm \epsilon$, where ν is a real standard number and ϵ is an infinitesimal.*

Proof. For the unicity of the decomposition, consider that if one would have $x + \epsilon = y + \mu$ where x and y are two usual real numbers and ϵ and μ are two infinitesimals numbers, one would indeed deduce that $x + y = \mu + \epsilon$. But, since this quantity is both a standard real number and an infinitesimal number, it must be equal to 0, which shows that one both has $x = y$ and $\epsilon = \mu$ as expected. On the other hand, let $x = [(x_n) : n \geqslant 0]$ be a finite non-standard real number. Let us then consider the subset B of $\mathbb{R}$ defined by setting $B = b \in \mathbb{R} : b < x$. Due to the finiteness of x, it is easy to see that B is a bounded set of real numbers which hence has a supremum $a \in \mathbb{R}$. Let us then set $\epsilon = x - a$ and let us suppose that there exists a strictly positive real number $r \in \mathbb{R}$ such that $r < |\epsilon|$. If $\epsilon > 0$, we would have $r < x - a$, i.e. $a < a + r < x$, in contradiction with the fact that a is the supremum of B (there would be a bigger real number $a + r \in B$ than a). On the other hand, if $\epsilon < 0$, we would have $r < a - x$, hence $x < a - r < a$, again contradicting the fact that a is the supremum of B ($a - r$ would be a small upper bound of B than a). Hence ϵ must be an infinitesimal, which ends the proof. □

We denote the extension of the original standard time domain T with infinitesimal numbers as $\overline{T}$. $\overline{T}$ is a totally ordered set of numbers.

NSA provides an axiomatization that allows one to apply all arithmetic operations and properties of traditional analysis in an intuitive way: for instance, the sum of two standard numbers is standard, the sum of two infinitesimal numbers is an infinitesimal number and the sum of an infinitesimal number with a standard number is a non-standard number.

The theory of NSA introduces, in addition to the notion of infinitesimal numbers and operations on them, the notion of infinite numbers (which are, intuitively, greater than any value in T), plus a rich set of results that make NSA an appealing framework for reasoning on both familiar and new objects. One of the most important result is that any standard domain T and its extension with infinitesimal numbers, $\overline{T}$, are *elementarily equivalent*, which means that the first order logical properties of T and $\overline{T}$ (expressed in the logical theory of ordered fields) are exactly the same. This property is in fact a special case of a more general result, called the *transfer principle*, which claims that, given a set-theoretical formula ϕ involving only variables on T which does not have any free variable, ϕ is true iff ϕ' is true, where ϕ' stands for the formula on non-standard domain obtained from ϕ by transfer, i.e. by changing each usual standard set a involved in ϕ by its non-standard version $\overline{a}$. As one can immediately see, the transfer principle is a very strong property of non-standard domains since it claims basically that every usual property of standard domains holds for non-standard domains up to replacing standard sets by their non-standard equivalents.

We exploit some of the terminology and concepts of NSA to provide an elegant characterization of zero-time steps, but I do not make use of the full power of the theory; for example, I do not deal with infinite numbers (i.e., we have that $ns(x)$ iff $x = \nu \pm \epsilon$, with $st(\nu)$ and ϵ infinitesimal), as they are useless when modelling zero-time transitions.

4.1.2 *X-TRIO syntax and semantics*

The syntax of X-TRIO is a straight extension of the TRIO one:

$$
\begin{aligned}
\phi &:= p(\tau_1, \ldots, \tau_n) \,|\, \neg\phi \,|\, \phi_1 \wedge \phi_2 \,|\, \mathrm{Dist}(\phi, \tau) \,|\, \forall \nu.\phi \,|\, X(\phi) \,|\, Y(\phi) \,|\, st(\tau) \\
\tau &:= \nu \,|\, k \,|\, f(\tau_1, \ldots, \tau_n)
\end{aligned}
\tag{17}
$$

Derived temporal operators can be defined as in Tables 1 and 2.

In keeping with the tradition of TRIO [29], X-TRIO can be interpreted over different temporal domains.

A model-theoretic semantics for X-TRIO is defined by following a fairly standard path on the basis of a temporal structure $S = \langle \overline{T}, \mathcal{D}, \beta, \nu, \sigma \rangle$, where:

- $\overline{T}$ is the time domain such that $\forall t \in \overline{T}$ it is $t \geqslant 0$.
- $\mathcal{D}$ is the union of the domains associated with functions and predicates (i.e., of their arguments and results).
- β associates, at every time instant, each function and predicate with its interpretation in that instant. For example, given a predicate p, $\beta(p,t)$ is the relation associated with p at instant t. β can be seen as the "system state", i.e., what holds in each time instant.
- $\nu : V \longrightarrow \mathcal{D}$ is an evaluation function that associates with each variable and constant a value in its domain.
- σ is the distinguishing element of the X-TRIO temporal structure; it is a (possibly infinite) sequence of time instants starting from the initial instant 0, called *History*. Intuitively, it represents the discrete sequence of instants when the system changes state; more precisely I have $\sigma = \{\sigma_i | i \in \mathbb{N}, \sigma_i \in \overline{T}, \sigma_0 = 0, \forall j \in \mathbb{N}$ s.t. $j < i$ it is $\sigma_j < \sigma_i$, and $\forall t \in \overline{T}$, if $\sigma_i < t < \sigma_{i+1}$, then for all function or predicate e it is $\beta(e, \sigma_i) = \beta(e,t)\}$.

Given a term τ, its value at time t is computed through a function α that is defined as follows: for each variable (resp. constant) it is $\alpha(\nu, t) = \nu(\nu)$ (resp. $\alpha(k,t) = \nu(k)$); if $\tau = f(\tau_1, \ldots, \tau_n)$, then $\alpha(\tau, t) = \beta(f,t)(\alpha(\tau_1,t), \ldots, \alpha(\tau_n,t))$.

The satisfaction relation $\models$ of an X-TRIO formula ϕ on a structure $S = \langle \overline{T}, \mathcal{D}, \beta, \nu, \sigma \rangle$ at a time instant $t \in \overline{T}$ is defined as follows:

$S, t \models p(\tau_1, \ldots, \tau_n)$ iff $\langle \alpha(\tau_1,t), \ldots, \alpha(\tau_n,t) \rangle \in \beta(p,t)$

$S, t \models \neg\phi$ iff $S, t \not\models \phi$

$S, t \models \phi_1 \wedge \phi_2$ iff $S, t \models \phi_1$ and $S, t \models \phi_2$

$S, t \models \mathrm{Dist}(\phi, d)$ iff $t + \alpha(d,t) \in \overline{T}$ and $S, t + \alpha(d,t) \models \phi$

$S, t \models X(\phi)$ iff there is $i \in \mathbb{N}$ s.t. $\sigma_i \leqslant t < \sigma_{i+1}$ and $S, \sigma_{i+1} \models \phi$

$S, t \models Y(\phi)$ iff there is $i \in \mathbb{N}$ s.t. $\sigma_{i-1} < t \leqslant \sigma_i$, $i > 0$ and $S, \sigma_{i-1} \models \phi$

$S, t \models \forall \nu.\phi$ iff for all ν' that differ from ν at most for ν, $\langle \overline{T}, \mathcal{D}, \beta, \nu', \sigma \rangle, t \models \phi$

Finally, a formula ϕ is *satisfiable* in a structure S iff $S, 0 \models \phi$. Notice that in this thesis we consider system evolutions that conventionally begin from time 0.

4.1.3 *Examples of usage of X-TRIO*

In its present general version, X-TRIO allows users to express any system property of interest (remember that it has full Turing computational power since it includes first-order arithmetic.). Let us show some examples of X-TRIO formulae.

Consider as time domain the following restriction of the hyperreals $^*\mathbb{R}$ [30]: $^*\mathbb{R}_{\geqslant 0} = \{x \in {}^*\mathbb{R} | x \geqslant 0\}$.

First of all, it will be useful in the following to identify the origin of the temporal domain (which is, by definition, mono-infinite). This can be done through the following X-TRIO formula, where k is any constant in $^*\mathbb{R}_{\geqslant 0}$:

$$\mathtt{orig} = \forall d.(0 < d < k \to \text{Dist}(\neg\text{Dist}(\top, -k), d))$$

In fact, formula $\mathtt{orig}$ holds only at instant 0, since $\text{Dist}(\neg\text{Dist}(\top, -k), d)$ is true in an instant $t \in {}^*\mathbb{R}_{\geqslant 0}$ iff at $t + d$ $\text{Dist}(\top, -k)$ is false, which occurs iff $t + d - k < 0$. Then, for $0 < t < k$, if $d = k - t/2$, it is $t + d - k = t/2 > 0$, so $\mathtt{orig}$ does not hold (similarly if $t \geqslant k$).

We can also introduce an abbreviation for the duration of the current step, which is the difference of the "timestamps" (i.e., the distance from the origin) between the end and the start of the step:

$$\mathtt{Dur}(d) = \exists d_1, d_2(\text{X}(\text{Past}(\mathtt{orig}, d_1) \wedge \text{Y}(\text{Past}(\mathtt{orig}, d_2))) \wedge d = d_1 - d_2)$$

We can distinguish micro-steps (which take an infinitesimal time) from macro-steps (which take a non-infinitesimal time) by means of the following new derived operators, where X_m (resp. X_M) stands for "the next step is a micro one" (resp. macro):

$$\begin{aligned} X_m(\phi) &= X(\phi) \wedge \forall d(\mathtt{Dur}(d) \to \mathtt{inf}(d)) \\ X_M(\phi) &= X(\phi) \wedge \forall d(\mathtt{Dur}(d) \to \neg\mathtt{inf}(d)) \end{aligned}$$

We introduce operators to state whether the next step ends in a standard or in a non-standard instant. For this, it is useful to introduce abbreviation $\text{NowST} = \exists t(t > 0 \wedge \text{Past}(\mathtt{orig}, t) \wedge \mathtt{st}(t))$, which holds exactly in all those instants that are standard.

$$\begin{aligned} X_{st}(\phi) &= X(\phi \wedge \text{NowST}) \\ X_{ns}(\phi) &= X(\phi \wedge \neg\text{NowST}) \end{aligned}$$

Now, we use X-TRIO formulae to define some interesting behaviours of systems that evolve through micro- and macro-steps.

The formula (18) states that "the system keeps going forever", i.e. at any time, the system will make progress with further steps, whether micro or macro:

$$\mathrm{AlwF}(\mathrm{SomF}(X(\top))) \tag{18}$$

Conversely, the following formula (19) claims that at some point the system will stop forever:

$$\mathrm{SomF}(X(\mathrm{AlwF}(\neg X(\top)))) \tag{19}$$

i.e., there will be a point at some time when the next step will cause the system to have no further steps. Thus the formula can be satisfied only by a finite sequence σ.

The following formula (20) is similar to (19):

$$\mathrm{SomF}(\mathrm{AlwF}(\neg X(\top))) \tag{20}$$

The formula (20), differently from (19), could also be satisfied by an infinite σ where the steps never advance, past a certain instant of time: this is a Zeno behaviour, first defined in section 2.1.2. Next I show how X-TRIO allows me to formalize and distinguish in a natural way various forms of such pathological behaviours, in particular different types of Zeno behaviours.

A first sufficient but not necessary condition to exhibit a Zeno behaviour is that, from some point on, only micro-steps occur:

$$\mathrm{SomF}(X_m(\top) \wedge \mathrm{AlwF}(X_m(\top) \rightarrow X_m(X_m(\top)))) \tag{21}$$

On domain $^*\mathbb{R}_{\geqslant 0}$, a Zeno behaviour can occur also with a σ consisting exclusively of macro-steps, which however have an always decreasing duration. The following formula (22) could be satisfied, e.g., by a sequence σ whose steps σ_i occur at time instants $\sum_{k=0}^{i-1} \frac{1}{2^k}$:

$$\mathrm{SomF}(\mathrm{AlwF}(\neg X(\top))) \wedge \mathrm{SomF}(X_M(\top) \wedge \mathrm{AlwF}(X_M(\top) \rightarrow X_M(X_M(\top)))) \tag{22}$$

If, instead, we restrict the time domain to be a discrete set augmented with infinitesimals, such as the one that will be used in section4.3, then only formula (21) captures Zeno behaviours.

We can also specify so-called "Berkeley behaviours" [31], i.e., those where time keeps advancing, but the step duration is ever decreasing or, more precisely it becomes shorter than any standard number:

$$\begin{gathered}\mathrm{AlwF}(\mathrm{SomF}(X_M(\top))) \wedge \\ \forall t(st(t) \rightarrow \mathrm{SomF}(\mathrm{AlwF}(X_M(\top) \rightarrow \forall d(Dur(d) \rightarrow d < t)))\end{gathered} \tag{23}$$

The formula (23) is satisfied, e.g., by a sequence whose steps occur at instants of time $\sum_{k=1}^{i} \frac{1}{k}$.

From the point of view of the physical intuition, the behaviours specified by formulae (20), (21), (22), and (23) could all be considered as "pathological" and could look as "almost indistinguishable", but the difference in the mathematical formalization could be used, for instance, to separate cases in which an unstable clock ever increases its frequency (formulae (22) and (23)) form cases where an unacceptable number of gate switches is supposed to occur within a clock period.

4.2 TOWARDS DECIDABLE VERSIONS OF X-TRIO

Since the advent of model checking, much research effort has been devoted to the definition of logic languages that exhibit a best trade-off between naturalness (ease of usage), expressiveness (computational power), and decidability[1] effort (complexity of decision procedures). In the case of TRIO, such an effort has produced several "LTL-oriented" versions of the original language and supporting tools. In this section, we trace a path to achieve the same goal for the new X-TRIO language. To this end, several directions are possible, depending on the combination of syntactic and domain restrictions chosen. In fact, the full generality of temporal domains augmented with infinitesimal numbers, as in NSA, is both too powerful to achieve decidability and often even useless in practical cases (infinite numbers have been already excluded since we are not interested in behaviours that take, say, an infinite time to perform a single step); thus, there are several ways of restricting time domains which clearly affect decidability and computational complexity properties. Some of these are effective only when combined with suitable syntactic restrictions, which in turn depend on the domain chosen. For instance, in most practical cases we can assume that a macro step ends always in a standard element of the time domain, i.e., $X_M(\top) \leftrightarrow X_{st}(\top)$. In this thesis, we will focus on one particular version of X-TRIO that exhibits a good trade-off between expressiveness and computational effort: it can be translated into LTL formulae (with past operators), it is PSPACE-complete, and it can be (and has been) implemented in satisfiability solvers for LTL, such as $\mathbb{Z}$ot.

First, however, we circumscribe the range of alternatives by exploring some more general variants of the logic, for which we analyse expressiveness and

1 As usual, in this thesis by *decidable logic language* we mean a language whose satisfiability problem is decidable.

decidability properties. This will give us a feel for the bounds within which it is reasonable to search for meaningful, though decidable variants of X-TRIO, and it will also point to further promising directions along which the logic can be extended. When looking for decidable fragments of temporal logic, a first natural (though not necessary [32]) syntactic restriction to consider is to avoid explicit first-order quantifications. From the point of view of the temporal domain, instead, discrete ones are natural candidates; however, dense time domains have also been widely investigated in literature for classic temporal logics [33, 34], so we will start by focusing on them.

4.2.1 *A propositional X-TRIO*

We start by restricting X-TRIO to a propositional, MTL-like syntax, in which the basic temporal operators are the metric Until and the metric Since. In addition, we specialize the X operator in two separate cases, X_{st} and X_{ns}. We also introduce a condition on the history σ that will entail that a "jump" from σ_i to σ_{i+1} corresponds to a macro-step (resp. micro-step) if and only if $st(\sigma_{i+1})$ (resp. $ns(\sigma_{i+1})$).

All in all, we give the following syntax for X-TRIO (where $p \in AP$, a and b are constant distances such that $a \leqslant b$, $a < \infty$, $b \leqslant \infty$, $\langle \in \{(, [\}$, $\rangle \in \{),]\}$, and $\rangle \neq]$ if $b = \infty$):

$$\begin{aligned} \phi \ := \ & p \,|\, \neg\phi \,|\, \phi_1 \wedge \phi_2 \,|\, \mathrm{Until}_{\langle a,b \rangle}(\phi_1, \phi_2) \,|\, \mathrm{Since}_{\langle a,b \rangle}(\phi_1, \phi_2) \,| \\ & X_{st}(\phi) \,|\, X_{ns}(\phi) \,|\, Y_{st}(\phi) \,|\, Y_{ns}(\phi) \end{aligned} \quad (24)$$

We denoted the new propositional X-TRIO logic, yet defined over the augmented domain $\overline{T}$ and including the condition on the history, with X-TRIO$^{P}_{\overline{T}}$, where the P letter means $\mathsf{Propositional}$.

As a consequence of the simplified syntax, the structures over which formulae are to be interpreted are also simpler. More precisely, a temporal structure is now a triple $S = \langle \overline{T}, \beta, \sigma \rangle$, where:

- $\overline{T}$ is the time domain (as before, $\forall t \in \overline{T}$ it is $t \geqslant 0$).
- $\beta : \overline{T} \longrightarrow 2^{AP}$ associates, to every instant of time, the propositions that hold in that instant.
- σ is defined as before, with the additional constraint that if $i > 0$ and $ns(\sigma_i)$, then for all $\sigma_{i-1} < j < \sigma_i$ it is $ns(j)$.

Then, the satisfaction relation $\models$ on a structure $S = \langle \overline{T}, \beta, \sigma \rangle$ at a time instant $t \in \overline{T}$ is defined as follows:

$S, t \models p$ iff $p \in \beta(t)$
$S, t \models \neg\phi$ iff $S, t \not\models \phi$
$S, t \models \phi_1 \wedge \phi_2$ iff $S, t \models \phi_1$ and $S, t \models \phi_2$
$S, t \models X_{st}(\phi)$ iff there is $i \in \mathbb{N}$ s.t. $\sigma_i \leqslant t < \sigma_{i+1}, S, \sigma_{i+1} \models \phi$ and $st(\sigma_{i+1})$
$S, t \models X_{ns}(\phi)$ iff there is $i \in \mathbb{N}$ s.t. $\sigma_i \leqslant t < \sigma_{i+1}, S, \sigma_{i+1} \models \phi$ and $ns(\sigma_{i+1})$
$S, t \models Y_{st}(\phi)$ iff there is $i \in \mathbb{N}$ s.t. $\sigma_{i-1} < t \leqslant \sigma_i, i > 0, S, \sigma_{i-1} \models \phi$ and $st(\sigma_{i-1})$
$S, t \models Y_{ns}(\phi)$ iff there is $i \in \mathbb{N}$ s.t. $\sigma_{i-1} < t \leqslant \sigma_i, i > 0, S, \sigma_{i-1} \models \phi$ and $ns(\sigma_{i-1})$
$S, i \models \text{Until}_{\langle a,b \rangle}(\phi, \psi)$ iff $\exists t \in \overline{T}$ s.t. $i + a \prec_1 t \prec_2 i + b$ and $S, t, \models \psi$
and $\forall t' \in \overline{T}$ s.t. $i \leqslant t' < t$ it is $S, t' \models \phi$
$S, i \models \text{Since}_{\langle a,b \rangle}(\phi, \psi)$ iff $\exists t \in \overline{T}$ s.t. $i - b \prec_2 t \prec_1 i - a$ and $S, t \models \psi$
and $\forall t' \in \overline{T}$ s.t. $t < t' \leqslant i$ it is $S, t' \models \phi$

Note that, in the semantics of the metric Until and Since, $\prec_1$ (resp. $\prec_2$) is $<$ or $\leqslant$ depending on whether the left (resp. right) endpoint is included or excluded.

In the next subsections 4.2.2 and 4.2.3 we study some relevant properties of the X-TRIO$^P_{\overline{T}}$. In particular, we study how its expressiveness and decidability are affected by the choice of temporal domain and by possible restrictions on the temporal operators used.

4.2.2 *Expressiveness of X-TRIO$^P_{\overline{T}}$*

When formalizing and analysing systems that evolve through micro- and macro-steps, it is often useful to be able to write formulae that can separate between the two kinds of instants. More precisely, we would like to be able to define an X-TRIO$^P_{\overline{T}}$ formula F that holds in an instant t if and only if t is standard. In this section, we investigate this issue, and, as a corollary, we derive some expressiveness results for X-TRIO$^P_{\overline{T}}$.

Let us consider, for the temporal domain $\overline{T}$, the following subset of the hyperreals:

$$\overline{\mathbb{R}}_+ = \{x \in {}^*\mathbb{R} | \exists \nu, \epsilon \geqslant 0 \text{ s.t. } st(\nu), inf(\epsilon) \text{ and } x = \nu + \epsilon\}$$

It includes all numbers of the form $\nu + \epsilon$, where $\nu \geqslant 0$ is a real number, and $\epsilon \geqslant 0$ is an infinitesimal number.

We have the following Theorem 4.2.1:

Theorem 4.2.1. *No X-TRIO$^{p}_{\overline{T}}$ formula F such that $AP = \varnothing$, evaluated over the temporal domain $\overline{T} = \overline{\mathbb{R}}_+$, that does not include instances of the* $\text{Since}_{\langle a,b \rangle}$ *operator, is such that F is true in an instant $t \in \overline{\mathbb{R}}_+$ iff t is such that $st(t)$.*

To prove Theorem 4.2.1 we first introduce the following abbreviations:

$$\begin{array}{rcl} \text{Until}(\phi, \psi) & = & \text{Until}_{(0,\infty)}(\phi, \psi) \\ \text{SomF}(\phi) & = & \text{Until}_{[0,\infty)}(\top, \phi) \\ \text{Futr}(\phi, d) & = & \text{Until}_{[d,d]}(\top, \phi) \\ \text{WithinF}_{\langle\rangle}(\phi, d) & = & \text{Until}_{\langle 0,d \rangle}(\top, \phi) \\ \text{Lasts}_{\langle\rangle}(\phi, d) & = & \neg \text{WithinF}_{\langle\rangle}(\neg\phi, d) \end{array}$$

and the following equivalences:

$$\begin{array}{rcl} \text{Until}_{(a,\infty)}(\phi, \psi) & \equiv & \text{Lasts}_{[)}(\phi, a) \wedge \text{Futr}(\text{Until}(\phi, \psi), a) \\ \text{Until}_{[a,\infty)}(\phi, \psi) & \equiv & \text{Lasts}_{[)}(\phi, a) \wedge \text{Futr}(\psi \vee \text{Until}(\phi, \psi), a) \\ \text{Until}_{(a,b\rangle}(\phi, \psi) & \equiv & \text{Lasts}_{[)}(\phi, a) \wedge \\ & & \text{Futr}\Big(\text{Until}(\phi, \psi) \wedge \text{WithinF}_{(\rangle}(\psi, b-a), a\Big) \\ \text{Until}_{[a,b\rangle}(\phi, \psi) & \equiv & \text{Lasts}_{[)}(\phi, a) \wedge \\ & & \text{Futr}\Big((\psi \vee \text{Until}(\phi, \psi)) \wedge \text{WithinF}_{[\rangle}(\psi, b-a), a\Big) \end{array} \tag{25}$$

Hence, in the following proofs, we use Until (non-metric), Futr and WithinF as basic temporal operators, instead of the metric $\text{Until}_{\langle a,b \rangle}$, without loss of generality. We can similarly define operators Since (non-metric), SomP, Past, WithinP and Lasted.

Before tackling the proof of Theorem 4.2.1, we introduce the auxiliary lemma 4.2.2:

Lemma 4.2.2. *If $AP = \varnothing$ and $S = \langle \overline{\mathbb{R}}_+, \beta, \sigma \rangle$ is such that the history σ contains only standard instants of time (i.e., $\forall i \in \mathbb{N}$ it is $st(\sigma_i)$), any X-TRIO$^{p}_{\overline{T}}$ formula F that does not include instances of the* $\text{Since}_{\langle a,b \rangle}$ *operator and with less than n instances of the Y_{st} operator is such that, given two instants $t_1 < t_2$ with $t_1 > \sigma_n$, it is $S, t_1 \models F$ iff $S, t_2 \models F$.*

Proof. First of all, note that in structure S, formulae $X_{ns}(\phi)$ and $Y_{ns}(\phi)$ are always false, independent of ϕ. The proof is by induction on the structure of F.

The base cases, in which $F = \top$ or $F = \bot$ are trivial, and so are the cases $F = \phi_1 \wedge \phi_2$ and $F = \neg\phi$.

Suppose F is $\text{Futr}(\phi, k)$; then, we have $S, t_1 \models F$ iff $S, t_1 + k \models \phi$; since $\sigma_n < t_1 < t_1 + k$, by inductive hypothesis this holds iff $S, t_2 + k \models \phi$, i.e., iff $S, t_2 \models \text{Futr}(\phi, k)$.

If $F = \text{WithinF}_{()}(\phi, k)$, $S, t_1 \models F$ iff there is $0 < d < k$ s.t. $S, t_1 + d \models \phi$.

Since $t_2 + d > t_1 + d > \sigma_n$, by inductive hypothesis, $S, t_1 + d \models \phi$ iff $S, t_2 + d \models \phi$, hence $S, t_2 \models F$. The cases for other variants of the WithinF operator and for the Until are similar.

If $F = X_{st}(\phi)$, $S, t_1 \models F$ iff $S, \sigma_i \models \phi$, where σ_i is the first element of σ such that $\sigma_i > t_1$ (note that by hypothesis it is $st(\sigma_i)$). If σ_j is the smallest element of σ s.t. $\sigma_j > t_2$ (also, it is $st(\sigma_j)$), by inductive hypothesis it is $S, \sigma_i \models \phi$ iff $S, \sigma_j \models \phi$, hence iff $S, t_2 \models X_{st}(\phi)$.

If $F = Y_{st}(\phi)$, $S, t_1 \models F$ iff $S, \sigma_i \models \phi$, where σ_i is the biggest element of σ such that $\sigma_i < t_1$ (for which $st(\sigma_i)$ by hypothesis). If σ_j is the biggest element of σ s.t. $\sigma_j < t_2$ (also, it is $st(\sigma_j)$), since the number of operators Y_{st} in ϕ is $n-1$, and $\sigma_j \geqslant \sigma_i > \sigma_{n-1}$, by inductive hypothesis it is $S, \sigma_i \models \phi$ iff $S, \sigma_j \models \phi$, hence iff $S, t_2 \models Y_{st}(\phi)$. □

Proof of Theorem 4.2.1. To prove Theorem 4.2.1, assume there is an X-TRIO$^{P}_{\overline{T}}$ formula F such that $AP = \varnothing$, F contains no instances of operator $\text{Since}_{\langle a,b \rangle}$ and such that, given a structure S, it is $S, t \models F$ iff $st(t)$. If N is the number of instances of operator Y_{st} in F, given a structure S as in Lemma 4.2.2, two instants t_1, t_2 such that $\sigma_N < t_1 < t_2$, $st(t_1)$ and $ns(t_2)$, by Lemma 4.2.2 it is $S, t_1 \models F$ iff $S, t_2 \models F$, which contradicts the assumption. □

It is easy to see that X-TRIO$^{P}_{\overline{T}}$ formula $\text{NowST} = \text{Lasts}_{[)}(\neg \text{Past}(\top, \epsilon), \epsilon)$, where $\epsilon > 0$ is any infinitesimal constant, holds exactly in the time instants $t \in \overline{\mathbb{N}}_+$ such that $st(t)$. As a consequence, we have the following Corollary 4.2.3:

Corollary 4.2.3. *Over the $\overline{\mathbb{R}}_+$ temporal domain, X-TRIO$^{P}_{\overline{T}}$ with past operators is strictly more expressive than its future-only counterpart.*

If we change the temporal domain from $\overline{\mathbb{R}}_+$ to $\overline{\mathbb{R}}_\pm$, in which we admit also non-negative instants of time of the form $\nu - \epsilon$, it is not possible anymore to separate standard and non-standard instants, as the following Theorem 4.2.4 holds:

Theorem 4.2.4. *Consider $AP = p$ and an X-TRIO$^{P}_{\overline{T}}$ formula F evaluated over the temporal domain $\overline{T} = \overline{\mathbb{R}}_\pm$; no such formula can express the property "p holds in all standard instants".*

Again, let us consider, as primitive temporal operators, Futr, Past, WithinF, WithinP, etc. The proof of Theorem 4.2.4 is somewhat similar to the one of [35]; more precisely, we rely on the following intermediate lemma 4.2.5:

Lemma 4.2.5. *Consider* $AP = p$ *and an X-TRIO*$^{P}_{T}$ *formula* F*; let* δ_{F_+} *and* δ_{F_-} *be the sum of all time bounds appearing in future and past operators in* F*, respectively, and* $\delta_F = \delta_{F_+} + \delta_{F_-}$*; let* $S_i = \langle \overline{\mathbb{R}}_\pm, \beta, \sigma \rangle$ *be a structure such that, for a fixed infinitesimal* ε *and for all* $k \in \mathbb{N}$ *it is* $\sigma_k = k\varepsilon$*, and for all* t *it is* $\beta(t) = \{p\}$*, except for one instant* $i > 1 + \delta_F$*, where* $\beta(i) = \emptyset$*. Then, given two structures* S_i *and* S_j *with* $j > i > 1 + \delta_F$, $S_i, 0 \models F$ *iff* $S_j, 0 \models F$.

Proof. Structures S_i correspond to Zeno behaviours in which all σ_i are at an infinitesimal distance from the origin. Then, for all t, ϵ (with ϵ infinitesimal) such that $t \geqslant 1 - \epsilon$ and all formula ϕ it is $S_i, t \not\models X_{ns}(\phi)$, $S_j, t \not\models X_{ns}(\phi)$, and similarly for $X_{st}(\phi)$, $Y_{st}(\phi)$, $Y_{ns}(\phi)$. In fact, $X_{st}(\phi)$ is always false, and $Y_{st}(\phi)$ is false for all $t > \varepsilon$.

We show the following:

- for all t such that $t \leqslant \delta_{F_-} + 1$ it is $S_i, t \models F$ iff $S_j, t \models F$;
- for all $t \geqslant i - \delta_{F_+}$ it is $S_i, t \models F$ iff $S_j, j - i + t \models F$, i.e., starting from δ_{F_+} instants before p becomes false in each structure S_i, S_j, F has the same values in S_i and S_j if the same offset is considered;
- for all $1 + \delta_{F_-} \leqslant t < i - \delta_{F_+}$, $S_i, t \models F$ iff $S_i, 1 + \delta_{F_-} \models F$; i.e. in all instants from $1 + \delta_{F_-}$ (included) to $i - \delta_{F_+}$ (excluded) F has the same value in structure S_i.

The proof is by induction on the structure of F.

The property holds in the base case $F = p$, as S_i and S_j are the same except in instants i and j, and $j > i > 1 + \delta_{F_-} + \delta_{F_+}$. The cases in which F is $\neg\phi$ or $\phi_1 \wedge \phi_2$ are trivial.

If $F = \text{Futr}(\phi, d)$, we have $S_i, t \models F$ iff $S_i, t + d \models \phi$. If $t \geqslant i - \delta_{F_+}$, then $t + d \geqslant i - \delta_{\phi_+}$, and by induction hypothesis $S_i, t + d \models \phi$ iff $S_j, j - i + t + d \models \phi$, i.e. $S_j, j - i + t \models F$. If $t + d < 1 + \delta_{F_-}$, then $S_i, t + d \models \phi$ iff $S_j, t + d \models \phi$, hence the result. If $1 + \delta_{F_-} \leqslant t < i - \delta_{F_+}$ or $1 + \delta_{F_-} \leqslant t + d < i - \delta_{F_+}$, then $t + d < i - \delta_{\phi_+} < j - \delta_{\phi_+}$, as $t < i - \delta_{F_+}$ and $\delta_{F_+} = \delta_{\phi_+} + d$, so $S_i, t + d \models \phi$ iff $S_i, 1 + \delta_{F_-} \models \phi$, iff $S_j, 1 + \delta_{F_-} \models \phi$, iff $S_j, t + d \models \phi$, hence the result. Similarly for $\text{WithinF}_{\langle\rangle}(\phi, d)$.

The reasoning is specular if $F = \text{Past}(\phi, d)$, for which $S_i, t \models F$ iff $t - d \in \overline{\mathbb{R}}_\pm$ and $S_i, t - d \models \phi$. If $t - d \geqslant i - \delta_{F_+}$, then by induction hypothesis $S_i, t - d \models \phi$

iff $S_j, j-i+t-d \models \phi$, i.e. $S_j, j-i+t \models F$. If $t < 1+\delta_{F_-}$, then $S_i, t-d \models \phi$ iff $S_j, t-d \models \phi$, hence the result. If $1+\delta_{F_-} \leqslant t-d < i-\delta_{F_+}$ or $1+\delta_{F_-} \leqslant t < i-\delta_{F_+}$, then $1+\delta_{\phi_-} \leqslant t-d$, as $1+\delta_{F_-} \leqslant t$ and $\delta_{F_-} = \delta_{\phi_-}+d$, so $S_i, t-d \models \phi$ iff $S_i, 1+\delta_{\phi_-} \models \phi$, iff $S_j, 1+\delta_{\phi_-} \models \phi$, iff $S_j, t-d \models \phi$, hence the result. Similarly for $\text{WithinP}_{\langle\rangle}(\phi, d)$.

If $F = X_{ns}(\phi)$, $S_i, t \not\models F$ and $S_j, t \not\models F$ if $t > 1-\epsilon$ for some infinitesimal ϵ. If $t = \epsilon$ for some infinitesimal ϵ, then $S_i, t \models F$ and $S_j, t \models F$ iff $S_i, \sigma_{j+1} \models \phi$, with $\sigma_j \leqslant t < \sigma_{j+1}$. Similarly for $X_{st}(\phi)$, $Y_{st}(\phi)$ and $Y_{ns}(\phi)$.

If $F = \text{Until}(\phi, \psi)$, then $S_i, t \models F$ if there is $t' > t$ such that $S_i, t' \models \psi$. We separate several cases. If $t \geqslant i-\delta_{F_+}$, then by induction hypothesis $S_i, t' \models \psi$ iff $S_j, j-i+t' \models \psi$, and also by induction hypothesis we have that for all $t \leqslant t'' < t'$ $S_i, t'' \models \phi$ iff for all $j-i+t \leqslant t'' < j-i+t'$ $S_j, t'' \models \phi$, hence the result. Similarly if $t < t' < 1+\delta_{F_-}$. If $t < i-\delta_{F_+} \leqslant t'$, then by induction hypothesis $S_i, t' \models \psi$ iff $S_j, j-i+t' \models \psi$, $S_i, t \models \phi$ iff $S_j, t \models \psi$, and for all $t \leqslant t'' < t'$ it is $S_i, t'' \models \phi$ iff for all $t \leqslant t'' < j-i+t'$ it $S_j, t'' \models \phi$, hence the result. The other cases are similar, and are not detailed here for brevity. The case $F = \text{Since}(\phi, \psi)$ is dual.

$S_i, 0 \models F$ iff $S_j, 0 \models F$ simply descends by observing that $0 < 1+\delta_{\phi_-}$. □

Proof of Theorem 4.2.4. Suppose there is an X-TRIO$^{P}_{T}$ formula F that captures property "p holds in all standard instants". Consider two instants i, j such that $st(i)$, $ns(j)$ and $j > i > 1+\delta_F$. Then, by lemma 4.2.5, I have that $S_i, 0 \models F$ iff $S_j, 0 \models F$, but F does not hold for S_i, so it does not hold for S_j, either, though by definition it should. □

4.2.3 *Decidability of X-TRIO$^{P}_{T}$*

As we expect, we have the following decidability result for the X-TRIO$^{P}_{T}$ logic:

Theorem 4.2.6. *The satisfiability problem of X-TRIO$^{P}_{T}$ formulae over the temporal domain* $nsDomT = \overline{\mathbb{R}}_+$ *is undecidable.*

Proof. To show the undecidability of X-TRIO$^{P}_{T}$, we reduce the satisfiability problem of MTL, which is well-known to be undecidable [36], to that of X-TRIO$^{P}_{T}$. Given an MTL formula F, it is possible to transform it in the X-TRIO$^{P}_{T}$ formula F' in which every subformula ϕ of F becomes $\text{NowST} \rightarrow \phi$. Then, given two structures S_1 and S_2 such that for all t s.t. $st(t)$ it is $\beta_1(t) = \beta_2(t)$, $S_1, 0 \models F'$ iff $S_2, 0 \models F'$, as $\text{NowST} \rightarrow \phi$ is true in all non-standard instants of time (where the antecedent is false). Then, a structure S is a model of F' iff its

restriction to the standard instants of time is a model of F, so any procedure to decide the satisfiability of F' would also decide the satisfiability of F, which is impossible. □

To achieve decidability, many different choices are possible. For example, in the standard MTL case, one way to make the logic decidable is to limit the kinds of intervals that can be written in the metric Until modality [33]. Decidability is also often obtained by considering a discrete temporal domain. In the next section 4.3 this second path is explored, without renouncing infinitesimals, however.

4.3 A DECIDABLE FRAGMENT OF X-TRIO

In this section, we focus the attention on discrete subsets of the hyperreals $^*\mathbb{R}$. In particular, we consider the temporal domain $\overline{\mathbb{N}}_+ = \{x \in {}^*\mathbb{R} | \exists \nu, k \in \mathbb{N} \text{ s.t. } x = \nu + k\varepsilon\}$, with $\varepsilon > 0$ a fixed infinitesimal. Note that $\overline{\mathbb{N}}_+$ is not the set of the hypernaturals [30], which does not include infinitesimal numbers, but, rather, a particular subset of the hyperreals.

For temporal domain $\overline{\mathbb{N}}_+$ it is easy to see that Theorem 4.2.1 and Corollary 4.2.3 still hold. In this case, however, the definition of NowST can be simplified in the following way: $\mathrm{NowST} = \neg\mathrm{Past}(\top, \varepsilon)$.

In X-TRIO$^{\mathrm{P}}_{\mathrm{T}}$, we consider only distances for the bounds a, b of the metric Until and Since operators that have the form $\nu \pm k\varepsilon$. On domain $\overline{\mathbb{N}}_+$, in addition to equivalences 25, we have a further set of results that allow us to consider only a subset of the logic X-TRIO$^{\mathrm{P}}_{\mathrm{T}}$ (which includes, for example, an unbounded number of forms for operator Futr, one for each possible bound), without loss of generality. More precisely, one can show that through operators $\mathrm{Futr}(\bullet, 1)$, $\mathrm{Futr}(\bullet, \varepsilon)$ and Until (and their past counterparts) it is possible to express all other temporal operators. Here we present some of the more interesting ones (note that $\mathrm{Until}_{[}(\phi, \psi) = \psi \vee \mathrm{Until}(\phi, \psi)$).

First of all, it is easy to show that $\mathrm{Futr}(\bullet, \nu \pm k\varepsilon)$ can be expressed in terms of $\mathrm{Futr}(\bullet, 1)$ and $\mathrm{Futr}(\bullet, \varepsilon)$, as in the following (where $k \geqslant 1$, $st(\nu)$ and $\nu \geqslant 1$):

$$
\begin{aligned}
\mathrm{Futr}(\phi, \nu + k\varepsilon) &\equiv \mathrm{Futr}(\mathrm{Futr}(\phi, k\varepsilon), \nu) \\
\mathrm{Futr}(\phi, \nu - k\varepsilon) &\equiv \mathrm{Futr}(\mathrm{Past}(\phi, k\varepsilon), \nu) \\
\mathrm{Futr}(\phi, \nu) &\equiv \mathrm{Futr}(\mathrm{Futr}(\phi, \nu - 1), 1) \\
\mathrm{Futr}(\phi, k\varepsilon) &\equiv \mathrm{Futr}(\mathrm{Futr}(\phi, (k-1)\varepsilon), \varepsilon)
\end{aligned}
\tag{26}
$$

Similar equivalences hold for the Past operator; note that in this case, when the bound is $\nu - k\varepsilon$, the equivalence is the following:

$$\text{Past}(\phi, \nu - k\varepsilon) \equiv \text{Futr}(\text{Past}(\phi, \nu), k\varepsilon) \tag{27}$$

In fact, if $\text{Past}(\phi, \nu - k\varepsilon)$ is evaluated in an instant t, it might be that $t - (\nu - k\varepsilon) \in \overline{\mathbb{N}}_+$, but $t - \nu \notin \overline{\mathbb{N}}_+$. Hence, in (26) it is Futr(Past, ...) instead of Past(Futr, ...), as the latter form might be false, no matter the value of ϕ, even if $t - \nu + k\varepsilon \in \overline{\mathbb{N}}_+$.

The equivalences for operator WithinF must take into account the peculiarities of the underlying domain. In a standard, discrete domain such as $\mathbb{N}$, $\text{WithinF}_{[]}(\phi, 1)$, for example, would simply be $\phi \vee \text{Futr}(\phi, 1)$. However, in domain $\overline{\mathbb{N}}_+$, between 0 and 1 there is an infinity of non-standard numbers of the form $k\varepsilon$, so $\text{WithinF}_{[]}(\phi, 1)$ actually reads "ϕ holds either in the current instant, or one instant from now, or in one of the non-standard instants between now and one instant from now". On the other hand, $\text{WithinF}_{[]}(\phi, 2\varepsilon)$ is still equivalent to $\phi \vee \text{Futr}(\phi, \varepsilon) \vee \text{Futr}(\phi, 2\varepsilon)$. Then, the following equivalences hold (with $st(\nu)$, $\nu \geqslant 1$ and $k \geqslant 0$).

$$\text{WithinF}_{[]}(\phi, k\varepsilon) \equiv \phi \vee \text{Futr}\Big(\text{WithinF}_{[]}(\phi, (k-1)\varepsilon), \varepsilon\Big) \tag{28}$$

$$\text{WithinF}_{[]}(\phi, \nu \pm k\varepsilon) \equiv \text{WithinF}_{[]}(\phi, 1) \vee \text{Futr}\Big(\text{WithinF}_{[]}(\phi, \nu - 1 \pm k\varepsilon), 1\Big) \tag{29}$$

$$\text{WithinF}_{[]}(\phi, 1 - k\varepsilon) \equiv \phi \vee \text{Futr}\Big(\text{Until}_{[}(\neg\text{NowST}, \phi \wedge \neg\text{NowST}), \varepsilon\Big) \vee \text{Futr}(\phi \vee \neg\text{Since}(\neg\phi, \text{NowST} \wedge \neg\phi), 1 - k\varepsilon) \tag{30}$$

Let us focus on equivalence (30). $S, t \models \text{WithinF}_{[]}(\phi, 1 - k\varepsilon)$ if $S, t \models \phi$, or if $S, t' \models \phi$ for any of the infinite instants of time in $[t + \varepsilon, t + 1 - k\varepsilon]$. Recall that $t \in \overline{\mathbb{N}}_+$ has the form $\nu + n\varepsilon$. Then, $[t + \varepsilon, t + 1 - k\varepsilon] = [t + \varepsilon, \nu + 1) \cup [\nu + 1 t + 1 - k\varepsilon]$. If ϕ holds in $[t + \varepsilon, \nu + 1)$, then there is an instant $t' > t$ such that $ns(t')$, $S, t' \models \phi$, and for all $t'' \in [t, t')$ it is $ns(t'')$. This corresponds to the second disjunct in equivalence (30). If ϕ holds in $[\nu + 1, t + 1 - k\varepsilon]$, then it must be $t + 1 - k\varepsilon \geqslant \nu + 1$, and either $S, t + 1 - k\varepsilon \models \phi$ (notice that it could be $t + 1 - k\varepsilon = \nu + 1$), or it is not possible that between $t + 1 - k\varepsilon$ and the preceding standard instant (i.e., ν), included, ϕ never holds. This is captured by the third disjunct in (30).

Similar equivalences hold for operator $\text{WithinP}_{[]}$.

Finally, all other forms of operators WithinF and WithinP ($\mathrm{WithinF}_{()}$, $\mathrm{WithinF}_{(]}$, etc.) can be expressed in terms of $\mathrm{WithinF}_{[]}$. We consider, for example, the case of operators $\mathrm{WithinF}_{()}$ and $\mathrm{WithinP}_{()}$. The following equivalences are trivial (with $d \geqslant 2\varepsilon$ and $k \geqslant 2$).

$$\mathrm{WithinF}_{()}(\phi, d) \equiv \mathrm{Futr}\Big(\mathrm{WithinF}_{[]}(\phi, d-2\varepsilon), \varepsilon\Big) \quad (31)$$

$$\mathrm{WithinP}_{()}(\phi, k\varepsilon) \equiv \mathrm{Past}\Big(\mathrm{WithinP}_{[]}(\phi, (k-2)\varepsilon), \varepsilon\Big) \quad (32)$$

The case for $\mathrm{WithinP}_{()}(\phi, 1-k\varepsilon)$, which is captured by equivalence (33), is a bit more involved, since it must separate the case in which the current instant is a standard one, from the one in which it is not.

$$\begin{aligned}\mathrm{WithinP}_{()}(\phi, 1-k\varepsilon) \equiv\ & \Big(\neg\mathrm{NowST} \wedge \mathrm{Past}\Big(\mathrm{WithinP}_{[]}(\phi, 1-(k+2)\varepsilon), \varepsilon\Big)\Big) \\ & \wedge(\mathrm{NowST}\wedge \\ & \mathrm{Past}(\mathrm{WithinF}_{[]}(\phi, 1-(k+2)\varepsilon), 1-(k+1)\varepsilon))\end{aligned} \quad (33)$$

In fact, if $ns(t)$, then $t - \varepsilon \in \overline{\mathbb{N}}_{+}$, so $\mathrm{WithinP}_{()}(\phi, 1-k\varepsilon)$ is simply $\mathrm{WithinP}_{[]}(\phi, 1-k\varepsilon-2\varepsilon)$ evaluated in $t-\varepsilon$. If, instead, $st(t)$, then $\mathrm{WithinP}_{()}(\phi, 1-k\varepsilon)$ is false if $t = 0$ (and in this case $1-k\varepsilon-\varepsilon \notin \overline{\mathbb{N}}_{+}$); if $t \geqslant 1$, then $1-k\varepsilon-\varepsilon \in \overline{\mathbb{N}}_{+}$, and $\mathrm{WithinP}_{()}(\phi, 1-k\varepsilon)$ iff in $1-k\varepsilon-\varepsilon$ it is $\mathrm{WithinF}_{[]}(\phi, 1-k\varepsilon-2\varepsilon)$.

The case $\mathrm{WithinP}_{()}(\phi, \nu \pm k\varepsilon)$, with $st(\nu)$ and $\nu \geqslant 1$ is not shown here for brevity.

Equivalences (25) and (26)-(33) suggest the following simplified, though equivalent for domain $\overline{\mathbb{N}}_{+}$, syntax for X-TRIO:

$$\begin{aligned}\phi := \ & p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathrm{Futr}(\phi, 1) \mid \mathrm{Past}(\phi, 1) \mid \mathrm{Futr}(\phi, \varepsilon) \mid \mathrm{Past}(\phi, \varepsilon) \mid \\ & \mathrm{Until}(\phi_1, \phi_2) \mid \mathrm{Since}(\phi_1, \phi_2) \mid X_{st}(\phi) \mid X_{ns}(\phi) \mid Y_{st}(\phi) \mid Y_{ns}(\phi)\end{aligned} \quad (34)$$

We denoted the new propositional fragment of X-TRIO, over the discrete domain $\overline{\mathbb{N}}_{+}$, with $\text{X-TRIO}^{P}_{\overline{\mathbb{N}}_{+}}$.

Despite the restrictions introduced in $\text{X-TRIO}^{P}_{\overline{\mathbb{N}}_{+}}$, however, it is undecidable, and the following Theorem 4.3.1 holds:

Theorem 4.3.1. *The satisfiability problem of the* $\text{X-TRIO}^{P}_{\overline{\mathbb{N}}_{+}}$ *logic is undecidable.*

Proof sketch. In classic fashion (see, e.g., [37]), we reduce the halting problem of a 2-counter machine to the satisfiability problem of X-TRIO$^{P}_{\mathbb{N}_+}$ formulae, by defining a set of X-TRIO$^{P}_{\mathbb{N}_+}$ formulae that formalize the increment and decrement of the 2 counters.

We associate one counter with the sequence of even standard numbers, and one with the sequence of odd standard numbers, as detailed below.

We associate two propositional letters, E and O, with each standard instant of σ so that when the current standard instant is even (resp., odd) then only E (resp., O) holds. They do not hold in non-standard instants. These constraints are represented by the following X-TRIO$^{P}_{\mathbb{N}_+}$ formulae (we show the case of even instants):

$$\begin{aligned} E \rightarrow\ & X_{ns}\Big(\mathrm{Until}_{[}(\neg O \wedge \neg E, X_{st}(O) \wedge \neg O \wedge \neg E)\Big) \\ E \leftrightarrow\ & \mathrm{Futr}(O, 1) \end{aligned}$$

Given two consecutive standard instants σ_j and σ_i in σ (i.e., such that $\sigma_i = \sigma_j + 1$), there is a finite nonempty sequence $\sigma_{[j,i)}$ of length $i - (j+1)$ of non-standard instants in σ between them since σ is discrete.

We introduce X-TRIO$^{P}_{\mathbb{N}_+}$ formulae (not shown here for brevity) to partition $\sigma_{[j,i)}$ into two subsequences such that, at each instant, either propositional letter A or propositional letter B) holds. We use letters A and B to "mark" each instant in $\sigma_{[j,i)}$ as shown in Figure 14. The sequence of B's ends in the last non-standard instant of $\sigma_{[j,i)}$.

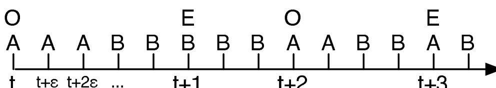

Figure 14: Part of trace representing counters.

The sequence of A and B are used to represent the two counters: the numbers of A's starting from standard numbers marked with E (resp. O) represent the first (resp. second) counter. We encode the operations *increase/decrease/check if 0*, by manipulating the length of the sequence of A's. For example, the counter associated with E increases its current value if the sequence of A's that starts at the next even standard instant is such that the last A of that sequence dists $2+\varepsilon$ from the last A of the current sequence of A's. This is encoded through the following X-TRIO$^{P}_{\mathbb{N}_+}$ formula, reminding that all the naturals numbers belong to the temporal domain:

$$E \to \quad (A \to \mathrm{Until}(A, B \land \mathrm{Futr}(A \land X_{ns}(B), 2)))\land$$
$$(B \to \mathrm{Futr}(A \land X_{ns}(B), 2))$$

The other cases are omitted for brevity.

The counter is zero when the sequence of A's is empty. In the case of the counter associated with even standard numbers we can encode this check with the formula $E \land B$.

Finally, at the initial instant of the sequence σ, which is an even number, E holds and the corresponding counter value is 0 (i.e., $E \land B$ holds at 0).

The halting of the formalized machine is expressed simply as reachability of a final state. Hence, we conclude that the satisfiability problem of X-TRIO$^{P}_{\mathbb{N}_+}$ is undecidable. □

In section 4.4, we introduce a sufficient condition that makes X-TRIO$^{P}_{\mathbb{N}_+}$ decidable, but still expressive enough for our purposes.

4.3.1 *A decision procedure for X-TRIO$^{P\text{-}R}_{\mathbb{N}_+}$*

In this section, we show the decidability of a fragment of X-TRIO$^{P}_{\mathbb{N}_+}$ by reducing its satisfiability problem to that of PLTLB (LTL with both future and past operators). The transformation is effective and has been implemented in the $\mathbb{Z}$ot satisfiability checker.

PLTLB extends classic LTL [38] with past operators; its syntax (as used in the rest of this thesis) is the following:

$$\phi := p | \neg\phi \,|\, \phi_1 \land \phi_2 \,|\, X_L(\phi) \,|\, Y_L(\phi) \,|\, \phi_1 \, U_L \, \phi_2 \,|\, \phi_1 \, S_L \, \phi_2 \tag{35}$$

We have also the usual abbreviations $F_L(\phi) = \top \, U_L \, \phi$ and $G_L(\phi) = \neg F_L(\neg\phi)$.

The semantics of PLTLB is defined over discrete *traces*. A trace is an infinite word $\pi = \pi(0)\pi(1)\ldots$ over the finite alphabet $\Sigma = 2^{AP}$, where each $\pi(i)$ represents the set of atomic propositions that are true in i.

$\models_L$ denotes the satisfiability relation of PLTLB. The definition of $\models_L$ is straightforward if one considers that, for any ϕ, $Y_L(\phi)$ is false at 0 [38].

As a first step, to encode X-TRIO$^{P}_{\mathbb{N}_+}$ into PLTLB, we restrict histories σ according to the following constraints:

1. Histories σ are infinite.

2. Either all standard natural numbers or a bounded interval thereof including 0 belong to σ.

3. If σ_{i+1} is non-standard ($ns(\sigma_{i+1})$), then $\sigma_{i+1} - \sigma_i = \varepsilon$.

Constraints 1, 2 and 3 are not strictly necessary to obtain decidability, but they are not overly restrictive and they reduce the number of cases to be considered in the encoding. For example, a finite history σ must be such that, after the last element of the sequence, the state does not change, which can be also represented as an infinite history in which, from a certain point on, all $\beta(\sigma_i)$ are the same.

Notice also that, if σ_{i+1} is standard ($st(\sigma_{i+1})$), then between σ_i and σ_{i+1} there is an infinite sequence of non-standard numbers $\sigma_i + \varepsilon, \sigma_i + 2\varepsilon, \ldots$ such that, for all $k \in \mathbb{N}$, $\beta(\sigma_i + k\varepsilon) = \beta(\sigma_i)$.

In order to reduce the satisfiability problem of X-TRIO$^{P}_{\mathbb{N}_+}$ (which is in general undecidable) to that of PLTLB (which is decidable), we need to apply further restrictions. The key to encoding a counting mechanism in X-TRIO$^{P}_{\mathbb{N}_+}$ is to evaluate formulae of the form $\text{Futr}(\phi, \nu + k\varepsilon)$ (with $\nu \geqslant 1$) in non-standard instants only. Then, every occurrence of $\text{Futr}(\phi, \nu + k\varepsilon)$ in an X-TRIO$^{P}_{\mathbb{N}_+}$ formula ϕ, with $\nu \geqslant 1$, will be intended as an abbreviation for the following formula:

$$\text{Futr}(\phi, \nu + k\varepsilon) \wedge \text{NowST} \tag{36}$$

Hence, the value of $\text{Futr}(\phi, \nu + k\varepsilon)$ in non-standard instants does not affect the value of the formula if $\nu \geqslant 1$. Similar considerations hold for the Past operator.

In addition, we introduce a further restriction that allows us to obtain a quite simple decision procedure for the logic, with a rather limited cost in expressiveness, as discussed below. The restriction consists of imposing that the value of formulae is meaningful only in instants that are "covered" by the history σ. In fact, as shown in Section 4.1.3, an infinite history σ could exhibit a Zeno behaviour, where there are instants $t \in \overline{\mathbb{T}}$ such that, for all i, $\sigma_i < t$. Then, by convention, we state that formulae that are evaluated after one such accumulation point are false. This can be achieved by considering every subformula ψ of an X-TRIO$^{P}_{\mathbb{N}_+}$ formula ϕ as an abbreviation for:

$$\psi \wedge \text{SomF}(X_{st}(\top) \vee X_{ns}(\top)) \tag{37}$$

We denote the logic X-TRIO$^{P}_{\overline{\mathbb{N}}_+}$ with the restrictions described in points 1, 2, 3 and the substitutions (36) and (37), with X-TRIO$^{P\text{-}R}_{\overline{\mathbb{N}}_+}$, where the letter R means $\mathsf{Restricted}$.

We have the following result:

Lemma 4.3.2. *Given an X-TRIO$^{P\text{-}R}_{\overline{\mathbb{N}}_+}$ formula ϕ, and given two structures $S_1 = \langle\overline{\mathbb{N}}_+, \beta_1, \sigma\rangle$, $S_2 = \langle\overline{\mathbb{N}}_+, \beta_2, \sigma\rangle$ (i.e., which have the same history σ) such that, for all $t \in \overline{\mathbb{N}}_+$ for which there is $i \in \mathbb{N}$ such that $t < \sigma_i$, it is $\beta_1(t) = \beta_2(t)$, then $S_1, 0 \models \phi$ iff $S_2, 0 \models \phi$.*

Proof. We show a stronger result, from which Lemma 4.3.2 descends as corollary. More precisely, we show that, given any $t \in \overline{\mathbb{N}}_+$, $S_1, t \models \phi$ iff $S_2, t \models \phi$. First of all, we remark that, if for each $t \in \overline{\mathbb{N}}_+$ there is a σ_i such that $t < \sigma_i$, then for all $t \in \overline{\mathbb{N}}_+$ it is $\beta_1(t) = \beta_2(t)$, hence the desired result. In addition, notice that, in this case, condition SomF($X_{st}(\top) \vee X_{ns}(\top)$) is true for all $t \in \overline{\mathbb{N}}_+$, so the value of ϕ does not depend on it.

In the rest of the proof, we consider the case in which there are instants t such that, for all i, $\sigma_i < t$. The set of such instants can be shown to have a minimum, which we indicate with $\bar{t}$, such that $st(\bar{t})$. Then, history σ accumulates at $\bar{t}$, and we separate two cases: $t < \bar{t}$ and $t \geqslant \bar{t}$. In the case $t \geqslant \bar{t}$, SomF($X_{st}(\top) \vee X_{ns}(\top)$) is false, hence for all ϕ both $S_1, t \not\models \phi$ and $S_2, t \not\models \phi$. Then, we only need to consider the case $t < \bar{t}$. The rest of the proof is by induction on the structure of ϕ: consider a subformula ψ of ϕ.

If $\psi = p$, by hypothesis $\beta_1(t) = \beta_2(t)$; hence the result. If $\psi = \mathrm{Futr}(\zeta, 1)$, then $S_1, t \models \psi$ iff $S_1, t+1 \models \zeta$, hence, by inductive hypothesis, iff $S_2, t+1 \models \zeta$, and iff $S_2, t \models \psi$.

The cases $\psi = \neg\zeta$ and $\psi = \psi_1 \wedge \psi_2$ are trivial.

Similarly for $\mathrm{Past}(\zeta, 1)$, $\mathrm{Futr}(\zeta, \varepsilon)$, and $\mathrm{Past}(\zeta, \varepsilon)$.

If $\psi = \mathrm{Until}(\psi_1, \psi_2)$, $S_1, t \models \psi$ iff there is $t' > t$ such that $S_1, t' \models \psi_2$, and for all $t \leqslant t'' < t'$ it is $S_1, t \models \psi_1$; by inductive hypothesis this occurs iff $S_2, t' \models \psi_2$, and for all $t \leqslant t'' < t'$ it is $S_2, t \models \psi_1$, i.e., iff $S_2, t \models \psi$. The case $\mathrm{Since}(\psi_1, \psi_2)$ is similar.

If $\psi = X_{st}(\zeta)$, then $S_1, t \models \psi$ iff there is $i \in \mathbb{N}$ such that $st(\sigma_{i+1})$, $\sigma_i < t \leqslant \sigma_{i+1}$ and $S_1, \sigma_{i+1} \models \zeta$; by inductive hypothesis this holds iff $S_2, \sigma_{i+1} \models \zeta$, hence the result. Similarly for $X_{ns}(\zeta)$, $Y_{st}(\zeta)$ and $Y_{ns}(\zeta)$.

□

As a consequence of Lemma 4.3.2, and also of the next lemma 4.3.3, in order to determine whether an X-TRIO$^{P\text{-}R}_{\overline{\mathbb{N}}_+}$ formula is satisfiable we only need to

focus on the sequence σ, disregarding the instants following an accumulation point, if any.

In order to introduce the PLTLB-based decision procedure for X-TRIO$^{\text{P-R}}_{\overline{\mathbb{N}}_+}$, we need a further intermediate result. We show that, in each interval (σ_i, σ_{i+1}) such that $st(\sigma_{i+1})$, there is an instant $\bar{t}$ such that the subformulae of ϕ have the same value in all $t \in [\bar{t}, \sigma_{i+1})$. In addition, $\bar{t}$ is a number of non-standard instants that is equal to the maximum nesting depth δ of Past$(\bullet, \varepsilon)$ operators far from σ_i .

More precisely, given a formula ϕ, the nesting δ_ϕ of Past$(\bullet, \varepsilon)$ operators is defined as follows: $\delta_p = 0$; $\delta_{\psi_1 \wedge \psi_2} = \max(\delta_{\psi_1}, \delta_{\psi_2})$, and similarly for the Until and Since operators; $\delta_{\neg\psi} = \delta_\psi$ (similarly for Futr$(\psi, \pm 1)$ and for Futr(ψ, ε)); finally, $\delta_{\text{Past}(\psi,\varepsilon)} = 1 + \delta_\psi$. Then, we have the following result:

Lemma 4.3.3. *Given an X-TRIO$^{P\text{-}R}_{\overline{\mathbb{N}}_+}$ formula ϕ and a structure $S = \langle \overline{\mathbb{N}}_+, \beta, \sigma \rangle$, if $st(\sigma_{i+1})$, then for any two instants $j, k \in \overline{\mathbb{N}}_+$ such that $\sigma_i + \delta_\phi \varepsilon < j < k < \sigma_{i+1}$, $S, j \models \phi$ iff $S, k \models \phi$.*

Proof. First of all, we notice that it must be $ns(j)$ and $ns(k)$, as, by constraint 2, $\sigma_i \geqslant \sigma_{i+1} - 1$. Then, the proof proceeds by induction on the structure of ϕ.

If $\phi = p \in AP$, then $p \in \beta(j)$ iff $p \in \beta(k)$, as $\beta(j) = \beta(k)$ by definition of σ, hence the result.

If $\phi = \neg\psi$, $S, j \models \phi$ iff $S, j \not\models \psi$, which holds iff $S, k \not\models \psi$ by inductive hypothesis, as $\delta_\phi = \delta_\psi$, i.e., iff $S, k \models \phi$. The case for Futr(ψ, ε) is similar.

If $\phi = \psi_1 \wedge \psi_2$, $S, j \models \phi$ iff both $S, j \models \psi_1$ and $S, j \models \psi_2$, which, by inductive hypothesis, holds iff both $S, k \models \psi_1$ and $S, k \models \psi_2$, as $\delta_\phi \geqslant \delta_{\psi_1}$ and $\delta_\phi \geqslant \delta_{\psi_2}$.

If $\phi = \text{Futr}(\psi, 1)$, then both $S, j \not\models \phi$ and $S, k \not\models \phi$, as Futr$(\psi, 1)$ is by convention false in non-standard instants. Similarly when $\phi = \text{Past}(\psi, 1)$.

If $\phi = \text{Until}(\psi_1, \psi_2)$, we have that $S, k \models \phi$ iff there is a $t > k$ s.t. $S, t \models \psi_2$, and for all $k \leqslant t' < t$ it is $S, t' \models \psi_1$. By inductive hypothesis, since $\delta_\phi \geqslant \delta_{\psi_1}$, for all t', t'' s.t. $j \leqslant t'' < k \leqslant t' < \sigma_{i+1}$, we have that $S, t' \models \psi_1$ iff $S, t'' \models \psi_1$. Hence, $S, t' \models \psi_1$ holds for all $k \leqslant t' < t$ iff also for all $j \leqslant t'' < t$ it is $S, t'' \models \psi_1$. Then, $S, k \models \phi$ iff $S, j \models \phi$. The case $\phi = \text{Since}(\psi_1, \psi_2)$ is similar.

If $\phi = X_{st}(\psi)$, $S, j \models \phi$ iff $S, \sigma_{i+1} \models \psi$, as $st(\sigma_{i+1})$. I have also $S, k \models \phi$ iff $S, \sigma_{i+1} \models \psi$, hence the result.

If $\phi = X_{ns}(\psi)$, both $S, j \not\models \phi$ and $S, k \not\models \phi$, as $st(\sigma_{i+1})$.

If $\phi = Y_{st}(\psi)$, both $S, j \models \phi$ and $S, k \models \phi$ hold iff $st(\sigma_i)$ and $S, \sigma_i \models \psi$. Similarly for the case $\phi = Y_{ns}(\psi)$.

If $\phi = \text{Past}(\psi, \varepsilon)$, $S, j \models \phi$ iff $S, j - \varepsilon \models \psi$; since $j - \varepsilon > \sigma_i + (\delta_\phi - 1)\varepsilon$ and $\delta_\phi = \delta_\psi + 1$, then $k - \varepsilon > j - \varepsilon > \sigma_i + \delta_\psi \varepsilon$ hence, by inductive hypothesis, we have $S, k - \varepsilon \models \psi$, i.e., $S, k \models \phi$. □

Notice that Lemma 4.3.3 does not hold if $j = \sigma_i + \delta_\phi \varepsilon$. For example, if $j = \sigma_0 = 0$, $\sigma_1 = 1$, and $p \in \beta(\sigma_0)$, with $\delta_\phi = 0$, then $Y_{st}(p)$ holds in $k = \sigma_0 + \varepsilon$, but not in j.

The basic idea of the encoding is, given an X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ formula ϕ, to build a corresponding PLTLB formula $\tau(\phi)$ such that each model $S = \langle \overline{\mathbb{N}}_+, \beta, \sigma \rangle$ of ϕ corresponds to a trace π that is a model of $\tau(\phi)$, where each $t \in \overline{\mathbb{N}}_+$ such that there is $\sigma_i > t$ is mapped onto an element $\pi(\rho_S(t))$, and $\beta(t) = \pi(\rho_S(t))$, where $\rho_S : \overline{\mathbb{N}}_+ \mapsto \mathbb{N}$ is monotonic. Then, we represent the transition $\sigma_i \longmapsto \sigma_{i+1}$ through the PLTLB operator X_L. Constraints 2 and 3 guarantee that the difference between σ_{i+1} and $\sigma_i = \nu + k\varepsilon$ is either $1 - k\varepsilon$ or ε, depending on whether σ_{i+1} is standard or not.

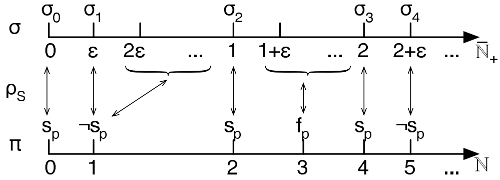

Figure 15: An example of ρ_S (with $\delta_\phi = 1$).

The encoding "flattens" the history σ over π: it represents each σ_i through an element of π. Then, to separate the elements of π that represent standard instants from those that represent non-standard ones, we introduce a PLTLB propositional letter s_p such that $s_p \in \pi(i)$ iff i in π corresponds to a standard number. In addition, we need to introduce "filling" elements in π to represent the (infinite) non-standard instants between σ_i and σ_{i+1} when $st(\sigma_{i+1})$. Lemma 4.3.3 suggests that the required number of these elements is finite, equal to $\delta_\phi + 1$; in fact, all non-standard instants such that $\sigma_i + \delta_\phi < t < \sigma_{i+1}$ are equivalent from the point of view of the truth of subformulae, hence they can be "condensed" in one single element. We mark the first δ_ϕ elements of π following the one corresponding to σ_i with proposition e_p, and the element corresponding to all instants $\sigma_i + \delta_\phi < t < \sigma_{i+1}$ with f_p. Figure 15 depicts an example of history σ, and its corresponding trace π.

Then, trace π must obey the following PLTLB constraint, where ns_p is an abbreviation for $\neg s_p \wedge \neg f_p \wedge \neg e_p$:

$$\begin{array}{l} s_p \wedge \\ G_L((e_p \rightarrow \neg f_p \wedge \neg s_p) \wedge (f_p \rightarrow \neg s_p) \wedge \\ \quad (s_p \rightarrow \\ \qquad X_L \left(ns_p \, U_L \left(\begin{array}{l} \bigwedge_{k=0}^{\delta_\phi - 1} X_L^k(e_p) \wedge \\ X_L^{\delta_\phi}(f_p \wedge X_L(s_p)) \end{array} \right) \right) \\ \qquad \vee \\ \qquad X_L(G_L(ns_p))) \wedge \\ \quad (e_p \vee f_p \rightarrow \bigwedge_{p \in AP} p \leftrightarrow Y_L(p))) \end{array} \tag{38}$$

PLTLB formula (38) imposes, respectively, that:

1. s_p holds in $\pi(0)$
2. e_p, f_p and s_p are mutually exclusive
3. each element marked s_p is followed either by an infinity of ns_p elements or by a finite number of ns_p elements, until there is a sequence of exactly δ_ϕ e_p elements followed, in turn, by a f_p element, which is in turn followed by a s_p element
4. if e_p or $f_p \in \pi(i+1)$, then all propositions that hold in $\pi(i+1)$ also hold in $\pi(i)$

In other words, traces π have one of the following two forms:

$$s_p(ns_p^* \; e_p^{\delta_\phi} \; f_p \; s_p)^\omega$$

or

$$s_p(ns_p^* \; e_p^{\delta_\phi} \; f_p \; s_p)^*(ns_p)^\omega$$

Transformation τ of Table 4 takes an X-TRIO$^{\text{P-R}}_{\overline{\mathbb{N}}_+}$ formula ϕ and produces an equisatisfiable PLTLB formula ϕ_L (for simplicity, we translate only the operator Until$_{[}$; the translation for operator Until is very similar).

Given a structure $S = \langle \overline{\mathbb{N}}_+, \beta, \sigma \rangle$, for all $t \in \overline{\mathbb{N}}_+$ such that there is $\sigma_i > t$, we define a function $\rho_S : \overline{\mathbb{N}}_+ \mapsto \mathbb{N}$ as follows(see Figure 15 for a graphical representation):

1. $\rho_S(0) = 0$ ($\sigma_0 = 0$);

$$\tau(p) = p$$
$$\tau(\neg\phi) = \neg\tau(\phi)$$
$$\tau(\phi_1 \wedge \phi_2) = \tau(\phi_1) \wedge \tau(\phi_2)$$
$$\tau(X_{ns}(\phi)) = X_L(ns_p \wedge \tau(\phi))$$
$$\tau(X_{st}(\phi)) = X_L((e_p \vee f_p)\, U_L\, (s_p \wedge \tau(\phi)))$$
$$\tau(Y_{ns}(\phi)) = Y_L((e_p \vee f_p)\, S_L\, (ns_p \wedge \tau(\phi)))$$
$$\tau(Y_{st}(\phi)) = Y_L((e_p \vee f_p)\, S_L\, (s_p \wedge \tau(\phi)))$$
$$\tau(\mathrm{Futr}(\phi, 0)) = \tau(\mathrm{Past}(\phi, 0)) = \tau(\phi)$$
$$\tau(\mathrm{Futr}(\phi, \varepsilon)) = (\neg f_p \wedge X_L(\tau(\phi))) \vee (f_p \wedge \tau(\phi))$$
$$\tau(\mathrm{Past}(\phi, \varepsilon)) = \neg s_p \;\wedge\; ((\neg f_p \wedge Y_L(\tau(\phi))) \vee (f_p \wedge \tau(\phi)))$$
$$\tau(\mathrm{Futr}(\phi, 1)) = s_p \wedge X_L(\neg s_p\, U_L\, (s_p \wedge \tau(\phi)))$$
$$\tau(\mathrm{Past}(\phi, 1)) = s_p \wedge Y_L(\neg s_p\, S_L\, (s_p \wedge \tau(\phi)))$$
$$\tau(\mathrm{Until}_{[}(\phi, \psi)) = \tau(\phi)\, U_L\, \tau(\psi)$$
$$\tau(\mathrm{Since}_{[}(\phi, \psi)) = \tau(\phi)\, S_L\, (f_p \wedge \tau(\phi) \wedge \tau(\psi)) \;\vee\; \tau(\phi)\, S_L\, (\neg f_p \wedge \tau(\psi))$$

Table 4: Translation schema τ.

2. if $ns(t)$, and $\exists i$ s.t. $t = \sigma_i$ (hence $t = \sigma_{i-1} + \varepsilon$), then $\rho_S(t) = \rho_S(\sigma_{i-1}) + 1$
3. if $ns(t)$, and $\exists i$ s.t. $\sigma_i + \delta_\phi \varepsilon < t < \sigma_{i+1}$ (hence $st(\sigma_{i+1})$), then $\rho_S(t) = \rho_S(\sigma_i) + \delta_\phi + 1$
4. if $t > 0$ and $st(t)$ (hence there is $i > 0$ s.t. $t = \sigma_i$), then $\rho_S(t) = \rho_S(\sigma_{i-1}) + \delta_\phi + 2$
5. if $ns(t)$, and $\exists i$ s.t. $\sigma_i < t \leqslant \sigma_i + \delta_\phi \varepsilon < \sigma_{i+1}$, then there is k s.t. $t = \sigma_i + k\varepsilon$, and $\rho_S(t) = \rho_S(\sigma_i) + k$

Notice that the above rule 3 is applied to all non-standard instants in between $\sigma_i + \delta_\phi \varepsilon$ and σ_{i+1}, and it maps each of them onto a "filling" element in π. As a consequence, the above rule 4 maps a standard instant onto $\rho_S(\sigma_{i-1}) + \delta_\phi + 2$, i.e., to the element in π that follows a "filling" one. Then, if $t < t'$, we have $\rho_S(t) \leqslant \rho_S(t')$. The following Theorem 4.3.4 holds:

Theorem 4.3.4. *Given an X-TRIO$^{P\text{-}R}_{\overline{\mathbb{N}}_+}$ formula ϕ, there is a structure $S = \langle\overline{\mathbb{N}}_+, \beta, \sigma\rangle$ such that $S, 0 \models \phi$ iff there exists a trace π such that $\pi \models_L \tau(\phi) \wedge (38)$.*

Proof. Suppose we have a structure $S = \langle\overline{\mathbb{N}}_+, \beta, \sigma\rangle$. The corresponding infinite word π is built from S as follows (where AP are the atomic propositions in ϕ): for each $p \in AP$ and $t \in \overline{\mathbb{N}}_+$ such that there is $\sigma_i > t$ (hence $\rho_S(t)$ is defined), $p \in \beta(t)$ iff $p \in \pi(\rho_S(t))$. In addition, $s_p \in \pi(\rho_S(t))$ iff $st(t)$, $f_p \in \pi(\rho_S(t))$ iff there i s.t. $\sigma_i + \delta_\phi < t < \sigma_{i+1}$ and $st(\sigma_{i+1})$, and $e_p \in \rho_S(t)$ iff $\sigma_i < t \leqslant \sigma_i + \delta_\phi < \sigma_{i+1}$. It can be shown that trace π built in this way satisfies formula (38).

Dually, if π is such that $\pi \models_L \tau_e(\phi) \wedge (38)$, structure $S = \langle\overline{\mathbb{N}}_+, \beta, \sigma\rangle$ is obtained in the following way.

Given an $l \in \mathbb{N}$, if $e_p, f_p \notin \pi(l)$ (i.e., l is not a "filling" element), $\langle l_0 \dots l_\nu\rangle$ (where $\forall j \in [0, \nu)$, we have $l_j < l_{j+1}$) are all the elements of the infinite word π such that $l_\nu \leqslant l$ and $\forall j \in [0, \nu] : s_p \in \pi(l_j)$ (i.e., they are all the elements that correspond to standard instants in π preceding element l), and $k = l - l_\nu$ (i.e., k is the number of elements between $\pi(l)$ and $\pi(l_\nu)$), then $\sigma_l = \nu + k\varepsilon$. This entails that, since σ_l is a standard number iff it is of the form $\nu + 0\varepsilon$, we have that $st(\sigma_l)$ iff $s_p \in \pi(l)$. In addition, by (38), $s_p \in \pi(0)$, hence $\sigma_0 = 0$ as expected. This defines the points of the history σ of S, which in turn defines ρ_S.

β is defined as follows: for each t such that there is $\sigma_i > t$, for each $p \in AP$, $p \in \beta(t)$ iff $p \in \pi(\rho_S(t))$. If, instead, there is no $\sigma_i > t$, we can choose the value of $\beta(t)$ arbitrarily, as, by Lemma 4.3.2, it does not affect the truth of ϕ in S; then, in this case we have $p \notin \beta(t)$ for all $p \in AP$.

We prove Theorem 4.3.4 by induction on the structure of formula ϕ. First of all, by Lemma 4.3.2, if there is an accumulation point $\bar{t}$, instants $t \geqslant \bar{t}$ do not affect the satisfiability of ϕ, hence we need only analyse instants $t < \bar{t}$. Then, we show that, for all $t < \bar{t}$ (where $\bar{t} = +\infty$ if there is no accumulation point), it is $S, t \models \psi$ (where ψ is a subformula of ϕ) iff $\pi, \rho_S(t) \models_L \tau(\psi)$.

If $\psi = p$, then $S, t \models p$ iff $p \in \beta(t)$, which holds, by construction, iff $p \in \pi(\rho_S(t))$, i.e. iff $\pi, \rho_S(t) \models_L p$, and $\tau(p) = p$, hence the result. The cases $\psi = \neg\zeta$ and $\psi = \psi_1 \wedge \psi_2$ are immediate.

If $\psi = X_{ns}(\zeta)$, $S, t \models \psi$ iff $\sigma_i \leqslant t < \sigma_{i+1}$, $ns(\sigma_{i+1})$ and $S, \sigma_{i+1} \models \zeta$; by inductive hypothesis this holds iff $\pi, \rho_S(\sigma_{i+1}) \models_L \tau(\zeta)$, and by construction $s_p, f_p, e_p \notin \pi(\rho_S(\sigma_{i+1}))$, hence $\pi, \rho_S(\sigma_{i+1}) - 1 \models_L X_L(\tau(\zeta) \wedge ns_p)$, and by construction $\rho_S(t) + 1 = \rho_S(\sigma_{i+1})$, since it must be $t = \sigma_i$, because $\sigma_{i+1} = \sigma_i + \varepsilon$.

If $\psi = X_{st}(\zeta)$, $S, t \models \psi$ iff $\sigma_i \leqslant t < \sigma_{i+1}$, $st(\sigma_{i+1})$ and $S, \sigma_{i+1} \models \zeta$; by inductive hypothesis this holds iff $\pi, \rho_S(\sigma_{i+1}) \models_L \tau(\zeta)$, and by construc-

tion $s_p \in \pi(\rho_S(\sigma_{i+1}))$, hence $\pi, \rho_S(\sigma_{i+1}) \models_L \tau(\zeta) \wedge s_p$. It is either $\sigma_i < t$, or $\sigma_i = t$. If $\sigma_i < t$, $f_p \in \pi(\rho_S(t))$ or $e_p \in \pi(\rho_S(t))$ and $\rho_S(\sigma_{i+1}) \geqslant \rho_S(t) + 1$. If $\sigma_i = t$, $f_p, e_p \notin \pi(\rho_S(t))$, and $\rho_S(\sigma_{i+1}) = \rho_S(t) + \delta_\phi + 2$. In both cases for all $\rho_S(t) < l < \rho_S(\sigma_{i+1})$ it is $f_p \in \pi(\rho_S(t))$ or $e_p \in \pi(\rho_S(t))$, hence $\pi, \rho_S(t) \models_L X_L((f_p \vee e_p) \, U_L \, (s_p \wedge \tau(\zeta)))$, i.e., $\pi, \rho_S(t) \models_L \tau(\psi)$. The cases for the Y_{ns} and Y_{st} operators are similar.

Futr$(\psi, 0)$ (and Past$(\psi, 0)$) is equivalent to ψ, hence this case is trivial.

If $\psi = \text{Futr}(\zeta, \varepsilon)$, $S, t \models \psi$ iff $S, t + \varepsilon \models \zeta$, which, by inductive hypothesis, holds iff $\pi, \rho_S(t + \varepsilon) \models_L \tau(\zeta)$. We have two cases: $\sigma_i \leqslant t \leqslant \delta_\phi < \sigma_{i+1}$ and $\sigma_i + \delta_\phi < t < \sigma_{i+1}$ (hence $st(\sigma_{i+1})$). In the first case, $f_p \notin \rho_S(t)$, $\rho_S(t + \varepsilon) = \rho_S(t) + 1$, hence $\pi, \rho_S(t) \models_L \neg f_p \wedge X_L(\tau(\zeta))$. In the second case, $f_p \in \rho_S(t)$, and by Lemma 4.3.3 $S, t + \varepsilon \models \zeta$ iff $S, t \models \zeta$, which in turn holds iff $\pi, \rho_S(t) \models_L \tau(\zeta)$, hence $\pi, \rho_S(t) \models_L f_p \wedge \tau(\zeta)$. All in all, $\pi, \rho_S(t) \models_L \neg f_p \wedge X_L(\neg s_p \wedge \tau(\zeta)) \vee f_p \wedge \tau(\zeta)$, i.e., $\pi, \rho_S(t) \models_L \tau(\psi)$.

The case $\psi = \text{Past}(\zeta, \varepsilon)$ is similar to the previous one, with the addition that $S, t \models \psi$ only if $t - \varepsilon \in \overline{\mathbb{N}}_+$ (i.e., $ns(t)$) which, by inductive hypothesis, holds iff $s_p \notin \pi(\rho_S(t))$, i.e., $\pi, \rho_S(t) \models_L \neg s_p$.

If $\psi = \text{Futr}(\zeta, 1)$, $S, t \models \psi$ iff $st(t)$, $t + 1 < \bar{t}$, and $S, t + 1 \models \zeta$. It is $st(t)$ iff $s_p \in \pi(\rho_S(t))$. Also, $t + 1 < \bar{t}$ holds iff there is $\sigma_i > t + 1$, i.e., iff $\rho_S(t + 1)$ is defined, and $s_p \in \pi(\rho_S(t + 1))$. In addition, when $\rho_S(t + 1)$ is defined, $S, t + 1 \models \zeta$ iff $\pi, \rho_S(t + 1) \models_L \tau(\zeta)$, by inductive hypothesis. As $\rho_S(t + 1) > \rho_S(t)$, and there are no standard instants in between, then $\pi, \rho_S(t) \models_L s_p \wedge X_L(\neg s_p \, U_L \, (s_p \wedge \tau(\zeta)))$, i.e., $\pi, \rho_S(t) \models_L \tau(\psi)$.

The case $\psi = \text{Past}(\zeta, 1)$ is similar.

If $\psi = \text{Since}_[(\psi_1, \psi_2)$, then $S, t \models \psi$ iff there is $0 \leqslant t' \leqslant t$ s.t. $S, t' \models \psi_2$ and for all $t' < t'' \leqslant t$ it is $S, t'' \models \psi_1$. By inductive hypothesis we have $\pi, \rho_S(t') \models_L \tau(\psi_2)$. We have two cases: $\sigma_i \leqslant t' \leqslant \sigma_i + \delta_\phi < \sigma_{i+1}$ for some i, or $\sigma_i + \delta_\phi < t' < \sigma_{i+1}$. In the former case, then for all $t' < t'' \leqslant t$ it is also $\rho_S(t') < \rho_S(t'') \leqslant \rho_S(t)$, hence, by inductive hypothesis, for all $\rho_S(t') < \rho_S(t'') \leqslant \rho_S(t)$ it is $\pi, \rho_S(t'') \models_L \tau(\psi_1)$, hence $\pi, \rho_S(t) \models_L \tau(\psi_1) \, S_L \, \tau(\psi_2)$. In the latter case, instead, there are some $t' < t'' < \sigma_{i+1}$ hence, by Lemma 4.3.3, $S, t' \models \psi_1$, and also $\rho_S(t') = \rho_S(t'')$ and $f_p \in \pi(\rho_S(t'))$. Then, by inductive hypothesis, we have $\pi, \rho_S(t') \models_L \tau(\psi_2)$ and for all $\rho_S(t') \leqslant \rho_S(t'') \leqslant \rho_S(t)$ it is $\pi, \rho_S(t'') \models_L \tau(\psi_1)$. Then, $\pi, \rho_S(t) \models_L \tau(\psi_1) \, S_L \, (f_p \wedge \tau(\psi_1) \wedge \tau(\psi_2))$. Overall, $\pi, \rho_S(t) \models_L \tau(\psi_1) \, S_L \, \tau(\psi_2) \vee \tau(\psi_1) \, S_L \, (f_p \wedge \tau(\psi_1) \wedge \tau(\psi_2))$, i.e., $\pi, \rho_S(t) \models_L \tau(\psi)$. The case for the Until$_[$ operator is similar. □

Finally, from translation schema τ and Theorem 4.3.4 we can prove the following Theorem 4.3.5:

Theorem 4.3.5. *The satisfiability problem of X-TRIO$^{P\text{-}R}_{\mathbb{N}_+}$ is PSPACE-complete.*

Proof. First of all, we remark that the satisfiability problem of PLTLB is PSPACE-complete [38]. Then, to show the PSPACE-hardness of the satisfiability problem for X-TRIO$^{P}_{\mathbb{N}_+}$ I reduce the satisfiability problem of PLTLB to that of X-TRIO$^{P\text{-}R}_{\mathbb{N}_+}$. To achieve this, given a PLTLB formula ϕ_L, we can build a corresponding X-TRIO$^{P\text{-}R}_{\mathbb{N}_+}$ formula simply by applying the following transformation: $U_L \mapsto \text{Until}_{[}$, $S_L \mapsto \text{Since}_{[}$, $X_L \mapsto \text{Futr}(\bullet, 1)$, $Y_L \mapsto \text{Past}(\bullet, 1)$, and by including the constraint $\text{AlwF}(\neg X_{ns}(\top))$.

To show the PSPACE-completeness, it is enough to note that, given an X-TRIO$^{P\text{-}R}_{\mathbb{N}_+}$ formula ϕ, transformation τ produces an equisatisfiable PLTLB formula ϕ_L, whose size is polynomial in the size of ϕ. ☐

As mentioned above, Lemma 4.3.3 does not hold if we relax the condition on j and allow it to be $j = \sigma_i + \delta_\phi$. However, if we allow $\sigma_i + \delta_\phi \leqslant j < k < \sigma_{i+1}$, the only cases where the proof fails are those for Y_{ns}, Y_{st}, and $\text{Futr}(\bullet, 1)$ (when $st(\sigma_i)$ and $\delta_\phi = 0$). This suggests that, if ϕ does not include instances of operator $\text{Past}(\bullet, \varepsilon)$ (in which case $\delta_\phi = 0$, and, by constraint (38), no elements marked with e_p appear in π), nor instances of the Y_{ns} and Y_{st} operators, the encoding can be simplified. In fact, in this case we can use the following modified encoding τ_f (we only show the cases that differ from τ):

$$\begin{aligned}
\tau_f(X_{ns}(\phi)) &= X_L(\tau_f(\phi) \wedge \neg s_p) \\
\tau_f(X_{st}(\phi)) &= X_L(s_p \wedge \tau_f(\phi)) \vee (f_p \wedge X_L(\tau_f(\phi))) \\
\tau_f(\text{Futr}(\phi, \varepsilon)) &= X_L(\neg s_p \wedge \tau_f(\phi)) \vee (X_L(s_p) \wedge \tau_f(\phi)) \\
\tau_f(\text{Since}_{[}(\phi, \psi)) &= \tau_f(\phi)\, S_L\, (X_L(\neg s_p) \wedge \tau_f(\psi)) \vee \\
&\qquad \tau_f(\phi)\, S_L\, (X_L(s_p) \wedge \tau_f(\phi) \wedge \tau_f(\psi))
\end{aligned}$$

Also, constraint (38) is replaced by the following formula (4.3.1):

$$s_p \wedge G_L \left(\begin{array}{l} (s_p \rightarrow X_L(f_p \vee \neg s_p)) \wedge (f_p \rightarrow (Y_L(s_p) \wedge \\ \neg s_p \wedge X_L(s_p))) \wedge (f_p \rightarrow \bigwedge_{p \in AP} p \leftrightarrow Y_L(p)) \end{array} \right)$$

As it happens with other logic formalisms, past operators allow users to obtain more succinct and intuitive formulae, paying a little price in terms of computational complexity.

Different optimizations of the encoding are possible, depending on the shape of the X-TRIO$^{P\text{-}R}_{\mathbb{N}_+}$ formulae to be analysed, but are not reported here for brevity.

4.4 X-TRIO ENCODING OF THE CASE STUDY

In this section, we encode the semantics of Stateflow using X-TRIO$^{\text{P-R}}_{\overline{\mathbb{N}_+}}$ formulae. We first focus on single Stateflow diagrams, then deal with the issue of composing diagrams in a hierarchy. The formulae are a modification of the formulae presented in section 3.3, so only the main differences between the two encoding are described. To simplify the formalization of the Stateflow semantics, restrict the distance between two consecutive non-standard instants has been restricted to be ϵ, so each micro-step has a predefined infinitesimal fixed length. In principle, it would be possible to define different infinitesimal lengths for each basic component, but this is outside of the scope of this thesis. Then, operator X_{ns} is related to the metric operator $\text{Dist}(\phi, \epsilon)$, as formula $\text{AlwF}(X_{ns}(\phi) \to \text{Dist}(\phi, \epsilon))$ holds.

Given a Stateflow diagram representing the behaviour of a module m, for each transition $H_{m,i} : s_{m,i} \overset{g_{m,i}/a_{m,i}}{\longrightarrow} s'_{m,i}$ with source state $s_{m,i}$ and target state $s'_{m,i}$ with condition $g_{m,i}$ and action $a_{m,i}$, we introduce the following formula:

$$\text{AlwF}\Big((\gamma_{m,i} \wedge s_m = s_{m,i}) \to ((X_{ns}\Big(s_m = s'_{m,i}\Big)) \wedge \alpha_{m,i})\Big) \tag{39}$$

Formula (39) formalizes the execution of a micro-step: it asserts that if the current state of module m is $s_{m,i}$ and the transition condition $\gamma_{m,i}$ holds, then in the next micro-step the active state is $s'_{m,i}$ and the transition actions $\alpha_{m,i}$ is executed. Since we want to encode a zero-time transition with an infinitesimal one, in this formula it is sufficient to replace the operator $\bigcirc$ of the formula (2) with the operator X_{ns}.

The formula (3) remains unchanged in the new encoding, since it does not involve temporal operators:

$$\text{AlwF}\Big((\textstyle\bigwedge_{i=1}^{|H_m|} \neg(\gamma_i \wedge s_i = s_{m,i})) \to \mathsf{NOCHANGE}\Big) \tag{40}$$

The "real" time advancement of the semantics is modelled through operator X_{st} instead of an explicit predicate `tick`. We restrict the distance between two consecutive standard instants (i.e. macro-steps) in a run to be exactly 1.

The following formula expresses a necessary condition for time advancement and substitutes the formula (4):

$$\text{AlwF}\left(X_{st}(\top) \rightarrow (\bigwedge_{i=1}^{|H_m|} \neg(\gamma_{m,i} \wedge s_m = s_{m,i}))\right). \tag{41}$$

Formula (41) asserts that the next instant of time is a standard one if the system is in a stable configuration.

Finally, we introduce a formula asserting that input variables $V_{I,m}$ of module m change values only at the beginning of a macro-step, i.e. in a standard instant of time. In other words, if the next instant of time is non-standard, then the values of the input variables must be the same as those in the current instant. This formula replaces the formula (5), which models the fact that the system is sensible to external events only when it is in a stable configuration:

$$\text{AlwF}\Big(X_{ns}(\top) \rightarrow (\bigwedge_{\nu \in V_{I,m}, x \in D_I}((\nu = x) \rightarrow X_{ns}(\nu = x)))\Big) \tag{42}$$

The formula MOD_m encoding the behaviour of a single component m is given by the conjunction of formulae $\bigwedge_{i=1}^{|H_m|}(39)_i$, (40-42), plus others not shown for brevity.

To encode the semantics of module composition, we use a modular approach to hide the details of time advancement of modules to other components, exploiting the local semantics encoded by the above formulae. The method is the same described in section 3.3.2, and uses a special integrator module M. Again, we introduce two X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ predicates (and related X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ formulae) that act as the interface of the module for the purpose of coordinating time advancement.

The first predicate, $\mathtt{stable}_m$ is true when a generic component module m reaches a stable configuration, and is equivalent to the predicate with the same name given in the TRIO encoding. This is formalized by the following formula, which replaces the formula (6):

$$\text{AlwF}\Big((\bigwedge_{i=1}^{|H_m|} \neg(\gamma_i \wedge s_m = s_{m,i})) \leftrightarrow \mathtt{stable}_m\Big) \tag{43}$$

The second predicate, $\mathtt{extST}_m$, is used to coordinate different modules; it is equivalent to the predicate $\mathtt{tick}_m^{ext}$ used in the TRIO encoding.

At the level of module m, time advancement obeys the following constraint:

$$\text{AlwF}(X_{st}(\top) \leftrightarrow (\mathtt{stable}_m \wedge \mathtt{extST}_m)) \tag{44}$$

This formula does not substitute the formula (41), but this is a further condition to enforce the module m to reach a stable configuration when the $\mathtt{extST}_m$

predicate is set to true by M, i.e. when the overall system is in a global stable configuration.

Module M has its own predicates $\mathtt{stable}_m$ and $\mathtt{extST}_m$. Furthermore, we have the following constraints, in addition to (43) and (44):

$$\text{AlwF}\left(\mathtt{stable}_m \leftrightarrow (\bigwedge_{i=1}^{n_M} \mathtt{stable}_{m_i})\right) \tag{45}$$

$$\text{AlwF}\left(\bigwedge_{i=1}^{n_M} (\mathtt{extST}_M \leftrightarrow \mathtt{extST}_{m_i})\right) \tag{46}$$

Formula (45) states that module M is stable only when all its components are stable, while (46) defines that the value of $\mathtt{extST}_M$ is passed on from M to its components; n_M is the number of components which compose the module M. These formulae encode the same behaviour of the formulae (8) and (9), respectively.

Finally, we formalize the relations between inputs and outputs of components of a Simulink graph such as those of Figure 11.

A link between an output variable of a component m_1 and an input variable of a component m_2 means that the corresponding data or event produced by m_1 is communicated and received by m_2. This corresponds to synchronizing the value of the input variable of m_2, to the value of the output variable of m_1 at the beginning of each macro-step, i.e., when the instant of time is standard.

This is captured by the following constraint, which uses predicate NowST introduced in Section 4.1.3:

$$\text{AlwF}\big(\text{NowST} \rightarrow (v_{m_1,out} = v_{m_2,in})\big) \tag{47}$$

where $v_{m_1,out}$ is an output variable of component m_1 linked to the $v_{m_2,in}$ input variable of the component m_2.

4.4.1 *Variations to the composition semantics*

As mentioned in section 2.1.2, many semantics exist for Statechart and its variants such as Stateflow. The X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$-based approach pursued in this thesis gives us great flexibility in adapting the formal semantics depending on the cases. In fact, changing semantics is as simple as changing X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ formulae.

For example, in some semantics input and output variables are synchronized not only at the end of a macro-step, but also during it [18]. To allow for this behaviour one would have to change constraint 43 with the following one (also, constraint 42 would have to be modified, but this is not shown here for brevity), which prescribes that connected input and output variables have the same values, unless one of the two components has reached a stable configuration:

$$\text{AlwF}\big(\neg \mathit{stable}_{m_1} \wedge \neg \mathit{stable}_{m_2} \rightarrow (v_{m_1,out} = v_{m_2,in})\big) \tag{48}$$

where $v_{m_1,out}$ is an output variable of component m_1 linked to the $v_{m_2,in}$ input variable of the component m_2.

Another possible variation could consist of imposing that there must be a maximum number K of micro-steps in a macro-step. This would reduce to forcing condition $\mathit{stable}_m \wedge \mathit{extST}_m$ (which, for formula (44), entails passing to a new macro-step) to occur within $K\epsilon$ instants from a standard one, which is simply formalized by the following formula (where $\text{WithinF}(\phi, K\epsilon)$ is an abbreviation for $\phi \vee \text{Dist}(\phi, \epsilon) \vee \ldots \vee \text{Dist}(\phi, K\epsilon)$):

$$\text{AlwF}(\text{NowST} \rightarrow \text{WithinF}(\mathit{stable}_m \wedge \mathit{extST}_m, K\epsilon)) \tag{49}$$

In a similar vein, another possible semantic variation might impose that a macro-step cannot last more than K micro-steps, even if the system has not yet reached a stable configuration. This semantics could be formalized, for example, by introducing an additional predicate, say $\mathit{adv_unstable}_m$ which holds exactly at distance $K\epsilon$ from a standard instant t if at $t + K\epsilon$ the module is still not stable. Then, formula (44) should be modified as follows (formulae (39)-(41) would also have to be modified, but this is not shown here for brevity):

$$\text{AlwF}(X_{st}(\top) \leftrightarrow (\mathit{stable}_m \vee \mathit{adv_unstable}_m) \wedge \mathit{extST}_m) \tag{50}$$

5 RELATED WORKS

As mentioned in the introduction, the work of this thesis comes from a collaboration between a group of automatic control engineers of ITIA and a group of informatics engineers of the DEIB of the Politecnico di Milano. The main objective of the work is to give a solution to the problem of specifying a development process for the control software of FMSs, using an MDE (Model-Driven engineering) approach. A commonly structured MDE methodology consists of several steps (or phases) which are briefly reminded hereafter and described in the introduction:

- *Control system specification*
- *Control system architecture and functional design*
- *Software implementation*
- *Verification and Validation (V&V)*

In the literature of the last decades, different frameworks based on the MDE approach have been developed. In the next section 5.1, some works which are similar to this one are analysed; the common ingredient is the use of formal methods to perform the V&V phase, but with differences with regard to the modelling language, the tools or the way formal methods are used, for example to obtain a set of tests to be performed successively.

A second group of works, that are shown in section 5.2, are related to the evolution of the work. They try to solve the problematic of time modelling of reactive systems that we solved introducing the X-TRIO metric temporal logic, using different approaches. For example, some works use an approach based on NSA to solve the problem of micro- macro steps and zero-time transitions, but do not address issues of decidability or verification; other works use the concept of *super-dense time* to model time in discrete-events and hybrid systems, but in a less intuitive and elegant way.

The different approaches to give a rigorous semantics to Statechart, and the various types of semantics, have already been described in the chapter 2.

5.1 MD APPROACHES FOR THE DEVELOPMENT PROCESS OF FMS

Verification is the process of checking the robustness and reliability of the designed control solution by proving its compliance with a given specification [39]. During the last few years, many research efforts focused on exploring and developing new methodologies supporting the verification process through formal approaches. Different types of formal models, as well as logics for the definition of the properties to be proved for the model have been investigated. The quantity of different solutions in literature is very high, mainly due to the fact that in the international community there is not a general consensus on which is the better formal or semi-formal language or standard to be used to model behaviour of FMS, or the best formal tools to check properties or the way to use it, so in this section have been selected only a part of the existed works, each of which is a good representative for its category of solutions.

The works in [39–43] define the model of the control system in terms of timed Signal Interpreted Petri Net (tSIPN) or timed Net Condition/Event Systems (NCES) [44], which are extensions of the classic Petri Nets. Respect to Stateflow, which is a graphical semi-formal modelling language, NCES and tSIPN have a rigorous semantics. NCES and tSIPN have a similar syntax and semantics, so only one of them has been explained here.

A timed NCES is a discrete state place-transition net formally described by the tuple:

$$NCES = \{P, T, F, M_0, C_N, E_N, C, E, B_C, B_E, C_S, D_t\}$$

where the tuple $PN = \{P, T, F, M_0\}$ is a classic Petri Net, and:

- C_N is the set of condition arcs $C_N \subseteq PxT$
- E_N is the set of event arcs $E_N \subseteq TxT$
- C is the set of conditions, further subdivided into C^{in} and C^{out}, the set respectively of input and output conditions
- E is the set of events, further subdivided into E^{in} and E^{out}, the set respectively of input and output events
- B_C is the set of NCES module condition input arcs $B_C \subseteq C^{in}xT$
- B_E is the set of NCES module event input arcs $B_E \subseteq E^{in}xT$

- C_S is the set of NCES module condition output arcs $C_S \subseteq PxC^{out}$
- D_t is the set of NCES module event output arcs $C_S \subseteq TxE^{out}$

So, respect to a classic PN, a NCES module is equipped with inputs and outputs which are of two types:

- Condition inputs/outputs carrying state information
- Event inputs/outputs carrying state transition information

The semantics of a NCES is of type *run-to-completion*, defined by the firing rules of the transitions. In addition to the fire rules of a classic PN, a transition is enabled by a condition signals if all source places of the condition signals are marked by at least one token (similar for event signals); if enabled, a transition is forced to fire and is executed in zero time.

NCES and tSIPN benefit also of the property of composition, so *basic* and *composite* modules exist. A *composite* module consists of a network of modules interconnected via event and condition arcs, in a similar way to the Simulink/Stateflow framework presented in this thesis.

The time in NCES is indicated by the *permeability intervals* attached to the incoming flow arcs of transitions. To every pre-arc of a transition, an interval $[eft, lft]$ is attached with $0 \leqslant eft \leqslant w$ (w is fixed). The interpretation is as follows: every place p bears a clock which is running if and only if the place is marked and switched off otherwise. All running clocks run at the same speed measuring the time the token status of its place has not been changed e.g. the clock on a marked place p shows the age of the youngest token on p. If a firing transition t is able to remove a token from the place p or adds a token to p then the clock of p is turned back to 0. In addition, a transition t is able to fire only if for any pre-place p of t the clock at place p shows a time $u(p)$ such that $eft(p,t) \leqslant u(p) \leqslant lft(p,t)$. Hence, the firing of transitions is restricted by the clock positions. Timed NCES is a powerful formalism, which expressiveness is enough to model discrete-events reactive systems.

In [40], an tSIPN sample model of a machining line with three working stations is verified using a symbolic model checker, Cadence SMV, and is then translated into the IEC 61131-SFC language [10], and is verified again using the same model checker. The translation from the tSIPN model to the linear temporal logic, which is used as input language of the model checker, is given by hand, in a similar way of the work of this thesis. In [39] an IEC 61499 sample model of a simple plant is analysed through the SESA model checker,

using a tool developed by the authors called VEDA. System properties to be checked are defined in Computation Tree Logic (CTL), a branching-time temporal logic incomparable with LTL. The semantics of CTL formulae is defined with respect to a reachability graph of the NCES model, whose states and paths are used in the satisfaction relation $\models$. In [43] authors use two extensions of CTL, ECTL (Extended CTL) and TCTL (Timed CTL), as input languages for the model checker SESA. In particular, TCTL essentially consists in attaching a time bound to the modalities of CTL. Details about how to translate an IEC 61499 model to an NCES model is given in another previous work of the same authors, [45], fixed a semantics of IEC 61499. Instead, [42] is a successive work of the same authors where they give a translation of IEC 61499 data types and algorithms in NCES, leaving as future works a more precise translation of an entire IEC 61499 application in NCES.

The works in [8] and [46] are examples of the use of Stateflows and Statecharts as a modeling language for FMS. A lot of other works related to the formal verification of Statecharts using model checking can be found in [47], which is a useful survey paper. In [8], a real model of the logic controller of a lasts warehouse deployed in the shoe manufacturing plant in Vigevano managed by ITIA has been developed in Simulink/Stateflow language exploiting modularization. The particularity of this work is that it uses formal methods to automatically generate test cases to reach the coverage of a Stateflow diagram. The properties to be proved are taken from the Model Coverage properties proposed by the DO-178B standard, which is the primary mean used by aviation software developers to obtain U.S. Federal Aviation Administration approval of airborne computer software. This standard has been proposed by the aerospace sector as a property set for verification of critical-safety solution, due to the lack in industry of commonly accepted quality indicators. There are three Model Coverage objectives:

- *Decision Coverage*, analyses decision points and determines whether all states of the Stateflow diagram can be reached at least once, then generates one reachability objective for each state.
- *Condition Coverage*, analyses the transitions of the Stateflow diagram and determines whether all conditions of each transition becomes true and false at least once, then it generates two objectives for each transition condition.
- *Modified Condition Decision Coverage*, analyses the transitions of the Stateflow diagram and determines to extent to which the test case checks the

independence of the transition conditions. Then, for each condition on the transition, determines if there is at least once that a change in the condition triggers the transition, then generates two objectives for each transition condition.

Simulink Design Verifier (SDV), that is a bounded model checker integrated in Matlab, has been considered by the authors to perform model automatic analysis. SDV is capable to generate test cases that achieve Model Coverage objectives. The generated test cases are automatically executed on the model by means of the integrated Simulink Verification and Validation tool, which, at the end of each execution, provides a report exposing achieved model coverage percentages. Authors applied the Model Coverage approach to the model of the lasts warehouse, showing the obtaining results.

[46] is a bit more theoretical paper about model checking of Statechart diagrams. Authors want to perform model checking given arbitrary properties defined in CTL over a *computation tree* of the model, the tree of all possible sequences of Statechart configurations. To do so, they give the definition of what is a generic *step semantics* for Statechart, and define what is a *step*, in a similar way of this thesis, overcoming the problem of all the existing different step semantics described in section 2.1.2. They claim, without demonstration, that the formulation of CTL properties on a computation tree is straightforward. To exploit existing model checkers, they show how to construct a Marked Kripke Structure whose computation tree is equivalent to the computation tree of the initial Statechart model and formally demonstrate it. Finally, they define an expansion operator to expand the Statechart CTL specifications so they can be used as equivalent specifications for the Marked Kripke Structure obtained through the translation; an example to validate the approach and the results is given, using the STATEMATE step semantics of Statechart.

The work [48] is an example of the use of UML 2.0 as modelling language. A brief introduction to functionality based controllers (FBC), which are logic controllers that can change their functionalities, is given. The various functionalities are modelled using IEC 61499 Function Blocks. Then, for modelling the behaviour of the FBs, Activity Diagrams of UML are investigated, which are based on the semantics of classic PNs. Authors used the Borland Together Control Center 6.2 tool to model Activity Diagrams for FBCs.

In this section, several formal methodologies for the verification of automation solutions, existed in literature, have been described, with differences regard to the types of model-checking tools exploited and the formalisms used to describe the control algorithm. Each methodology has its specific benefits

and limitations, but none of the approaches mentioned above is commonly adopted in the current development practice. In particular, the works come from the automatic and control engineering field, where the focus is on the approaches and methodologies supporting the verification process of FMS through formal methods, without going in-depth in the more theoretical part of the problems, for example how to model time in real-time and reactive systems. This thesis has a more general approach for the modelling of reactive systems, since it presents a new metric temporal logic to deal with zero-time transitions and step semantics; this logic is a "glue" between a front-end part, the modelling language (Simulink/Stateflow, but also other languages with a similar semantics can be modelled in X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$, such as tSPIN, NCES or the Activity Diagrams of UML 2.0) and a back-end part, the model checker tool who takes X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ as input language.

5.2 FORMAL SPECIFICATIONS FOR REACTIVE SYSTEMS

In the last decades, a lot of efforts have been spent in the development of formal specification methods for real-time and reactive systems. In this thesis, in the evolution of the original work, we introduced a metric temporal logic which deals with the time modelization problems of this type of systems, in particular about the notions of zero-time transitions and micro- and macro-steps. These notions appear very naturally when reasoning about computations of embedded systems too, so they arise in real-time temporal logics (metric and not metric). Since the very early developments in this field, approaches were introduced that admit zero-time transitions at the price of associating multiple states to single time instants, as in [49]. The approach of this thesis is akin to that of [27], which defines, for the first time, a first-order temporal logic to support the definition of *quantitative* real-time properties, the Metric Temporal Logic (MTL), which extends the classic temporal logic. It introduces a *distance function* $d : TxT \to D$ over the pointwise T temporal domain of classic temporal logic, to measure time. D is the range (or metric) domain of d; the author says that there are no reasons to use reals as range domain, but that a structure with addition and a zero element is enough and gives the minimum restrictions. After it, he defines a *metric point structure*, which is a two-sorted structure $M = \{T, D, <, d, +, 0\}$ where:

- $<$ is a total order relation over T, $< \subseteq TxT$
- d is surjective

- $(D, +, 0)$ is an algebraic structure where $+ : DxD \rightarrow D$ and $0 \in D$ are respectively the usual addition operation and the zero element of D.

Furthermore, he introduces a particular metric point structure, which support the notion of micro- and macro-steps and is the first and more general example of the so called *super-dense time* approach, which is an alternative approach to model time in systems with the presence of zero-time transitions. This particular metric point structure is defined in the following way, given $T = \mathbb{N}x\mathbb{N}$ and $D = \{0\}x\mathbb{N} \cup \mathbb{N}_+ x\mathbb{Z}$:

- $(n, n') \prec (m, m')$ iff $(n < m$ or $(n = m$ and $n' < m'))$ is the lexicographic order between couple of natural numbers
-
$$d((n, n'), (m, m')) = \begin{cases} (0, |n' - m'|) & \text{if } n = m \\ (m - n, m' - n') & \text{if } n < m \\ (n - m, n' - m') & \text{if } n > m \end{cases}$$
 is the distance function over couple of natural numbers
- $(n, m) + (n', m') = (n + n', m + m')$ is the addition operation over couple of natural numbers
- the element 0 corresponds to the couple $(0, 0)$

Informally, the idea is that a change in the first component of the couple corresponds to the evolution of the real time, in the same way as a change in the standard part of non-standard numbers in this thesis; keeping fixed the first component, a change in the second component corresponds to the evolution of the logic time and represents a micro-step, in the same way as a change in the infinitesimal part of non-standard numbers. In the rest of the paper, it defines a group of basic metric operators and demonstrates that it is possible to derive the classical qualitative operators of temporal logic quantifying existentially, over time, the basic metric operators. Finally, it gives equivalences between the basic metric operators, a proof system and several examples to show how to express real-time properties in MTL. A lack of the work is that it does not address issues of decidability and verification.

A group of works which are based on the *super-dense time* approach are [50–52]. They are only a little part of works which exploit this approach, but the more representatives.

The work [50], the oldest, proposes a framework for the formal specification of timed and hybrid systems. In the first part of the paper, authors

define a semantics for *timed transition systems*, which are adapted to model discrete-events and reactive systems. Given a time domain T, for example the totally ordered set of real numbers $\mathbb{R}$, a timed transition system is a transition systems with an interval $[l, u]$ associated to each transition, with $u \geqslant l$ and $u, l \in T$, which are respectively the minimal and maximal delay of the transition. The timed transition system has a *run-to-completion* semantics, with an explicit clock. Transitions take zero-time to be performed, so it is possible to have different value for a variable at the same instant of time: this is a great difference with my approach, where the evaluation of the variables and the current state, that the authors called a $situation$, is a *function* of time, which is a more natural and intuitive way to model it. After the introduction of timed transition systems, they present Timed Statechart, an extension of the normal Statechart notation, by annotating each transition with an interval $[l, u]$, denoting the lower and upper time bound of that transition. A particularity of the semantics of this system is that an event persists until time progresses, without consume it, and disappear when time advances again, which is similar to the STATEMATE semantics of Statechart. To specify properties for timed transition systems they use two different styles of specification, one is based on MTL, the other one on a age-based approach with explicit clocks over the operators of a classic temporal logic; in the age-based approach, an age function is introduced that, for each formula ϕ, measures the length of the largest time interval ending in the current position, on which the formula ϕ holds continuously. Then, a proof system is given for timed transition systems.

In the second part of the paper, authors define a semantics for hybrid systems. Informally speaking, an hybrid system is a real-time heterogeneous system that include continuous-time subsystems interacting with discrete events, such as a digital controller that controls a continuous environment. The time model is an extension of the super-dense time structure, where the time domain is the set of non-negative real numbers instead of the set of natural numbers. A time structure can be viewed as consisting of alternations of discrete and continuous phases. A *discrete phase* is a sequence of instants of the time structure where only the second component increases; it corresponds to a zero-time transition of the system. Instead, a *continuous phase* is a sequence of instants of the time structure where only the first component, the real time, increases: the evolution of variables during a continuous phase is given by the definition of a continuous function in the interval of real numbers $[t_i, t_{i+1}]$, which are respectively the initial instant and the final instant of the continuous phase. The semantics is of type *run-to-completion*. To model hybrid systems, authors define a generalization of a timed transition system,

called *phase transition system*. The main difference is that a phase transition system adds a finite set of *activities* to the original formalism, where each activity is associated with a conditional differential equation which constraints the value of a continuous variable of the system. Transitions start activities by enabling their activation conditions, and viceversa activities start transitions: this mechanism models the activation of discrete and continuous phases. To specify properties, they use again the two different styles of specifications developed for timed transition systems.

The other two works [51, 52] are related to the specification of a semantics for hybrid systems of the Ptolemy project [53]. In the first work, authors define an operational *run-to-completion* semantics for these systems, with, as in [50], an alternation of continuous and discrete phases. The paper presents a simulation tool, HiVisual, used to develop an example of such type of systems. Authors represent continuous evolving and discrete signals with a continuous function on, respectively, a continuous or discrete domain represented by an extension of the super-dense time structure that they called the *tag set*; each element of this time structure is a possible *tag* for the signal. The evolution of the system is represented with a set of ODE (Ordinary Differential Equations), which is the usual mathematical way to describe the trajectory of a continuous signal. The work is complicated by the issue of smoothness, Lipschitzness, existence and uniqueness of solutions, which are not addressed by this thesis since they are typical of hybrid systems.

In the second work [52], the same authors give a semantics to discrete-event models, which generalizes the semantics of a synchronous/reactive system; the continuous-time semantics of hybrid systems is presented as a generalization of the discrete-event one. The work shows that all three semantic models can be used in actor-oriented composition languages, which can be concretely represented in several ways. An *actor-oriented design* is a component methodology very close to the component methodology presented in this thesis and the one of the IEC 61499 Function Block Network, where a series of *atomic actors* with a well-defined interface could be composed to form a *composite actor*. The interface includes *ports* and *parameters*, where the latters are used to configure the operations of an actor. Ports are connected by explicit *communication channels*: a difference with Simulink/Stateflow is that actors cannot communicate directly but only through the channels connected to their output ports. The composition of arbitrary models of computation is made tractable by an *abstract semantics*, which abstracts how communication and flow of control work. The abstract semantics is not the union of interesting semantics, but rather the intersection. It is abstract in the sense that it rep-

resents the common features of models of computation as opposed to their collection of features. In this semantics, actors execute in three phases:

1. a *setup* phase
2. a sequence of *iterations*
3. a *wrap-up* phase

An iteration is a sequence of operations that read input data from a port, produce output data to a port, and update the internal state of an actor: these operations are described in detailed in the paper and represented with abstract functions.

Authors start defining the semantics of synchronous/reactive (SR) models. The execution of an SR model follows ticks of a global clock, in a synchronized way. At each tick, every variable may have a value, that is calculated by a function. To give a semantics to SR systems, the authors specialize the tagged signal model introduced in [51]. They define a *signal* to be the entire history of a communication between an output port and an input port. Given the set of natural numbers as tag set, with the usual numerical order, then a signal s is a partial function $s : T \rightharpoonup V$ defined on an initial segment of T, which is the time domain; V is the domain of the range values of the signal. After it, they define the semantics for discrete-events (DE) and continuous-time (CT) models as an extension of the semantics of SR models. Both of them use the tagged signal model, with the difference that in CT models, some signals, between two discrete phases, have continuous evolving values governed by a set of ODE. Both extensions use the set of non-negative reals as the first component of the time structure. Given the different time models, all the three semantics are of type *run-to-completion*. A limitation of these works is that the Zeno behaviours must be detected in advance and avoided, since all the execution traces must progress *a priori*, instead in this thesis it is possible to specify a X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ formula to detect and analyse the presence of various type of Zeno behaviours.

Other works [54, 55, 12] have tried to deal with the problem of giving a formal and rigorous semantics to semi-formal languages based on the micro-macro step abstraction, using non-standard analysis (NSA). In [54] they use a NSA semantics approach to solve a set of ODE describing an hybrid system modelled in Simulink, in presence of cascaded mode changes produced by the "zero-crossing" problem. In particular, authors define a denotational, a constructive and a Kahn Process Network semantics for hybrid systems for a

minimalist language called *SimpleHybrid*. They define a principle, called *standardization principle*, which characterize the class of hybrid systems which can only be given a standard semantics, for any choice of the time base (which corresponds to the constant ϵ, introduced in the chapter 4) of the non-standard semantics. Despite the similarities between their approach and the one of this thesis, the latter is more general, since it is possible to view it as a "schema" to extend existed metric temporal logics to use a NSA based framework.

In [55], authors define a complete system theory, discussing in detail the notion of "system" and investigating its computability issues. They use a formalization close to that of Turing machines to define the concept of "system", starting by the formal definition of a *time scale*, which is an increasing succession of non-standard positive numbers where, for each instant, exists a predecessor and a successor which belongs to the time scale itself. Then, they define *dataflows*, which are mapping functions from a time scale T to symbols of an alphabet A, and the concept of *snapshot* of a dataflow, which is, given an instant t, the set of symbols produced by the dataflow at an instant t' such that $t' = min\{u \in T | t \leqslant u\}$. Later in the paper, they define the concept of *transfer function*, which can be seen as a real-time dataflow transformer satisfying a so called *causality condition*. In the non-standard framework, it is the classic strong causality condition, since the concept of working in zero time is not defined. Transfer functions can be "composed" through interfaces, in such a way that a "system" can be the result of the composition of two or more "subsystems". Finally, they define a "system" as a particular extension of a Turing machine, adding an input/output mechanism that consists of:

- an input channel x capable of receiving, only on instants of a given input time scale, elements that belong to a given input alphabet, called the *input domain* of the system
- an output channel y capable of emitting, only on instants of a given output time scale, elements that belong to a given output alphabet, called the *output domain* of the system

The memory tape is indexed by the set of non-standard natural numbers and the control mechanism updates the internal state at instants that belong to an internal time scale, that could be also a non-standard set. The authors observe that it is possible to associate to each system a transfer function which is the mapping that associates to any input flow, the output flow generated by the system. Conversely, they define the class of *implementable transfer function*, which is the class of functions that can be associated to a system. The work

ends with the definition of a *computable function*, which is a function on real numbers that can be approximated by an implementable transfer function, given a suitable encoding of non-standard numbers: in this way, the authors demonstrate formally that a large class of real physical systems (such as dynamical and Hamiltonian systems, hybrid systems and so on) can be defined using their formalism. This work is more general than the one in this thesis, since it tries to define a formalism to describe a large class of real-time systems, but it addresses the problematic only on a theoretical viewpoint.

In [12], authors deal with the problem of zero-time transitions giving an NSA semantics to timed Petri Nets (TPN). They use the TRIO temporal logic to obtain an axiom system to describe the non-standard semantics in a way similar to the one in this thesis. A timed Petri Nets is very close to a tSIPN, without events and condition arcs and the input/output mechanism. In particular, every transition is labelled by a pair $[l, u]$, with $u \geqslant l \geqslant 0$ and $u, l \in T$, where T is the standard time domain and u and l are respectively the minimal and maximal delay of the transition. To implement the non-standard semantics, they change the fire rules of a transition, permitting to augment the lower and the upper-bound of each transition by an infinitesimal positive constant amount. They introduce a first axiom which asserts that, given a transition ν and its lower-bound l, if the transition fire at a certain instant of time t, than it consumes a token which has been produced strictly more than l time units before t: in this way, if $l = 0$, the axiom excludes a zero-time transition and constraints the transition to fire in an infinitesimal positive time. Another axiom asserts that, given a transition ν and its upper-bound u, exists an infinitesimal number ϵ such that each previous transition, which belongs to the set of transitions producing the token which enables the transition ν, fires at a distance d in the past, with $d \leqslant u + \epsilon$. Adding axioms stating the token uniqueness, authors obtain that:

- there are no firings occurring exactly in null time
- no transitions can fire exactly at the same instant, but at least at two instants whose distance is infinitesimal

This work is less general than the one in this thesis and is similar to the first part, where TRIO has been used to develop a set of axioms modelling a particular semantics of reactive systems. Instead, in this thesis we realize a new metric temporal logic, X-TRIO, which intrinsically exploits a non-standard time model to deal with the problem of the modelling of micro- macro-step systems with zero-time transitions.

Finally, the proposal in [56] provides notations for modelling micro-steps in the framework of Duration Calculus, which, unlike TRIO, is a logic based on intervals: it defines a decidable fragment of the notation but does not give algorithms or builds tools supporting verification. Other works are only partially connected to this thesis, since they deal with issues concerning the modelling and development of embedded systems at various time scales: [57] and [31] deal with issues of sampling and digitization, [58] and [59] discuss issues related with time granularity, and [60] provides a refinement method based on assume-guarantee induction over different time scales.

6 TOOLS

In this chapter are described the tools that have been used in this thesis to model and perform the formal verification of the robotic cell case study. In the section 6.1 is described $\mathbb{Z}$ot, a bound model checker developed internally at the Politecnico di Milano, written in the *Common Lisp* programming language. It has been used to perform the verification of an axiomatic X-TRIO$^{\text{P-R}}_{\overline{\mathbb{N}}_+}$ model of the robotic cell. This model has been developed using the Sf2Trio tool, a plugin for $\mathbb{Z}$ot described in section 6.2, which allows the manual translation of a Stateflow/Simulink graphical model into a logic model in a simple and straightforward way, without writing the necessary X-TRIO$^{\text{P-R}}_{\overline{\mathbb{N}}_+}$ formulae directly. Another tool, a plugin for $\mathbb{Z}$ot, called *X-TRIO*, has been developed during the work for this thesis, to implement the decidable fragment of X-TRIO$^{\text{P-R}}_{\overline{\mathbb{N}}_+}$. It is described briefly in section 6.1.

6.1 ZOT VERIFICATION TOOL

$\mathbb{Z}$ot [23] is a Bounded Model/Satisfiability checker that takes as input specifications written in a variety of temporal logics, and determines whether they are satisfiable or not. Through this basic mechanism, it can perform verification of user-defined properties for the desired models.

Let us call S the temporal logic-based model of the system being designed. If S is fed to the $\mathbb{Z}$ot tool "as is", $\mathbb{Z}$ot will look for an execution trace of the modelled system; if it does not find one (i.e., if the model is unsatisfiable), then the model is contradictory, hence it contains some flaw that make it unrealizable in practice. Now, suppose we introduce a further temporal logic formula, P, which formalizes a user-defined property to be checked for the system. We can ask $\mathbb{Z}$ot to check whether the formula $S \wedge \neg P$ is satisfiable or not. If $S \wedge \neg P$ is unsatisfiable, this means that there is no execution trace that satisfies the system (i.e., S), that also satisfies $\neg P$, that is, that violates the property P. If no system trace violates property P, then the latter actually holds for the system. If, on the other hand, $S \wedge \neg P$ is satisfiable, this means that there is at least one system trace that satisfies both S and $\neg P$; that is, there is at least one execution trace of the system that violates the property,

so the property does not hold for the system. If $\mathbb{Z}$ot determines that $S \wedge \neg P$ is satisfiable, then the tool produces an execution trace that satisfies it, i.e. a counterexample trace that is compatible with the system model but that violates the property.

$\mathbb{Z}$ot performs the checks outlined previously by encoding temporal logic formulae into the input language of various solvers. In particular, $\mathbb{Z}$ot supports two kinds of solvers: SAT solvers and SMT solvers. SAT solvers are capable of taking, as input, formulae written in propositional logic and determine whether they are satisfiable or not. SMT solvers do the same, but they accept, as input, formulae written in logics (fragments of First-Order Logic) that are richer than the simple propositional logic. Over the last few years, both SAT solvers and SMT solvers have made great strides in terms of their performances, so that they have become viable engines for fully automated logic-based verification approaches such as the one realized by $\mathbb{Z}$ot. In addition, most SAT/SMT solvers accept inputs written in a standard format (DIMACS format for SAT solvers and the SMT-LIB/SMT-LIB2 format for SMT solvers), which makes them easily interchangeable. This is very useful, since different solvers implement different heuristics, and the "best" solver does not exist in absolute terms, but only on a model-by-model case.

At its core, $\mathbb{Z}$ot encodes specifications written in a variety of temporal logics into the input languages of SAT/SMT solvers. The tool supports several different encodings, which the user can choose by setting suitable options in the verification scripts. $\mathbb{Z}$ot is plugin-based: every encoding is realized by a plugin, and to select the encoding to be used the user selects the corresponding plugin. Figure 16 depicts the architecture of the $\mathbb{Z}$ot tool. At the basis, there are the third-party SAT and SMT solvers. Suitable modules interface $\mathbb{Z}$ot with the underlying solvers. Since the two kinds of solvers accept different inputs and have different features, the interface modules are also different, depending on the solver: "sat-interface" and "zot2cnf" interface $\mathbb{Z}$ot with SAT solvers, while "smt-interface" does the same for SMT solvers.

The plugins implementing the different encodings define how temporal logic formulae are translated into the input languages of SAT/SMT solvers. The temporal logic supported by $\mathbb{Z}$ot are depicted in Figure 16 in the lightest grey. They are:

1. LTLB: propositional LTL with both future and past operators.
2. CLTLB: constraint LTLB, wich has the same temporal operators as LTLB, but which admits, as atomic elements, also a limited set of first-order arithmetic constraints (e.g. $x \geqslant 2$).

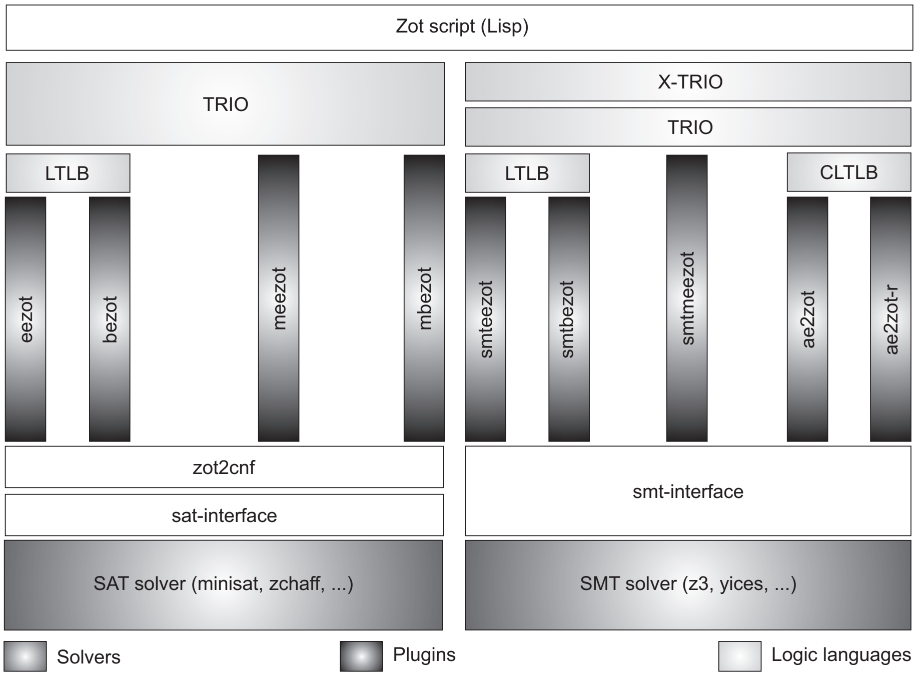

Figure 16: Overall architecture of ℤot

3. TRIO: the metric linear temporal logic described in section 3.3.1.
4. X-TRIO: the new metric temporal logic described in this thesis in chapter 4. The plugin implements the decidable fragment X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$.

Since both LTLB and CLTLB are subsets of the more expressive TRIO, to encode TRIO formulae into the input languages of SAT/SMT solvers some plugins (e.g., eezot in Figure 16) first translate TRIO formulae into LTLB/-CLTLB, then they encode the latters into the language of the corresponding solver. ℤot plugins implement the primitives that are used in ℤot scripts to define the models to be analysed. Since the plugins are implemented as Lisp modules, the primitives are essentially Lisp declarations. Then, ℤot scripts, which contain both the model to be analysed and the necessary commands to invoke the desired solver, are a collection of Lisp statements. The available ℤot plugins are the following:

- **eezot**, which encodes LTLB formulae into the input language of SAT solvers; eezot also translates TRIO formulae into LTLB ones, by transforming the TRIO metric operator Dist into a series of next/yesterday operators.
- **smteezot**, which encodes LTLB formulae in the input language of SMT solvers; smteezot is akin to eezot (for example as far as the translation of the TRIO Dist operator is concerned), but it exploits the features of SMT solvers to produce a more compact encoding than eezot.
- **ae2zot**, which encodes CLTLB formulae in the input language of SMT solvers; ae2zot differs from smteezot in that it supports arithmetic constraints over integer numbers such as $x \geqslant 2$. However it is similar to smteezot in that it translates the TRIO Dist operator as a sequence of next/yesterday operators.

As shown in Figure 16, $\mathbb{Z}$ot includes also other plugins that are not used in this thesis.

In the following, the X-TRIO plugin is explained briefly, which is the implementation of the decidable fragment X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ logic. In this thesis has been demonstrated that it is possible to reduce the satisfiability of a X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ formula ϕ to a suitable LTLB formula ψ which is equisatisfiable under some restrictions shown in section 4.3. The transformation is syntax-directed and is based on the translation schema illustrated in table 4, under the further conditions imposed by the formula (38) over the LTLB trace π; the formula must be true at the instant 0. In other words, to enforce the condition over the LTLB trace, the X-TRIO plugin wraps the function zot of the TRIO plugin, which is the interface to invoke the model checker, in such a way that the X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ formula ϕ is translated in the LTLB formula $\pi \wedge (38)$. The X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ operators implemented in the X-TRIO plugin are the following:

- *(X-somf F)* is the X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ operator SomF(F)
- *(X-alwf F)* is the X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ operator AlwF(F)
- *(X-until F G)* is the X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ operator Until(F, G)
- *(X-since F G)* is the X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ operator Since(F, G)
- *(X-next-ns F)* is the X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ operator $X_{ns}(F)$

- *(X-next-st F)* is the X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ operator $X_{st}(F)$
- *(X-yesterday-ns F)* is the X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ operator $Y_{ns}(F)$
- *(X-yesterday-st F)* is the X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ operator $Y_{st}(F)$
- *(X-dist F S N)* , where S and N are respectively the standard and non-standard part of the non-standard number $T = S + N\varepsilon$, implements the $\text{Futr}(F, T)$ operators if $T \geqslant 0$ and the $\text{Past}(F, -T)$ operators if $T < 0$.

To allow the construction of more concise formulae, the list above contains also derived operators. In the future, we will implement natively in the plugin other derived operators described in section 4.2. s_p, f_p and e_p are implemented internally in the X-TRIO plugin as the Lisp terms NowST, Ep and Fp respectively, since the TRIO plugin uses Lisp terms to implements the propositional letters of the temporal logic.

6.2 SF2TRIO TOOL: ENCODING STATEFLOW/SIMULINK INTO X-TRIO$^{\text{P-R}}_{+}$

Sf2Trio is a tool that supports designers in the construction of Simulink/Stateflow models, avoiding the burden to write manually the necessary X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ formulae. It constitutes a first step towards the development of a completely automatic tool to translate a system specified by a Simulink/Stateflow diagram into a set of X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ formulae that characterize its behaviour. Sf2Trio has been developed as a plugin for $\mathbb{Z}$ot and is based entirely over two other plugins: X-TRIO, which implements the encoding given in Section 4.4 for the X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ metric temporal logic, and **ae2zot**, which is used by Sf2Trio to support checking of arithmetic finite integer variables in an efficient way. The current version of the tool provides a set of commands to specify a Stateflow diagram for a basic component and to define the composition and interactions of several basic components into a composed one, according to the Simulink graph.

We first illustrate the commands to define the Stateflow diagram of a basic component. Each command is specified by its syntax, with the name of the command in bold and parameters in italic (parameters with a colon, as in *:parameter-name*, are optional).

1. (**make-module** $ModName$)
   A *module* is the entity that, in Sf2Trio, represents a (basic or composed) component. Parameter *ModName* is the module name.

2. (**def-variable** $VName\ Type\ DomType\ :initvalue\ \ :range\ (minvalue\ \ maxvalue))$
Command **def-variable** defines a single Stateflow variable, with the following parameters:
   - *VName* is the name of the variable.
   - *Type* is one of three predefined values (***Input***, ***Output*** and ***Intern***) qualifying the variable as input, output or internal.
   - *DomType* denotes the type of the variable; in the actual version of the tool, only integer (***Int***) and boolean (***Bool***) variables may be declared; integer variables range over finite domains.
   - *:init* specifies an optional initial value, at time 0, for the variable; if it is not specified, the default value is nil, which corresponds to false, for boolean variables, and 0 for integer variables.
   - *:range* specifies the range of an integer variable; by default, the value is in the range $[-100, 100]$.
3. (**def-state** $SName\ :init\ :entering\ Acts\ :during\ Acts\ :exiting\ Acts\ :inv\ \ Expr)$
Command **def-state** defines a single state of the Stateflow diagram. Its parameters are the following:
   - *SName* is the state identifier.
   - *:init* indicates that this state is the unique initial state for the Stateflow diagram. The uniqueness is not enforced by the plugin, it is responsibility of the user.
   - *:entering*, *:during* and *:exiting* each specify a list of **actions** *Acts* that are executed when the state is, respectively , entered, exited, or throughout the permanence in the state. The syntax of a single action has the form (*var expression*), where *expression* denotes the value assigned to *var*. Arithmetic operators include the usual $+$, $*$, $-$ and $/$; constant expressions must be integers.
   - *:inv* specifies a time invariant over a state. The syntax has the form (*operator val*), where *val* is an integer constant value greater or equal than one and *operator* is one of the following:
     - **<=** every outgoing transition from the state *SName* is enabled up to *val* standard instants of time in the future, starting from the instant of time in which the state *SName* is entered. If the

state is entered in a non-standard instant of time, *val* must be added to its standard part to obtain the standard instant of time from which the outgoing transitions are disabled.

- **>=** every outgoing transition from the state *SName* is disabled up to *val* standard instants of time in the future, starting from the instant of time in which the state *SName* is entered. If the state is entered in a non-standard instant of time, *val* must be added to its standard part to obtain the standard instant from which the outoing transitions are enabled.
- **=** every outgoing transition from the state *SName* is disabled up to *val* standard instants of time in the future, starting from the instant of time in which the state *SName* is entered, and enabled between *val* and $val + 1$ standard instants of time in the future. If the state is entered in a non-standard instant of time, *val* must be added to its standard part to obtain the standard instant from which the outgoing transitions are disabled.

4. (**def-transition** $\mathrm{SourceSName}$ $\mathrm{DestSName}$ $:\mathrm{condition}$ Expr $:\mathrm{action}$ Act)
   Command **def-transition** defines a single transition of the Stateflow diagram, with the following parameters:
   - *SourceSName* and *SourceDestSName* are, respectively, the label names of the source and destination states of the transition.
   - *:condition* specifies the transition condition as a boolean expression *Expr*. If absent, the condition is always true.
   - *:action* specifies a transition action, which is executed when the transition fires.

The following Sf2Trio commands are used to declare a composed component.

1. (**make-composed-module** $\mathrm{ModName}$ $\mathrm{ComposedModules}$)
   Command **make-composed-module** specifies the set of components of the specified composed module.
   - *ModName* is the name of the composed module.
   - *ComposedModules* is a list of pairs ($\mathrm{ModName}$ Path), where the first parameter is the name of a module to compose and the second is the file that contains its specification.

2. (**def-connection** SourceVName DestVName)
   Command **def-connection** defines an equivalence between a pair of variables (*SourceVName*, *DestVName*); this corresponds to a *link* of the Simulink graph that describes the composed module. The connected variables must be of the same type, but of different modules. In a composed module, a variable is referenced in a different way respect to a basic module, using the syntax ModName.VName, where the first part is the name of the module and the second is the name of the variable defined in the module ModName.

The following commands, which can be declared in any module, are used to define and verify user-defined properties thereof.

1. (**def-axiom** Formula)
   Command **def-axiom** defines a TRIO temporal logic formula that specifies a temporal property of the model, for example a constraint which specifices that a machine must remain in a busy state for a fixed number of time instants (its working time).
   - *Formula* is the X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$ formula, written in the input language of $\mathbb{Z}$ot, which formalizes the desired property. States and variables can be referred to in the formula using the corresponding predicates automatically introduced by the Sf2Trio tool, which have the form VName if the formula is defined in a basic component, or ModName.VName if the formula is defined in a composed component. The special form ModName.State refers to the actual state of the module ModName.

2. (**analyze** BoundLength Property)

   Command **analyze** is the interface to the $\mathbb{Z}$ot model checker. It performs the verification of a user-defined property for the specified module, using the plugin **ae2zot** and the Z3 solver [25].
   - *BoundLength* is the maximum length of runs analysed by $\mathbb{Z}$ot. It is the length of execution traces analysed by the solver.
   - *Property* is the user-defined property, written in X-TRIO$^{\text{P-R}}_{\mathbb{N}_+}$, that is to be analysed by $\mathbb{Z}$ot.

Figure 17 is the Sf2Trio model of the Stateflow diagram of the *Machine 1* of Figure 10. Figure 17 is the Sf2Trio model of the composed component of the robotic cell, which corresponds to the Simulink graph of Figure 11.

```
(make-module 'Machine1)

;;; Input Variables Definitions
(def-variable 'FM1 '*Input* '*Bool*)
(def-variable 'SwitchAutoMan '*Input* '*Bool*)

;;; Output Variables Definitions
(def-variable 'M1 '*Output* '*Bool*)

;;; State Definitions
(def-state 'Start t :init t :entering ((M1 nil)))
(def-state 'Working :inv '(>= 2))
(def-state 'WorkingEnd :entering ((M1 t)))
(def-state 'Dispatch :entering ((M1 nil)))

;;; Transition Definitions
(def-transition 'Start 'Working :condition '(&& FM1 (!! SwitchAutoMan)))
(def-transition 'Working 'Start :condition 'SwitchAutoMan)
(def-transition 'Working 'WorkingEnd :condition '(&& (!! FM1) (!! SwitchAutoMan)))
(def-transition 'WorkingEnd 'Working :condition 'SwitchAutoMan)
(def-transition 'WorkingEnd 'Start :condition 'SwitchAutoMan)
(def-transition 'WorkingEnd 'Dispatch :condition '(&& FM1 (!! SwitchAutoMan)))
(def-transition 'Dispatch 'Start :condition '(|| (!! FM1) SwitchAutoMan))
```

Figure 17: Model of Machine 1 in the input syntax of the Sf2Trio tool.

The two figures are used to exemplify how to use the tool Sf2Trio in a real case study, the robotic cell, illustrating some of the commands defined in the two Lisp scripts. In Figures 17 and 18 there are no integer variables, so it is not possible to show how to define and use such type of variables in actions or user-defined properties. The following are a list of some examples of commands:

- *(**def-variable** 'SwitchAutoMan '*Input* '*Bool*)* is an example of the definition of a boolean input variable, *SwitchAutoMan*. It corresponds to the Stateflow variable with the same name which is used to signal that the robot arm is entered in manual mode. It is initialized by default to false and is an input variable for the module of the *Machine 1*, as it is possible to see in Figure 11.

```
(make-composed-module 'RoboticCell '(
(Machine1 "path\Machine1.lisp")
(Machine2 "path\Machine2.lisp")
(ConveyorIn "path\ConveyorIn.lisp")
(Robot "path\Robot.lisp") (Controller "path\Controller.lisp")))

;;; ConveyorIn
(def-connection 'ConveyorIn.A 'Controller.A)
(def-connection 'ConveyorIn.B 'Controller.B)
(def-connection 'ConveyorIn.FCIn 'Robot.FCIn)

;;; Machine1
(def-connection'Robot.FM1 'Machine1.FM1)
(def-connection 'Controller.M1 'Machine1.M1)

;;; Machine2
(def-connection 'Robot.FM2 'Machine2.FM2)
(def-connection 'Controller.M2 'Machine2.M2)

;;; Robot
(def-connection 'Controller.ToP0 'Robot.ToP0)
(def-connection 'Controller.ToM1 'Robot.ToM1)
(def-connection 'Controller.ToM2 'Robot.ToM2)
(def-connection 'Controller.ToCin 'Robot.ToCin)
(def-connection 'Controller.ToCout 'Robot.ToCout)
(def-connection 'Controller.FP0 'Robot.FP0)
(def-connection 'Controller.FM1 'Robot.FM1)
(def-connection'Controller.FM2 'Robot.FM2)
(def-connection 'Controller.FCIn 'Robot.FCIn)
(def-connection 'Controller.FCOut 'Robot.FCOut)
(analyze 20 (X-somf ([=] Controller.State Controller.Start)))
```

Figure 18: Model of the composed module of the robotic cell in the input syntax of the Sf2Trio tool.

- *(**def-state** 'Start t :init t :entering ((M1 nil)))* is an example of the definition of the *Start* state of the *Machine 1*, with an entry action (M1nil). The action corresponds to the assignment of the false value to the Stateflow variable M1 of the *Machine 1*, which is used to signal that the machine is free to process a new piece.

- *(**def-state** 'Working :inv '(>= 2))* is an example of the definition of the *Working* state of the *Machine 1*, with an invariant over the state. The invariant models the self-transition of the *Working* state in Figure 10: it means that the *Machine 1* needs at least two standard instants of time to process a piece, i.e. it models the working time of the machine. To enforce it, the outgoing transitions are disabled when the *Working* state is entered until the working time is expired.

- *(**def-transition** 'Start 'Working :condition '(&& FM1 (!! SwitchAutoMan)))* is an example of the definition of the transition between the *Start* state and the *Working* state, with a guard over the transition. The guard is modelled with the formula *(&& FM1 (!! SwitchAutoMan))*, which is an example of the definition of a conditional expression: *&&* and *!!* are the Lisp operators which correspond to the two X-TRIO$^{P\text{-}R}_{\mathbb{N}_+}$ operators $\wedge$ and $\neg$. The notation is the usual Lisp prefix notation for functions and operators. The guard means that the machine starts to process a new piece (i.e. it enters the *Working* state) if the robot arm is not in manual mode and it has just delivered a new piece to the machine, signalled using the Stateflow variable $FM1$.

- *(**make-composed-module** 'RoboticCell '((Machine1 "path\Machine1.lisp") ...))* is an example of the definition of a composed module, the *RoboticCell* module. (Machine1 "*path*\Machine1.lisp") is an example of how to define the path of one of the modules of the robotic cell, the *Machine 1* component, where $path$ is the directory path of the Lisp script which define the component.

- *(**def-connection** 'ConveyorIn.A 'Controller.A)* is an example of the connection between the variable A of the *Conveyor-in* and the variable of the same name of the *Controller* of the robotic cell. Since *RoboticCell* is a composed module, the notation to refer the variables is of the form $ModName.VName$, which is needed also to disambiguate variables with the same name of different modules.

- *(**analyze** 20 (X-somf ([=] Controller.State Controller.Start)))* is an example of the definition of a property to be verified by the $\mathbb{Z}$ot model checker on the *RoboticCell* model. *(X-somf ([=] Controller.State Controller.Start))* is an example of an user-defined property. In particular, X-somf is the Lisp function which implement the X-TRIO$^{P\text{-}R}_{\mathbb{N}_+}$ SomF operator; *([=] Controller.State Controller.Start))* instead is an example of the use of the spe-

cial form ModName.State: in this particular case, the operator [=] is used to check if the actual state of the *Controller* module is the *Start* state. In the same way of action's expressions and conditions, the notation is the usual Lisp prefix notation for functions and operators.

Is is also possible to use composed modules as components of higher level composed modules, but the case study is too simple to show this feature of the tool.

7 CONCLUDING REMARKS

In this thesis, an approach to the formal verification of the modelling of control code for FMSs has been presented and analysed. The thesis has demonstrated the validity of the approach, and its orthogonality respect to the choice of modelling language, formalism and model checking tool; the use of a high level graphical modelling language allows designers to work with a powerful, easy to use, familiar notation, without imposing them the use of the formalism of the underlying verification tool. The translation in a classic linear temporal logic (TRIO formulae are translated into LTL) permits to use a large class of different model checker tools that take this logical formalism as input language, but it is also possible to use a different logical formalism, for example CTL or MTL; the particularity of the approach is the separation of concerns about the encoding of the semantics of the language and the specific model, which must be separated, and the possibility to integrate in a unique Model-driven environment the entire development process.

Furthermore, the particular axiomatization of the run-to-completion semantics of Stateflow-like notation allows designers to check the model for the presence of so-called Zeno behaviours. In addition, the logic-based approach facilitates a rather fine analysis of the temporal behaviour of Stateflow diagrams, since it allows users to separate between different micro-steps of the same macro-step (e.g., the first vs. the last micro-step of the same macro-step) and predicate over them, both in a qualitative (the order of execution of the transitions of the Stateflow diagram) or in a quantitative way (whether, for example, within x micro-step the system reaches a global stable configuration).

The new X-TRIO logic, introducing the notion of infinitesimal duration for micro-steps and by borrowing the elegant terminology of NSA to formalize them, overcomes the limitations of the approaches based on zero-time transitions, which collapses the duration of some action to zero, and the approaches based on different time granularities, such as [59], where different but positive standard and comparable time scales are adopted at different levels of abstraction, and generalize them: on the one side, unlike traditional mappings of different but positive standard time granularities, infinitesimal steps may accumulate in unbounded or unpredictable way, thus allowing for the analysis of usually pathological cases such as Zeno behaviours, in the same

way as the formalization in TRIO described in section 3.3.2; on the other side, by imposing that the effect of an event strictly follows in time its cause, we are closer to the traditional view of dynamical system theory, and we can reason explicitly about possible synchronizations between different components, even at the level of micro-steps, in a more elegant way than in the original formalization.

The choice, made in this thesis, of using one time unit for micro-steps and one for macro-steps is good enough for Stateflow and FMS, but it is not a necessary restriction: different, fixed or even variable durations for micro-steps could be used to model different components of a system and their synchronization at the micro-level; macro- steps, too, could have different durations. Also, non-zero infinitesimal durations for micro-steps are well-suited to investigate dangerous behaviours such as Zeno ones; however, once these have been excluded, one could revert to a finite metric of micro-steps, perhaps exploiting different time granularities: something similar occurs during hardware design where, in various contexts, the designer analyses the risk of critical races and the duration of precise finite sequences of micro-steps, or collapses all such sequences in an abstract zero-time. X-TRIO allows the designer to manage all such "phases" in a uniform and general way.

7.1 FUTURE WORK

Future work will focus on creating a complete Model-driven environment through which FMS experts can seamlessly move from the modelling to the verification of their designs, then receive feedback from the formal verification tool, without having to directly access the formal concepts underlying the environment. The prototype formal verification tools, Sf2Trio and the X-TRIO plugins, that support the technique presented in this paper, show the feasibility of the approach. In the future we will study mechanisms for improving the efficiency of the verification phase, in particular by exploiting the hierarchical and modular nature of the analysed Simulink/Stateflow diagrams. We will also explore the application of the approach to other industry standards besides IEC 61499, such as timed Petri Nets.

With reference to the introduction of the new X-TRIO logic, we also plan to exploit its flexibility in decidability issues. Infact, the trade-off between expressiveness and decidability (efficiency) offers many opportunities. Other fragments of X-TRIO, more general than X-TRIO$^{P}_{\mathbb{N}_+}$, and decision procedures

different from, or complementary to, the translation into PLTLB will be investigated.

BIBLIOGRAPHY

[1] M. Khalgui, O. Mosbahi, H.M. Hanisch, and Z. Li. A multi-agent architectural solution for coherent distributed reconfigurations of function blocks. *Journal of Intelligent Manufacturing*, 23:2531–2549, 2012.

[2] H. Pranevicius. Formal specification and analysis of distributed systems. *Journal of Intelligent Manufacturing*, 9:559–569, 1998.

[3] A. Brusaferri, A. Ballarino, and E. Capanzano. Reconfigurable knowledge-based control solutions for responsive manufacturing systems. In *Studies in Informatics and Control (SIC) Vol. 20*, pages 31–42, 2011.

[4] F. Basile, P. Chiacchio, V. Vittorini, and N. Mazzocca. Modeling and logic controller specification of flexible manufacturing systems using behavioral traces and petri net building blocks. *Journal of Intelligent Manufacturing*, 15:351–371, 2004.

[5] J. Wang and Y. Deng. Incremental modeling and verification of flexible manufacturing systems. *Journal of Intelligent Manufacturing*, 10:485–502, 1999.

[6] D. Zhang and A. Anosike. Modelling and simulation of dynamically integrated manufacturing systems. *Journal of Intelligent Manufacturing*, 23:2367–2382, 2012.

[7] H.M. Hanisch, A. Lobov, J.M. Lastra, R. Tuokko, and V. Vyatkin. Formal validation of intelligent-automated production systems: towards industrial applications. In *International Journal of Manufacturing Technology and Management Vol. 8, No. 1*, pages 75–106, 2006.

[8] M. Mazzolini, A. Brusaferri, and E. Carpanzano. Model-checking based verification approach for advanced industrial automation solutions. In *Proceedings of the International Conference on Emerging Technologies and Factory Automation*, pages 1–8, 2010.

[9] Mathworks. Stateflow online documentation. http://www.mathworks.it/help/toolbox/stateflow/, 2011.

[10] IEC. *International Standard IEC61131-3, Programming Languages for Programmable Controllers*. International Electro-technical Commission, (IEC), 2.0 edition, January 2003.

[11] IEC. *International Standard IEC61499, Function Blocks, Part 1 - 4*. International Electro-technical Commission, (IEC), 1.0 edition, January 2005.

[12] A. Gargantini, D. Mandrioli, and A. Morzenti. Dealing with zero-time transitions in axiom systems. *Information and Computation*, 150(2):119–131, 1999.

[13] ICS Triplex ISaGRAF. Isagraf6 developer web site and online documentation. http://www.isagraf.com, 2012.

[14] A. Ballarino and E. Carpanzano. Modular automation systems design using the IEC 61499 standard and the simulink/stateflow toolboxes. In *Proceedings of the Asme Japan-Usa Symposium on Flexible Automation*, 2002.

[15] A. Robinson. *Non-standard analysis*. Princeton University Press, 1996.

[16] Harel. Statecharts: A visual formalism for complex systems. *Sci. of Comp. Prog.*, 8(3):231–274, 1987.

[17] Harel and Naamad. The STATEMATE semantics of statecharts. *ACM TOSEM*, 5(4):293–333, 1996.

[18] R. Eshuis. Reconciling statechart semantics. *Sci. of Comp. Prog.*, 74:65–99, 2009.

[19] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. *Lecture notes in computer science*, 526:2544 – 265, 1991.

[20] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure. Technical report, OMG, 2010. formal/2010-05-05.

[21] R.W. Lewis. Modelling control systems using iec 61499. applying function blocks to distributed systems. *IEEE Publishing*, 2001.

[22] E. Ciapessoni, A. Coen-Porisini, E. Crivelli, D. Mandrioli, P. Mirandola, and A. Morzenti. From formal models to formally-based methods: an industrial experience. *ACM Transactions on Software Engineering and Methodology*, 8(1):79–113, 1999.

[23] M. Pradella, A. Morzenti, and P. San Pietro. Refining real-time system specifications through bounded model- and satisfiability-checking. In *Proc. of ASE*, pages 119–127, 2008.

[24] E. Carpanzano, L. Ferrucci, D. Mandrioli, M. Mazzolini, A. Morzenti, and M. Rossi. Automated formal verification of flexible manufacturing systems. *Journal of Intelligent Manufacturing*, pages 1–15, 2013.

[25] Microsoft. Z3 online documentation. http://z3.codeplex.com/, 2012.

[26] C.A. Furia and M. Rossi. Integrating discrete- and continuous-time metric temporal logics through sampling. In Eugene Asarin and Patricia Bouyer, editors, *Proceedings of the 4th International Conference on the Formal Modeling and Analysis of Timed Systems (FORMATS'06)*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[27] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[28] F. Diener and M. Diener. *Nonstandard analysis in practice*. Springer-Verlag, 1995.

[29] A. Morzenti, D. Mandrioli, and C. Ghezzi. A model parametric real-time logic. *ACM Transactions on Programming Languages and Systems*, 14 (4):521–573, 1992.

[30] R. Goldblatt. *Lectures on the Hyperreals: An Introduction to Nonstandard Analysis*. Springer, 1998.

[31] C.A. Furia and M. Rossi. A theory of sampling for continuous-time metric temporal logic. *ACM Transactions on Computational Logic*, 12(1): 1–40, 2010. Article 8.

[32] S. Demri and D. D'Souza. An automata-theoretic approach to constraint LTL. *Information and Computation*, 205(3):380–415, 2007.

[33] R. Alur, T. Feder, and T.A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

[34] Y. Hirshfeld and A.M. Rabinovich. Logics for real time: Decidability and complexity. *Fundamenta Informaticae*, 62(1):1–28, 2004.

[35] P. Wolper. Temporal logic can be more expressive. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, pages 340–348, 1981.

[36] R. Alur and T.A. Henzinger. Real-time logics: Complexity and expressiveness. *Information and Computation*, 104(1):35–77, 1993.

[37] R. Alur and T.A. Henzinger. A really temporal logic. *Journal of the ACM*, 41:181 – 203, 1994.

[38] P. Schnoebelen. The complexity of temporal logic model checking. In *Advances in Modal Logic*, pages 393–436, 2002.

[39] V. Vyatkin and H.M. Hanisch. Verification of distributed control systems in intelligent manufacturing. *Journal of Intelligent Manufacturing*, 14:123–136, 2003.

[40] S. Klein, X. Weng, G. Frey, J. Lesage, and L. Litz. Controller design for an FMS using signal interpreted Petri nets and SFC. In *Proceedings of the American Control Conference*, pages 4141–4146, 2002.

[41] V. Vyatkin, H.M. Hanisch, and T. Pfeiffer. Object-oriented modular place/transition formalism for systematic modeling and validation of industrial automation systems. In *Proceedings of the IEEE International Conference on Industrial Informatics*, pages 224–232, 2003.

[42] C. Pang and V. Vyatkin. Towards formal verification of iec61499: modelling of data and algorithms in nces. In *Proceedings of 5th IEEE International Conference on Industrial Informatics*, pages 879–884, 2007.

[43] H. Khalgui and H.M. Hanisch. Nces-based modelling and ctl-based verification of reconfigurable benchmark production systems. In *Proceedings of International Symposium on Industrial Embedded Systems*, pages 1–10, 2008.

[44] M. Rausch and H.M. Hanisch. Net condition/event systems with multiple condition outputs. In *Proceedings of the Symposium on Emerging Technologies and Factory Automation*, pages 592–600, 1995.

[45] H.M. Hanisch and V. Vyatkin. Development of adequate formalisms for verification of iec 61499 distributed applications. In *Proceedings of the 39th SICE Annual Conference*, pages 73–78, 2000.

[46] Q. Zhao and B.H. Krogh. Formal verification of statecharts using finite-state model checkers. In *Proceeding of the American Control Conference*, pages 313–318, 2001.

[47] P. Bhaduri and S. Ramesh. Model checking of statechart models: Survey and research directions. *CoRR*, cs.SE/0407038, 2004.

[48] S. Panjaitan and G. Frey. Functional design for iec 61499 distributed control systems using uml activity diagrams. In *Proceedings of the 2005 International Conference on Instrumentation, Communications and Information Technology*, pages 64–70, 2005.

[49] J.S. Ostroff. *Temporal Logic for Real Time Sytems*. Advanced Software Development Series. John Wiley & Sons, 1989.

[50] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems, 1992.

[51] E.A. Lee and H. Zheng. Operational semantics of hybrid systems. *Lecture Notes in Computer Science*, 3414:25–53, 2005.

[52] E.A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, 2007.

[53] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4:155–182, 1994.

[54] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. Non-standard semantics of hybrid systems modelers. *J. of Comp. and Sys. Sci.*, 78(3):877 – 910, 2012.

[55] S. Bliudze and D. Krob. Modelling of complex systems: Systems as dataflow machines. *Fundam. Inf.*, 91:251–274, 2009.

[56] D.P. Guelev and D. Van Hung. Prefix and projection onto state in duration calculus. *Electr. Notes Theor. Comput. Sci.*, 65(6):101–119, 2002.

[57] T.A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Proc. of the Int. Coll. on Aut., Lang. and Prog.*, volume 623 of *LNCS*, pages 545–558, 1992.

[58] A. Burns and I.J. Hayes. A timeband framework for modelling real-time systems. *Real-Time Systems*, 45(1–2):106–142, 2010.

[59] E. Corsetti, E. Crivelli, D. Mandrioli, A. Morzenti, A. Montanari, P. San Pietro, and E. Ratto. Dealing with different time scales in formal specifications. In *Proc. of the 6th Int. Work. on Software Specification and Design*, pages 92–101, 1991.

[60] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Assume-guarantee refinement between different time scales. In *Proc. of CAV*, volume 1633 of *LNCS*, pages 208–221, 1999.